WEBSITE BOOK EDITION

# Pixels → Perception

From Pixels to Understanding

Foundations • Vision • Language • Learning • Theory

**Volume 3 of 4**

Machine Learning and Deep Learning for Vision

English Edition • Volume 3

2026-11-05

https://blog.shyandram.dev/

# About this edition

This is volume 3 of 4 of the Pixels → Perception course, *Machine Learning and Deep Learning for Vision*. It contains 44 published tutorials and 0 blog posts, in the order of the website's learning tracks. This is a typeset edition of the website, not a separately peer-reviewed publication.

The prose, code, equations, and references retain the website's content. Exercise hints are expanded for print. Static figures remain in the book, image comparisons show their input and output images, and interactive demonstrations link to the website. Reference numbers are local to each tutorial.

The website is still under review. AI tools assisted its production, and errors or omissions may remain. Read it alongside the original sources and consult the website for subsequent corrections.

English and Traditional Chinese, and each volume, are separate PDFs. Links to tutorials in this volume navigate within the PDF. Links to tutorials in other volumes, to specific web sections, interactive demonstrations, notebooks, and external sources remain online. Content snapshot: 2026-11-05

Source revision: `2b3c30d`

https://blog.shyandram.dev/

# Contents


**I Machine Learning** **1**

**ML 01 ·Introduction to Machine Learning** **2**

ML 01A ·Linear Algebra for Learning . . . . . 31

ML 01B ·Probability and Information Theory . . . . . 53

ML 01C ·Numerical Computation and Gradient Checks . . . . . 76

**ML 02 ·Neural Networks from Perceptron to Deep Networks** **101**

**ML 03 ·Convolution, Image Filtering and CNNs** **128**

**ML 04 ·Recurrent and Recursive Networks** **152**

**ML 05 ·Attention and Vision Transformers** **175**

**ML 06 ·Representation Learning and Self-Supervision** **195**

ML 06A ·Linear Factor Models and Autoencoders . . . . . 217

**ML 07 ·Generative Models: Autoencoders, VAEs, GANs and Flows** **240**

**ML 08 ·Diffusion Models and Flow Matching** **264**

**ML 09 ·Autoregressive Vision: Pixels, Tokens and Scales** **300**

**ML 10 ·Foundation Models and Vision–Language Learning** **322**

**ML 11 ·Evaluation, Efficiency and Deployment** **344**

**II Deep Learning for Computer Vision** **372**

**DLCV 01 ·Deep Learning for Image Processing and Computer Vision** **373**

DLCV 01A ·Training Deep Vision Models in Practice . . . . . 398

DLCV 01B ·Building Blocks and How to Compose Them: Classification and Restoration Networks . . . 428
DLCV 01C ·Model Compression and Efficient Inference . . . 458

**DLCV 02 ·Deep Image Classification** **487**
DLCV 02A ·Face Recognition and Metric Learning . . . 523

**DLCV 03 ·Object Detection: From R-CNN to DETR** **553**
DLCV 03A ·One-Stage Detection, Read Through YOLOv4 . . . 580
DLCV 03B ·3D Object Detection: LiDAR, Cameras and Bird's-Eye View . . . 623

**DLCV 04 ·Image Segmentation: Semantic, Instance and Panoptic** **654**

**DLCV 05 ·Human Pose and Keypoint Estimation** **682**

**DLCV 06 ·Text in Images: OCR and Document Understanding** **710**

**DLCV 07 ·Object Tracking: From Single-Object to Multi-Object** **740**

**DLCV 08 ·Video Understanding and Action Recognition** **769**

**DLCV 09 ·Deep Image Restoration: Low-Level Vision** **796**
DLCV 09A ·Non-Blind Image Super-Resolution . . . 841
DLCV 09A.1 ·Blind and Real-World Image Super-Resolution . . . 872
DLCV 09A.2 ·One-Step Diffusion Super-Resolution . . . 898
DLCV 09A.3 ·Video Super-Resolution: Using Time to Recover Detail . . . 912
DLCV 09B ·Generative Image Restoration: Sampling a Plausible Answer . . . 940
DLCV 09C ·Low-Light Image Enhancement . . . 966

**DLCV 10 ·Image Fusion: Complementary Images, Useful Outputs** **1000**

**DLCV 11 ·3D Reconstruction: From Multi-View Geometry to NeRF and Gaussian Splatting** **1035**
DLCV 11A ·Depth Estimation and Optical Flow . . . 1066

**DLCV 12 ·Image Generation: From GANs to Text-to-Image Systems** **1094**
DLCV 12A ·Video Generation . . . 1119
DLCV 12B ·One-Step Generation . . . 1146

DLCV 13 ·World Models: Video, Latent Dynamics and Planning 1162

DLCV 14 ·Robustness: Distribution Shift, Adversarial Examples and Anomalies 1184

# Part I

# Machine Learning

# ML 01 • Introduction to Machine Learning

What it means for a computer to learn from data: the learning paradigms, linear and logistic regression, cost functions, gradient descent, overfitting and evaluation, built from scratch in NumPy.

Online edition: https://blog.shyandram.dev/en/learn/ml/introduction/

## What you'll learn

- Define inputs, features, targets, parameters and losses.
- Distinguish learning paradigms by the feedback they use.
- Fit linear regression with gradient descent and feature scaling.
- Use logistic regression for classification and validation to diagnose fitting.
- Choose metrics and baselines that match the task.

## The big picture

Most programs are recipes: a person thinks hard about a problem and writes down every step. Machine learning flips this. We write a *flexible* program with many adjustable numbers, show it many examples of the job done correctly, and let an algorithm adjust the numbers until the program does the job well. The examples replace the recipe.

> **Explain like I'm five**
>
> Think about how you learned to tell cats from dogs. Nobody gave you a rulebook ("pointy ears plus whiskers minus barking equals cat"). Grown-ups pointed and said "cat", "dog", "cat" many times, and you made mistakes and got corrected until you just *knew*. Machine learning is the same: we show the computer lots of examples with the right answer, it guesses, we tell it how wrong it was, and it nudges its guesses a tiny bit each time until it is rarely wrong.

Why this matters: almost every modern computer-vision system, from phone face unlock to medical image analysis, is a learned model, not a hand-written program. The ideas in this tutorial (a model with parameters, a cost that measures error, gradient descent that lowers the cost, and separate data to check the result) are the same ideas that train today's largest neural networks. Only the size of the model changes. Once these few ideas are clear, the rest of the series (ML 02 and the deep-learning tutorials) is mostly about *which* model and *which* cost.

> **About this tutorial**
>
> This tutorial is an expanded, formal version of the earlier notes Introduction to Machine Learning [1], which followed Andrew Ng's Machine Learning Specialization [4] and the *Deep Learning* book [3].

## What is machine learning?

**Plain version.** A machine-learning system improves at a task by looking at data, instead of having every decision written by a programmer.

### Learning instead of programming

The idea is old. In 1959, Arthur Samuel described a program that learned to play checkers better than he did, by playing many games and adjusting how it scored board positions [2]. His point was that a computer which learns from experience removes much of the detailed, hand-written programming the task would otherwise need. The famous one-line definition, machine learning as the "field of study that gives computers the ability to learn without being explicitly programmed", is a later paraphrase of this idea, usually attributed to Samuel, rather than a sentence from his paper.

A more operational definition, common in textbooks, has three parts: a **task** (for example, recognizing digits), a **performance measure** (the fraction recognized correctly), and **experience** (a collection of labeled digit images). A program *learns* if its performance on the task improves with experience [3], [7].

### An image example: rules versus learning

Our running example is the scikit-learn digits dataset [5], [6]: 1,797 grayscale images of hand-written digits, each only $8 \times 8$ pixels with values from 0 to 16. As in Images as Arrays, each image is just a small array of numbers, and flattening it gives a vector of 64 numbers.

Suppose we only need to tell a **0** from a **1**. A programmer might write a rule: "a 1 is thin, a 0 is wide, so count how many columns contain ink; five columns or fewer means 1." This rule is easy to read, and it is right for most upright 1s. But people write 1s with slants, serifs and bases, and a slanted 1 covers as many columns as a 0 (Figure 1). The rule gets about 79% of the cases right. Fixing it means more rules ("unless it slants..."), and each new rule creates new exceptions.

The machine-learning approach writes *no* digit-specific rule. It picks a flexible family of functions (here, logistic regression, which we build later), and lets an algorithm choose the 65 numbers inside it from examples. On held-out images, it separates 0s from 1s perfectly, and exactly the same code handles all ten digits with no new rules:

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

digits = load_digits()                      # 1,797 images, 8x8 pixels, values 0..16
X, y = digits.data, digits.target           # X: (1797, 64) flattened pixels; y: 0..9

# Hand-written rule for "0 versus 1": a 1 is thin, a 0 is wide.
```

```
keep = (y == 0) | (y == 1)
imgs, labels = digits.images[keep], y[keep]
width = (imgs.max(axis=1) > 4).sum(axis=1)   # how many columns contain ink
rule = np.where(width <= 5, 1, 0)
print("hand rule, 0 vs 1:", round((rule == labels).mean(), 3))          # 0.789

# Learned rule for the same task, judged on images it never saw.
X01 = imgs.reshape(len(imgs), -1)
Xa, Xb, ya, yb = train_test_split(X01, labels, test_size=0.3, random_state=0)
print("learned, 0 vs 1:", LogisticRegression(max_iter=1000).fit(Xa, ya).score(Xb, yb))   #
 ↪  1.0

# The same recipe scales to all ten digits with no new rules.
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, random_state=0)
print("learned, 10 digits:",
      round(LogisticRegression(max_iter=5000).fit(X_tr, y_tr).score(X_te, y_te), 3))  # 0.953
```

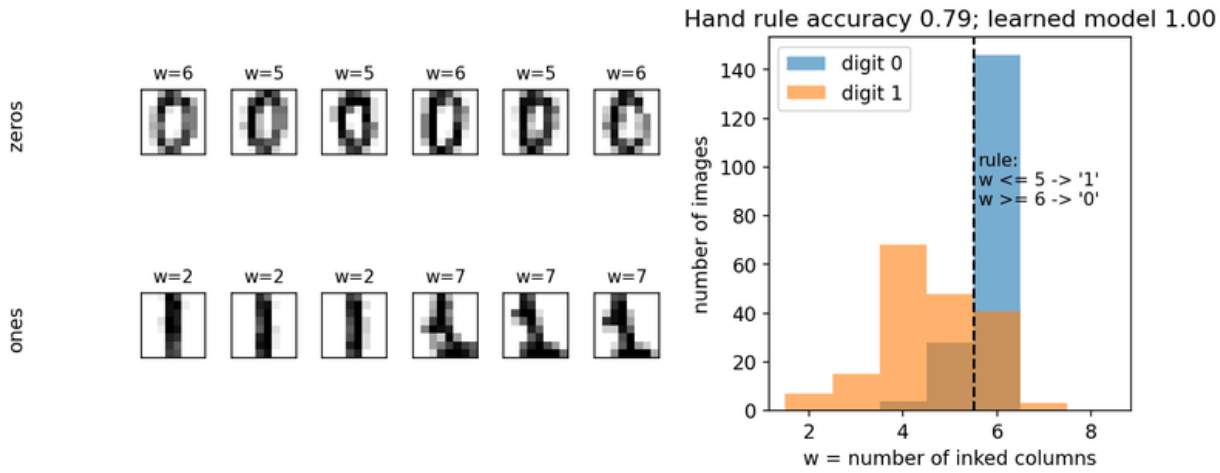



*Figure 1 — Why hand-written rules are brittle. Left: zeros and ones from the digits dataset; slanted ones are as wide as zeros. Right: the 'count inked columns' feature overlaps heavily between the two classes, so any threshold on it makes many mistakes. A learned model uses all 64 pixels at once.*

This small experiment contains the whole argument for machine learning. When the rule is hard to put into words but examples are cheap, learn the rule from examples. It also shows the price: the learned model is a list of 65 numbers that is much harder to read than "count the columns."

## The learning paradigms

**Plain version.** Ask where feedback comes from: supplied labels, structure in unlabeled data, targets derived from the data itself, or rewards for actions.

| Paradigm | What the data looks like | What is learned | Typical example |
|---|---|---|---|
| Supervised | pairs $(x, y)$: input and the "right answer" | a function from $x$ to $y$ | digit image → digit label |
| Unsupervised | inputs $x$ only | structure: groups, directions, outliers | grouping similar digits |
| Self-supervised | inputs $x$ only, but part of $x$ is hidden and used as the target | a general-purpose representation | predicting a masked word or image patch |
| Reinforcement | an agent's actions and the rewards they earn | a policy: which action to take in each situation | game playing, tuning chat models |

My original notes listed the first, second and fourth; self-supervised learning has since become the main way large models are pre-trained, so it gets its own row [9], [10].

### Supervised learning

In supervised learning, the learner is given the right answers. The data is a set of examples $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, where $x^{(i)}$ is an input, $y^{(i)}$ is the desired output (the **label** or **target**), and $m$ is the number of examples. The goal is a function $f$ such that $f(x)$ is close to $y$, *also for new inputs not in the data*. That last part, called **generalization**, is the real goal; memorizing the examples is easy.

### Unsupervised learning

In unsupervised learning there are no answers, only inputs $\{x^{(i)}\}$. The goal is to find something interesting in the data: groups of similar examples, a compact description, or examples that do not fit. There is no single "correct" output, which makes these methods useful for exploration and harder to evaluate.

### Self-supervised learning

Self-supervised learning uses unlabeled data but turns it into a supervised problem by hiding part of each example and asking the model to predict the hidden part. In text, hide a word and predict it from the rest of the sentence. In images, hide most of the patches and reconstruct them: masked autoencoders hide 75% of an image's patches and train a network to fill them in [12]. LeCun and Misra describe self-supervised learning as a way for models to acquire broad background knowledge from raw observation, because the labels come for free [9].

The point is not the fill-in-the-blank task itself; it is that a model which can fill in blanks well must have learned a lot about the structure of the data. The learned internal **representation** can then be adapted to many other tasks with few labels. This is the recipe behind **foundation models**: models trained on broad data at scale (usually self-supervised) and then adapted to a wide range of downstream tasks [10]. GPT-3, trained simply to predict the next word on a large text corpus, was able to perform many language tasks from a few examples given in the prompt, without further training [11].

### Reinforcement learning

In reinforcement learning (RL), an **agent** interacts with an **environment**. At each step it observes a **state**, chooses an **action**, and receives a **reward** (a number) and a new state. Nobody tells it the right action; it must discover which actions lead to high total reward over time, often when the reward arrives long after the action that earned it. What is learned is a **policy**, a rule for choosing actions in each state [8]. Samuel's checkers player is an early example of this setting [2].

Today, RL is best known for tuning large language models. In **reinforcement learning from human feedback** (RLHF), as described by Ouyang et al. [13], a pre-trained language model is first fine-tuned on demonstrations written by people (supervised learning). People then rank several model answers to the same prompt, and a separate **reward model** is trained to predict those rankings. Finally, the language model is optimized with RL to produce answers that the reward model scores highly. The authors report that labelers preferred the outputs of their 1.3-billion-parameter tuned model over those of the 175-billion-parameter GPT-3 [13]. More recently, DeepSeek-R1 used large-scale RL with automatically checkable rewards (such as whether a math answer is correct) to improve step-by-step reasoning [14].

Notice that RLHF chains three paradigms: self-supervised pre-training, supervised fine-tuning, and reinforcement learning. Real systems mix paradigms freely; the table above is a map, not a set of walls.

## Supervised learning: regression and classification

**Plain version.** Supervised learning comes in two main kinds. **Regression** predicts a number; **classification** predicts a category.

### Regression

In regression the target $y$ is a real number, so there are infinitely many possible outputs. Examples: predicting a stock price, a student's final grade from earlier test scores, or a person's age from a face image. The last one shows that the input can be an image while the output is still a single number.

### Classification

In classification the target is one of a small, fixed set of categories (classes). Examples: spam versus not spam (two classes, **binary classification**), the shape category of a face, or a tumor type such as benign, malignant type 1 or malignant type 2 (**multi-class classification**). Classes are names, not quantities: "class 2" is not twice "class 1". A classifier usually outputs a probability for each class and then picks the most likely one.

```
from sklearn.datasets import make_regression, make_classification
```

```
from sklearn.linear_model import LinearRegression, LogisticRegression

# Regression: the target is a real number.
X_r, y_r = make_regression(n_samples=100, n_features=1, noise=10.0, random_state=0)
reg = LinearRegression().fit(X_r, y_r)
print(reg.predict(X_r[:3]).round(1), y_r[:3].round(1))   # numbers vs numbers

# Classification: the target is one of a few labels.
X_c, y_c = make_classification(n_samples=100, n_features=2, n_redundant=0,
                               n_clusters_per_class=1, random_state=0)
clf = LogisticRegression().fit(X_c, y_c)
print(clf.predict(X_c[:5]), y_c[:5])                     # labels vs labels
print(clf.predict_proba(X_c[:2]).round(3))               # class probabilities
```

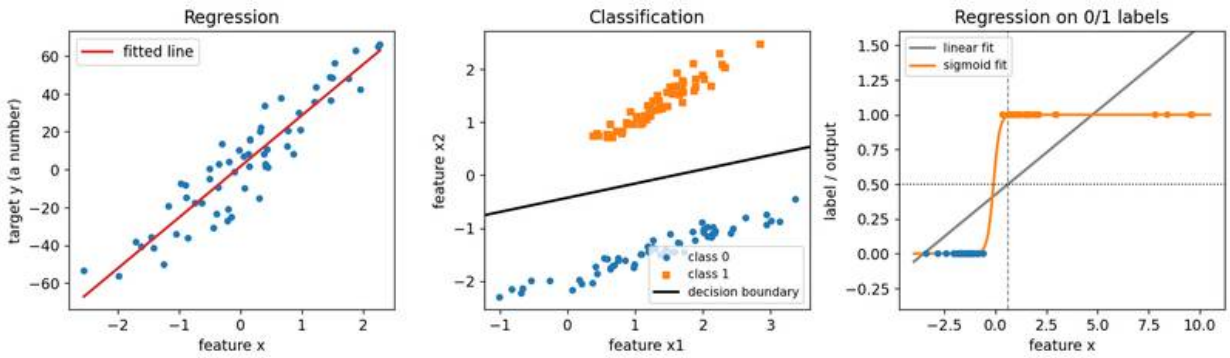



*Figure 2 — Left: regression fits a curve through numeric targets. Middle: classification separates labeled points with a decision boundary. Right: fitting a straight regression line to 0/1 labels (gray) gives outputs outside $[0, 1]$, and far-away points drag its 0.5 crossing; a sigmoid (logistic regression, orange) is the right tool.*

### Why not use regression for classification?

It is tempting to code the classes as 0 and 1, fit a regression line, and call anything above 0.5 a 1. This can work on easy data, but it usually performs badly, for two reasons visible in Figure 2 (right). First, the line outputs values like $-0.4$ or 1.7, which are not probabilities. Second, the squared error punishes a point that is *correctly* classified but far from the boundary, so a few extreme but easy examples pull the line and move its 0.5 crossing to the wrong place. Logistic regression (later in this tutorial) fixes both problems.

## Unsupervised learning

**Plain version.** Without any answers, we can still ask: which examples belong together, how can each example be described with fewer numbers, and which examples are strange?

### Clustering

**Clustering** groups similar examples. The classic algorithm is **k-means** [15]. Choose the number of groups $K$. Start with $K$ center points, then repeat two steps until nothing changes: assign each example to its nearest center, and move each center to the mean of the examples assigned to it. Each step can only lower the total squared distance from examples to their centers,

$$\sum_{i=1}^{m} \|x^{(i)} - \mu_{c^{(i)}}\|^2,$$

where $\mu_k$ is the center of cluster $k$ and $c^{(i)}$ is the cluster that example $i$ is assigned to. So the algorithm always stops, but possibly at a poor local solution, which is why it is run from several random starts. Jain's review of fifty years of clustering research [15] stresses that there is no best clustering algorithm in general: the "right" groups depend on what similarity means for the data.

A common confusion: **k-nearest neighbors** (KNN) is *not* a clustering method. KNN classifies a new example by a vote among the $k$ most similar *labeled* training examples, so it is a supervised method [5]. The "k" in k-means counts clusters; the "k" in KNN counts neighbors.

### Dimensionality reduction

**Dimensionality reduction** describes each example with fewer numbers while keeping as much useful information as possible. **Principal component analysis** (PCA) finds the directions along which the data varies most and projects the data onto the top few of them; the new coordinates are uncorrelated and successively capture as much variance as possible [16]. Two PCA coordinates can turn 64-pixel digit images into points on a plane that we can look at (Figure 3, middle).

### Anomaly detection

**Anomaly detection** finds examples that do not fit the pattern of the rest: a fraudulent transaction, a defective part, a corrupted image. Chandola, Banerjee and Kumar's survey [17] organizes the many techniques by the assumption each one makes about what "normal" looks like. A simple version uses PCA: fit a few components on the data, rebuild each example from them, and flag those with a large reconstruction error, because the components only describe what is common.

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA

X, y = load_digits(return_X_y=True)             # y is used only to check, never to fit

# Clustering: ten groups, found without looking at y.
km = KMeans(n_clusters=10, n_init=10, random_state=0).fit(X)
purity = sum(np.bincount(y[km.labels_ == k]).max() for k in range(10)) / len(y)
print("cluster purity:", round(purity, 3))      # 0.792: most clusters are one digit
```

```
# Dimensionality reduction: 64 numbers per image -> 2 numbers.
Z = PCA(n_components=2).fit_transform(X)        # shape (1797, 2)

# Anomaly detection: images that a 20-component PCA rebuilds badly are unusual.
pca20 = PCA(n_components=20).fit(X)
err = ((X - pca20.inverse_transform(pca20.transform(X))) ** 2).sum(axis=1)
print("most unusual images:", np.argsort(err)[-5:])
```

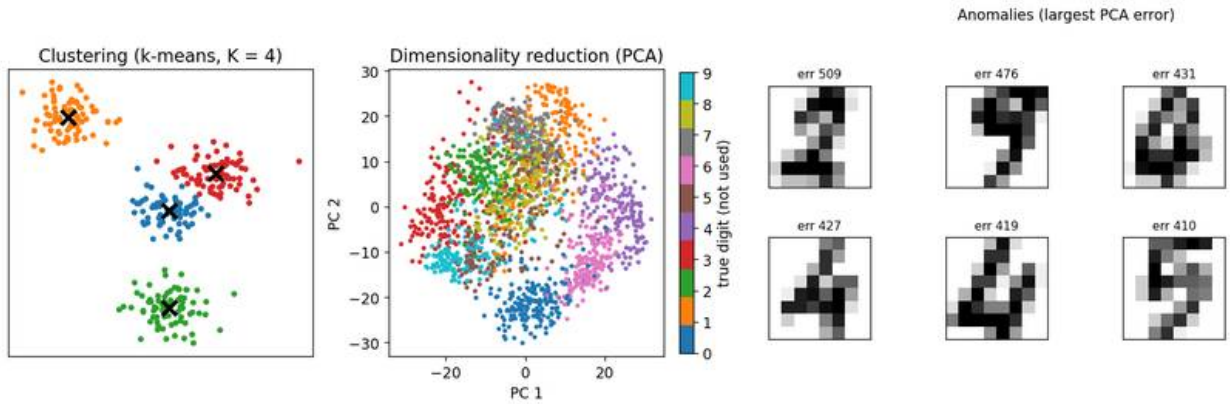



*Figure 3 — Unsupervised learning. Left: k-means groups points around K centers. Middle: PCA compresses each 64-pixel digit to two numbers (colors show the true digits, which PCA never saw). Right: the digits with the largest PCA reconstruction error are the oddest-looking ones.*

The purity check in the snippet uses the labels only *after* clustering, to see whether the discovered groups match the digits. About 79% of the images fall in a cluster dominated by their own digit, even though k-means never saw a label.

## The vocabulary of a learning problem

**Plain version.** Before any math, we need names for the parts: what goes in, what comes out, what the right answer is, what the model is, and how we measure its mistakes.

### Inputs, outputs and targets

- **Features** (inputs), $x$: the numbers that describe one example. For a digit image, the 64 pixel values; for a house, its size and number of rooms. With $n$ features we write $x = (x_1, \dots, x_n)$, and $x_j$ is the $j$-th feature.
- **Target**, $y$: the correct output, also called the label or **ground truth**. Only supervised learning has targets.
- **Prediction**, $\hat{y}$ ("y-hat"): what the model outputs for an input. The hat means "estimated".
- **Training example** $i$: the pair $(x^{(i)}, y^{(i)})$. The superscript in parentheses is an index, not a power. $x_j^{(i)}$ is feature $j$ of example $i$.

### The model and its parameters

The **model** is the function that maps features to a prediction. We write it $f_{w,b}$ to show that it depends on adjustable numbers called **parameters**, here $w$ (the **weights**) and $b$ (the **bias**):

$$\hat{y} = f_{w,b}(x).$$

Choosing the *form* of $f$ (a line, a curve, a neural network) is a design decision. Choosing the *values* of $w$ and $b$ is what learning does. Numbers we set by hand, such as the learning rate or polynomial degree, are called **hyperparameters** to distinguish them from the learned parameters.

### Cost function

A **loss** $L(\hat{y}, y)$ measures how wrong one prediction is. The **cost function** $J(w, b)$ averages the loss over the training set, so it is a single number that says how wrong the whole model is for a given choice of parameters. The model is $f$; the cost is $J$. Keeping these two apart avoids a lot of confusion: $f_{w,b}(x)$ takes an input and returns a prediction, while $J(w, b)$ takes parameters and returns an error.

The **objective** of training is then precise: find the parameters that make $J$ as small as possible, so that $\hat{y}^{(i)}$ is close to $y^{(i)}$ for all training examples.

### Training, validation and test sets

The data is split into three parts with different jobs:

- the **training set** is used to fit the parameters;
- the **validation set** is used during development to compare models and choose hyperparameters;
- the **test set** is used once, at the end, to estimate how the final model will do on new data.

The test set must not influence any decision, or the estimate becomes optimistic. We come back to *why* this split matters in the section on overfitting.

*Online only: Domingos — A Few Useful Things to Know About Machine Learning — https://blog.shyandram.dev/en/learn/ml/introduction/papers/domingos–a-few-useful-things-to-know-about-machine-learning/*

## Linear regression

**Plain version.** Fit a straight line through the data. The line has two knobs: its slope and its height. Learning means turning the knobs until the line passes as close as possible to the points.

### The model

With one feature, the linear regression model is

$$f_{w,b}(x) = wx + b,$$

where $x$ is the feature, $w$ is the slope (how much the prediction changes when $x$ grows by 1) and $b$ is the intercept (the prediction when $x = 0$). Changing $w$ tilts the line; changing $b$ moves it up and down (Figure 4, left).

With $n$ features, the same model is a weighted sum,

$$f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b = \sum_{j=1}^{n} w_j x_j + b,$$

where $\mathbf{w} = (w_1, \dots, w_n)$ holds one weight per feature. A common trick appends a constant feature $x_0 = 1$, so the bias becomes just another weight $w_0$ and the model shortens to $\mathbf{w} \cdot \mathbf{x}$. Either way, "linear" means linear in the *parameters*; we will later feed it squared or cubed features and it remains a linear regression.

### The cost function: mean squared error

For each example, the **error** (residual) is $f_{w,b}(x^{(i)}) - y^{(i)}$. Squaring makes errors positive and punishes big ones more than small ones. Averaging over the $m$ training examples gives the **mean squared error** (MSE) cost:

$$J(w, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)}\right)^2,$$

where $m$ is the number of training examples and $(x^{(i)}, y^{(i)})$ is the $i$-th example. The extra factor $\frac{1}{2}$ is a convenience: it cancels the 2 that appears when we differentiate the square, and it does not change where the minimum is.

Picture the cost as a landscape over the parameters. If we fix $b$ and vary only $w$, $J$ is a parabola: lines that are too flat or too steep both have large error, and one slope in between is best (Figure 4, middle). With both $w$ and $b$, $J$ is a bowl, which we draw as a contour map of ellipses (Figure 4, right). The best line sits at the bottom of the bowl.

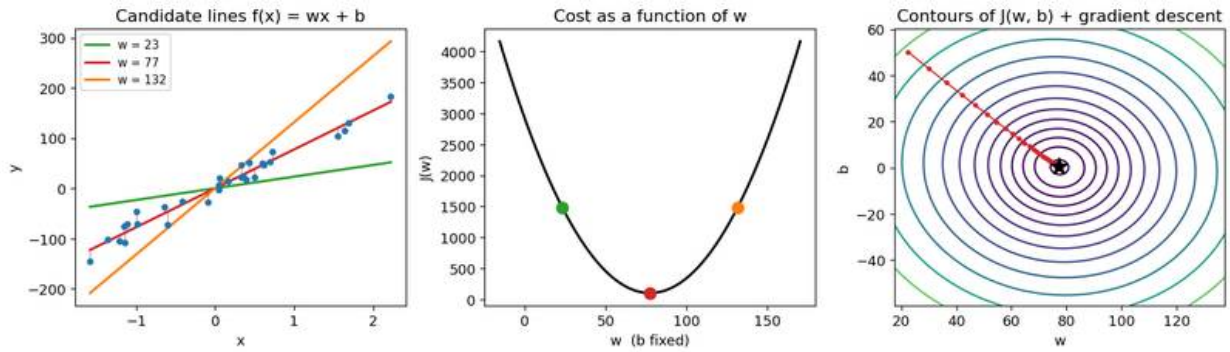



*Figure 4 — From lines to costs. Left: three candidate lines; vertical segments are the residuals of the best one. Middle: with b fixed, $J(w)$ is a parabola, and each line is one point on it. Right: $J(w, b)$ is a bowl (contours), and gradient descent walks downhill to its bottom.*

### Why this bowl has one bottom

For linear regression, the MSE cost is a **convex** function of $(w, b)$: any straight segment between two points on its surface lies on or above the surface. Every local minimum is global, but the minimizing parameters are unique only when the design matrix (including the bias column) has full column rank. Redundant features can produce a flat set of equally good solutions. For neural networks in ML 02, a small gradient can also indicate a saddle point; neither convexity nor a global optimum should be assumed.

Linear least squares can also be solved directly, without iterative gradient descent. The normal equations characterize its stationary points, but numerical solvers need not form them: `np.polyfit` uses a least-squares solve and reports the scaled design matrix's singular values [27]. ML 01A explains why this matters for conditioning. We introduce gradient descent because it extends to models without such a direct solve.

## Gradient descent

**Plain version.** Like walking downhill in fog, gradient descent takes small steps against the local slope. Flat ground alone does not tell you whether you reached the bottom.

### The update rule

The **derivative** $\partial J/\partial w$ tells us how $J$ changes when $w$ increases a little: positive means "uphill to the right", negative means "uphill to the left". Its sign gives the direction, and its size says how steep the slope is. Gradient descent moves each parameter a small step *against* its derivative:

$$w \leftarrow w - \alpha \frac{\partial J(w,b)}{\partial w},$$
$$b \leftarrow b - \alpha \frac{\partial J(w,b)}{\partial b},$$

where $\alpha > 0$ is the **learning rate** (step size) and $\leftarrow$ means "replace with". Both derivatives are computed at the *current* $(w, b)$ before either is changed; this is called a **simultaneous update**. The vector of all partial derivatives is the **gradient** $\nabla J$, which points in the direction of steepest increase, so the update moves in the direction of steepest decrease [3].

For the MSE cost of linear regression, the chain rule gives

$$\frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right)x^{(i)}, \qquad \frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right).$$

Read them in words: the gradient for $w$ is the average error times the feature; the gradient for $b$ is the average error. If predictions are too high on average, $b$ goes down. If predictions are too high mainly where $x$ is large, $w$ goes down. With $n$ features, $\partial J/\partial w_j$ uses $x_j^{(i)}$ in place of $x^{(i)}$.

### Implementing it from scratch

```
import numpy as np
from sklearn.datasets import make_regression

X, y = make_regression(n_samples=100, n_features=1, noise=10.0, random_state=0)
x = X[:, 0]                                  # one feature, shape (m,)
m = len(x)

def cost(w, b):
    return ((w * x + b - y) ** 2).sum() / (2 * m)

w, b, alpha = 0.0, 0.0, 0.1
for step in range(200):
    err = w * x + b - y                      # f_wb(x_i) - y_i for every i
    dj_dw = (err * x).sum() / m
    dj_db = err.sum() / m
    w, b = w - alpha * dj_dw, b - alpha * dj_db   # simultaneous update
    if step % 50 == 0:
        print(f"step {step:3d}  J = {cost(w, b):8.2f}")

print("gradient descent:", round(w, 3), round(b, 3))   # 42.619 -0.814
print("closed form     :", np.polyfit(x, y, 1).round(3))  # [42.619 -0.814]
```

The cost drops from about 800 to 57 within 50 steps and then stops changing. Gradient descent and the exact formula agree to three decimals. The remaining cost is not zero, because the data contains noise that no straight line can explain.

### Convergence: where the derivative is zero

"Repeat until convergence" needs a definition. At a minimum, the landscape is flat, so the **derivative is zero** there, and the update $w - \alpha \cdot 0$ leaves $w$ unchanged. Note that it is the *derivative*, not the cost, that becomes zero: the cost at the minimum is just the smallest value available, 57 in the example above. In practice we stop when the cost decreases by less than a small tolerance between iterations, or when the gradient's size falls below a threshold, or after a fixed budget of steps.

A useful consequence: as we approach a minimum, the derivative shrinks, so even with a *fixed* $\alpha$ the steps $\alpha\, \partial J/\partial w$ get smaller automatically. Gradient descent naturally slows down near the bottom.

### Choosing the learning rate

The learning rate is the most important hyperparameter of gradient descent (Figure 5):

- **Too small:** every step is tiny. The algorithm does go downhill, but it may need a very large number of steps.
- **About right:** the cost falls quickly and settles.
- **Too large:** a step jumps past the bottom to the other side of the valley, possibly higher than where it started. The parameters bounce back and forth, may never settle (**overshoot**), and can fly off to infinity (**divergence**).

For this fixed, full-batch quadratic, plot the same cost $J$ after each update: a sufficiently small constant learning rate makes it decrease. An increase is a useful reason to inspect the step size or gradient. This is **not** a rule that every training loss must fall on every update: mini-batch sampling, augmentation and stochastic layers change the quantity observed, and even useful stochastic updates can raise the full-data loss [18]. Inspect a consistent evaluation loss and its trend over many updates. For the deterministic example, try learning rates spaced by about a factor of 3 and choose one whose cost decreases smoothly.

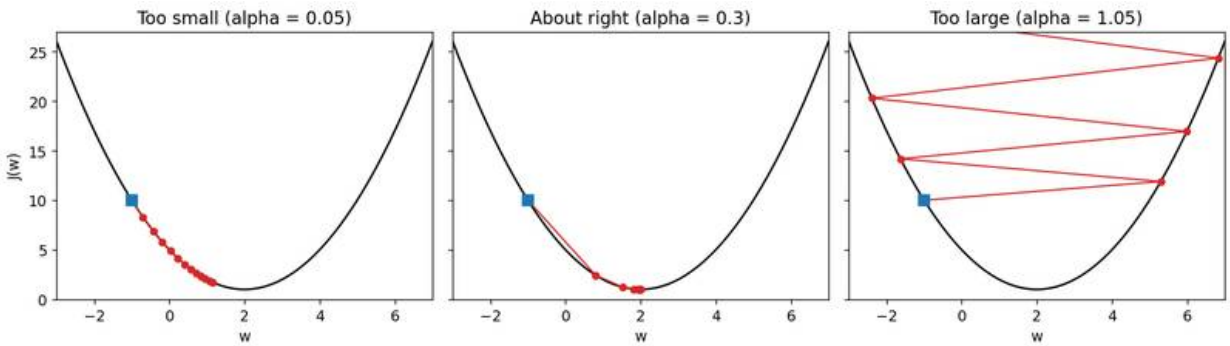



*Figure 5 — The same cost $J(w)$, starting from the same point, with three learning rates. Too small: slow progress. About right: fast convergence. Too large: each step overshoots further, and the iterates diverge.*

Many optimizers adapt the step size during training, either by a **schedule** that lowers $\alpha$ over time

or by scaling each parameter's step using running statistics of its past gradients, as Adam does [19]. These make "the steps become smaller near the minimum" an explicit design choice rather than just a side effect.

### Batch, mini-batch and stochastic gradient descent

The gradient formula sums over all $m$ examples. That is fine for 100 examples and a problem for 100 million. The key insight, explained clearly in [3] and analyzed in depth by Bottou, Curtis and Nocedal [18], is that the gradient is an *average*, that is, an expectation over the data, and an average can be estimated from a sample.

- **Batch gradient descent** uses the *whole* training set for every update. Each step is exact and expensive.
- **Mini-batch gradient descent** shuffles the data and, for each update, uses a small random subset (a **mini-batch**) of $B$ examples, typically 32 to a few hundred. Each step is a noisy but unbiased estimate of the full gradient, and costs $B/m$ as much.
- **Stochastic gradient descent** (SGD), in the strict sense, uses one example per update ($B = 1$). In practice, "SGD" often refers to the mini-batch version too.

One full pass through the training data is an **epoch**. Beware of naming in code: the `batch_size` argument in deep-learning libraries means the *mini-batch* size, not the batch-gradient-descent "whole set".

The noise in mini-batch gradients is not only tolerable but often helpful: in the same number of epochs, mini-batch descent makes many more updates than batch descent. Bottou et al. argue that this is why stochastic methods dominate large-scale learning: they reach a useful accuracy with far less computation than exact methods [18].

```
import numpy as np
from sklearn.datasets import make_regression

X, y = make_regression(n_samples=10_000, n_features=5, noise=10.0, random_state=0)
m, n = X.shape
rng = np.random.default_rng(0)

def minibatch_gd(batch_size, alpha=0.05, epochs=5):
    w, b = np.zeros(n), 0.0
    for epoch in range(epochs):
        order = rng.permutation(m)                    # shuffle once per epoch
        for start in range(0, m, batch_size):
            idx = order[start:start + batch_size]
            err = X[idx] @ w + b - y[idx]
            w -= alpha * X[idx].T @ err / len(idx)    # gradient estimated on the batch
            b -= alpha * err.mean()
    return ((X @ w + b - y) ** 2).mean() / 2

for bs in [m, 64, 1]:                                 # batch, mini-batch, stochastic
    print(f"batch size {bs:5d}: J = {minibatch_gd(bs, alpha=0.05 if bs > 1 else 0.005):.2f}")
# batch size 10000: J = 6127.00   (only 5 updates in 5 epochs)
```

```
# batch size    64: J = 49.93     (785 updates)
# batch size     1: J = 50.64     (50,000 noisy updates)
```

With the same five passes over the data, batch descent has made only five steps and is still far from the bottom, while mini-batch descent has essentially converged (the noise level of this data puts the best achievable $J$ near 50). Single-example SGD gets there too, but needs a smaller learning rate because its gradients are noisier, and it cannot use vectorized arithmetic efficiently.

## Feature scaling

**Plain version.** Unequal feature scales can make gradient descent zigzag. Scaling can improve the geometry, but does not guarantee a round loss surface.

### Why scale matters

Suppose feature $x_1$ (say, house area in square feet) ranges over thousands and feature $x_2$ (number of bedrooms) ranges from 1 to 5. A small change in $w_1$ changes the predictions a lot; a similar change in $w_2$ barely matters. So $J$ is extremely steep along $w_1$ and very flat along $w_2$: the contours are thin ellipses. A learning rate small enough not to diverge along the steep direction makes almost no progress along the flat one (Figure 6, left).

### Standardization and min–max scaling

The most common fix is **standardization** (z-score normalization):

$$x'_j = \frac{x_j - \mu_j}{\sigma_j},$$

where $\mu_j$ and $\sigma_j$ are the mean and standard deviation of feature $j$ *computed on the training set*. Each scaled feature then has mean 0 and standard deviation 1. An alternative is **min–max scaling**, $x'_j = (x_j - \min_j)/(\max_j - \min_j)$, which maps the training range to $[0, 1]$ and is the usual choice for pixel intensities (divide by 255 for 8-bit images). Both are available in scikit-learn as `StandardScaler` and `MinMaxScaler` [5].

Two rules prevent subtle bugs. Compute $\mu_j, \sigma_j$ on the training set only, then apply the *same* numbers to validation, test and future data. And remember that scaling changes the meaning of the learned weights: a weight now says how much the prediction changes per standard deviation of the feature.

```
import numpy as np
from sklearn.datasets import make_regression

X, y = make_regression(n_samples=200, n_features=2, noise=5.0, random_state=1)
X = X * np.array([1000.0, 1.0])              # feature 1 now has a ~1000x wider range
```

```
def gd(X, y, alpha, steps=500):
    m, n = X.shape
    w, b = np.zeros(n), 0.0
    for _ in range(steps):
        err = X @ w + b - y
        w -= alpha * X.T @ err / m
        b -= alpha * err.mean()
    return ((X @ w + b - y) ** 2).mean() / 2

print("raw,    alpha=1e-7:", round(gd(X, y, 1e-7), 2))    # 158.01; alpha=3e-6 diverges
mu, sigma = X.mean(axis=0), X.std(axis=0)
Xs = (X - mu) / sigma                          # z-score standardization
print("scaled, alpha=0.1 :", round(gd(Xs, y, 0.1), 2))    # 13.47
```

On the raw features, any learning rate above about $2 \times 10^{-6}$ diverges, and with a safe one the cost is still stuck near 158 after 500 steps (it is still near 158 after 5,000), because the second weight barely moves. After standardization, $\alpha = 0.1$ reaches the minimum (about 13, the noise level) easily.

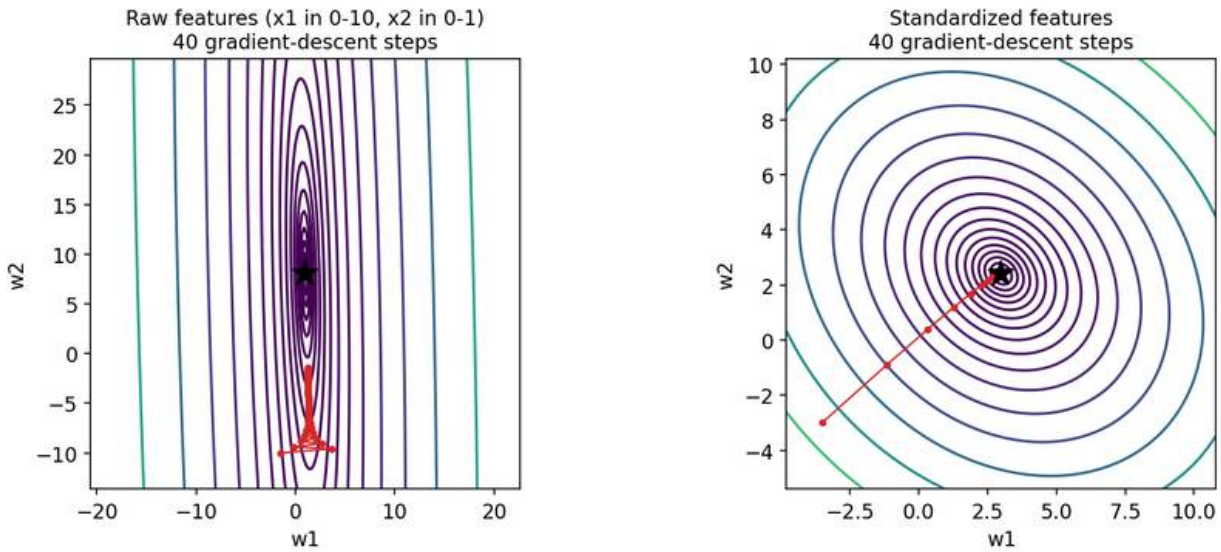



*Figure 6 — Feature scaling reshapes the cost landscape. Left: features on very different scales give a narrow valley, and gradient descent zigzags. Right: after standardization, the contours are nearly circular, and the same number of steps reaches the minimum.*

## Logistic regression for classification

**Plain version.** Keep the weighted sum from linear regression, but squash its output into the range 0 to 1 so it can be read as a probability. Then choose the weights so that the probability is high for the true class.

### The sigmoid function

The **sigmoid** (logistic) function is

$$g(z) = \frac{1}{1 + e^{-z}},$$

where $z$ is any real number. It maps $-\infty$ to 0, $+\infty$ to 1 and 0 to exactly 0.5, with an S-shaped curve in between (Figure 7, left). It is also symmetric, $g(-z) = 1 - g(z)$, and has the convenient derivative $g'(z) = g(z)\,(1 - g(z))$.

### The model and the decision boundary

**Logistic regression** feeds the linear model into the sigmoid:

$$f_{\mathbf{w},b}(\mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + e^{-(\mathbf{w}\cdot\mathbf{x}+b)}},$$

and interprets the output as the estimated probability that the label is 1, $f_{\mathbf{w},b}(\mathbf{x}) = P(y = 1 \mid \mathbf{x})$. Despite the name, it is a classification method. To make a decision, predict 1 when this probability is at least 0.5. Since $g(z) \geq 0.5$ exactly when $z \geq 0$, the rule is

$$\hat{y} = 1 \quad \text{if} \quad \mathbf{w} \cdot \mathbf{x} + b \geq 0, \qquad \hat{y} = 0 \text{ otherwise.}$$

The set where $\mathbf{w} \cdot \mathbf{x} + b = 0$ is the **decision boundary**. With two features it is a straight line; in general it is a flat hyperplane, with $\mathbf{w}$ perpendicular to it. Points far from the boundary get probabilities near 0 or 1; points near it get probabilities near 0.5. To obtain a curved boundary, feed the model nonlinear features such as $x_1^2$ or $x_1 x_2$; the model stays linear in its parameters.

### The cost: cross-entropy (log loss)

Why not reuse squared error? Combined with the sigmoid, the squared-error cost is *not* convex in $(\mathbf{w}, b)$: it develops flat plateaus and multiple valleys. Instead, logistic regression uses the **log loss**, also called **binary cross-entropy**:

$$L(f, y) = -y \log f \; - \; (1 - y) \log(1 - f),$$

where $f = f_{\mathbf{w},b}(\mathbf{x})$ is the predicted probability and $y \in \{0, 1\}$ is the true label. Only one of the two terms is active for each example. If $y = 1$, the loss is $-\log f$: zero when $f = 1$, and growing without bound as $f \to 0$. If $y = 0$, the loss is $-\log(1 - f)$, the mirror image (Figure 7, middle). Confident wrong answers are punished very heavily; confident right answers cost almost nothing.

The cost is the average over the training set:

$$J(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \Big[ -\, y^{(i)} \log f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - (1 - y^{(i)}) \log\left(1 - f_{\mathbf{w},b}(\mathbf{x}^{(i)})\right) \Big].$$

This choice is not arbitrary. If we treat each label as a coin flip with probability $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ of coming up 1, then $J$ is exactly the negative log-likelihood of the training labels, divided by $m$. Minimizing

it is **maximum likelihood estimation** [3]. And with the sigmoid, this cost is convex, so gradient descent again finds the single global minimum.

### Gradient descent, again

Using $g' = g(1-g)$, the derivative of the log loss simplifies remarkably:

$$\frac{\partial J}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}, \qquad \frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}\right).$$

This has *exactly the same form* as the linear-regression gradient: average error times feature. The only difference is inside $f$. This is not a coincidence; it holds for a whole family of models that pair an output function with its matching likelihood, and it is one reason gradient descent code is so reusable.

```
import numpy as np
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=200, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, class_sep=1.5, random_state=3)

def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

def log_loss(p, y, eps=1e-12):
    p = np.clip(p, eps, 1 - eps)              # avoid log(0)
    return -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()

w, b, alpha = np.zeros(2), 0.0, 0.5
for step in range(1000):
    p = sigmoid(X @ w + b)                    # predicted P(y = 1 | x)
    w -= alpha * X.T @ (p - y) / len(y)       # same form as linear regression
    b -= alpha * (p - y).mean()

p = sigmoid(X @ w + b)
print("log loss:", round(log_loss(p, y), 3))  # 0.06
print("accuracy:", ((p >= 0.5) == y).mean())  # 0.99
print("boundary: %.2f*x1 + %.2f*x2 + %.2f = 0" % (w[0], w[1], b))
```

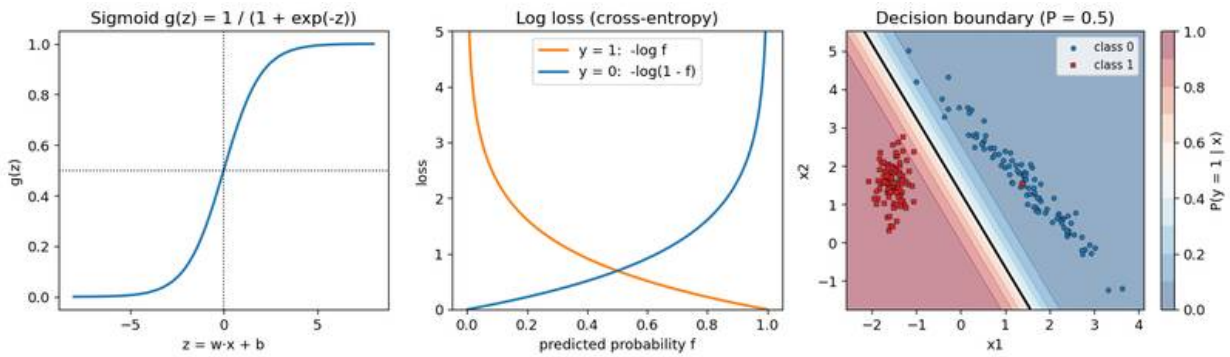


*Figure 7 — Logistic regression. Left: the sigmoid squashes any score into $(0, 1)$. Middle: the log loss for $y = 1$ and $y = 0$ punishes confident mistakes heavily. Right: a model trained from scratch; shading is the predicted probability, and the black line is the decision boundary $w \cdot x + b = 0$.*

### More than two classes

For $K > 2$ classes, the standard generalization gives each class its own weight vector and score $z_k = \mathbf{w}_k \cdot \mathbf{x} + b_k$, and turns the scores into probabilities that sum to 1 with the **softmax** function, $P(y = k \mid \mathbf{x}) = e^{z_k} / \sum_{l=1}^{K} e^{z_l}$. The cost is again the average negative log probability of the true class (categorical cross-entropy) [3]. This is what scikit-learn's `LogisticRegression` did for the ten digits at the start of this tutorial, and it is exactly the output layer of most classification networks in ML 02.

## Overfitting, underfitting and regularization

**Plain version.** High training error suggests underfitting or optimization trouble. Low training error with high validation error suggests overfitting; regularization can help.

### Seeing it happen

Take 25 noisy samples of a smooth curve and fit polynomials of different degrees, which is linear regression on the features $x, x^2, \ldots, x^d$ (Figure 8):

- **Degree 1 (underfitting, high bias).** A straight line cannot follow the curve. The error is large on the training set *and* on new data.
- **Degree 3 (a good fit).** The curve follows the trend and ignores the noise. Errors are small on both.
- **Degree 15 (overfitting, high variance).** The curve passes almost exactly through every training point, so the training error is nearly zero. Between and beyond the points it swings wildly, and the error on new data explodes.

The training error alone cannot tell the good model from the overfitted one; it even prefers the overfitted one. Only data that the model did not see can reveal the difference. This is why we keep a validation set.

### Using the validation and test sets

The workflow is:

1. Fit each candidate model (each degree, each regularization strength) on the **training set**.
2. Compare the candidates on the **validation set** and keep the best one.
3. Report the chosen model's error on the **test set**, once.

Why three sets and not two? Because picking the best of many candidates on the validation set is itself a form of fitting: the winner is partly the one that got lucky on those particular examples. The untouched test set gives an unbiased final estimate. When data is scarce, **cross-validation** replaces the single validation set: split the training data into $k$ folds, train $k$ times, each time validating on a different fold, and average.

The pattern to look for: if training and validation errors are both high, the model underfits (use a more flexible model or better features). If training error is low but validation error is much higher, the model overfits (get more data, simplify the model, or regularize).

### L2 regularization

**Regularization** keeps a flexible model but discourages extreme parameter values. The wild curve of the degree-15 fit needs huge, finely balanced weights. **L2 regularization** adds a penalty on the squared size of the weights to the cost:

$$J_{\text{reg}}(\mathbf{w}, b) = J(\mathbf{w}, b) + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2,$$

where $J$ is the original cost (MSE or log loss), $\lambda \geq 0$ is the **regularization strength** (a hyperparameter), and the sum runs over the weights but conventionally not the bias $b$. With $\lambda = 0$ we recover the original model; as $\lambda$ grows, the weights are pulled toward zero and the fitted function becomes smoother; too large a $\lambda$ underfits. For linear regression, this is **ridge regression**, introduced by Hoerl and Kennard to stabilize least-squares estimates when features are strongly correlated [20].

The gradient-descent update shows what the penalty does:

$$w_j \leftarrow w_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{\partial J}{\partial w_j}.$$

Before each ordinary step, every weight is multiplied by a number slightly less than 1. This is why L2 regularization is also called **weight decay** in neural-network training [3].

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.metrics import mean_squared_error

rng = np.random.default_rng(0)
x = rng.uniform(0, 1, 25)
y = np.sin(2 * np.pi * x) + rng.normal(0, 0.25, 25)      # true curve + noise
X = x[:, None]

# 60% train, 20% validation, 20% test
X_tmp, X_te, y_tmp, y_te = train_test_split(X, y, test_size=0.2, random_state=0)
X_tr, X_va, y_tr, y_va = train_test_split(X_tmp, y_tmp, test_size=0.25, random_state=0)
```

```
def fit(degree, lam=0.0):
    reg = Ridge(alpha=lam) if lam > 0 else LinearRegression()   # sklearn calls lambda
     ↪  "alpha"
    model = make_pipeline(PolynomialFeatures(degree), StandardScaler(), reg)
    return model.fit(X_tr, y_tr)

for degree, lam in [(1, 0), (3, 0), (15, 0), (15, 1e-2)]:
    model = fit(degree, lam)
    tr = mean_squared_error(y_tr, model.predict(X_tr))
    va = mean_squared_error(y_va, model.predict(X_va))
    print(f"degree {degree:2d}, lambda {lam:g}: train MSE {tr:.3f}, val MSE {va:.3g}")

best = fit(15, 1e-2)                                        # chosen on validation only
print("test MSE:", round(mean_squared_error(y_te, best.predict(X_te)), 3))
```

The output tells the whole story: degree 1 has train/validation MSE of 0.144/0.292 (underfit), degree 3 has 0.031/0.052 (good), and unregularized degree 15 has a training MSE of 0.003 but a validation MSE in the millions, because some validation points fall in wide gaps between training points (for example between $x \approx 0.27$ and $x \approx 0.61$), where the polynomial swings wildly. Adding a mild L2 penalty ($\lambda = 0.01$) to the same degree-15 model gives 0.041/0.035, and its test MSE is 0.075.

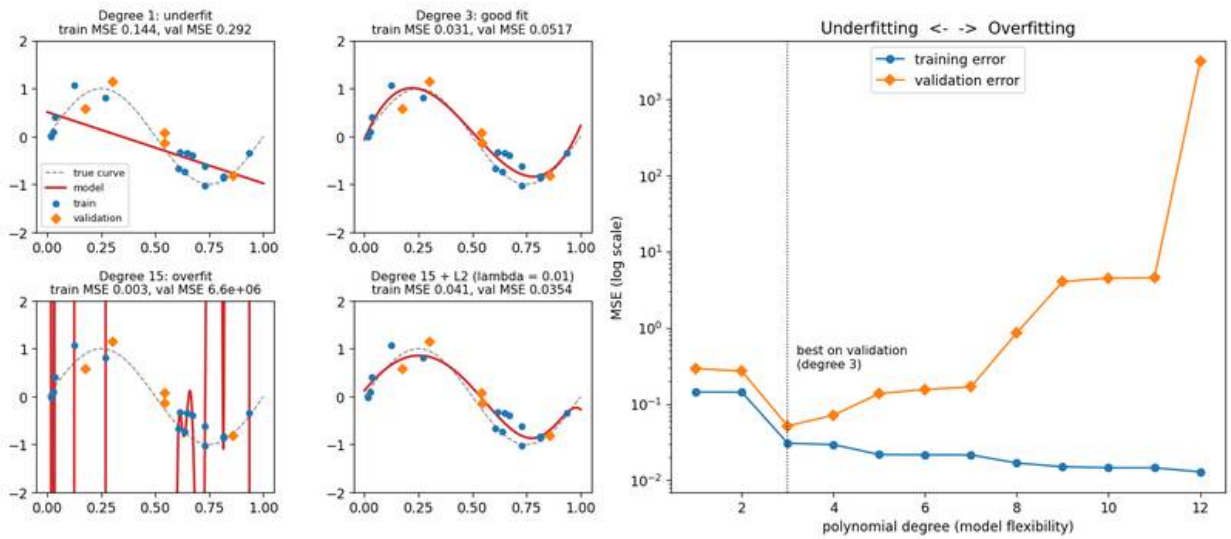



*Figure 8 — Underfitting, a good fit, overfitting, and overfitting tamed by L2 regularization (training points in blue, validation points in orange). Right: as the degree grows, training error keeps falling while validation error falls and then rises; the bottom of the validation curve is the model to choose.*

## Evaluating a model

**Plain version.** A single number never tells the whole story. Choose a metric that matches what a mistake costs, and always compare against a trivial baseline.

### Regression metrics

For regression, the most common metric is the MSE, $\frac{1}{m}\sum_i(\hat{y}^{(i)} - y^{(i)})^2$, here without the $\frac{1}{2}$ that was only a training convenience. Its square root, the **RMSE**, is in the same units as $y$ and is easier to interpret ("off by about 3 points on average"). The **mean absolute error**, $\frac{1}{m}\sum_i|\hat{y}^{(i)} - y^{(i)}|$, is less sensitive to a few large errors [5].

### Classification metrics

For classification, **accuracy** is the fraction of correct predictions. It is easy to understand and easy to be fooled by. In the digits data, about 10% of the images are 9s. A "model" that always answers "not a 9" is 90% accurate and completely useless.

The **confusion matrix** counts the four outcomes for a binary problem, with "positive" meaning the class we care about:

| | predicted positive | predicted negative |
|---|---|---|
| **actually positive** | true positive (TP) | false negative (FN) |
| **actually negative** | false positive (FP) | true negative (TN) |

From it,

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}, \qquad \text{precision} = \frac{TP}{TP + FP}, \qquad \text{recall} = \frac{TP}{TP + FN}.$$

**Precision** answers "when the model says positive, how often is it right?" **Recall** answers "of all the real positives, how many did the model find?" They trade off: lowering the 0.5 decision threshold of logistic regression finds more positives (higher recall) but raises more false alarms (lower precision). Which matters more depends on the cost of each mistake: a cancer screen should not miss tumors (high recall), while a spam filter should not hide real mail (high precision). The **F1 score**, the harmonic mean of precision and recall, combines both into one number [5].

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score

X, y = load_digits(return_X_y=True)
y = (y == 9).astype(int)                  # "is it a 9?": only about 10% positives
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.3, stratify=y, random_state=0)

lazy = np.zeros_like(y_te)                # always answers "not a 9"
model = LogisticRegression(max_iter=5000).fit(X_tr, y_tr)
pred = model.predict(X_te)

print("lazy accuracy :", round(accuracy_score(y_te, lazy), 3), " recall:", recall_score(y_te,
 ↪  lazy))
```

```
print("model accuracy:", round(accuracy_score(y_te, pred), 3),
      " precision:", round(precision_score(y_te, pred), 3),
      " recall:", round(recall_score(y_te, pred), 3))
print(confusion_matrix(y_te, pred))       # rows: true 0/1, columns: predicted 0/1
# lazy accuracy : 0.9   recall: 0.0
# model accuracy: 0.974  precision: 0.87  recall: 0.87
```

The lazy baseline reaches 90% accuracy with zero recall. The real model's 97.4% accuracy sounds only a little better, but its recall of 0.87 shows that it actually finds most of the 9s. Always report the baseline, and always look past accuracy when classes are imbalanced. The same metrics reappear, with more care about thresholds and multiple classes, in the DIP chapter on pattern classification.

## Modern view

The mechanics in this tutorial (a parameterized model, a cost, gradient-based optimization, and held-out evaluation) have not changed in decades, and every system discussed below still uses them. What has changed is *where the features come from*, *where the labels come from*, and *how large the models are*. A handful of reviews map these shifts well.

**Machine learning as a field.** Jordan and Mitchell's review in *Science* [7] frames machine learning around a single question, how to build computers that improve automatically through experience, and places it at the intersection of computer science and statistics, at the core of artificial intelligence and data science. They attribute its rapid spread across science, medicine, manufacturing, education and finance to three things arriving together: new learning algorithms and theory, an explosion in the amount of available data, and cheap computation. That framing is a useful antidote to hype: the method is only as good as the data and the experience it is given.

**From feature engineering to representation learning.** For a long time, the success of a classical model depended mostly on hand-designed features, the "count the inked columns" step of Figure 1 done by experts: edge histograms, texture descriptors, keypoint descriptors (see IP 12). Bengio, Courville and Vincent [21] argue that this dependence is a weakness, because the right representation is what makes a task easy or hard, and propose learning the representation itself from data. Their review covers the tools for this (probabilistic models, autoencoders, manifold learning and deep networks) and the general priors that make a representation good, such as disentangling the separate explanatory factors behind the data. In the language of this tutorial, a deep network is still "features followed by logistic regression", except that the features are produced by earlier layers whose weights are learned by the same gradient descent. That is the subject of ML 02.

**Self-supervised learning.** Learning representations needs data, and labels are the expensive part. Jing and Tian's survey of self-supervised visual feature learning [22] sorts the label-free "pretext tasks" into four families: generation-based (for example, reconstructing or colorizing images), context-based (exploiting spatial or temporal structure, such as the arrangement of image patches), free-semantic-label-based (using labels that come automatically, for example from rendering engines or classical methods), and cross-modal (checking whether two signals, such as video and audio, belong together). It also sets out the evaluation protocol that is still standard:

judge a representation by how well it transfers to downstream tasks such as classification, detection and segmentation. The later survey by Gui et al. [23] reorganizes the field around context-based methods, contrastive learning (with negative examples, self-distillation, or feature decorrelation), generative methods dominated by masked image modeling, and hybrids of contrastive and generative objectives, reflecting the shift toward masked prediction after masked autoencoders [12]. For practitioners, Balestriero et al.'s *Cookbook of Self-Supervised Learning* [24] groups modern methods into a deep metric learning family, a self-distillation family, a canonical correlation analysis family and masked image modeling, and is candid about how much the outcome depends on details such as data augmentation and the small "projector" network used during training.

**Foundation models.** The report by Bommasani et al. [10] named the resulting paradigm: models trained on broad data at scale, usually self-supervised, then adapted to many downstream tasks. It highlights two consequences. **Emergence**: capabilities such as following instructions from a few examples in the prompt [11] appear with scale rather than being designed in. **Homogenization**: many applications are now adapted from the same few base models, which gives great leverage but means that a defect in the base model is inherited by everything built on it. The report also stresses how limited our understanding is of how these models work and when they fail.

**Reinforcement learning from human feedback.** RL has moved from games to the final stage of training language models [13], [14]. Casper et al. [25] survey what can go wrong. They follow the three steps of RLHF (collecting human feedback, fitting a reward model, and optimizing the policy) and catalog problems at each: feedback from people is noisy, inconsistent and can be manipulated; a reward model is only an imperfect proxy for what people want; and a policy optimized hard against that proxy can exploit its errors. They separate problems that better methods within RLHF could solve from fundamental limitations that would require a different approach, and they argue for transparency and for combining RLHF with other safety measures rather than relying on it alone.

**Optimization at scale.** Bottou, Curtis and Nocedal [18] explain why the humble stochastic gradient method, and not more sophisticated optimizers from classical numerical analysis, became the workhorse of machine learning: when data is plentiful, a cheap noisy step beats an expensive exact one. Adaptive methods such as Adam [19] add per-parameter step sizes on top. Every model mentioned in this section, however large, is trained with a descendant of the mini-batch loop in this tutorial.

**Open problems.** Despite the scale, several questions from this beginner's tutorial remain the hard ones. *Generalization*: very large models fit their training data almost perfectly yet still generalize, which does not match the simple overfitting picture of Figure 8, and why this happens is not fully understood. *Evaluation*: when a model is trained on a large fraction of the public internet, it is hard to guarantee that the test set was not in the training data, so a clean train/test separation, the foundation of honest evaluation, is difficult to verify. *Objectives*: the cost function is the only thing the model optimizes, and RLHF shows how hard it is to write a cost that captures what people actually want. And *data*: what a model learns is bounded by the data and labels it sees, with all their gaps and biases.

For computer vision specifically, the classical route (hand-made features plus a simple classifier)

is developed in the DIP chapter on pattern classification, and the learned route continues in ML 02 and Deep Learning for Image Processing.

## Key takeaways

- A model maps inputs to predictions; training adjusts its parameters to reduce a loss.
- Supervised learning uses targets, self-supervision builds targets from data, and reinforcement learning uses rewards; unsupervised methods seek structure without supplied targets.
- Gradient descent follows the negative gradient; a zero gradient alone does not establish a minimum.
- Fit scaling statistics on training data only; tune the learning rate with validation feedback.
- Linear regression predicts numbers; binary logistic regression converts a linear score into a probability.
- Compare training and validation errors, choose regularization on validation data, and reserve the test set for final reporting.

## Exercises

1. **A derivative by hand.** For the one-feature MSE cost $J(w,b) = \frac{1}{2m}\sum_i (wx^{(i)} + b - y^{(i)})^2$, derive $\partial J/\partial w$ and $\partial J/\partial b$. Then, for the three points $(1,2)$, $(2,4)$, $(3,6)$ starting at $w = 0, b = 0$, compute one gradient-descent step with $\alpha = 0.1$.

   **Hint**

   The chain rule on each square gives $2(wx^{(i)}+b-y^{(i)})\cdot x^{(i)}$, and the 2 cancels the $\frac{1}{2}$, giving the formulas in the gradient-descent section. At $w = b = 0$ the errors are $-2, -4, -6$. So $\partial J/\partial w = (-2\cdot 1 - 4\cdot 2 - 6\cdot 3)/3 = -28/3$ and $\partial J/\partial b = -12/3 = -4$. After one step, $w = 0.1 \cdot 28/3 \approx 0.933$ and $b = 0.4$.

2. **Find the divergence threshold.** For $J(w) = \frac{1}{2m}\sum_i (wx^{(i)} - y^{(i)})^2$ (no bias), show that one gradient step multiplies the distance to the optimum $w^*$ by $1 - \alpha s$, where $s = \frac{1}{m}\sum_i (x^{(i)})^2$. For which $\alpha$ does gradient descent converge? Check your answer with the `gd` function from the feature-scaling section.

   **Hint**

   $\partial J/\partial w = s\,(w - w^*)$ because $J$ is a parabola with curvature $s$ and minimum at $w^*$. So $w_{\text{new}} - w^* = (1-\alpha s)(w - w^*)$. It converges when $|1-\alpha s| < 1$, that is $0 < \alpha < 2/s$; at $\alpha = 1/s$ it lands on the minimum in one step. With several features, $s$ is replaced by the largest eigenvalue of $\frac{1}{m}X^\top X$, which is why one huge-scale feature forces a tiny $\alpha$.

3. **Regression for classification.** Generate 1-D data with `make_classification(n_features=1, n_informative=1, n_redundant=0, n_clusters_per_class=1)`, then add five extra examples of class 1 with very large $x$. Fit both `LinearRegression` (thresholding at 0.5) and

`LogisticRegression`. Compare their accuracy before and after adding the extra points, and explain the difference.

**Hint**

The extra points are easy to classify, but the squared error still wants the line to pass near them. The line's slope drops, its 0.5 crossing moves toward class 1, and some ordinary class-1 points near the boundary become misclassified. The log loss of a correctly classified, far-away point is already close to 0, so logistic regression barely moves.

4. **Scaling the digits.** Train the from-scratch logistic regression of this tutorial on "9 versus not 9" with the digits data, once on raw pixels (0 to 16) and once on pixels divided by 16. Use the same number of steps and find the largest learning rate that works for each. Does scaling change the final accuracy, the speed, or both?

**Hint**

All pixels share one unit here, so the effect is milder than in Figure 6, but the raw inputs are 16 times larger, which multiplies the gradient's size and the landscape's curvature. The safe learning rate on raw pixels is roughly $16^2 = 256$ times smaller. With properly tuned learning rates, both reach similar accuracy; the scaled version is simply easier to tune.

5. **Choosing lambda.** Using the polynomial example, fix degree 15 and try $\lambda \in \{10^{-6}, 10^{-4}, 10^{-2}, 1, 100\}$. Plot training and validation MSE against $\lambda$ on a log axis. Which end underfits, which overfits, and which $\lambda$ would you pick? Why must you not use the test set to make this choice?

**Hint**

Tiny $\lambda$ behaves like no regularization (overfitting: low training error, high validation error). Huge $\lambda$ flattens the curve toward a constant (underfitting: both errors high). Pick the $\lambda$ with the lowest validation error. If the test set guided the choice, it would no longer be an unbiased estimate of performance on new data.

6. **Which paradigm?** Classify each problem as supervised (regression or classification), unsupervised, self-supervised or reinforcement learning, and name the target or reward: (a) estimating the remaining battery life of a phone from usage logs with recorded outcomes; (b) grouping a million unlabeled product photos for browsing; (c) pre-training an image model by predicting the missing quarter of each photo; (d) teaching a robot arm to stack blocks with a score given only when the stack stands; (e) flagging unusual readings in a factory sensor stream with no labeled failures.

**Hint**

(a) Supervised regression; target: remaining hours. (b) Unsupervised clustering. (c) Self-supervised; the target is the hidden pixels, made from the data itself. (d) Reinforcement learning; reward: whether the stack stands, delayed until the end. (e) Unsupervised anomaly detection.

## Continue through ML/DL

If the mathematics is new, take linear algebra, probability and information theory, and numerical computation before neural networks.

The rest of the track connects image filtering and CNNs with recurrent/recursive networks, transformers and representation learning. It then builds from factor models and autoencoders to VAEs, GANs, normalizing flows, diffusion/flow matching, autoregressive models and reusable foundation models. Finish with evaluation and deployment. For the statistical decision and inference viewpoint, follow Pattern Recognition; for task-specific systems, follow Deep Learning for Computer Vision.

## A first route through the papers

Read the main chapters first. These 21 core guides are optional checkpoints, grouped by question rather than publication date; they do not add prerequisites to the course. Each has six foldable parts, an original paper figure, a reading question and a next step. Follow one stage at a time, and use the listed chapter prerequisites before jumping between themes.

### 1 • Frame the problem and its mathematics

**Know first:** ML 01, ML 01A and ML 01B: data splits, vectors, covariance and probability.

Practical ML → PCA → Shannon

**Checkpoint:** Which assumptions connect the representation, objective and evaluation?

### 2 • Make training and architectures understandable

**Know first:** ML 01C → ML 02 → ML 03 → ML 04 → ML 05: numeric range, gradients, convolution, recurrence and attention.

Mixed Precision → Adam → ResNet → LSTM → ViT

**Checkpoint:** What failure does each design target, and which comparison isolates its effect?

### 3 • Ask what a representation preserves

**Know first:** ML 06 and ML 06A; take ML 10 before CLIP. Know contrastive learning, Gaussian latent models and transfer protocols.

Probabilistic PCA → SimCLR → CLIP

**Checkpoint:** What creates the learning target, and what does downstream evaluation actually test?

### 4 • Compare generative objectives and sampling

**Know first:** ML 07–ML 09: latent variables, KL divergence, noise schedules, differential equations and token priors.

VAE → GAN → VQ-VAE → DDPM → Flow Matching

**Checkpoint:** Separate reconstruction, density or bound, sample quality and generation cost. Which evidence measures each?

### 4A • Continue beyond Flow Matching

**Know first:** ML 08, instantaneous versus average velocity, total derivatives, guidance and pixel/latent representations. First follow interval learning; then connect it to direct image output.

MeanFlow → Improved MeanFlow → JiT → Pixel MeanFlow

**Checkpoint:** Which method changes the time interval, the regression inputs, the direct output or the generation space? Why is JiT alone not a one-step model?

### 5 • Close the loop with reliable evaluation

**Know first:** ML 11: logits, validation/test separation and reliability diagrams.

Calibration

**Checkpoint:** Can confidence improve without changing accuracy, and what happens after data shift?

## References


1. Shyandram, "Introduction to Machine Learning," Medium post, 2024; updated 2026. link
2. A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959. doi
3. I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016. book site
4. A. Ng, E. Shyu, A. Bagul and G. Ladwig, "Machine Learning Specialization," DeepLearning.AI and Stanford Online, Coursera. course page
5. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. link
6. E. Alpaydin and C. Kaynak, "Optical Recognition of Handwritten Digits," UCI Machine Learning Repository, 1998. doi
7. M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015. doi
8. R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., MIT Press, 2018. book page

9. Y. LeCun and I. Misra, “Self-supervised learning: The dark matter of intelligence,” Meta AI blog, 2021. link
10. R. Bommasani, D. A. Hudson, E. Adeli et al., “On the opportunities and risks of foundation models,” arXiv:2108.07258, 2021. arXiv
11. T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan et al., “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems* (*NeurIPS*), vol. 33, 2020. link
12. K. He, X. Chen, S. Xie, Y. Li, P. Dollár and R. Girshick, “Masked autoencoders are scalable vision learners,” in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), pp. 15979–15988, 2022. arXiv
13. L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright et al., “Training language models to follow instructions with human feedback,” in *Advances in Neural Information Processing Systems* (*NeurIPS*), vol. 35, 2022. link
14. DeepSeek-AI, “DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning,” *Nature*, vol. 645, pp. 633–638, 2025. arXiv
15. A. K. Jain, “Data clustering: 50 years beyond K-means,” *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, 2010. doi
16. I. T. Jolliffe and J. Cadima, “Principal component analysis: A review and recent developments,” *Philosophical Transactions of the Royal Society A*, vol. 374, no. 2065, 20150202, 2016. doi
17. V. Chandola, A. Banerjee and V. Kumar, “Anomaly detection: A survey,” *ACM Computing Surveys*, vol. 41, no. 3, 2009. doi
18. L. Bottou, F. E. Curtis and J. Nocedal, “Optimization methods for large-scale machine learning,” *SIAM Review*, vol. 60, no. 2, pp. 223–311, 2018. doi
19. D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2015. arXiv
20. A. E. Hoerl and R. W. Kennard, “Ridge regression: Biased estimation for nonorthogonal problems,” *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970. doi
21. Y. Bengio, A. Courville and P. Vincent, “Representation learning: A review and new perspectives,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013. doi
22. L. Jing and Y. Tian, “Self-supervised visual feature learning with deep neural networks: A survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 11, pp. 4037–4058, 2021. doi
23. J. Gui, T. Chen, J. Zhang, Q. Cao, Z. Sun, H. Luo and D. Tao, “A survey on self-supervised learning: Algorithms, applications, and future trends,” arXiv:2301.05712, 2023. arXiv
24. R. Balestriero, M. Ibrahim, V. Sobal, A. Morcos, S. Shekhar et al., “A cookbook of self-supervised learning,” arXiv:2304.12210, 2023. arXiv
25. S. Casper, X. Davies, C. Shi et al., “Open problems and fundamental limitations of reinforcement learning from human feedback,” *Transactions on Machine Learning Research*, 2023. TMLR ·arXiv
26. P. Domingos, “A Few Useful Things to Know About Machine Learning,” *Communications of the ACM*, 55(10), pp. 78–87, 2012. doi
27. NumPy developers, “numpy.polyfit,” NumPy documentation, accessed 2026-11-05. Least-squares solver and singular values

# ML 01A • Linear Algebra for Learning

Vectors, matrices and tensors, norms, eigendecomposition, the SVD, the pseudo-inverse, trace and determinant, and a step-by-step derivation of PCA, with tested NumPy code.

Online edition: https://blog.shyandram.dev/en/learn/ml/linear-algebra/

## What you'll learn

- Track array shapes and interpret matrix multiplication.
- Use rank, null spaces and norms to reason about linear systems.
- Read eigenvalues and singular values as geometric information.
- Solve least-squares problems and diagnose sensitivity to noise.
- Derive PCA and connect matrix identities to learning.

## The big picture

Almost every operation inside a neural network is a linear map followed by a simple nonlinear function. A fully connected layer multiplies by a weight matrix. A convolution is a structured matrix product. Attention computes dot products between vectors and then mixes vectors with a matrix of weights. Linear algebra is the language for all of these. It tells us what a map keeps, what it throws away, what it stretches, and how to compute it without losing accuracy.

> **Explain like I'm five**
>
> A vector is a list of measurements, like the height, weight and age of one person. A matrix is a machine that takes one list and mixes it into a new list. Some machines just turn the list around (rotation), some make some parts bigger and others smaller (stretching), and some squash parts flat so the information is gone forever. Linear algebra is the instruction manual for these machines: it tells you which way each machine stretches, by how much, and whether you can undo it.

This chapter follows the "Applied Math" part of Goodfellow, Bengio and Courville's *Deep Learning* [1] and the linear algebra chapters of *Mathematics for Machine Learning* [2]. Strang's textbook [3] is the gentlest full course if you want more practice, and Golub and Van Loan [4] and Trefethen and Bau [5] are the standard references for how these computations are done reliably on a computer. Read this chapter before ML 02 if matrix notation is unfamiliar. The companion chapters are ML 01B ·Probability and Information Theory and ML 01C ·Numerical Computation.

The chapter has a single thread. We start with objects and shapes, move to products and solving equations, then measure size with norms. Eigendecomposition and the SVD then explain the "personality" of a matrix. The pseudo-inverse, trace and determinant are tools built on top of them. Finally, we use every one of these tools to derive PCA, the classic method for compressing data, exactly as it is derived in [1].

### Scalars, vectors, matrices and tensors

**Plain version.** Every number in machine learning lives in an array with named axes. Name every axis before you multiply anything.

A **scalar** is a single number, written in italics: $s \in \mathbb{R}$. A **vector** is an ordered list of $n$ numbers, written in bold lowercase or plain lowercase in this chapter: $x \in \mathbb{R}^n$, with entries $x_1, \ldots, x_n$. By convention a vector is a column. A **matrix** is a 2-D grid of numbers: $A \in \mathbb{R}^{m \times n}$ has $m$ rows and $n$ columns, and $A_{ij}$ is the entry in row $i$ and column $j$. A **tensor** is an array with any number of axes.

These objects map directly onto data:

- A grayscale image of height $H$ and width $W$ is a matrix in $\mathbb{R}^{H \times W}$. Flattened, it becomes a vector $x \in \mathbb{R}^{HW}$. (See Images as Arrays.)
- A dataset of $n$ examples with $d$ features each, stored one example per row, is a **design matrix** $X \in \mathbb{R}^{n \times d}$.
- A fully connected layer with weights $W \in \mathbb{R}^{d \times m}$ and bias $b \in \mathbb{R}^m$ maps the whole batch at once: $Y = XW + \mathbf{1}b^\top \in \mathbb{R}^{n \times m}$, where $\mathbf{1} \in \mathbb{R}^n$ is a vector of ones.
- An image batch in PyTorch is usually a 4-D tensor of shape $B \times C \times H \times W$ (batch, channels, height, width). A batch of token embeddings in a Transformer is usually $B \times N \times D$ (batch, sequence length, embedding width).

The **transpose** $A^\top$ swaps rows and columns: $(A^\top)_{ij} = A_{ji}$. It satisfies $(A^\top)^\top = A$ and $(AB)^\top = B^\top A^\top$. Note the reversed order.

### Broadcasting

Writing $\mathbf{1}b^\top$ is mathematically honest but tedious, so libraries let you add a vector to every row of a matrix directly. This is called **broadcasting**: when two arrays have different shapes, axes of length 1 (or missing leading axes) are stretched to match. Broadcasting is convenient and dangerous, because a wrong shape often still runs. The most common bug is subtracting a column vector of shape $(n, 1)$ from a flat vector of shape $(n,)$, which silently produces an $n \times n$ matrix instead of $n$ residuals.

```
import numpy as np

rng = np.random.default_rng(0)
X = rng.normal(size=(32, 64))      # batch of 32 examples, 64 features
W = rng.normal(size=(64, 10))      # linear map 64 -> 10
b = rng.normal(size=10)            # one bias per output
Y = X @ W + b                      # b broadcasts over the 32 rows
assert Y.shape == (32, 10)

# A classic broadcasting bug: (32,) - (32, 1) silently becomes (32, 32)
y_true = rng.normal(size=32)
y_pred = rng.normal(size=(32, 1))
print("buggy residual shape:", (y_true - y_pred).shape)
```

```
print("fixed residual shape:", (y_true - y_pred[:, 0]).shape)

# Four equivalent views of the same matrix product
A, B = rng.normal(size=(3, 4)), rng.normal(size=(4, 2))
entry = np.array([[A[i] @ B[:, j] for j in range(2)] for i in range(3)])
cols = np.stack([A @ B[:, j] for j in range(2)], axis=1)
outer = sum(np.outer(A[:, k], B[k]) for k in range(4))
assert np.allclose(A @ B, entry) and np.allclose(A @ B, cols) and np.allclose(A @ B, outer)
print("all views agree")
```

The output shows `(32, 32)` for the buggy line and `(32,)` for the fixed one. A mean over a $32 \times 32$ array still returns a single number, so a loss computed this way trains without errors while optimizing the wrong thing. Put `assert` statements on shapes in any code you care about.

### Multiplying matrices and vectors

**Plain version.** A matrix product chains two linear machines into one. Each output number is a dot product between a row of the first matrix and a column of the second.

For $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$, the **matrix product** $C = AB \in \mathbb{R}^{m \times p}$ is

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

Here $i$ indexes rows of $A$ (and of $C$), $j$ indexes columns of $B$ (and of $C$), and $k$ runs over the shared inner dimension $n$. The inner dimensions must match; that is the first thing to check when a shape error appears.

The same product can be read in four ways, and each one is useful:

1. **Entry view.** $C_{ij}$ is the dot product of row $i$ of $A$ with column $j$ of $B$. This is how attention scores $QK^\top$ are read: every query is compared with every key.
2. **Column view.** Column $j$ of $C$ is $A$ times column $j$ of $B$. Each column of $C$ is a combination of the columns of $A$.
3. **Row view.** Row $i$ of $C$ is row $i$ of $A$ times $B$. In $Y = XW$, each output example is its input row pushed through the same weights.
4. **Outer-product view.** $C = \sum_k a_{:k} b_{k:}$, a sum of $n$ rank-one matrices, each formed from column $k$ of $A$ and row $k$ of $B$. This view explains low-rank approximation and LoRA later in the chapter.

The code above checks that all four views give the same answer.

Matrix multiplication is **associative**, $(AB)C = A(BC)$, and **distributive**, $A(B + C) = AB + AC$, but **not commutative**: $AB \neq BA$ in general, and often one of the two products is not even defined. The **elementwise** (Hadamard) product $A \odot B$ multiplies matching entries instead and needs equal shapes. In NumPy and PyTorch, `A @ B` is the matrix product and `A * B` is elementwise. The **dot product** of two vectors $x, y \in \mathbb{R}^n$ is the special case $x^\top y = \sum_i x_i y_i$, and it is symmetric: $x^\top y = y^\top x$.

### Identity, inverse and systems of equations

A system of $m$ linear equations in $n$ unknowns can be written compactly as

$$Ax = b, \qquad A \in \mathbb{R}^{m \times n},\ x \in \mathbb{R}^{n},\ b \in \mathbb{R}^{m}.$$

Each row of $A$ holds the coefficients of one equation, $x$ holds the unknowns, and $b$ holds the right-hand sides.

The **identity matrix** $I_n$ has ones on the diagonal and zeros elsewhere; it leaves every vector unchanged, $I_n x = x$. A square matrix $A$ is **invertible** if there is a matrix $A^{-1}$ with $A^{-1}A = AA^{-1} = I$. Then the system has exactly one solution, $x = A^{-1}b$.

That formula is excellent for derivations and poor for computation. Explicitly forming $A^{-1}$ costs more and loses more accuracy than solving the system directly with an LU, Cholesky or QR factorization [4], [5]. In NumPy, write `np.linalg.solve(A, b)`, not `np.linalg.inv(A) @ b`. In this chapter, $A^{-1}$ is a thinking tool.

### Linear dependence, span and rank

**Plain version.** Rank counts how many genuinely different directions a matrix can produce. Repeated or redundant columns add nothing new.

A **linear combination** of vectors $v^{(1)}, \dots, v^{(k)}$ is $\sum_i c_i v^{(i)}$ for scalars $c_i$. The **span** of a set of vectors is the set of all their linear combinations. Because $Ax = \sum_j x_j\, a_{:j}$, the vector $Ax$ is a linear combination of the columns $a_{:j}$ of $A$. So $Ax = b$ has a solution exactly when $b$ lies in the span of the columns, called the **column space** (or range) of $A$.

A set of vectors is **linearly independent** if no vector in it is a linear combination of the others; equivalently, $\sum_i c_i v^{(i)} = 0$ only when every $c_i = 0$. The **rank** of $A$ is the number of linearly independent columns, which always equals the number of linearly independent rows. For $A \in \mathbb{R}^{m \times n}$, $\operatorname{rank}(A) \le \min(m, n)$; if equality holds, $A$ has **full rank**.

The **null space** of $A$ is the set of vectors $z$ with $Az = 0$. Those are the input directions that $A$ destroys. The **rank–nullity theorem** says

$$\operatorname{rank}(A) + \dim \operatorname{null}(A) = n,$$

where $n$ is the number of columns and dim is the number of independent directions in the null space. This one equation explains the three regimes of $Ax = b$:

| Shape and rank | What happens | Typical ML case |
|---|---|---|
| square, full rank | exactly one solution | a well-posed change of coordinates |
| tall ($m > n$), full column rank | usually no exact solution; solve in the least-squares sense | regression with more examples than features |
| wide ($m < n$) or rank-deficient | infinitely many solutions (if any) | more parameters than constraints, as in deep networks |

For a square matrix, "invertible", "full rank", "columns linearly independent" and "null space contains only zero" are all the same statement. A square matrix that fails them is called **singular**.

### Norms: measuring size

**Plain version.** A norm turns a vector into one non-negative number that says how big it is. Different norms disagree about what "big" means.

A **norm** is any function $f$ with $f(x) = 0 \Rightarrow x = 0$, $f(x + y) \le f(x) + f(y)$ (the triangle inequality), and $f(\alpha x) = |\alpha| f(x)$ for every scalar $\alpha$. The most common family is the $L^p$ norm,

$$\|x\|_p = \Big( \sum_i |x_i|^p \Big)^{1/p}, \qquad p \ge 1.$$

- $\|x\|_2$, the **Euclidean** norm, is ordinary length; we often write $\|x\|$. Its square $\|x\|_2^2 = x^\top x$ is easier to differentiate and appears in mean-squared-error losses and weight decay.
- $\|x\|_1 = \sum_i |x_i|$ grows at the same rate everywhere, even near zero. That makes it useful when the difference between exactly zero and slightly nonzero matters, which is why $L^1$ penalties encourage sparse solutions.
- $\|x\|_\infty = \max_i |x_i|$, the **max norm**, is the size of the largest entry. Adversarial-robustness work often bounds perturbations in this norm.
- The count of nonzero entries is sometimes called the "$L^0$ norm", but it is not a norm: scaling $x$ by 2 does not change it.

The dot product connects to the Euclidean norm through angles: $x^\top y = \|x\|_2 \|y\|_2 \cos\theta$, where $\theta$ is the angle between $x$ and $y$. The **cosine similarity**

$$\cos(x, y) = \frac{x^\top y}{\|x\|_2 \|y\|_2}$$

is defined for nonzero vectors and ignores scale. That is useful for some embeddings (contrastive learning normalizes features for this reason) and wrong when magnitude carries meaning, such as image brightness.

### Matrix norms

For matrices we use two norms constantly. The **Frobenius norm** treats the matrix as one long vector,

$$\|A\|_F = \sqrt{\sum_{i,j} A_{ij}^2}.$$

The **spectral norm** (operator 2-norm) measures the largest factor by which $A$ can stretch a unit vector,

$$\|A\|_2 = \max_{\|x\|_2 = 1} \|Ax\|_2 = \sigma_{\max}(A),$$

where $\sigma_{\max}(A)$ is the largest singular value, defined in the SVD section below. In terms of all singular values $\sigma_1 \geq \sigma_2 \geq \ldots$, we have $\|A\|_F^2 = \sum_i \sigma_i^2$ while $\|A\|_2 = \sigma_1$. So many small singular values can make the Frobenius norm large without changing the largest amplification. The spectral norm bounds how much a layer can magnify a perturbation of its input, which is why it appears in robustness and GAN-stability work [17].

```
import numpy as np

x = np.array([3.0, -4.0, 0.0, 1.0])
for p in [1, 2, np.inf]:
    print(f"L{p} norm:", np.linalg.norm(x, p))
print("L0 'norm' (count of nonzeros):", np.count_nonzero(x))

A = np.array([[3.0, 0.0], [4.0, 5.0]])
s = np.linalg.svd(A, compute_uv=False)
print("singular values:", s.round(4))
print("Frobenius:", np.linalg.norm(A, "fro"), "=", np.sqrt((s**2).sum()))
print("spectral :", np.linalg.norm(A, 2), "=", s[0])
assert np.isclose(np.linalg.norm(A, "fro")**2, np.trace(A.T @ A))
```

For $x = (3, -4, 0, 1)$ this prints $L^1 = 8$, $L^2 = \sqrt{26} \approx 5.10$, $L^\infty = 4$ and three nonzeros. For the $2 \times 2$ matrix the singular values are $3\sqrt{5} \approx 6.71$ and $\sqrt{5} \approx 2.24$, giving $\|A\|_F = \sqrt{50} \approx 7.07$ and $\|A\|_2 \approx 6.71$.

### Special kinds of matrices and vectors

**Plain version.** Some matrices are much easier to work with than others. Learning to spot them saves computation and explains many design choices.

- A **diagonal** matrix $D = \operatorname{diag}(v)$ has nonzero entries only on its main diagonal. Multiplying by it scales each coordinate: $\operatorname{diag}(v)x = v \odot x$. Its inverse, when every $v_i \neq 0$, is $\operatorname{diag}(1/v_1, \ldots, 1/v_n)$. Per-channel scaling in normalization layers is a diagonal matrix in disguise. Rectangular diagonal matrices also exist; they appear in the SVD.
- A **symmetric** matrix satisfies $A = A^\top$. Covariance matrices, Gram matrices $X^\top X$, Hessians of smooth functions and distance matrices are symmetric.
- A **unit vector** has $\|x\|_2 = 1$. Two vectors are **orthogonal** if $x^\top y = 0$; if they are also unit vectors, they are **orthonormal**.
- An **orthogonal matrix** $Q$ is square with orthonormal rows and columns, so $Q^\top Q = QQ^\top = I$ and $Q^{-1} = Q^\top$. Its inverse is free to compute. It preserves lengths and angles, $\|Qx\|_2 = \|x\|_2$, so geometrically it is a rotation, a reflection, or both.

If $Q \in \mathbb{R}^{n \times k}$ has $k < n$ orthonormal columns, then $Q^\top Q = I_k$ but $QQ^\top \neq I_n$. The matrix $P = QQ^\top$ is the **orthogonal projection** onto the span of the columns of $Q$. The projection $Px$ is the closest point to $x$ in that subspace, and the residual $x - Px$ is perpendicular to every column of $Q$. Projecting twice changes nothing: $P^2 = P$. PCA, below, is exactly the search for the best such $Q$.

### Eigendecomposition

**Plain version.** A real eigenvector stays on the same line after transformation. Its eigenvalue sets the scale and may reverse its direction.

An **eigenvector** of a square matrix $A \in \mathbb{R}^{n \times n}$ is a nonzero vector $v$ with

$$Av = \lambda v,$$

where the scalar $\lambda$ is the corresponding **eigenvalue**. If $v$ is an eigenvector, so is $cv$ for any $c \neq 0$, so we usually normalize eigenvectors to unit length.

Suppose $A$ has $n$ linearly independent eigenvectors $v^{(1)}, \dots, v^{(n)}$ with eigenvalues $\lambda_1, \dots, \lambda_n$. Stack them as columns of $V$ and put the eigenvalues in $\Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$. Then $AV = V\Lambda$, so

$$A = V\Lambda V^{-1}.$$

This is the **eigendecomposition**. It says: change to eigenvector coordinates $(V^{-1})$, scale each coordinate independently $(\Lambda)$, and change back $(V)$. Not every matrix has one. The shear $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ has only one independent eigenvector. A rotation by 90° has eigenvalues $\pm i$, which are complex, because no real direction is left unturned.

### The symmetric case

Symmetric matrices are the friendly case, and they are the ones ML meets most. The **spectral theorem** says that every real symmetric matrix can be written as

$$A = Q\Lambda Q^{\top},$$

where $Q$ is orthogonal (its columns are orthonormal eigenvectors) and $\Lambda$ is real and diagonal [3], [4]. Because $Q^{-1} = Q^{\top}$, no inversion is needed. Geometrically, $A$ stretches space by $\lambda_i$ along the direction $q^{(i)}$, the $i$-th column of $Q$. The decomposition is unique up to the order of the eigenvalues and the choice of basis within any repeated eigenvalue; by convention we sort eigenvalues from largest to smallest.

For a symmetric $A$, the **quadratic form** $f(x) = x^{\top}Ax$ restricted to unit vectors is bounded by the extreme eigenvalues:

$$\lambda_{\min} \leq x^{\top}Ax \leq \lambda_{\max} \qquad \text{for } \|x\|_2 = 1,$$

with the maximum reached at the top eigenvector and the minimum at the bottom one. The ratio $x^{\top}Ax/x^{\top}x$ is called the **Rayleigh quotient**. This fact is the engine of the PCA derivation later.

### Positive definite matrices

The signs of the eigenvalues classify symmetric matrices:

| Name | Eigenvalues | Equivalent condition |
|---|---|---|
| positive definite (PD) | all $\lambda_i > 0$ | $x^\top Ax > 0$ for every $x \neq 0$ |
| positive semidefinite (PSD) | all $\lambda_i \geq 0$ | $x^\top Ax \geq 0$ for every $x$ |
| negative (semi)definite | all $\lambda_i < 0$ (or $\leq 0$) | the reverse inequalities |
| indefinite | both signs present | $x^\top Ax$ takes both signs |

Every covariance matrix and every Gram matrix $X^\top X$ is PSD, because $v^\top X^\top Xv = \|Xv\|_2^2 \geq 0$. A Gaussian density needs a PD covariance so that it can be inverted. The Hessian of a loss is PD at a strict local minimum and indefinite at a saddle point; ML 01C uses this to explain why optimization can be slow. A PD matrix has a **Cholesky factorization** $A = LL^\top$ with $L$ lower triangular, which is the fastest stable way to solve systems with it and to sample from a Gaussian [4].

```
import numpy as np

rng = np.random.default_rng(1)
M = rng.normal(size=(4, 4))
S = M @ M.T + 0.1 * np.eye(4)        # symmetric positive definite
lam, Q = np.linalg.eigh(S)           # ascending eigenvalues, orthonormal Q
assert np.allclose(Q @ np.diag(lam) @ Q.T, S)
assert np.allclose(Q.T @ Q, np.eye(4))
print("eigenvalues:", lam.round(3), "all positive:", bool((lam > 0).all()))

# The quadratic form x^T S x is bounded by the extreme eigenvalues
x = rng.normal(size=4); x /= np.linalg.norm(x)
print("Rayleigh quotient in range:", lam[0] <= x @ S @ x <= lam[-1])

# A non-symmetric matrix can have complex eigenvalues
R = np.array([[0.0, -1.0], [1.0, 0.0]])  # 90-degree rotation
print("rotation eigenvalues:", np.linalg.eigvals(R))
```

Use `np.linalg.eigh` for symmetric matrices: it is faster, returns real eigenvalues in ascending order and orthonormal eigenvectors. `np.linalg.eig` handles the general case and may return complex numbers, as the rotation shows.

### Singular value decomposition

**Plain version.** SVD decomposes a real matrix into orthogonal changes of coordinates and axis-wise scaling. Those coordinate changes may include reflections.

Eigendecomposition needs a square matrix and may not exist. The **singular value decomposition** always exists, for every real matrix $A \in \mathbb{R}^{m \times n}$:

$$A = U\Sigma V^\top.$$

- $U \in \mathbb{R}^{m \times m}$ is orthogonal. Its columns are the **left singular vectors**, directions in the output space.

- $\Sigma \in \mathbb{R}^{m \times n}$ is rectangular diagonal. Its diagonal entries $\sigma_1 \geq \sigma_2 \geq \ldots \geq 0$ are the **singular values**.
- $V \in \mathbb{R}^{n \times n}$ is orthogonal. Its columns are the **right singular vectors**, directions in the input space.

Read from right to left, $Ax = U\Sigma V^\top x$ first expresses $x$ in the $V$ coordinates (a rotation or reflection), then scales coordinate $i$ by $\sigma_i$, then places the result along the $U$ directions. Figure 1 shows this for a $2 \times 2$ matrix: the unit circle becomes an ellipse whose semi-axes have lengths $\sigma_1$ and $\sigma_2$.

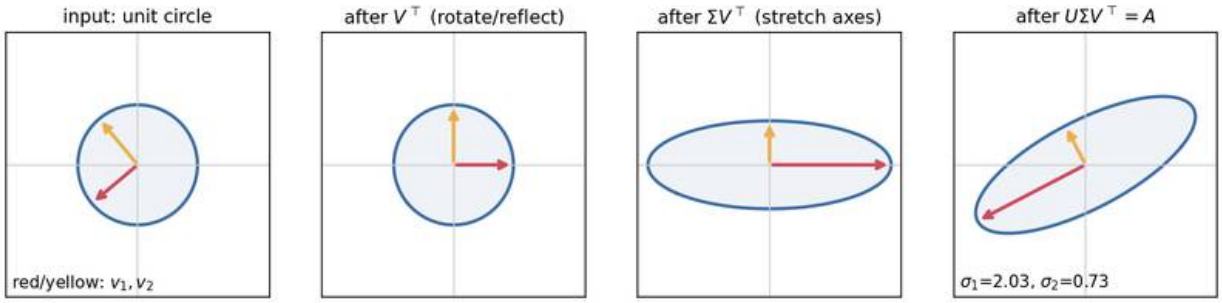



*Figure 1 — The SVD as rotate, stretch, rotate. The right singular vectors v1 and v2 (red, yellow) are first rotated onto the axes by V-transpose, stretched by the singular values, then rotated by U. The final ellipse is the image of the unit circle under A.*

The SVD and eigendecomposition are closely related. Since $A^\top A = V\Sigma^\top \Sigma V^\top$ and $AA^\top = U\Sigma\Sigma^\top U^\top$, the right singular vectors are eigenvectors of $A^\top A$, the left singular vectors are eigenvectors of $AA^\top$, and $\sigma_i = \sqrt{\lambda_i(A^\top A)}$. For a symmetric PSD matrix, the SVD and the eigendecomposition coincide.

The number of nonzero singular values equals the rank $r$. Keeping only those gives the **compact** (thin) SVD $A = U_r \Sigma_r V_r^\top = \sum_{i=1}^{r} \sigma_i u^{(i)} v^{(i)\top}$, a sum of $r$ rank-one matrices. That is the outer-product view of a matrix product again.

### The best low-rank approximation

Truncating that sum after $k$ terms gives $A_k = \sum_{i=1}^{k} \sigma_i u^{(i)} v^{(i)\top}$. The **Eckart–Young theorem** [6] says that $A_k$ is the best rank-$k$ approximation of $A$ in the Frobenius norm:

$$\min_{\operatorname{rank}(B) \leq k} \|A - B\|_F^2 = \|A - A_k\|_F^2 = \sum_{i>k} \sigma_i^2.$$

The same $A_k$ is also optimal in the spectral norm, with error $\|A - A_k\|_2 = \sigma_{k+1}$ [4]. In words: throw away the smallest singular values, and the energy you lose is exactly what you threw away. Storing $A_k$ needs $k(m + n + 1)$ numbers instead of $mn$.

Figure 2 applies this to a $256 \times 256$ grayscale photograph. Natural images have rapidly decaying singular values, so a rank-50 approximation already keeps the main structure. The streaky

artifacts at low rank come from the fact that each rank-one term is an outer product of a column pattern and a row pattern.




*Figure 2 — Rank-k approximations of the 256×256 'camera' image from scikit-image, with the relative Frobenius error in each title. Right: singular values on a log scale; dashed lines mark $k = 5, 20, 50$. The error of each approximation is set by the singular values to the right of its line.*

```
import numpy as np

rng = np.random.default_rng(0)
A = rng.normal(size=(12, 8))
U, s, Vt = np.linalg.svd(A, full_matrices=False)
assert np.allclose(U @ np.diag(s) @ Vt, A)
k = 3
Ak = (U[:, :k] * s[:k]) @ Vt[:k]
err_F = np.linalg.norm(A - Ak, "fro")**2
err_2 = np.linalg.norm(A - Ak, 2)
assert np.isclose(err_F, (s[k:]**2).sum()) and np.isclose(err_2, s[k])
print("rank-k squared Frobenius error:", round(err_F, 3))
print("rank-k spectral error = sigma_{k+1}:", round(err_2, 3), round(s[k], 3))
# Eigenvalues of A^T A are squared singular values of A
assert np.allclose(np.sort(np.linalg.eigvalsh(A.T @ A))[::-1], s**2)
print("storage: full", A.size, "numbers; rank-3 factors", k * (12 + 8 + 1))
```

`full_matrices=False` returns the thin SVD, which is what you almost always want. `(U[:, :k] * s[:k])` scales the columns of $U$ by broadcasting instead of building a diagonal matrix. Both Eckart–Young identities hold to machine precision.

One caveat: the directions with the largest singular values carry the most energy, not necessarily the most useful information. For classification, a small-variance direction can be the one that separates the classes. PR 05 contrasts PCA with discriminant methods for exactly this reason.

### The Moore–Penrose pseudo-inverse

**Plain version.** When a matrix has no inverse, the pseudo-inverse gives the next best thing: the answer that fits the data as closely as possible, and among equally good answers, the smallest

one.

Only square, full-rank matrices have inverses. For any $A \in \mathbb{R}^{m\times n}$, the **Moore–Penrose pseudo-inverse** $A^+ \in \mathbb{R}^{n\times m}$ is the unique matrix satisfying the four Penrose conditions [7]:

$$AA^+A = A, \quad A^+AA^+ = A^+, \quad (AA^+)^\top = AA^+, \quad (A^+A)^\top = A^+A.$$

It is easiest to compute from the SVD. If $A = U\Sigma V^\top$, then

$$A^+ = V\Sigma^+U^\top,$$

where $\Sigma^+ \in \mathbb{R}^{n\times m}$ is formed by transposing $\Sigma$ and replacing each nonzero singular value $\sigma_i$ by $1/\sigma_i$ (zeros stay zero). Goodfellow et al. give the equivalent limit $A^+ = \lim_{\alpha\searrow 0}(A^\top A + \alpha I)^{-1}A^\top$ [1], which already hints at ridge regression.

The pseudo-inverse gives the right answer in every regime of the rank table:

- **Tall, full column rank** ($m > n$). $Ax = b$ usually has no exact solution. Then $x = A^+b = (A^\top A)^{-1}A^\top b$ is the **least-squares** solution: it minimizes $\|Ax - b\|_2^2$.
- **Wide, full row rank** ($m < n$). There are infinitely many solutions. Then $x = A^+b = A^\top(AA^\top)^{-1}b$ is the solution with the smallest $\|x\|_2$, the **minimum-norm** solution. Any other solution adds a null-space vector, which can only increase the norm.
- **Rank-deficient.** $A^+b$ is both: among all least-squares minimizers, the one with the smallest norm.

The minimum-norm property is a small window into a modern question. Gradient descent on an under-determined least-squares problem, started from zero, converges to the minimum-norm solution, because every gradient $A^\top(Ax - b)$ lies in the row space of $A$. Over-parameterized networks are not linear, but this simple case is the usual starting point for thinking about which of many perfect fits a training algorithm picks.

### Least squares, the normal equations and conditioning

For a regression design matrix $X \in \mathbb{R}^{n\times d}$ and targets $y \in \mathbb{R}^n$, least squares minimizes $\|Xw - y\|_2^2$. Setting the gradient to zero gives the **normal equations**

$$X^\top Xw = X^\top y.$$

These are correct but numerically risky. The sensitivity of a linear problem is measured by the **condition number** $\kappa_2(A) = \sigma_{\max}/\sigma_{\min}$. Roughly, relative errors in the data can be amplified by up to a factor $\kappa$ in the solution [5]. Forming $X^\top X$ squares the condition number, $\kappa(X^\top X) = \kappa(X)^2$, so a mildly ill-conditioned problem becomes a severely ill-conditioned one. Library least-squares solvers (`np.linalg.lstsq`, `torch.linalg.lstsq`) work with QR or SVD factorizations of $X$ directly and avoid that squaring [4].

When two features are almost identical, $\sigma_{\min}$ is tiny, and the individual coefficients become extremely sensitive even when predictions are fine. **Ridge regression** [8] adds a penalty $\lambda\|w\|_2^2$ with $\lambda > 0$:

$$w_{\text{ridge}} = (X^\top X + \lambda I)^{-1} X^\top y = \sum_i \frac{\sigma_i}{\sigma_i^2 + \lambda} \, (u^{(i)\top} y) \, v^{(i)}.$$

The SVD form on the right shows what ridge does. Plain least squares uses the factor $1/\sigma_i$, which explodes for small $\sigma_i$. Ridge uses $\sigma_i/(\sigma_i^2+\lambda)$, which stays bounded by $1/(2\sqrt{\lambda})$. Large-$\sigma$ directions are nearly untouched; small-$\sigma$ directions are suppressed. The price is bias. In neural networks the same penalty is called **weight decay**.

```
import numpy as np

rng = np.random.default_rng(0)
# Over-determined: more equations than unknowns -> least squares
x = np.linspace(-1, 1, 40)
X = np.c_[np.ones_like(x), x]
y = 2 + 3*x + rng.normal(0, .05, len(x))
w, residuals, rank, sv = np.linalg.lstsq(X, y, rcond=None)
print("intercept and slope:", w.round(3), "rank:", rank)
assert np.allclose(w, np.linalg.pinv(X) @ y)

# Under-determined: fewer equations than unknowns -> minimum-norm solution
A = rng.normal(size=(2, 5)); b = rng.normal(size=2)
w_min = np.linalg.pinv(A) @ b
null = np.linalg.svd(A)[2][-1]             # a direction with A @ null = 0
w_other = w_min + 0.7 * null
print("both solve Aw=b:", np.allclose(A @ w_min, b), np.allclose(A @ w_other, b))
print("norms:", np.linalg.norm(w_min).round(3), "<", np.linalg.norm(w_other).round(3))

# Near-collinear features: condition number and ridge
z = rng.normal(size=200)
F = np.c_[z, z + 1e-4 * rng.normal(size=200)]
t = z + 0.01 * rng.normal(size=200)
print("cond(F) = %.1e, cond(F^T F) = %.1e" % (np.linalg.cond(F), np.linalg.cond(F.T @ F)))
w_ls = np.linalg.lstsq(F, t, rcond=None)[0]
lam = 1e-2
w_ridge = np.linalg.solve(F.T @ F + lam * np.eye(2), F.T @ t)
print("least squares:", w_ls.round(2), " ridge:", w_ridge.round(3))
```

The line fit recovers intercept $\approx 2.00$ and slope $\approx 3.01$, and `pinv` agrees with `lstsq`. The under-determined system has two exact solutions; the pseudo-inverse one has the smaller norm (0.885 versus 1.128). In the collinear case, $\kappa(F) \approx 2 \times 10^4$ while $\kappa(F^\top F) \approx 4 \times 10^8$, its square. Least squares returns wild coefficients $(-2.63, 3.63)$ whose sum is still about 1, so its predictions are fine. Ridge returns the stable split $(0.5, 0.5)$. The intercept column in the first example lets the line have a nonzero offset. In a real pipeline, fit any centering and scaling on the training set only, then apply it unchanged to validation and test data.

### The trace operator

**Plain version.** The trace adds up the diagonal. It looks too simple to matter, but it lets you rewrite awkward sums as matrix expressions and move matrices around inside them.

The **trace** of a square matrix is

$$\mathrm{Tr}(A) = \sum_i A_{ii}.$$

Its useful properties are:

- **Frobenius norm.** $\|A\|_F = \sqrt{\mathrm{Tr}(AA^\top)} = \sqrt{\mathrm{Tr}(A^\top A)}$.
- **Transpose.** $\mathrm{Tr}(A) = \mathrm{Tr}(A^\top)$.
- **Cyclic property.** $\mathrm{Tr}(ABC) = \mathrm{Tr}(CAB) = \mathrm{Tr}(BCA)$ whenever the products are defined, even when the matrices are not square and the intermediate products have different shapes. In particular $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$ for $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times m}$. You may cycle, but not arbitrarily reorder: $\mathrm{Tr}(ABC) \neq \mathrm{Tr}(ACB)$ in general.
- **Scalars.** A scalar is its own trace, $a = \mathrm{Tr}(a)$. This turns a quadratic form into a trace: $x^\top Ax = \mathrm{Tr}(x^\top Ax) = \mathrm{Tr}(Axx^\top)$.
- **Eigenvalues.** $\mathrm{Tr}(A) = \sum_i \lambda_i$. For a covariance matrix, that is the total variance, the sum of the variances along every principal direction.

The scalar trick is how many expectations are evaluated. For a random vector $x$ with mean zero and covariance $\Sigma$, $\mathbb{E}[x^\top Ax] = \mathbb{E}[\mathrm{Tr}(Axx^\top)] = \mathrm{Tr}(A\Sigma)$, because trace and expectation are both linear.

### The determinant

**Plain version.** The absolute determinant gives volume scaling; its sign records orientation. A zero determinant means a square matrix collapses a dimension and is not invertible.

The **determinant** $\det(A)$ of a square matrix is a single number with these properties:

- $\det(A) = \prod_i \lambda_i$, the product of the eigenvalues.
- $|\det(A)|$ is the factor by which $A$ scales volume. The unit cube maps to a parallelepiped of volume $|\det(A)|$.
- The sign records orientation. A negative determinant means $A$ includes a reflection.
- $\det(A) = 0$ exactly when $A$ is singular: some direction is collapsed to zero, so information is lost.
- $\det(AB) = \det(A)\det(B)$, $\det(A^\top) = \det(A)$, and $\det(A^{-1}) = 1/\det(A)$.

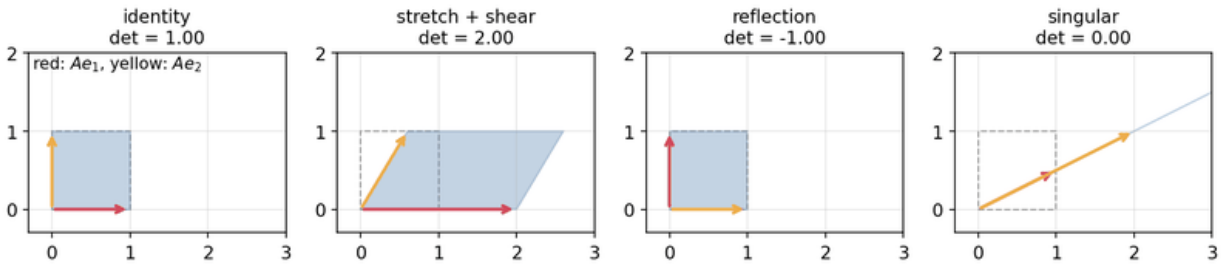



*Figure 3 — The determinant as signed area. Dashed: the unit square. Shaded: its image, spanned by Ae1*

(*red*) and *Ae2* (*yellow*). *A shear does not change area; a reflection flips the sign; a singular matrix collapses the square onto a line* ($\det = 0$).

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

Determinants appear in machine learning in two main places. The multivariate Gaussian density contains $\det(\Sigma)^{-1/2}$, the volume normalization of the covariance; ML 01B uses it. Normalizing flows (ML 07) need $\log|\det J|$ of a Jacobian matrix $J$ to account for how an invertible network stretches probability mass. In both cases you need the logarithm, never the raw value: the determinant of a $500 \times 500$ diagonal matrix with entries $10^{-3}$ is $10^{-1500}$, which underflows to zero in float64.

```
import numpy as np

rng = np.random.default_rng(2)
A, B, C = rng.normal(size=(3, 4)), rng.normal(size=(4, 5)), rng.normal(size=(5, 3))
print("cyclic trace:", np.isclose(np.trace(A @ B @ C), np.trace(C @ A @ B)),
      np.isclose(np.trace(A @ B @ C), np.trace(B @ C @ A)))

S = rng.normal(size=(4, 4))
lam = np.linalg.eigvals(S)
print("trace = sum of eigenvalues:", np.isclose(np.trace(S), lam.sum().real))
print("det = product of eigenvalues:", np.isclose(np.linalg.det(S), np.prod(lam).real))

# log-determinant of a covariance without overflow/underflow
Sigma = np.diag(np.full(500, 1e-3))
print("det underflows:", np.linalg.det(Sigma))
sign, logdet = np.linalg.slogdet(Sigma)
print("slogdet:", sign, round(logdet, 2))
L = np.linalg.cholesky(Sigma)
print("via Cholesky:", round(2 * np.log(np.diag(L)).sum(), 2))
```

The direct determinant prints `0.0`, while `slogdet` and the Cholesky route both give $\log\det = 500\log 10^{-3} \approx -3453.88$. For a PD matrix, $\det(A) = \prod_i L_{ii}^2$ when $A = LL^\top$, so $\log\det A = 2\sum_i \log L_{ii}$.

### Worked example: deriving principal component analysis

**Plain version.** PCA looks for the flat subspace that the data cloud lies closest to. Projecting onto it compresses each point to a few numbers while losing as little as possible.

PCA goes back to Pearson's "lines and planes of closest fit" [9] and Hotelling's analysis of "principal components" [10]; Jolliffe and Cadima give a modern review [11]. We derive it in the style of [1], using only the tools above.

**Setup.** We have $n$ points $x^{(1)}, \dots, x^{(n)} \in \mathbb{R}^d$ and want to store each one with only $l < d$ numbers. First subtract the mean $\mu = \frac{1}{n}\sum_i x^{(i)}$ from every point, so we can assume the data are centered. Stack the centered points as rows of $X \in \mathbb{R}^{n\times d}$.

**Step 1: choose the form of the code.** We use a linear **decoder** $g(c) = Dc$ with $D \in \mathbb{R}^{d \times l}$. To make the problem well-posed we require the columns of $D$ to be orthonormal, $D^\top D = I_l$. (Without this, we could shrink $c$ and enlarge $D$ forever.)

**Step 2: find the best code for a fixed decoder.** For a given $x$, choose $c$ to minimize the reconstruction error:

$$c^* = \arg\min_c \|x - Dc\|_2^2 = \arg\min_c \left( -2x^\top Dc + c^\top D^\top Dc \right) = \arg\min_c \left( -2x^\top Dc + c^\top c \right).$$

We expanded the square, dropped $x^\top x$ (it does not depend on $c$) and used $D^\top D = I$. The gradient with respect to $c$ is $-2D^\top x + 2c$. Setting it to zero gives the **encoder**

$$f(x) = D^\top x.$$

So encoding is a matrix–vector product, and the reconstruction is $r(x) = g(f(x)) = DD^\top x$, the orthogonal projection of $x$ onto the span of $D$.

**Step 3: find the best decoder.** Now choose $D$ to minimize the total reconstruction error over all points:

$$D^* = \arg\min_D \|X - XDD^\top\|_F^2 \quad \text{subject to } D^\top D = I_l.$$

Start with $l = 1$, so $D$ is a single unit vector $d$. Write the Frobenius norm as a trace and expand:

$$\begin{aligned} \|X - Xdd^\top\|_F^2 &= \operatorname{Tr}\left((X - Xdd^\top)^\top (X - Xdd^\top)\right) \\ &= \operatorname{Tr}(X^\top X) - 2\operatorname{Tr}(X^\top Xdd^\top) + \operatorname{Tr}(dd^\top X^\top Xdd^\top) \\ &= \operatorname{Tr}(X^\top X) - 2\, d^\top X^\top Xd + d^\top X^\top Xd\, (d^\top d) \\ &= \operatorname{Tr}(X^\top X) - d^\top X^\top Xd. \end{aligned}$$

The cyclic property moved $d^\top$ to the front in the second and third terms, and $d^\top d = 1$ simplified the last line. The first term does not depend on $d$. So minimizing the error is the same as

$$d^* = \arg\max_d \; d^\top X^\top Xd \quad \text{subject to } d^\top d = 1.$$

That is a Rayleigh quotient of the symmetric PSD matrix $X^\top X$. By the eigendecomposition section, it is maximized by the **eigenvector of $X^\top X$ with the largest eigenvalue**.

**Step 4: more than one component.** Repeating the argument for general $l$ (for example, by induction, requiring each new direction to be orthogonal to the previous ones) gives the answer: the columns of $D$ are the eigenvectors of $X^\top X$ for the $l$ largest eigenvalues [1]. The minimum error equals the sum of the discarded eigenvalues.

**Step 5: the variance view and the SVD link.** Dividing by $n$ turns $X^\top X$ into the sample covariance $S = \frac{1}{n} X^\top X$ (some texts use $n - 1$). The quantity $d^\top Sd$ is the variance of the projected data $Xd$. So "minimize reconstruction error" and "maximize retained variance" select the same directions: Pearson's and Hotelling's views agree. Finally, if the centered data matrix has thin SVD $X = U\Sigma V^\top$, then $X^\top X = V\Sigma^2 V^\top$. The principal directions are the right singular vectors $V$, the eigenvalues are $\sigma_i^2$, and the codes are $XV_l = U_l\Sigma_l$. In practice PCA is computed with the SVD of $X$ rather than by forming the covariance, for the conditioning reason we met in least squares.

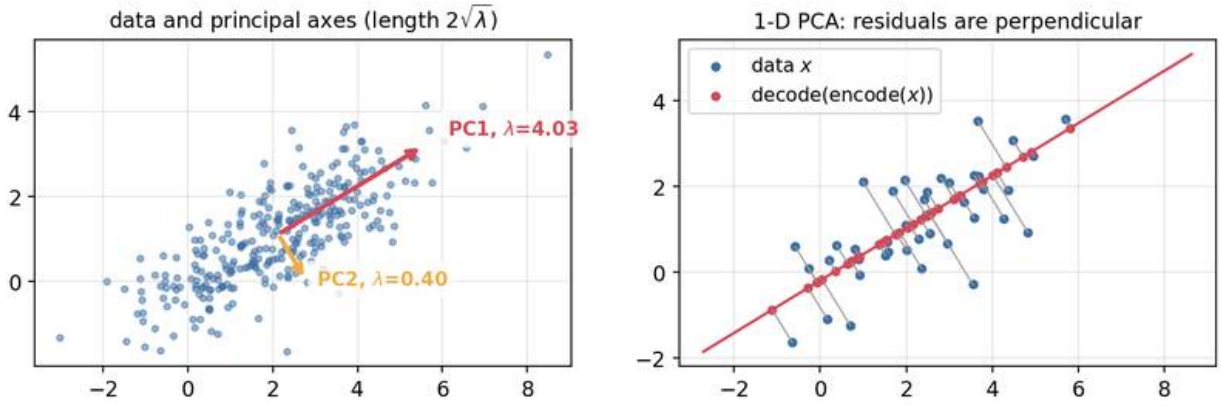



*Figure 4 — PCA on 300 correlated 2-D points. Left: principal axes from the eigendecomposition of the sample covariance, drawn with length $2\sqrt{\lambda}$. Right: a 1-D PCA reconstruction $\mathrm{decode}(\mathrm{encode}(x))$; each gray residual meets the red line at a right angle, because the reconstruction is an orthogonal projection.*

```
import numpy as np

rng = np.random.default_rng(0)
n, d, l = 500, 5, 2
Z = rng.normal(size=(n, l))
X = Z @ rng.normal(size=(l, d)) + 0.1 * rng.normal(size=(n, d)) + 3.0

mu = X.mean(axis=0)
Xc = X - mu                                  # centre first
S = Xc.T @ Xc / n                            # sample covariance (d x d)
lam, Q = np.linalg.eigh(S)
lam, Q = lam[::-1], Q[:, ::-1]               # descending
D = Q[:, :l]                                 # decoder matrix, orthonormal columns

codes = Xc @ D                               # encode: f(x) = D^T (x - mu)
X_hat = codes @ D.T + mu                     # decode: g(c) = D c + mu
mse = np.mean(np.sum((X - X_hat)**2, axis=1))
print("explained variance ratio:", (lam[:l].sum() / lam.sum()).round(4))
print("reconstruction MSE:", mse.round(5), " sum of discarded eigenvalues:",
 ↪  lam[l:].sum().round(5))
assert np.isclose(mse, lam[l:].sum())

# Same subspace from the SVD of the centred data matrix
U, s, Vt = np.linalg.svd(Xc, full_matrices=False)
assert np.allclose(s[:l]**2 / n, lam[:l])
assert np.allclose(np.abs(Vt[:l] @ D), np.eye(l), atol=1e-8)
print("SVD and eigendecomposition agree")
```

The synthetic data are 5-D points that really live near a 2-D plane, plus small noise. Two components explain 99.36% of the variance, and the average squared reconstruction error equals the sum of the three discarded eigenvalues, exactly as Step 4 predicts. The SVD route finds the same directions (up to sign, which is arbitrary for eigenvectors).

Three practical notes. Always center the data, or the first "component" just points at the mean. Standardize features if they have incomparable units, because PCA is not scale-invariant. And fit the mean and directions on training data only. PCA is the linear ancestor of autoencoders; ML 06A develops the probabilistic version and its nonlinear successors.

*Online only: PCA — Jolliffe and Cadima's Review — https://blog.shyandram.dev/en/learn/ml/linear-algebra/papers/pca–jolliffe-and-cadimas-review/*

### Matrix calculus you will meet

**Plain version.** Training means following gradients, and most gradients in ML are a handful of matrix formulas used over and over.

For a scalar function $f(x)$ of a vector $x \in \mathbb{R}^n$, the **gradient** $\nabla_x f \in \mathbb{R}^n$ collects the partial derivatives $\partial f / \partial x_i$. For a vector function $f : \mathbb{R}^n \to \mathbb{R}^m$, the **Jacobian** $J \in \mathbb{R}^{m \times n}$ has entries $J_{ij} = \partial f_i / \partial x_j$. The rules you will use most are:

| Function | Gradient | Note |
|---|---|---|
| $a^\top x$ | $a$ | linear function |
| $x^\top A x$ | $(A + A^\top)x$ | becomes $2Ax$ when $A$ is symmetric |
| $\frac{1}{2}\lVert Xw - y \rVert_2^2$ | $X^\top(Xw - y)$ | least-squares loss |
| $\lVert x \rVert_2^2$ | $2x$ | weight decay |
| $\operatorname{Tr}(AB)$ w.r.t. $A$ | $B^\top$ | used with the trace tricks above |

Each can be verified by writing the expression as a sum over indices and differentiating term by term [2]. The **chain rule** for vector functions multiplies Jacobians: if $y = g(x)$ and $L = f(y)$ is a scalar, then $\nabla_x L = J_g^\top \nabla_y L$. Backpropagation applies this rule layer by layer, multiplying transposed local Jacobians by the incoming gradient without ever building the full Jacobians, which would be enormous. ML 02 builds this up in detail, and ML 01C adds the Hessian, the matrix of second derivatives.

Setting the least-squares gradient to zero recovers the normal equations from the previous section. The Hessian of that loss is $X^\top X$, which is PSD, so the loss is convex and every stationary point is a global minimum. A quick finite-difference check confirms the formula:

```
import numpy as np

rng = np.random.default_rng(3)
X, y = rng.normal(size=(30, 4)), rng.normal(size=30)
w = rng.normal(size=4)
L = lambda w: 0.5 * np.sum((X @ w - y)**2)
grad = X.T @ (X @ w - y)                    # analytic gradient
eps = 1e-6
num = np.array([(L(w + eps*e) - L(w - eps*e)) / (2*eps) for e in np.eye(4)])
print("max gradient mismatch:", np.abs(grad - num).max())
# Setting the gradient to zero gives the normal equations
w_star = np.linalg.lstsq(X, y, rcond=None)[0]
print("gradient at solution ~ 0:", np.abs(X.T @ (X @ w_star - y)).max() < 1e-10)
```

The analytic and numerical gradients agree to about $10^{-8}$, the size expected from finite-difference rounding.

### Modern view

The definitions in this chapter are a century old, but the way they are used in machine learning keeps changing. Four lines of work are worth knowing about.

**Randomized numerical linear algebra.** Exact SVDs of large matrices are expensive, roughly $O(mn \min(m, n))$ operations. Halko, Martinsson and Tropp's SIAM Review survey [12] made a simple idea standard: multiply $A$ by a thin random matrix $\Omega$ to "sketch" its range, orthonormalize the result to get a basis $Q$, and compute an exact SVD of the small matrix $Q^\top A$. With a little oversampling and a few power iterations, the result is accurate for matrices whose spectrum decays. Martinsson and Tropp's *Acta Numerica* survey [13] covers the theory and a wide range of algorithms: norm estimation, random embeddings, regression, low-rank approximation, Nyström approximation of PSD matrices, streaming algorithms and kernel matrices from machine learning. Murray et al. [14] (LAPACK Working Note 299, 2023) take the software view. They note that RandNLA methods have entered MATLAB, the NAG Library, NVIDIA's cuSOLVER and scikit-learn, and they argue for standard "RandBLAS" and "RandLAPACK" libraries. scikit-learn's randomized PCA solver is a direct descendant. The core algorithm fits in a few lines:

```
import numpy as np

def randomized_svd(A, k, oversample=10, power_iters=2, seed=0):
    rng = np.random.default_rng(seed)
    Omega = rng.normal(size=(A.shape[1], k + oversample))  # random test matrix
    Y = A @ Omega                                          # sample the range of A
    for _ in range(power_iters):                           # sharpen a slow spectrum
        Y = A @ (A.T @ Y)
    Q, _ = np.linalg.qr(Y)                                 # orthonormal basis
    B = Q.T @ A                                            # small (k+p) x n matrix
    Ub, s, Vt = np.linalg.svd(B, full_matrices=False)
    return (Q @ Ub)[:, :k], s[:k], Vt[:k]

rng = np.random.default_rng(1)
m, n, r = 2000, 500, 20
A = rng.normal(size=(m, r)) @ rng.normal(size=(r, n)) + 0.01 * rng.normal(size=(m, n))
U, s, Vt = randomized_svd(A, k=r)
s_exact = np.linalg.svd(A, compute_uv=False)[:r]
print("max relative error in top singular values:", np.max(np.abs(s - s_exact) / s_exact))
rel = np.linalg.norm(A - (U * s) @ Vt) / np.linalg.norm(A)
print("relative reconstruction error:", round(rel, 5))
```

On this nearly rank-20 matrix the top 20 singular values match the exact ones to about $10^{-15}$ relative error, and the rank-20 reconstruction error (about 0.2%) is set by the added noise. The power iterations are what make it work when the spectrum decays slowly. For a matrix with a flat spectrum, no rank-$k$ method can do well, randomized or not.

**Low-rank structure in large models.** The outer-product view of matrix products has become a practical tool for adapting large networks. LoRA [15] freezes a pretrained weight $W_0 \in \mathbb{R}^{d\times k}$ and learns an update $\Delta W = BA$ with $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$ and $r \ll \min(d,k)$. Only $r(d+k)$ numbers are trained instead of $dk$; for $d = k = 4096$ and $r = 8$, that is 65,536 instead of about 16.8 million per matrix. $B$ starts at zero, so training begins exactly at the pretrained model. GaLore [16] (ICML 2024) keeps all weights trainable but projects the *gradients* onto a low-rank subspace, which shrinks the optimizer's memory. Its authors argue that restricting the weight update itself to a low-rank subspace, as LoRA does, can underperform full-rank training. Both rely on the Eckart–Young intuition that much of a matrix's useful content lives in a few singular directions. Whether that holds for a given task is an empirical question, not a theorem.

**Controlling the spectral norm.** A network's sensitivity to input perturbations is bounded by the product of its layers' spectral norms (for 1-Lipschitz activations such as ReLU). Spectral normalization [17] divides each weight matrix by an estimate of its largest singular value, obtained cheaply with one step of power iteration per training step. It was introduced to stabilize the discriminator in GANs (ML 07), and the same idea appears wherever a Lipschitz bound is wanted.

**Beyond matrices.** Convolution kernels and video features are naturally tensors with three or more axes. Kolda and Bader's SIAM Review survey [18] is the standard introduction to tensor decompositions. It covers the CP decomposition (a sum of rank-one tensors, generalizing the outer-product view) and the Tucker decomposition (a small core tensor multiplied by a matrix along each mode, generalizing the SVD). Unlike matrices, computing the rank of a tensor is hard in general, and best low-rank approximations need not exist. The clean Eckart–Young picture does not carry over. These decompositions are used to compress layers and to analyze multi-way data.

**Open questions.** Why do trained weight matrices often have rapidly decaying spectra, and how does that relate to generalization? How low can the rank of an adaptation be before quality drops, and does that depend on the task or the model? How should randomized algorithms be combined with low-precision hardware, where rounding errors are much larger than in float64? None of these have settled answers. The tools in this chapter are what you need to read the papers that study them. For broader textbook treatments that connect linear algebra to deep learning, see [1], [2] and [19].

### Key takeaways

- Name each axis and check shapes before multiplying or broadcasting arrays.
- Rank and null space constrain possible solutions; use a linear solver instead of explicitly inverting a matrix to solve a system.
- Real symmetric matrices admit an orthonormal eigenbasis; eigenvalue signs describe their quadratic forms.
- Truncated SVD gives a best low-rank approximation under spectral or Frobenius error.
- The pseudo-inverse selects the minimum-norm least-squares solution; small singular values can amplify noise.
- PCA on centered data maximizes retained variance and minimizes squared reconstruction er-

ror for a fixed subspace dimension.

## Exercises

1. **Shapes and bias.** A batch has shape $32 \times 64$ and a weight matrix has shape $64 \times 10$. What shape should the bias have, and what goes wrong if you use a bias of shape $(32, 1)$ instead?

   **Hint**

   A bias of shape $(10,)$ broadcasts over the 32 rows: one offset per output unit, shared by every example. A $(32, 1)$ bias broadcasts across the 10 columns, giving one offset per *example*. The code runs, but the "bias" now depends on the position of an example in the batch, which is meaningless and changes with batch size.

2. **Projection.** Let $Q \in \mathbb{R}^{n \times k}$ have orthonormal columns and $P = QQ^\top$. Show that $P^2 = P$, $P^\top = P$, and that $x - Px$ is orthogonal to every column of $Q$. What are the eigenvalues of $P$?

   **Hint**

   $P^2 = Q(Q^\top Q)Q^\top = QIQ^\top = P$. For orthogonality compute $Q^\top(x - Px) = Q^\top x - Q^\top QQ^\top x = 0$. From $P^2 = P$, any eigenvalue satisfies $\lambda^2 = \lambda$, so $\lambda \in \{0, 1\}$: eigenvalue 1 with multiplicity $k$ (the subspace) and 0 with multiplicity $n - k$ (its orthogonal complement). Hence $\mathrm{Tr}(P) = k$.

3. **SVD by hand.** Find the singular values of $A = \begin{bmatrix} 3 & 0 \\ 4 & 5 \end{bmatrix}$ by computing the eigenvalues of $A^\top A$. Check that $\|A\|_F^2 = \sigma_1^2 + \sigma_2^2$ and $|\det A| = \sigma_1 \sigma_2$.

   **Hint**

   $A^\top A = \begin{bmatrix} 25 & 20 \\ 20 & 25 \end{bmatrix}$ has eigenvalues 45 and 5, so $\sigma_1 = 3\sqrt{5}$, $\sigma_2 = \sqrt{5}$. Then $\sigma_1^2 + \sigma_2^2 = 50 = 9 + 16 + 25$, and $\sigma_1 \sigma_2 = 15 = |\det A| = |3 \cdot 5 - 0 \cdot 4|$. In general $|\det A| = \prod_i \sigma_i$ for a square matrix, because $|\det U| = |\det V| = 1$.

4. **Conditioning and ridge.** Starting from the least-squares code, replace the second feature by an exact copy of the first. What does `np.linalg.lstsq` return, and why is it the minimum-norm solution? Then derive the SVD form of the ridge solution and show that each factor $\sigma/(\sigma^2 + \lambda)$ is at most $1/(2\sqrt{\lambda})$.

   **Hint**

   With identical columns, the rank is 1 and the solution set is a line $w_1 + w_2 = c$. `lstsq` uses the SVD and returns the pseudo-inverse solution, which splits the weight equally, $w_1 = w_2 = c/2$, because that point on the line is closest to the origin. For the bound, maximize $g(\sigma) = \sigma/(\sigma^2 + \lambda)$: $g'(\sigma) = 0$ at $\sigma = \sqrt{\lambda}$, giving $g = 1/(2\sqrt{\lambda})$.

5. **PCA error.** Using the derivation in this chapter, show that the total reconstruction error of $l$-component PCA equals $\sum_{i>l} \lambda_i(X^\top X)$, and that the fraction of variance retained is $\sum_{i \le l} \lambda_i / \sum_i \lambda_i$. Then run PCA on `sklearn.datasets.load_digits()` and find the smallest $l$ that keeps 90% of the variance.

   **Hint**

   Write $\|X - XDD^\top\|_F^2 = \mathrm{Tr}(X^\top X) - \mathrm{Tr}(D^\top X^\top X D)$ (use $D^\top D = I$ and the cyclic property). With $D$ the top-$l$ eigenvectors, the second trace is $\sum_{i \le l} \lambda_i$, and $\mathrm{Tr}(X^\top X) = \sum_i \lambda_i$. For the digits, center the $1797 \times 64$ matrix, compute its singular values, and take the cumulative sum of $\sigma_i^2$. Report your own number rather than trusting a quoted one.

6. **Determinants and volume.** A 2×2 matrix maps the unit square to a parallelogram with vertices $0$, $(2,1)$, $(1,3)$ and $(3,4)$. Write down a matrix that does this, compute its determinant, and explain why a normalizing flow must track $\log|\det J|$ for every layer.

   **Hint**

   Put the edge vectors in the columns: $A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$, with $\det A = 5$, so areas grow by a factor of 5. If a map stretches volume by 5, probability density must shrink by the same factor to keep total probability equal to 1. Stacking layers multiplies these factors, so their logarithms add, which is why flows sum log-determinants layer by layer.

## References


1. I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016, ch. 2 “Linear Algebra”. Book
2. M. P. Deisenroth, A. A. Faisal and C. S. Ong, *Mathematics for Machine Learning*, Cambridge University Press, 2020. Book
3. G. Strang, *Introduction to Linear Algebra*, 6th ed., Wellesley-Cambridge Press, 2023. Book site
4. G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed., Johns Hopkins University Press, 2013. DOI
5. L. N. Trefethen and D. Bau III, *Numerical Linear Algebra*, SIAM, 1997. DOI
6. C. Eckart and G. Young, “The Approximation of One Matrix by Another of Lower Rank,” *Psychometrika*, vol. 1, pp. 211–218, 1936. DOI
7. R. Penrose, “A Generalized Inverse for Matrices,” *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 51, pp. 406–413, 1955. DOI
8. A. E. Hoerl and R. W. Kennard, “Ridge Regression: Biased Estimation for Nonorthogonal Problems,” *Technometrics*, vol. 12, pp. 55–67, 1970. DOI
9. K. Pearson, “On Lines and Planes of Closest Fit to Systems of Points in Space,” *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, pp. 559–572, 1901. DOI
10. H. Hotelling, “Analysis of a Complex of Statistical Variables into Principal Components,” *Journal of Educational Psychology*, vol. 24, pp. 417–441, 1933. DOI

11. I. T. Jolliffe and J. Cadima, “Principal Component Analysis: A Review and Recent Developments,” *Philosophical Transactions of the Royal Society A*, vol. 374, 20150202, 2016. DOI
12. N. Halko, P.-G. Martinsson and J. A. Tropp, “Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions,” *SIAM Review*, vol. 53, no. 2, pp. 217–288, 2011. DOI ·arXiv
13. P.-G. Martinsson and J. A. Tropp, “Randomized Numerical Linear Algebra: Foundations and Algorithms,” *Acta Numerica*, vol. 29, pp. 403–572, 2020. DOI ·arXiv
14. R. Murray, J. Demmel, M. W. Mahoney, N. B. Erichson, M. Melnichenko, O. A. Malik, L. Grigori, P. Luszczek, M. Dereziński, M. E. Lopes, T. Liang, H. Luo and J. Dongarra, “Randomized Numerical Linear Algebra: A Perspective on the Field With an Eye to Software,” LAPACK Working Note 299, 2023. arXiv
15. E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang and W. Chen, “LoRA: Low-Rank Adaptation of Large Language Models,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2022. arXiv
16. J. Zhao, Z. Zhang, B. Chen, Z. Wang, A. Anandkumar and Y. Tian, “GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection,” *ICML*, 2024. arXiv
17. T. Miyato, T. Kataoka, M. Koyama and Y. Yoshida, “Spectral Normalization for Generative Adversarial Networks,” *ICLR*, 2018. arXiv
18. T. G. Kolda and B. W. Bader, “Tensor Decompositions and Applications,” *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009. DOI
19. C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts*, Springer, 2024. Book

# ML 01B • Probability and Information Theory

Random variables, common distributions, Bayes' rule, expectation and covariance, likelihood, entropy, cross-entropy, KL divergence and structured models, connected to how neural networks are trained and evaluated.

Online edition: https://blog.shyandram.dev/en/learn/ml/probability-information/

## What you'll learn

- Distinguish probabilities, densities and distributions.
- Relate joint, marginal and conditional probabilities.
- Describe variation and dependence without confusing correlation with independence.
- Use Bayes' rule, likelihood and priors to estimate unknown quantities.
- Connect entropy, cross-entropy and KL divergence to learning objectives.

## The big picture

A neural network never sees the whole world. It sees a finite training set, measured by imperfect sensors, labelled by busy people, and then it is asked about images it has never seen. Every step of that story involves **uncertainty**. Probability theory is the mathematics of uncertainty: it tells us how to describe what we do not know, how to combine evidence, and how to make predictions that admit doubt. Information theory, its close relative, measures *how much* uncertainty there is and how costly it is to use the wrong distribution. Together they explain almost every loss function in this site's ML and DL tracks.

> **Explain like I'm five**
>
> Imagine a bag of marbles you cannot see into. **Probability** is your best guess about what you will pull out next: "probably red, maybe blue, almost never green." Every time you pull a marble, you learn a little more about the bag and update your guess — that updating is **Bayes' rule**. **Entropy** is how surprised you expect to be: a bag of all-red marbles never surprises you; a bag with ten colours in equal amounts surprises you a lot. A machine learning model is a very fancy guesser, and training means making its guesses less surprised by the real marbles.

This chapter follows the "Applied Math" part of *Deep Learning* by Goodfellow, Bengio and Courville [1], with additional material from Bishop and Bishop [2], Murphy [3] and *Mathematics for Machine Learning* [4]. It sits between ML 01A ·Linear Algebra and ML 01C ·Numerical Computation. If you are heading to the Pattern Recognition track, PR 01 ·Bayes Decisions builds directly on the Bayes-rule section here.

## Why probability?

**Plain version.** Probability gives learning systems a language for uncertain observations, predictions and model assumptions.

Goodfellow et al. [1] list three sources of uncertainty:

1. **Inherent stochasticity.** Some systems are genuinely random — photon arrival at a camera sensor (shot noise), the roll of a die, or the exact moment a pedestrian steps off a curb.
2. **Incomplete observability.** Even a deterministic system looks random if we cannot see all of its variables. A single photo of a car does not show whether its brake lights are about to turn on.
3. **Incomplete modelling.** A model must discard information to be useful. A classifier that sees $32 \times 32$ thumbnails cannot recover details that were lost when the image was downsampled, so its predictions about fine-grained classes must remain uncertain.

It is often more practical to use a simple rule with uncertainty ("most birds fly") than a complex rule that is always true ("birds fly, except penguins, ostriches, injured birds, ..."). 

### Two readings of a probability

A probability can be read in two ways, and both obey the same mathematical rules.

- **Frequentist probability** describes long-run frequencies of repeatable events: "if we toss this coin many times, about 70% of tosses land heads".
- **Bayesian probability** describes a *degree of belief* about something that might not be repeatable: "given this X-ray, I am 40% confident the patient has the disease".

If degrees of belief are required to satisfy a few common-sense consistency properties, they turn out to obey exactly the same rules as frequencies, which is why one calculus serves both readings [1], [3]. In practice, deep learning mixes the two: the data are treated as samples from a distribution (frequentist), while the model's output is often read as a belief about the label (Bayesian).

### Random variables and probability distributions

**Plain version.** A random variable is a quantity whose value is not known in advance. A probability distribution says how likely each value is.

We write random variables in roman letters (for example x in [1]); in this tutorial we use $X$ for the variable and $x$ for one of its values. A random variable can be **discrete** (a finite or countable set of values, such as a class label in $\{1, \dots, K\}$) or **continuous** (real values, such as a pixel intensity treated as a real number).

### Probability mass functions

A discrete variable has a **probability mass function** (PMF) $P(X = x)$, often shortened to $P(x)$. It must satisfy

$$0 \le P(x) \le 1 \text{ for every } x, \qquad \sum_{x} P(x) = 1.$$

Here the sum runs over every possible value $x$ of $X$. The first condition says no event is more than certain; the second says that *something* always happens. A joint PMF $P(X = x, Y = y)$ gives the probability that both events happen together.

### Probability density functions

A continuous variable has a **probability density function** (PDF) $p(x)$, which must satisfy

$$p(x) \geq 0, \qquad \int p(x)\, dx = 1.$$

A density is *not* a probability. The probability that $X$ falls in an interval $[a, b]$ is the area under the density:

$$P(a \leq X \leq b) = \int_a^b p(x)\, dx.$$

The probability of any single exact value is zero. Because only areas must be at most one, a density can exceed one: a narrow Gaussian with standard deviation 0.25 peaks at about 1.6 (Figure 1, top right). A common mistake is to report "probability 1.6" for a density value.

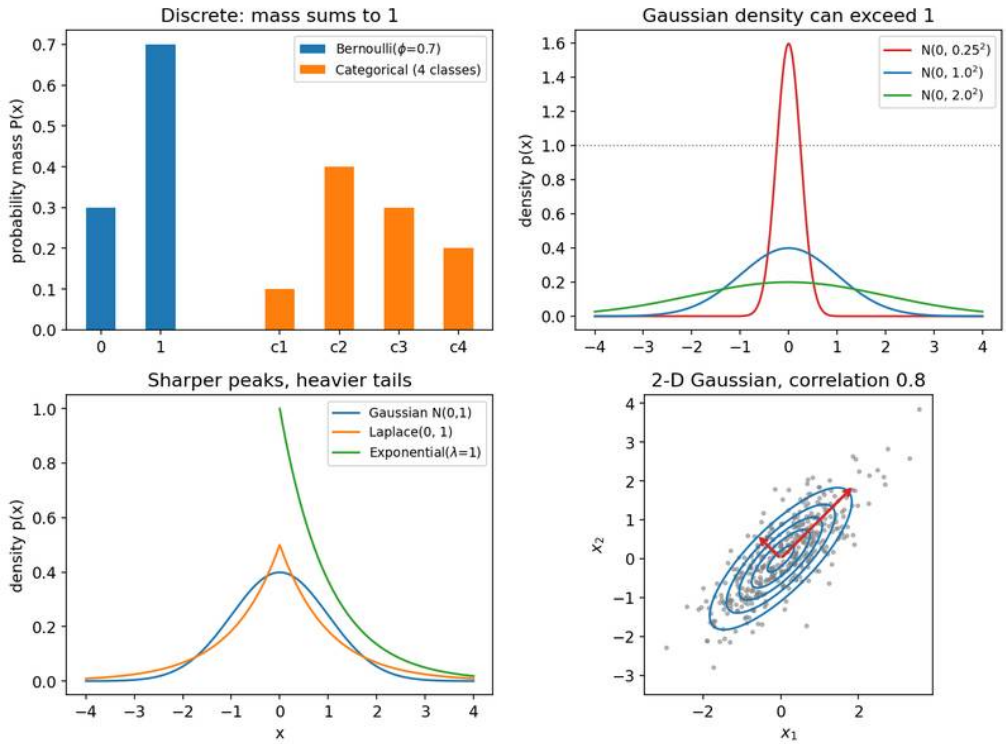



*Figure 1 — Common distributions. Top left: discrete masses sum to one. Top right: a density can exceed one because only its area must equal one. Bottom left: the Laplace density has a sharp peak and heavier tails than the Gaussian; the exponential lives on $x \geq 0$. Bottom right: a 2-D Gaussian with correlation 0.8; red arrows show the eigenvectors of its covariance, scaled by two standard deviations.*

### Marginal probability

**Plain version.** If you know how two things vary together, you can recover how one of them varies on its own by "summing out" the other.

Given a joint distribution over $X$ and $Y$, the **marginal** distribution of $X$ is

$$P(x) = \sum_{y} P(x, y) \quad \text{(discrete)}, \qquad p(x) = \int p(x, y)\, dy \quad \text{(continuous)}.$$

This is the **sum rule**. If the joint is written as a table with rows for $x$ and columns for $y$, the marginal of $x$ is the row sums — historically written in the margin of the page, hence the name.

### Conditional probability

**Plain version.** A conditional probability answers "how likely is this, *given that* I already know that?"

$$P(y \mid x) = \frac{P(x, y)}{P(x)}, \qquad P(x) > 0.$$

$P(y \mid x)$ is the probability of $Y = y$ after we have observed $X = x$. It is only defined when the condition can actually happen. Rearranging gives the **product rule** $P(x, y) = P(y \mid x)P(x)$.

Do not confuse conditioning with intervention. "The probability that the street is wet given that people carry umbrellas" is high, but *forcing* people to carry umbrellas will not wet the street. Questions about interventions belong to causal inference, which needs more than a joint distribution.

### The chain rule of probability

Applying the product rule repeatedly factorizes any joint distribution over $n$ variables:

$$P(x^{(1)}, \dots, x^{(n)}) = P(x^{(1)}) \prod_{i=2}^{n} P(x^{(i)} \mid x^{(1)}, \dots, x^{(i-1)}).$$

Here $x^{(i)}$ is the value of the $i$-th variable. This identity is *exact* — no assumption is involved. It is the backbone of autoregressive models: a language model or an autoregressive image model predicts one token or pixel at a time given all previous ones (ML 09, NLP 01). The modelling choice lies in *how* each conditional is parameterized, not in the factorization itself.

### Independence and conditional independence

Two variables are **independent**, written $X \perp Y$, if

$$p(x, y) = p(x)\, p(y) \quad \text{for all } x, y.$$

They are **conditionally independent** given $Z$, written $X \perp Y \mid Z$, if

$$p(x, y \mid z) = p(x \mid z)\, p(y \mid z) \quad \text{for all } x, y, z.$$

Ice-cream sales and sunburn are dependent, but roughly independent once you condition on the weather. Conditional independence is the main tool for making high-dimensional models tractable — naive Bayes assumes the features are independent given the class; a hidden Markov model assumes the future is independent of the past given the present state. These are *assumptions* [3]: they should be justified by the problem, not adopted only because they make the algebra easy.

### Code: a joint table, marginals and conditionals

The following example builds a small joint table for weather and umbrella use, computes marginals and conditionals, and tests independence. It also shows a case where two variables are clearly dependent yet have (almost) zero covariance.

```
import numpy as np

# Joint table P(weather, umbrella): rows = sunny/rainy, columns = no/yes
P = np.array([[0.56, 0.14],
              [0.06, 0.24]])
assert np.isclose(P.sum(), 1)

p_weather = P.sum(axis=1)            # marginal over umbrella
p_umbrella = P.sum(axis=0)           # marginal over weather
p_umb_given_w = P / p_weather[:, None]   # each row sums to 1
print("P(weather):", p_weather)
print("P(umbrella):", p_umbrella)
print("P(umbrella | weather):\n", p_umb_given_w)
print("independent?", np.allclose(P, np.outer(p_weather, p_umbrella)))

# Zero covariance does not imply independence
rng = np.random.default_rng(0)
x = rng.uniform(-1, 1, 200_000)
y = x**2                              # y is a deterministic function of x
print("cov(x, y) ~", np.cov(x, y)[0, 1].round(4))
```

Output: the marginals are $[0.7, 0.3]$ and $[0.62, 0.38]$; the conditional table is $[[0.8, 0.2], [0.2, 0.8]]$; the variables are not independent; and the sample covariance of $x$ and $x^2$ is about $-0.0007$, essentially zero, even though $y$ is completely determined by $x$.

### Expectation, variance and covariance

**Plain version.** The expectation is the long-run average; the variance is how far values typically spread from that average; the covariance says whether two quantities tend to go up and down together.

The **expectation** of a function $f$ under a distribution is

$$\mathbb{E}_{x\sim P}[f(x)] = \sum_x P(x) f(x), \qquad \mathbb{E}_{x\sim p}[f(x)] = \int p(x) f(x)\, dx.$$

The subscript $x \sim P$ names the distribution we average over; we drop it when it is obvious. Expectation is **linear**: $\mathbb{E}[\alpha f(x) + \beta g(x)] = \alpha \mathbb{E}[f(x)] + \beta \mathbb{E}[g(x)]$ for constants $\alpha, \beta$. Almost every training loss is an expectation over the data distribution, estimated by an average over a mini-batch.

The **variance** measures spread around the mean $\mu = \mathbb{E}[f(x)]$:

$$\mathrm{Var}(f(x)) = \mathbb{E}\big[(f(x) - \mu)^2\big] = \mathbb{E}[f(x)^2] - \mu^2.$$

Its square root is the **standard deviation**, which has the same units as $f(x)$.

The **covariance** of two quantities measures how they vary together:

$$\mathrm{Cov}(f(x), g(y)) = \mathbb{E}\big[(f(x) - \mathbb{E}[f(x)])\,(g(y) - \mathbb{E}[g(y)])\big].$$

A positive covariance means both tend to be above (or below) their means at the same time; a negative covariance means one tends to be high when the other is low. Its magnitude depends on the units, so we often normalize it into the **correlation**

$$\rho = \frac{\mathrm{Cov}(X, Y)}{\sqrt{\mathrm{Var}(X)\,\mathrm{Var}(Y)}} \in [-1, 1].$$

Independence implies zero covariance, but **zero covariance does not imply independence**. Covariance measures only *linear* dependence. In the code above, $y = x^2$ with $x$ uniform on $[-1, 1]$: knowing $x$ tells you $y$ exactly, but the covariance is zero because the relationship is symmetric and nonlinear. Mutual information, introduced later, captures every kind of dependence.

### The covariance matrix

For a random vector $\mathbf{x} \in \mathbb{R}^n$ with mean $\boldsymbol{\mu} = \mathbb{E}[\mathbf{x}]$, the **covariance matrix** is

$$\Sigma = \mathrm{Cov}(\mathbf{x}) = \mathbb{E}\big[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top\big], \qquad \Sigma_{ij} = \mathrm{Cov}(x_i, x_j).$$

Its diagonal holds the variances $\Sigma_{ii} = \mathrm{Var}(x_i)$. $\Sigma$ is symmetric and positive semidefinite, so (by ML 01A) it has real nonnegative eigenvalues and orthogonal eigenvectors. The eigenvectors are the principal axes of the data cloud and the eigenvalues are the variances along them — the red arrows in Figure 1 — which is exactly what PCA computes. For an affine map $\mathbf{y} = A\mathbf{x} + \mathbf{b}$, $\mathbb{E}[\mathbf{y}] = A\boldsymbol{\mu} + \mathbf{b}$ and $\mathrm{Cov}(\mathbf{y}) = A\Sigma A^\top$.

### Common probability distributions

**Plain version.** A handful of distributions appear over and over. Choosing one is choosing an assumption about your data or your target.

### Bernoulli

A single binary variable $X \in \{0, 1\}$ with parameter $\phi \in [0, 1]$:

$$P(X = 1) = \phi, \quad P(X = 0) = 1 - \phi, \quad P(X = x) = \phi^x (1 - \phi)^{1-x}.$$

Its mean is $\phi$ and its variance $\phi(1 - \phi)$. A binary classifier with a sigmoid output predicts the parameter $\phi$ of a Bernoulli distribution over the label.

### Categorical (multinoulli)

A single discrete variable with $K$ states, parameterized by a probability vector $\mathbf{p} \in [0, 1]^K$ with $\sum_k p_k = 1$:

$$P(X = k) = p_k.$$

If the label is encoded one-hot as $\mathbf{y} \in \{0, 1\}^K$, then $P(\mathbf{y}) = \prod_k p_k^{y_k}$. A softmax classifier outputs exactly such a vector. (The *multinomial* distribution counts outcomes over several draws; for one draw it reduces to the categorical, which Goodfellow et al. call "multinoulli" [1].)

### Gaussian (normal)

The most widely used continuous distribution:

$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\Big( - \frac{(x - \mu)^2}{2\sigma^2} \Big).$$

$\mu$ is the mean (and the peak), $\sigma^2$ the variance. Sometimes we use the **precision** $\beta = 1/\sigma^2$ instead, which is cheaper to evaluate when the variance changes often.

Two reasons make the Gaussian the default. The **central limit theorem** says that the sum of many independent variables with finite variance tends towards a Gaussian, so measurement noise built from many small effects is often near-Gaussian. And among all distributions with a given mean and variance on the real line, the Gaussian has the **maximum entropy**: it builds in the least extra structure.

In $n$ dimensions,

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Sigma) = \frac{1}{\sqrt{(2\pi)^n \det \Sigma}} \exp\Big( - \tfrac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^\top \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}) \Big),$$

where $\boldsymbol{\mu} \in \mathbb{R}^n$ is the mean vector and $\Sigma \in \mathbb{R}^{n \times n}$ is a symmetric positive-definite covariance matrix. The **precision matrix** $\Lambda = \Sigma^{-1}$ has a neat meaning: $\Lambda_{ij} = 0$ exactly when $x_i$ and $x_j$ are conditionally independent given all other coordinates. An **isotropic** Gaussian has $\Sigma = \sigma^2 I$ — the noise distribution used by diffusion models (ML 08).

A fundamental link to the losses you will use: if a regression network predicts the mean of a Gaussian with fixed variance, maximizing the likelihood is the same as minimizing the **mean squared error**.

### Exponential and Laplace

Sometimes we want a sharp point at a specific location.

$$p(x;\lambda) = \lambda\,\mathbf{1}_{x\geq 0}\,e^{-\lambda x}, \qquad \text{Laplace}(x;\mu,\gamma) = \frac{1}{2\gamma}\exp\Big(-\frac{|x-\mu|}{\gamma}\Big).$$

$\lambda > 0$ is the rate, $\mathbf{1}_{x\geq 0}$ is the indicator that is 1 when $x \geq 0$ and 0 otherwise, $\mu$ is the location and $\gamma > 0$ the scale. Compared with the Gaussian, the Laplace density has a sharper peak and heavier tails (Figure 1, bottom left). Predicting the location of a Laplace distribution by maximum likelihood is equivalent to minimizing the **absolute error** ($L_1$ loss), which is why $L_1$ is more tolerant of outliers than $L_2$; a Laplace *prior* on weights gives $L_1$ regularization (sparsity).

### Dirac and empirical distributions

The **Dirac delta** $\delta(x-\mu)$ puts all of its mass at the point $\mu$: it is zero everywhere else yet integrates to one. It is a generalized function, defined by what it does inside integrals: $\int \delta(x-\mu)f(x)\,dx = f(\mu)$.

A training set $\{\mathbf{x}^{(1)},\dots,\mathbf{x}^{(m)}\}$ defines the **empirical distribution**

$$\hat{p}(\mathbf{x}) = \frac{1}{m}\sum_{i=1}^{m}\delta(\mathbf{x}-\mathbf{x}^{(i)}),$$

which puts mass $1/m$ on each of the $m$ examples. Training a model by averaging a loss over the dataset is the same as taking an expectation under $\hat{p}$. It is also the distribution that maximizes the likelihood of the training data — which is precisely why minimizing training loss alone overfits.

### Mixtures and latent variables

**Plain version.** A mixture first picks a "type" at random, then draws a sample from the distribution belonging to that type.

$$P(\mathbf{x}) = \sum_{k=1}^{K} P(z=k)\,P(\mathbf{x}\mid z=k).$$

$z$ is a categorical **latent variable** — a variable we never observe directly — chosen with probabilities $\pi_k = P(z=k)$, and $P(\mathbf{x}\mid z=k)$ is the $k$-th component. A **Gaussian mixture model** (GMM) uses Gaussian components $\mathcal{N}(\mathbf{x};\boldsymbol{\mu}_k,\Sigma_k)$; with enough components it can approximate any smooth density to any desired nonzero error [1]. Given an observation $\mathbf{x}$, Bayes' rule gives the **posterior responsibility** $P(z=k\mid\mathbf{x})$: how likely it is that component $k$ generated that point. Fitting a GMM with expectation–maximization is the subject of PR 07.

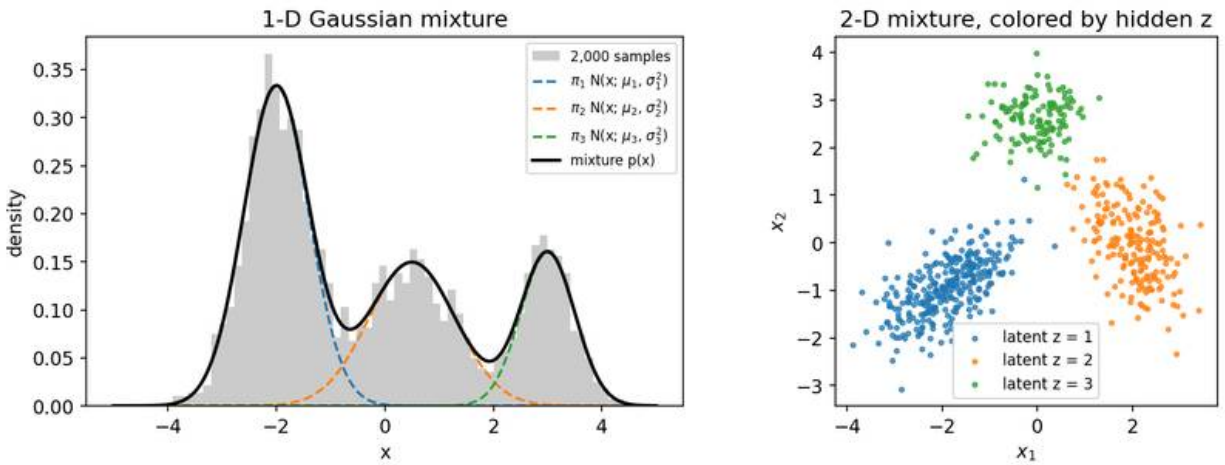



*Figure 2 — A Gaussian mixture. Left: the black density is the weighted sum of three components; the grey histogram is 2,000 samples drawn by first picking a component and then sampling from it. Right: in two dimensions, colour shows the hidden component $z$, which a real dataset would not reveal.*

Latent variables are central to modern generative models. A VAE (ML 07) is, informally, a mixture with a *continuous* latent $\mathbf{z}$ and a neural network for $p(\mathbf{x} \mid \mathbf{z})$ [8]; the marginal $p(\mathbf{x}) = \int p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})\, d\mathbf{z}$ is then intractable and must be approximated.

```
import numpy as np

rng = np.random.default_rng(0)
pi = np.array([0.5, 0.3, 0.2])      # mixing weights P(z = k)
mu = np.array([-2.0, 0.5, 3.0])     # component means
sd = np.array([0.6, 0.8, 0.5])      # component standard deviations

# Ancestral sampling: first the hidden component z, then x given z
z = rng.choice(3, size=100_000, p=pi)
x = rng.normal(mu[z], sd[z])

def log_normal(x, m, s):
    return -0.5*((x - m)/s)**2 - np.log(s) - 0.5*np.log(2*np.pi)

def mixture_logpdf(x):
    # log sum_k pi_k N(x; mu_k, sd_k^2), computed with log-sum-exp
    a = np.log(pi) + log_normal(x[:, None], mu, sd)
    m = a.max(axis=1, keepdims=True)
    return (m + np.log(np.exp(a - m).sum(axis=1, keepdims=True))).ravel()

grid = np.linspace(-6, 6, 4001)
print("density integrates to",
 ↪  (np.exp(mixture_logpdf(grid)).sum()*(grid[1]-grid[0])).round(4))
print("sample mean", x.mean().round(3), "vs exact", (pi*mu).sum().round(3))
# Posterior responsibility P(z | x) for one observation
a = np.log(pi) + log_normal(np.array([0.0])[:, None], mu, sd)
resp = np.exp(a - a.max()); resp /= resp.sum()
print("P(z | x=0):", resp.round(3))
```

The density integrates to 1.0, the sample mean ($-0.255$) matches the exact mean $\sum_k \pi_k \mu_k = -0.25$, and the point $x = 0$ is assigned to the middle component with probability 0.99. The log-sum-exp trick used here is explained in ML 01C.

### Useful functions: sigmoid and softplus

**Plain version.** The sigmoid squashes any real number into a probability between 0 and 1; the softplus is a smooth version of $\max(0, x)$ that is always positive.

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \qquad \zeta(x) = \log\left(1 + e^{x}\right).$$

The **logistic sigmoid** $\sigma$ produces the parameter $\phi$ of a Bernoulli distribution; it **saturates** (becomes flat) for large $|x|$, which makes gradients small there. The **softplus** $\zeta$ produces a positive number, such as a standard deviation $\sigma$ or a precision $\beta$; its name comes from being a softened "positive part" $x^+ = \max(0, x)$.

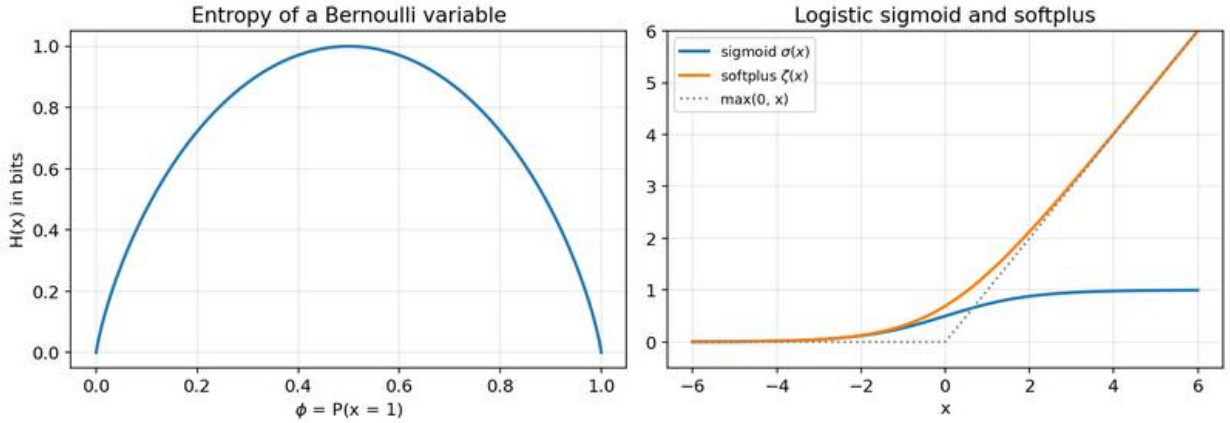



*Figure 3 — Left: entropy of a Bernoulli variable, maximal (1 bit) at $\varphi = 0.5$ and zero when the outcome is certain. Right: the logistic sigmoid $\sigma(x)$ and the softplus $\zeta(x)$, a smooth version of $\max(0, x)$.*

These identities are worth memorizing, because they appear every time a loss is simplified [1]:

$$1 - \sigma(x) = \sigma(-x), \qquad \frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)),$$

$$\log \sigma(x) = -\zeta(-x), \qquad \frac{d}{dx}\zeta(x) = \sigma(x),$$

$$\sigma^{-1}(p) = \log \frac{p}{1-p}, \qquad \zeta(x) - \zeta(-x) = x.$$

$\sigma^{-1}$ is the **logit** (log-odds) function. The identity $\log \sigma(x) = -\zeta(-x)$ shows why binary cross-entropy should be computed from logits: $-\log \sigma(x) = \zeta(-x)$, and softplus can be evaluated without overflow.

```
import numpy as np
```

```
def sigmoid(x):
    return 1/(1 + np.exp(-x))

def softplus(x):
    return np.logaddexp(0, x)          # log(1 + exp(x)) without overflow

x = np.linspace(-30, 30, 13)
assert np.allclose(1 - sigmoid(x), sigmoid(-x))
assert np.allclose(np.log(sigmoid(x)), -softplus(-x))
assert np.allclose(softplus(x) - softplus(-x), x)
h = 1e-6
assert np.allclose((softplus(x+h) - softplus(x-h))/(2*h), sigmoid(x), atol=1e-6)
p = sigmoid(x[4:9])
assert np.allclose(np.log(p/(1-p)), x[4:9])   # logit inverts sigmoid
print("naive log(1+exp(1000)):", np.log1p(np.exp(np.float64(1000))))
print("stable softplus(1000):", softplus(1000.0))
```

All identities pass. The naive expression returns `inf` (with an overflow warning) while `np.logaddexp` returns 1000.0.

### Bayes' rule

**Plain version.** Bayes' rule combines how well a hypothesis explains the evidence with how plausible it was beforehand.

From the product rule written both ways, $P(a, b) = P(a \mid b)P(b) = P(b \mid a)P(a)$, so

$$P(a \mid b) = \frac{P(b \mid a)\,P(a)}{P(b)}, \qquad P(b) = \sum_{a'} P(b \mid a')\,P(a').$$

- $P(a)$ is the **prior**: how plausible $a$ is before seeing the evidence.
- $P(b \mid a)$ is the **likelihood** of the evidence $b$ under hypothesis $a$.
- $P(a \mid b)$ is the **posterior**: the updated plausibility.
- $P(b)$ is the **evidence** (or marginal likelihood), the normalizer obtained by summing over all hypotheses $a'$.

#### A worked example: the base-rate effect

Consider a hypothetical defect detector with **sensitivity** $P(+ \mid \text{defect}) = 0.9$, **false-positive rate** $P(+ \mid \text{no defect}) = 0.05$, and a **prevalence** (base rate) $P(\text{defect}) = 0.01$. The probability that an item flagged as positive really is defective is

$$P(\text{defect} \mid +) = \frac{0.9 \cdot 0.01}{0.9 \cdot 0.01 + 0.05 \cdot 0.99} \approx 0.154.$$

Only about 15% of alarms are real, even though the detector catches 90% of defects. Counting makes this obvious: among 10,000 items, 100 are defective and 90 of them raise an alarm, but 9,900

are fine and 495 of them also raise an alarm (Figure 4, right). The same detector at 10% prevalence gives $0.09/(0.09 + 0.045) = 2/3$. This is a mathematical illustration, not a performance claim about any real system. Its lesson carries over to every deployed classifier: **sensitivity and false-positive rate alone do not determine how trustworthy a positive prediction is; the deployment prevalence matters.** ML 11 returns to this when discussing precision under distribution shift.

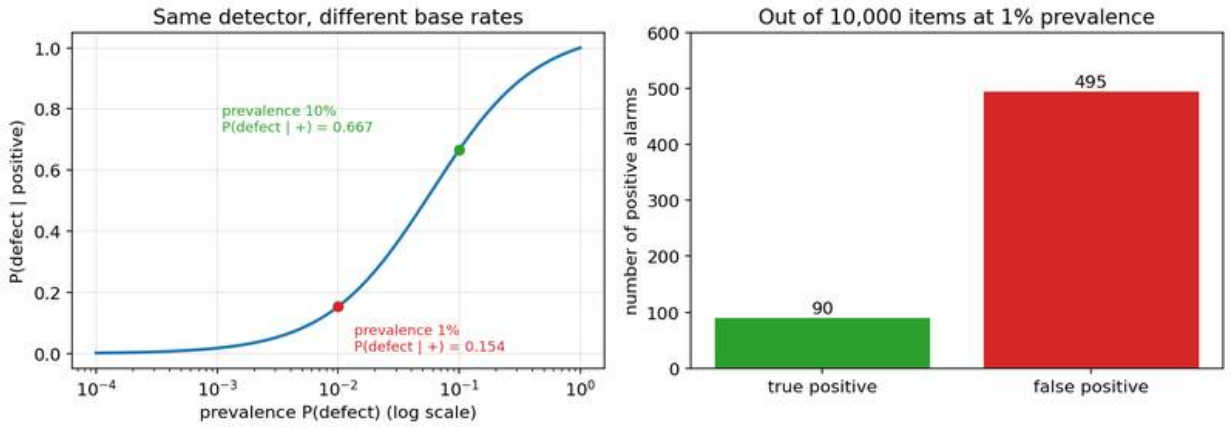



*Figure 4 — The base-rate effect for a hypothetical detector with sensitivity 0.9 and false-positive rate 0.05. Left: the posterior $P(defect \mid +)$ depends strongly on prevalence. Right: at 1% prevalence, false alarms outnumber true alarms more than five to one.*

```
import numpy as np

def posterior_positive(prevalence, sensitivity=0.9, false_pos=0.05):
    evidence = sensitivity*prevalence + false_pos*(1 - prevalence)   # P(+)
    return sensitivity*prevalence / evidence                          # P(defect | +)

for prev in [0.001, 0.01, 0.1, 0.5]:
    print(f"prevalence {prev:>5}: P(defect | +) = {posterior_positive(prev):.3f}")

# Simulation check with 1,000,000 synthetic items
rng = np.random.default_rng(0)
defect = rng.random(1_000_000) < 0.01
alarm = np.where(defect, rng.random(defect.size) < 0.9, rng.random(defect.size) < 0.05)
print("simulated:", defect[alarm].mean().round(3))
```

The formula gives 0.018, 0.154, 0.667 and 0.947 for the four prevalences, and a one-million-item simulation reproduces 0.154.

Bayes' rule also underlies the **Bayes classifier**: predicting the class $k$ with the largest posterior $P(y = k \mid \mathbf{x}) \propto p(\mathbf{x} \mid y = k)P(y = k)$ minimizes the expected error rate. PR 01 develops this decision theory in depth.

### Change of variables for densities

**Plain version.** If you stretch a rubber sheet, the ink on it gets thinner. When a random variable is transformed, its density must be rescaled by how much the transformation stretches space.

A tempting mistake: if $y = g(x)$ and $x$ has density $p_x$, then $p_y(y) = p_x(g^{-1}(y))$. This is wrong, because it does not integrate to one. The correct rule for an invertible, differentiable $g$ in one dimension is

$$p_y(y) = p_x(g^{-1}(y)) \left| \frac{d\, g^{-1}(y)}{dy} \right|.$$

The probability in a small interval must be preserved: $|p_y(y)\, dy| = |p_x(x)\, dx|$. In $n$ dimensions, with $\mathbf{y} = g(\mathbf{x})$,

$$p_x(\mathbf{x}) = p_y(g(\mathbf{x})) \left| \det \frac{\partial g(\mathbf{x})}{\partial \mathbf{x}} \right|,$$

where $\partial g / \partial \mathbf{x}$ is the $n \times n$ **Jacobian matrix** of $g$ (entry $(i, j)$ is $\partial y_i / \partial x_j$) and $|\det(\cdot)|$ is the absolute value of its determinant — the factor by which $g$ scales a small volume (see ML 01A for determinants). This formula is the entire foundation of **normalizing flows**, which stack invertible networks with cheap Jacobian determinants to obtain an exact likelihood [9].

```
import numpy as np

rng = np.random.default_rng(0)
x = rng.normal(0, 1, 500_000)        # x ~ N(0, 1)
y = np.exp(x)                        # y = g(x), invertible on (0, inf)

def p_x(x):
    return np.exp(-0.5*x**2)/np.sqrt(2*np.pi)

def p_y(y):
    # p_y(y) = p_x(g^{-1}(y)) * |d g^{-1}(y) / dy| = p_x(log y) / y
    return p_x(np.log(y))/y

counts, edges = np.histogram(y, bins=np.linspace(0.05, 4, 80), density=False)
centers = 0.5*(edges[:-1] + edges[1:])
empirical = counts/(y.size*np.diff(edges))
print("max |histogram - formula|:", np.abs(empirical - p_y(centers)).max().round(3))
print("wrong formula (no Jacobian) error:", np.abs(empirical -
 ↪  p_x(np.log(centers))).max().round(3))
```

With the Jacobian factor $1/y$ the histogram matches the formula to within 0.011 (sampling noise); without it the error is 0.46.

### Likelihood, maximum likelihood and MAP

**Plain version.** Hold the observed data fixed and ask which parameter values make that data most plausible. If you also have prior knowledge about the parameters, weigh that in too.

Suppose a model $p_{\boldsymbol{\theta}}(\mathbf{x})$ has parameters $\boldsymbol{\theta}$, and we observe a dataset $\mathcal{D} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ of independent and identically distributed (i.i.d.) examples. The **likelihood** is

$$L(\boldsymbol{\theta}) = p_{\boldsymbol{\theta}}(\mathcal{D}) = \prod_{i=1}^{m} p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}).$$

As a function of $\boldsymbol{\theta}$, this is *not* a probability distribution over parameters: it need not integrate to one over $\boldsymbol{\theta}$. The **maximum likelihood estimate** (MLE) is

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) = \arg\min_{\boldsymbol{\theta}} \; \mathbb{E}_{\mathbf{x}\sim\hat{p}}[\, -\log p_{\boldsymbol{\theta}}(\mathbf{x})\,].$$

We take logarithms because a product of many numbers below one underflows, while a sum of logs is numerically safe and has the same maximizer. The right-hand form says MLE **minimizes the average negative log-likelihood** (NLL) under the empirical distribution $\hat{p}$. For a classifier $p_{\boldsymbol{\theta}}(y \mid \mathbf{x})$ the same idea gives the conditional NLL $-\log p_{\boldsymbol{\theta}}(y \mid \mathbf{x})$, which is the cross-entropy loss we meet below.

Two classic results: for a Bernoulli coin with $k$ heads in $n$ flips, $\phi_{\mathrm{ML}} = k/n$; for a Gaussian, $\mu_{\mathrm{ML}}$ is the sample mean and $\sigma^2_{\mathrm{ML}} = \frac{1}{m}\sum_i (x^{(i)} - \mu_{\mathrm{ML}})^2$ — a slightly *biased* estimate (dividing by $m-1$ removes the bias).

### MAP estimation

**Bayesian** inference treats $\boldsymbol{\theta}$ itself as a random variable with a **prior** $p(\boldsymbol{\theta})$ and computes the **posterior**

$$p(\boldsymbol{\theta} \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})}{p(\mathcal{D})} \;\propto\; p(\mathcal{D} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta}).$$

The full posterior is often intractable, so a common shortcut is its mode, the **maximum a posteriori** (MAP) estimate:

$$\boldsymbol{\theta}_{\mathrm{MAP}} = \arg\max_{\boldsymbol{\theta}} \left[\, \log p(\mathcal{D} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) \,\right].$$

With a Gaussian prior $p(\boldsymbol{\theta}) = \mathcal{N}(\mathbf{0}, \frac{1}{\lambda} I)$, the term $-\log p(\boldsymbol{\theta})$ equals $\frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2$ plus a constant: **weight decay is MAP with a Gaussian prior**. A Laplace prior gives $L_1$ regularization. This links probabilistic assumptions to the regularizers in ML 02. But a MAP point estimate is still a single parameter vector; it does not tell you how uncertain the parameters are. Approximating the whole posterior is the topic of PR 08 and of variational inference [10].

```
import numpy as np

rng = np.random.default_rng(0)
flips = rng.random(10) < 0.7          # 10 coin flips, true P(heads) = 0.7
k, n = flips.sum(), flips.size

phi = np.linspace(1e-3, 1 - 1e-3, 999)
```

```
log_lik = k*np.log(phi) + (n - k)*np.log(1 - phi)
a, b = 5.0, 5.0                          # Beta(5, 5) prior, centered on 0.5
log_prior = (a - 1)*np.log(phi) + (b - 1)*np.log(1 - phi)

print(f"{k} heads in {n} flips")
print("MLE  grid:", phi[log_lik.argmax()].round(3), " closed form:", k/n)
print("MAP  grid:", phi[(log_lik + log_prior).argmax()].round(3),
      " closed form:", round((k + a - 1)/(n + a + b - 2), 3))

# Gaussian MLE: sample mean and the 1/m (biased) variance
x = rng.normal(2.0, 3.0, 50)
print("Gaussian MLE mean, var:", x.mean().round(3), x.var().round(3),
      " unbiased var:", x.var(ddof=1).round(3))
```

With this seed, 6 heads appear in 10 flips. The MLE is 0.6; the Beta$(5, 5)$ prior pulls the MAP estimate towards 0.5, giving 0.556, matching the closed form $(k + a - 1)/(n + a + b - 2)$. With more data the likelihood dominates and the two estimates converge. The Gaussian MLE variance (7.71) is smaller than the unbiased estimate (7.87), as expected.

### Information theory

**Plain version.** Information theory measures surprise. A rare event carries a lot of information when it happens; a certain event carries none. Averaging surprise gives entropy, and the extra surprise from using the wrong distribution gives KL divergence.

Claude Shannon founded the field to study how many bits are needed to send messages over a noisy channel [5]; the textbook treatment is Cover and Thomas [7]. For machine learning, the useful idea is to quantify how different two distributions are.

### Self-information and entropy

The **self-information** of an outcome $x$ is

$$I(x) = -\log P(x).$$

With the natural logarithm the unit is the **nat**; with $\log_2$ it is the **bit** (one nat $= 1/\ln 2 \approx 1.443$ bits). Certain events ($P = 1$) carry zero information; unlikely events carry a lot; and information from independent events adds, because logs turn products into sums.

The **Shannon entropy** is the expected self-information:

$$H(P) = \mathbb{E}_{x\sim P}[I(x)] = -\sum_{x} P(x) \log P(x).$$

We use the convention $0 \log 0 = 0$, justified by the limit $p \log p \to 0$ as $p \to 0$. Entropy is the minimum average number of bits (or nats) needed to encode outcomes drawn from $P$. It is zero for a deterministic outcome and maximal, $\log K$, for a uniform distribution over $K$ outcomes. For

a Bernoulli variable with parameter $\phi$, $H = -\phi \log \phi - (1-\phi)\log(1-\phi)$, which peaks at one bit when $\phi = 0.5$ (Figure 3, left).

For continuous variables, $h(p) = -\int p(x) \log p(x)\, dx$ is the **differential entropy**. Treat it carefully: it can be negative (a narrow uniform distribution on $[0, \frac{1}{2}]$ has $h = -\log 2$) and it changes when you rescale the coordinates. It is not a limiting version of the discrete count of bits.

### KL divergence

The **Kullback–Leibler divergence** [6] measures how different a distribution $Q$ is from a reference $P$:

$$D_{\mathrm{KL}}(P \,\|\, Q) = \mathbb{E}_{x\sim P}\Big[\log \frac{P(x)}{Q(x)}\Big] = \sum_x P(x) \log \frac{P(x)}{Q(x)}.$$

In coding terms, it is the *extra* number of nats needed to encode samples from $P$ when the code was designed for $Q$. Its key properties:

- **Nonnegative**: $D_{\mathrm{KL}}(P\|Q) \geq 0$, with equality if and only if $P = Q$ (for discrete variables; "almost everywhere" for continuous ones). This follows from Jensen's inequality, because $-\log$ is convex.
- **Infinite under unsupported mass**: if $Q(x) = 0$ for some $x$ with $P(x) > 0$, the divergence is infinite.
- **Asymmetric**: in general $D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$, and it does not satisfy the triangle inequality. KL is therefore *not* a distance metric, even though it is often described as "the distance between distributions".

### Forward versus reverse KL

The asymmetry has visible consequences when the approximating family is too simple. Figure 5 fits a single Gaussian $q$ to a target $p$ with two well-separated modes.

- Minimizing **forward KL** $D_{\mathrm{KL}}(p\|q)$ heavily penalizes $q$ for being small where $p$ has mass. The best Gaussian therefore **covers both modes**, placing its mean between them ($\mu = 0$, $\sigma \approx 2.62$) and putting a lot of mass in the empty valley. For Gaussian $q$, the forward-KL optimum simply matches the mean and variance of $p$.
- Minimizing **reverse KL** $D_{\mathrm{KL}}(q\|p)$ heavily penalizes $q$ for being large where $p$ is small. The best Gaussian therefore **locks onto one mode** ($\mu \approx 2.49$, $\sigma \approx 0.81$) and ignores the other.

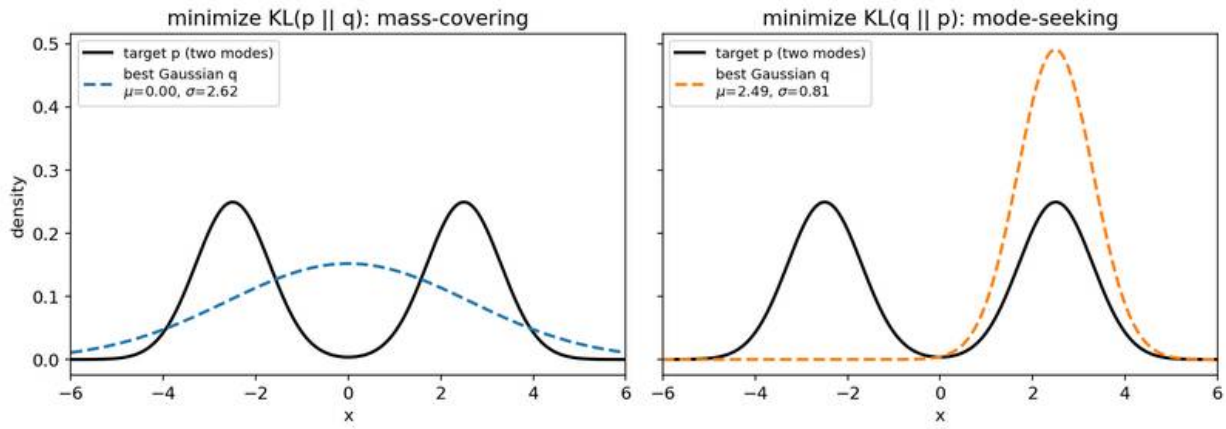



*Figure 5 — KL is asymmetric. Fitting one Gaussian $q$ to a two-mode target $p$: forward $KL(p\|q)$ gives a mass-covering fit centred between the modes; reverse $KL(q\|p)$ gives a mode-seeking fit on one peak (the other peak gives an equally good mirror solution).*

Which direction you minimize is a design choice. **Maximum likelihood minimizes forward KL** from the data distribution to the model, so likelihood-trained models (autoregressive models, flows, VAEs) tend to spread probability over all the data, sometimes producing blurry samples. **Variational inference usually minimizes reverse KL** from the approximation to the true posterior [10], and so tends to underestimate posterior uncertainty.

### Cross-entropy

The **cross-entropy** of $Q$ relative to $P$ is

$$H(P, Q) = -\mathbb{E}_{x\sim P}[\log Q(x)] = H(P) + D_{\mathrm{KL}}(P \,\|\, Q).$$

Because $H(P)$ does not depend on $Q$, minimizing cross-entropy with respect to $Q$ is the same as minimizing forward KL. In classification, $P$ is the empirical label distribution — a one-hot vector for each example — and $Q$ is the model's softmax output $\hat{\mathbf{y}}$. For one example with true class $y$,

$$H(\mathbf{e}_y, \hat{\mathbf{y}}) = -\sum_k [\mathbf{e}_y]_k \log \hat{y}_k = -\log \hat{y}_y,$$

where $\mathbf{e}_y$ is the one-hot vector for class $y$. Averaged over a dataset, this is exactly the conditional negative log-likelihood. So "cross-entropy loss", "log loss" and "negative log-likelihood of a categorical model" are three names for the same objective. Any loss consisting of a negative log-likelihood is a cross-entropy between the empirical distribution and the model — which is why the mean squared error is also "the cross-entropy" for a Gaussian model [1].

### Mutual information

**Mutual information** measures how much knowing one variable reduces uncertainty about another:

$$I(X;Y) = D_{\mathrm{KL}}(p(x,y) \,\|\, p(x)p(y)) = H(X) - H(X \mid Y) = H(X) + H(Y) - H(X,Y).$$

Here $H(X \mid Y) = -\mathbb{E}_{x,y}[\log p(x \mid y)]$ is the conditional entropy and $H(X,Y)$ the joint entropy. $I(X;Y) \geq 0$, with equality if and only if $X$ and $Y$ are independent — unlike covariance, it detects any form of dependence. It is symmetric in $X$ and $Y$.

### Code: information quantities

```
import numpy as np

p = np.array([.7, .2, .1])
q = np.array([.5, .3, .2])
entropy = -(p*np.log(p)).sum()
cross_entropy = -(p*np.log(q)).sum()
kl = (p*np.log(p/q)).sum()
reverse_kl = (q*np.log(q/p)).sum()
assert np.allclose(cross_entropy - entropy, kl) and kl >= 0
print("H(p), H(p,q), KL(p||q), KL(q||p) in nats:",
      np.round([entropy, cross_entropy, kl, reverse_kl], 4))
print("H(p) in bits:", (entropy/np.log(2)).round(4))

# Mutual information from a joint table (same weather/umbrella table)
P = np.array([[0.56, 0.14],
              [0.06, 0.24]])
px, py = P.sum(1), P.sum(0)
mi = (P*np.log(P/np.outer(px, py))).sum()
H = lambda t: -(t[t > 0]*np.log(t[t > 0])).sum()
assert np.isclose(mi, H(px) + H(py) - H(P.ravel()))
print("I(weather; umbrella) =", mi.round(4), "nats")

# Cross-entropy loss on one-hot labels equals mean -log q(y)
probs = np.array([[0.7, 0.2, 0.1], [0.1, 0.8, 0.1], [0.3, 0.3, 0.4]])
labels = np.array([0, 1, 2])
print("mean cross-entropy:", (-np.log(probs[np.arange(3), labels])).mean().round(4))
```

Output: $H(p) = 0.8018$, $H(p,q) = 0.8869$, $D_{\mathrm{KL}}(p\|q) = 0.0851$ and $D_{\mathrm{KL}}(q\|p) = 0.0920$ nats — the two directions differ. $H(p)$ is 1.1568 bits. The weather–umbrella mutual information is 0.1637 nats, and both formulas for it agree. The mean cross-entropy of the three predictions is 0.4987 nats. In real training code, compute the cross-entropy from logits with a fused function rather than from probabilities (see ML 01C).

*Online only: Shannon — A Mathematical Theory of Communication — https://blog.shyandram.dev/en/learn/ml/probability-information/papers/shannon–a-mathematical-theory-of-communication/*

### Structured probabilistic models

**Plain version.** Graphical models use conditional-independence structure to factor a joint distribution into smaller pieces. The graph alone does not establish causation.

A joint table over 100 binary variables would need $2^{100} - 1$ numbers. **Structured probabilistic models**, also called **graphical models**, exploit conditional independence to factorize the joint into small factors [1], [3]. There are two main kinds.

**Directed models** (Bayesian networks) use a directed acyclic graph $\mathcal{G}$. Each variable depends only on its parents $Pa_{\mathcal{G}}(x_i)$:

$$p(\mathbf{x}) = \prod_i p(x_i \mid Pa_{\mathcal{G}}(x_i)).$$

For example, with $a \to b \to c$ and $a \to c$ we have $p(a, b, c) = p(a)\, p(b \mid a)\, p(c \mid a, b)$; deleting the edge $a \to c$ asserts $c \perp a \mid b$ and shortens the last factor to $p(c \mid b)$. Directed graphs are natural for generative stories ("first choose the class, then the pixels"), and a mixture model is the simplest example: $z \to \mathbf{x}$.

**Undirected models** (Markov random fields) use an undirected graph. Each **clique** $C^{(i)}$ (a set of mutually connected nodes) has a nonnegative **factor** $\phi^{(i)}$, and

$$p(\mathbf{x}) = \frac{1}{Z} \prod_i \phi^{(i)}(C^{(i)}), \qquad Z = \sum_{\mathbf{x}} \prod_i \phi^{(i)}(C^{(i)}).$$

The **partition function** $Z$ normalizes the product; computing it is often the hard part. Undirected models suit symmetric interactions, like neighbouring pixels in an image prior or a conditional random field for segmentation.

A graph is a *description* of a distribution, not a different kind of distribution: any distribution can be written either way, although one may be much more compact. Deep learning often uses these ideas implicitly — an autoregressive model is a fully connected directed model, and a VAE is a two-node directed model $\mathbf{z} \to \mathbf{x}$ whose conditional is a neural network [8].

## Modern view

The probability and information theory in this chapter is classical. What has changed in the deep-learning era is *how* it is used and what goes wrong in practice. This section surveys the review literature and recent directions; it summarizes their framing rather than reproducing benchmark numbers.

### Calibration: are predicted probabilities honest?

A classifier is **calibrated** if, among all inputs for which it predicts confidence 0.8, about 80% are correct. Guo et al. [11] (ICML 2017) found that modern deep networks are often **overconfident**, and identified depth, width, weight decay and batch normalization as factors that affect calibration. Their main recommendation, **temperature scaling** — dividing all logits by a single scalar $T$ fitted on a validation set — is a one-parameter variant of Platt scaling and remains a strong baseline because it changes confidence without changing the predicted class. Wang's survey [12] organizes calibration methods into four families: post-hoc calibration, regularization during training, uncertainty estimation, and compositions of these; its revised version also covers calibration

of large language models. The probabilistic lesson: a softmax output is a model of $P(y \mid \mathbf{x})$, and like any model it can be wrong in its uncertainty, not only in its argmax.

### Uncertainty quantification

Two major reviews map the field of uncertainty in deep learning. Abdar et al. [13] (*Information Fusion*, 2021) review Bayesian approximations, ensembles and their applications across many domains. Gawlikowski et al. [14] (*Artificial Intelligence Review*, 2023) distinguish **reducible model (epistemic) uncertainty** from **irreducible data (aleatoric) uncertainty** — roughly the third and first sources of uncertainty from the start of this chapter — and compare deterministic single networks, Bayesian methods, ensembles and test-time augmentation, together with calibration measures.

Two primary studies are especially influential. **Deep ensembles** [15] train several networks from different random initializations and average their predictive distributions; this simple, parallel method was competitive with approximate Bayesian neural networks and expressed higher uncertainty on out-of-distribution inputs. Ovadia et al. [16] (NeurIPS 2019) benchmarked uncertainty methods under **dataset shift** and found that post-hoc calibration on in-distribution validation data did not carry over well, while methods that marginalize over models (such as ensembles) held up better.

A complementary, assumption-light approach is **conformal prediction**. Angelopoulos and Bates [17] give an accessible introduction: using a held-out calibration set, conformal methods turn any pre-trained model's scores into prediction *sets* that contain the true label with a user-chosen probability, assuming the calibration and test data are exchangeable. It does not require the model's probabilities to be calibrated, which makes it attractive for vision and language systems that are used as black boxes.

### Mutual information in representation learning

Mutual information became a popular objective for self-supervised learning (ML 06). Contrastive Predictive Coding [18] introduced the **InfoNCE** loss, which classifies a positive pair among negatives and can be shown to maximize a lower bound on mutual information. Poole et al. [19] (ICML 2019) unified many variational MI bounds and showed that they degrade when the true MI is large, trading high bias against high variance. Tschannen et al. [20] (ICLR 2020) argued that the success of these methods "cannot be attributed to the properties of MI alone": the encoder architecture and the parameterization of the estimator matter at least as much. The modern consensus is to treat InfoNCE-style objectives as well-behaved contrastive losses with an information-theoretic motivation, not as precise MI estimators.

### Divergences and latent variables in generative models

Generative modelling is applied probability. VAEs [8] maximize an evidence lower bound that combines a reconstruction likelihood with a KL term; Blei et al. [10] review variational inference,

which approximates intractable posteriors by minimizing (reverse) KL within a tractable family. Normalizing flows [9] are built entirely on the change-of-variables formula. Diffusion and flow-matching models (ML 08) combine Gaussian noise, conditional expectations and likelihood bounds. The choice of divergence (forward KL, reverse KL, or something else) shapes whether a model covers all the data or concentrates on its modes — the same picture as Figure 5, at scale.

### Open problems

- **Calibration under shift.** Calibration that holds on a validation set can fail on new cameras, hospitals or prompts [16]; detecting when this happens is an open question.
- **Separating uncertainty types.** Decomposing predictive uncertainty into epistemic and aleatoric parts is conceptually clear but hard to do reliably for large networks [14].
- **Estimating information in high dimensions.** Variational MI bounds have fundamental bias–variance limits [19], so information-theoretic claims about deep representations need care.
- **Uncertainty for foundation models.** Applying calibration and conformal methods to large vision–language and language models, whose outputs are open-ended, is an active research area [12], [17].

### Key takeaways

- Discrete probabilities sum to one; continuous densities integrate to one and may exceed one at a point.
- Marginalization and the chain rule are identities; independence is an additional assumption.
- Zero covariance does not generally imply independence.
- Likelihood and prior assumptions determine the resulting loss and regularization; keep their scale parameters explicit.
- Bayes' rule updates prior beliefs with evidence; density transformations also require a volume correction.
- Cross-entropy separates into entropy and KL. KL is asymmetric; mode-covering or mode-seeking behavior depends on the approximation family and problem.

### Exercises

1. **Units and certainty.** (a) Convert a loss of 1 nat into bits. (b) Compute the entropy, in bits, of a fair coin, a certain outcome, and a fair 8-sided die.

   **Hint**

   (a) Divide by $\ln 2$: about 1.443 bits. (b) 1 bit, 0 bits and $\log_2 8 = 3$ bits. A uniform distribution over $K$ outcomes has entropy $\log K$, the maximum for $K$ outcomes.

2. **Prevalence shift.** Using the detector in the Bayes section, find the prevalence at which a positive result is exactly a coin flip, $P(\text{defect} \mid +) = 0.5$.

**Hint**

Set $0.9\pi = 0.05(1-\pi)$, where $\pi$ is the prevalence. Then $\pi = 0.05/0.95 \approx 0.053$, about 5.3%. Below that, most alarms are false.

3. **Dependent but uncorrelated.** Let $X$ take values $-1, 0, 1$ with equal probability and $Y = X^2$. Show that $\mathrm{Cov}(X, Y) = 0$ and compute $I(X; Y)$ in bits.

**Hint**

$\mathbb{E}[X] = 0$ and $\mathbb{E}[XY] = \mathbb{E}[X^3] = 0$, so the covariance is zero. Because $Y$ is a function of $X$, $H(Y \mid X) = 0$ and $I(X; Y) = H(Y)$. $Y = 1$ with probability $2/3$ and $Y = 0$ with probability $1/3$, so $I \approx 0.918$ bits.

4. **Gaussian MLE equals squared error.** Show that maximizing $\prod_i \mathcal{N}(y^{(i)}; f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}), \sigma^2)$ over $\boldsymbol{\theta}$ (with $\sigma$ fixed) is equivalent to minimizing the mean squared error. What changes if $\sigma$ is also predicted by the network?

**Hint**

The negative log-likelihood is $\sum_i \frac{(y^{(i)} - f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))^2}{2\sigma^2} + m \log \sigma + \text{const}$. With $\sigma$ fixed only the squared error depends on $\boldsymbol{\theta}$. If the network also predicts $\sigma(\mathbf{x})$ (for example through a softplus), each example's error is weighted by $1/\sigma^2$ and a $\log \sigma$ penalty prevents the variance from growing without bound — a heteroscedastic regression model.

5. **Change of variables.** If $X \sim \mathcal{U}(0, 1)$ (uniform) and $Y = -\frac{1}{\lambda} \log X$ with $\lambda > 0$, what is the density of $Y$? Check your answer by sampling in NumPy.

**Hint**

$g^{-1}(y) = e^{-\lambda y}$ and $|dg^{-1}/dy| = \lambda e^{-\lambda y}$, while $p_X = 1$ on $(0, 1)$. So $p_Y(y) = \lambda e^{-\lambda y}$ for $y \geq 0$: the exponential distribution. This is the inverse-transform sampling trick.

6. **KL direction.** For $p = (0.5, 0.5, 0)$ and $q = (0.4, 0.4, 0.2)$, compute $D_{\mathrm{KL}}(p \| q)$ and $D_{\mathrm{KL}}(q \| p)$. Which one is infinite, and what does that tell you about training a model by maximum likelihood?

**Hint**

$D_{\mathrm{KL}}(p \| q) = \log(0.5/0.4) \approx 0.223$ nats (the third term is $0 \log 0 = 0$). $D_{\mathrm{KL}}(q \| p) = \infty$ because $q$ puts mass where $p$ has none. Maximum likelihood minimizes $D_{\mathrm{KL}}(p_{\text{data}} \| q_{\text{model}})$, which becomes infinite if the model assigns zero probability to any observed example — so likelihood-trained models must spread some probability everywhere the data might be.

## References


1. I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016, ch. 3 “Probability and Information Theory”. Book
2. C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts*, Springer, 2024. Book
3. K. P. Murphy, *Probabilistic Machine Learning: An Introduction*, MIT Press, 2022. Book
4. M. P. Deisenroth, A. A. Faisal and C. S. Ong, *Mathematics for Machine Learning*, Cambridge University Press, 2020. Book
5. C. E. Shannon, “A Mathematical Theory of Communication,” *Bell System Technical Journal*, vol. 27, pp. 379–423, 1948. DOI
6. S. Kullback and R. A. Leibler, “On Information and Sufficiency,” *The Annals of Mathematical Statistics*, vol. 22, pp. 79–86, 1951. DOI
7. T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed., Wiley, 2006. DOI
8. D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” in *Proc. International Conference on Learning Representations (ICLR)*, 2014. arXiv
9. G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed and B. Lakshminarayanan, “Normalizing Flows for Probabilistic Modeling and Inference,” *Journal of Machine Learning Research*, vol. 22, no. 57, pp. 1–64, 2021. arXiv
10. D. M. Blei, A. Kucukelbir and J. D. McAuliffe, “Variational Inference: A Review for Statisticians,” *Journal of the American Statistical Association*, vol. 112, no. 518, pp. 859–877, 2017. DOI
11. C. Guo, G. Pleiss, Y. Sun and K. Q. Weinberger, “On Calibration of Modern Neural Networks,” *ICML*, 2017. arXiv
12. C. Wang, “Calibration in Deep Learning: A Survey of the State-of-the-Art,” arXiv:2308.01222, 2023 (revised 2025). arXiv
13. M. Abdar et al., “A Review of Uncertainty Quantification in Deep Learning: Techniques, Applications and Challenges,” *Information Fusion*, vol. 76, pp. 243–297, 2021. DOI
14. J. Gawlikowski et al., “A Survey of Uncertainty in Deep Neural Networks,” *Artificial Intelligence Review*, vol. 56, pp. 1513–1589, 2023. DOI
15. B. Lakshminarayanan, A. Pritzel and C. Blundell, “Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles,” *NeurIPS (NIPS)*, 2017. arXiv
16. Y. Ovadia et al., “Can You Trust Your Model’s Uncertainty? Evaluating Predictive Uncertainty Under Dataset Shift,” *NeurIPS*, 2019. arXiv
17. A. N. Angelopoulos and S. Bates, “A Gentle Introduction to Conformal Prediction and Distribution-Free Uncertainty Quantification,” arXiv:2107.07511, 2021. arXiv
18. A. van den Oord, Y. Li and O. Vinyals, “Representation Learning with Contrastive Predictive Coding,” arXiv:1807.03748, 2018. arXiv
19. B. Poole, S. Ozair, A. van den Oord, A. A. Alemi and G. Tucker, “On Variational Bounds of Mutual Information,” *ICML*, 2019. arXiv
20. M. Tschannen, J. Djolonga, P. K. Rubenstein, S. Gelly and M. Lucic, “On Mutual Information Maximization for Representation Learning,” *ICLR*, 2020. arXiv

# ML 01C • Numerical Computation and Gradient Checks

Floating point, overflow and underflow, stable softmax, conditioning, gradients, Jacobians and Hessians, Newton-type methods, KKT conditions and mixed precision: make the mathematical model survive actual computation.

Online edition: https://blog.shyandram.dev/en/learn/ml/numerical-computation/

## What you'll learn

- Distinguish floating-point range from precision.
- Rewrite losses and normalization formulas to avoid numerical failure.
- Separate problem conditioning from algorithm stability.
- Use gradients and curvature to understand optimization and constraints.
- Check derivatives and validate mixed-precision computation.

## The big picture

Machine learning is written in the language of real numbers, but computers do not have real numbers. They have a finite set of floating-point numbers, and every arithmetic operation rounds its result to one of them. Most of the time this rounding is invisible. Sometimes it is catastrophic: an exponential overflows to infinity, a tiny probability becomes exactly zero and its logarithm becomes minus infinity, or two nearly equal numbers are subtracted and almost all their significant digits disappear.

The second half of the story is optimization. Training a model means minimizing a function of millions or billions of parameters, and we can only do that numerically, by taking many small steps guided by derivatives. How fast and how reliably those steps make progress depends on the shape of the loss surface — its slopes and its curvatures — and on how we take the steps. This chapter follows the "Numerical Computation" chapter of Goodfellow, Bengio and Courville [1], adds the floating-point details that modern low-precision training relies on, and connects each idea to small NumPy experiments.

> **Explain like I'm five**
>
> Imagine a calculator that can show only seven digits. If you ask it for one million plus 0.1, it shows one million: the 0.1 fell off the screen. If you then subtract one million, you get zero instead of 0.1. Good numerical code arranges its steps so the important digits never fall off the screen. Optimization is a different game: you are blindfolded on a hilly field and want to reach the lowest point. You can feel the slope under your feet (the gradient) and, with more effort, how the ground curves (the Hessian). Numerical computation is about walking down safely without your calculator lying to you.

Read ML 01B first if softmax and cross-entropy are new, and keep ML 01A nearby for eigenvalues and the singular value decomposition. ML 02 then uses everything here to train neural networks.

### Floating-point numbers

**Plain version.** Floating point stores a sign, significand and exponent. Relative spacing is roughly constant for normal numbers, but worsens near zero in the subnormal range.

### How a float is stored

The IEEE 754 standard [3] defines binary floating-point formats used by essentially all CPUs and GPUs. A normal (non-special) number is stored as

$$x = (-1)^s \times \left(1 + \sum_{i=1}^{t} b_i\, 2^{-i}\right) \times 2^{e},$$

where $s \in \{0,1\}$ is the sign bit, $b_1, \ldots, b_t \in \{0,1\}$ are the $t$ stored fraction bits of the significand (the leading 1 is implicit), and $e$ is an integer exponent restricted to a fixed range. Two numbers matter most:

- **Machine epsilon** $\varepsilon_{\text{mach}} = 2^{-t}$ is the gap between 1 and the next larger representable number.
- **Unit roundoff** $u = \varepsilon_{\text{mach}}/2$ bounds the relative error of rounding to nearest: for any real $x$ in the normal range, the stored value is $\mathrm{fl}(x) = x(1+\delta)$ with $|\delta| \le u$.

The standard also requires that each basic operation $(+, -, \times, \div, \sqrt{\cdot})$ is computed as if exactly and then rounded once: $\mathrm{fl}(a \circ b) = (a \circ b)(1+\delta)$, $|\delta| \le u$. This one rule is the foundation of rounding-error analysis [2], [4].

Below the smallest normal number the format uses **subnormal** numbers, which give up leading precision to fill the gap down to zero gradually. Special values complete the picture: $\pm\infty$ for results that are too large, and **NaN** ("not a number") for invalid operations such as $0/0$, $\infty - \infty$ or $\infty \times 0$. Once a NaN appears, almost every operation that touches it returns NaN, which is why a single bad value can poison a whole training run.

### The four formats you will meet

The table below was computed with `numpy.finfo` (and `torch.finfo` for bfloat16, which NumPy lacks). "Bits" lists sign/exponent/fraction.

| Format | Bits | $\varepsilon_{\text{mach}}$ | Largest finite | Smallest normal | Smallest subnormal |
|---|---|---|---|---|---|
| float16 (IEEE half) | 1/5/10 | $9.77 \times 10^{-4}$ | 65504 | $6.10 \times 10^{-5}$ | $5.96 \times 10^{-8}$ |
| bfloat16 | 1/8/7 | $7.81 \times 10^{-3}$ | $3.39 \times 10^{38}$ | $1.18 \times 10^{-38}$ | $9.18 \times 10^{-41}$ |
| float32 (single) | 1/8/23 | $1.19 \times 10^{-7}$ | $3.40 \times 10^{38}$ | $1.18 \times 10^{-38}$ | $1.40 \times 10^{-45}$ |
| float64 (double) | 1/11/52 | $2.22 \times 10^{-16}$ | $1.80 \times 10^{308}$ | $2.23 \times 10^{-308}$ | $4.94 \times 10^{-324}$ |

Read the table in two directions. The **exponent bits** set the *range*: float16 overflows above 65504, while bfloat16 keeps float32's 8 exponent bits and therefore its range. The **fraction bits** set the

*precision*: float32 carries about 7 significant decimal digits, float64 about 16, float16 about 3, and bfloat16 only about 2. That trade-off — bfloat16's wide range with coarse precision versus float16's narrow range with finer precision — is the reason the two half-precision formats need different training recipes (see the mixed-precision section below and [5], [12]).

```
import numpy as np

for dtype in (np.float16, np.float32, np.float64):
    info = np.finfo(dtype)
    print(f"{info.dtype}: eps={float(info.eps):.3e}  max={float(info.max):.3e}  "
          f"tiny={float(info.tiny):.3e}  subnormal={float(info.smallest_subnormal):.3e}")

# Rounding: 1 + eps/2 rounds back to 1 in float32
e = np.finfo(np.float32).eps
print(np.float32(1) + np.float32(e/2) == np.float32(1))   # True
print(np.float32(1) + np.float32(e) == np.float32(1))     # False

# Addition is not associative
a, b, c = np.float32(1e8), np.float32(-1e8), np.float32(1.0)
print((a + b) + c, a + (b + c))                           # 1.0 0.0
```

The last line is worth pausing on. In exact arithmetic both orders give 1. In float32, $-10^8 + 1$ rounds back to $-10^8$ because the spacing between float32 numbers near $10^8$ is 8, so the 1 is lost. Consequences in practice: summing the same numbers in a different order — for example on a different number of GPUs, or with a different reduction kernel — can change the result in the last digits. This is a major source of run-to-run nondeterminism, and it is not a bug in your code [13].

### Overflow, underflow and cancellation

**Plain version.** Overflow means a result is too large to store; underflow means it is too small and becomes zero (or loses precision as a subnormal). Cancellation means subtracting nearly equal numbers, which keeps the rounding error but throws away the true signal.

### Overflow and underflow

The exponential is the usual culprit in machine learning. Because $\exp(z)$ exceeds the largest float32 once $z > \ln(3.40 \times 10^{38}) \approx 88.7$, a logit of 100 overflows in float32 (and a logit of about 11.1 already overflows in float16, since $\ln 65504 \approx 11.09$). In the other direction, $\exp(-110) \approx 1.7 \times 10^{-48}$ is below the smallest float32 subnormal and becomes exactly 0. Neither event raises an error by default; they produce `inf` and `0`, and the next operation turns them into `nan` or $-\infty$.

```
import numpy as np

with np.errstate(over="ignore", under="ignore", invalid="ignore"):
    print(np.exp(np.float32(100.0)))        # inf: overflow
```

```
    print(np.exp(np.float32(-110.0)))       # 0.0: underflow
    big = np.exp(np.float32(100.0))
    print(big / big)                        # nan: inf / inf

# Catastrophic cancellation and its cure
x = 1e-10
print(np.log(1 + x), np.log1p(x))           # roundoff vs accurate
print(np.exp(x) - 1, np.expm1(x))

# Variance: the one-pass formula cancels badly when the mean is large
rng = np.random.default_rng(0)
data = (1e4 + rng.normal(0, 1, 100_000)).astype(np.float32)
one_pass = np.mean(data**2) - np.mean(data)**2
two_pass = np.mean((data - data.mean())**2)
print("one-pass:", one_pass, " two-pass:", two_pass)
```

On this machine the one-pass variance comes out as $-8.0$ — a negative variance — while the two-pass formula gives the correct value of about 1.0. The data have mean $10^4$ and variance 1, so $\mathbb{E}[X^2] \approx 10^8$ and $(\mathbb{E}X)^2 \approx 10^8$; float32 can only resolve differences of about 8 at that magnitude, so the true difference of 1 is pure noise.

### Why subtraction is the dangerous operation

Suppose $\hat{a} = a(1+\delta_a)$ and $\hat{b} = b(1+\delta_b)$ are stored with relative errors $|\delta_a|, |\delta_b| \le u$. Even if the subtraction itself is exact,

$$\frac{|(\hat{a}-\hat{b}) - (a-b)|}{|a-b|} = \frac{|a\delta_a - b\delta_b|}{|a-b|} \le u\,\frac{|a|+|b|}{|a-b|}.$$

Here $a, b$ are the true values and $u$ is the unit roundoff. The factor $(|a|+|b|)/|a-b|$ is huge when $a \approx b$: the subtraction does not create error, it *reveals* the error already present in the inputs by removing the digits they shared. The cures are algebraic: rewrite the formula so the subtraction never happens. Examples you will use:

- `log1p(x)` computes $\log(1+x)$ and `expm1(x)` computes $e^x - 1$ accurately for small $x$, where the naive forms lose almost all digits (see the output above).
- Variance: subtract the mean first (two-pass), or use Welford's streaming update.
- Softmax and log-sum-exp: subtract the maximum first, as the next section shows.
- Distances: $\|x-y\|^2 = \|x\|^2 + \|y\|^2 - 2x^\top y$ is fast but cancels badly when $x \approx y$; clamp it at zero or compute $x - y$ directly when accuracy matters.

### Stable softmax and log-sum-exp

**Plain version.** Subtract a shared constant before exponentiating, and compute logarithms of probabilities directly instead of first creating tiny probabilities and then taking their logarithm.

The softmax turns a vector of logits $z \in \mathbb{R}^K$ into probabilities:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}, \qquad i = 1, \dots, K.$$

It is invariant to adding the same constant $c$ to every logit, because the factor $\exp(c)$ cancels between numerator and denominator. Choose $c = -m$ with $m = \max_i z_i$:

$$p_i = \frac{\exp(z_i - m)}{\sum_j \exp(z_j - m)}.$$

Now every exponent is $\le 0$, so nothing overflows, and the largest term is exactly $\exp(0) = 1$, so the denominator is at least 1 and cannot underflow to zero. The same trick stabilizes the **log-sum-exp** function,

$$\text{LSE}(z) = \log \sum_j \exp(z_j) = m + \log \sum_j \exp(z_j - m), \qquad \log p_i = z_i - \text{LSE}(z).$$

LSE is a smooth version of the maximum: $\max_i z_i \le \text{LSE}(z) \le \max_i z_i + \log K$. Its gradient is exactly the softmax, $\partial\,\text{LSE}/\partial z_i = p_i$.

For a target class $y$, the cross-entropy loss is $-\log p_y = \text{LSE}(z) - z_y$, and its gradient with respect to the logits is the famously simple $p - e_y$, where $e_y$ is the one-hot vector of the target. Computing `softmax` first and then `log` can still fail: if $p_y$ underflows to 0, its logarithm is $-\infty$ even though $\text{LSE}(z) - z_y$ is a perfectly ordinary finite number. This is why frameworks provide fused `log_softmax` and `cross_entropy` functions that take logits.

### Code: logits that overflow naive exponentiation

This standalone NumPy example checks stable probabilities, log probabilities and invariance to shifting logits.

```
import numpy as np

def logsumexp(z):
    m = np.max(z)
    return m + np.log(np.exp(z-m).sum())
def logsoftmax(z):
    return z-logsumexp(z)

z = np.array([1000., 1001., 999.], dtype=np.float64)
logp = logsoftmax(z)
p = np.exp(logp)
assert np.allclose(p.sum(), 1)
assert np.allclose(logsoftmax(z+12345), logp)
print("probabilities:", p)
print("cross-entropy for class 1:", -logp[1])
with np.errstate(over="ignore", invalid="ignore"):
    naive = np.exp(z) / np.exp(z).sum()
print("naive softmax:", naive)
```

The stable version prints probabilities $[0.245, 0.665, 0.090]$ and a cross-entropy of 0.408 nats for class 1; the naive version prints `[nan nan nan]`, because $\exp(1000)$ overflows even in float64 (whose limit is $\exp(709.78)$).

### Stable logistic (binary cross-entropy) loss

Binary classification has the same issue in a different form. With logit $z$, label $y \in \{0, 1\}$ and sigmoid $\sigma(z) = 1/(1 + e^{-z})$, the loss $-y \log \sigma(z) - (1 - y) \log(1 - \sigma(z))$ simplifies algebraically to

$$\ell(z, y) = \operatorname{softplus}(z) - yz, \qquad \operatorname{softplus}(z) = \log(1 + e^{z}) = \max(z, 0) + \log\bigl(1 + e^{-|z|}\bigr).$$

The right-hand form never exponentiates a positive number, and `log1p` keeps the small correction accurate. Prefer `binary_cross_entropy_with_logits` to a hand-built sigmoid followed by a logarithm.

```
import numpy as np

def bce_naive(z, y):
    s = 1 / (1 + np.exp(-z))
    return -(y*np.log(s) + (1-y)*np.log(1-s))

def bce_stable(z, y):
    # softplus(z) - y*z, with softplus(z) = max(z,0) + log1p(exp(-|z|))
    return np.maximum(z, 0) + np.log1p(np.exp(-np.abs(z))) - y*z

z = np.array([-800., -30., 0., 30., 800.])
y = np.array([1., 1., 0., 0., 0.])
with np.errstate(over="ignore", divide="ignore", invalid="ignore"):
    print("naive :", bce_naive(z, y))
print("stable:", bce_stable(z, y))
```

The naive version returns `inf` for the two confident mistakes ($z = \pm 800$) and 30.001 instead of 30.000 at $z = 30$, because $1 - \sigma(30)$ is computed by cancellation. The stable version returns 800, 30, $\log 2$, 30 and 800 — the correct values.

## Poor conditioning

**Plain version.** Some problems are inherently sensitive: a tiny change in the input produces a large change in the answer, no matter how carefully you compute. The **condition number** measures that sensitivity.

### Condition number of a linear system

Consider solving $Ax = b$ for a square invertible matrix $A$. If the data change to $b + \delta b$, the solution changes to $x + \delta x$ with

$$\frac{\|\delta x\|}{\|x\|} \le \kappa(A)\,\frac{\|\delta b\|}{\|b\|}, \qquad \kappa(A) = \|A\|\,\|A^{-1}\| = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)} \text{ (2-norm).}$$

Here $\|\cdot\|$ is the Euclidean vector norm and its induced matrix norm, and $\sigma_{\max}, \sigma_{\min}$ are the largest and smallest singular values of $A$. For a symmetric positive-definite matrix, singular values equal eigenvalues and $\kappa = \lambda_{\max}/\lambda_{\min}$. For a general invertible matrix, an eigenvalue ratio is not sufficient: use singular values, as NumPy's default `cond` does [23]. A large condition number permits strong error amplification; the actual error also depends on the perturbation's direction [4].

```
import numpy as np

A = np.array([[1.0, 1.0],
              [1.0, 1.0001]])
print("condition number:", np.linalg.cond(A))

b = np.array([2.0, 2.0001])
x = np.linalg.solve(A, b)
db = np.array([0.0, 0.0001])                # tiny change in the data
x2 = np.linalg.solve(A, b + db)
rel_in = np.linalg.norm(db) / np.linalg.norm(b)
rel_out = np.linalg.norm(x2 - x) / np.linalg.norm(x)
print("x:", x, " x after perturbation:", x2)
print(f"relative change in b: {rel_in:.1e}, in x: {rel_out:.1e}, "
      f"amplification: {rel_out/rel_in:.0f}")
```

The two columns of $A$ are almost identical, $\kappa(A) \approx 4 \times 10^4$, and a relative change of $3.5 \times 10^{-5}$ in $b$ moves the solution from $(1, 1)$ to $(0, 2)$ — a 100% change, an amplification of about 28,000, within the bound $\kappa$. Nothing was computed wrongly. The *problem* is ill-conditioned.

### Ill-conditioned problem versus unstable algorithm

Keep two ideas apart:

- An **ill-conditioned problem** amplifies small changes in its input. No algorithm can fix that; you need better data, regularization (which changes the problem) or a reformulation.
- An **unstable algorithm** adds avoidable error to a problem that may be perfectly well conditioned. The naive softmax and the one-pass variance are unstable algorithms; changing the algorithm fixes them.

Least squares shows both. If features are nearly collinear, the coefficients are ill-conditioned (the problem). Solving the normal equations $X^\top X w = X^\top y$ makes things worse than necessary because $\kappa(X^\top X) = \kappa(X)^2$ (the algorithm); a QR- or SVD-based solver works with $X$ directly. Ridge regression, which adds $\lambda\|w\|^2$, changes the problem to $(X^\top X + \lambda I)w = X^\top y$, whose condition number $(\sigma_{\max}^2 + \lambda)/(\sigma_{\min}^2 + \lambda)$ is smaller — a theme ML 01A develops further.

### Gradient-based optimization

**Plain version.** To minimize a function, look at its slope and step downhill. The gradient points uphill most steeply, so step in the opposite direction, with a step size small enough that the slope you measured is still roughly valid.

### Derivatives, gradients and directional derivatives

We minimize an **objective** (also called cost, loss or error) function $f : \mathbb{R}^n \to \mathbb{R}$. For one variable, the derivative $f'(x)$ gives the first-order approximation $f(x+\epsilon) \approx f(x) + \epsilon f'(x)$, so moving a small step against the sign of $f'(x)$ decreases $f$. For many variables, the **gradient** $\nabla_x f(x) \in \mathbb{R}^n$ collects the partial derivatives $\partial f / \partial x_i$. The **directional derivative** in a unit direction $u$ is the slope along $u$:

$$\frac{\partial}{\partial \alpha} f(x + \alpha u)\Big|_{\alpha=0} = u^\top \nabla_x f(x) = \|\nabla_x f(x)\|_2 \cos\theta,$$

where $\theta$ is the angle between $u$ and the gradient. It is most negative when $\theta = \pi$, that is, when $u$ points along $-\nabla_x f(x)$. This is the **method of steepest descent**, or **gradient descent**:

$$x \leftarrow x - \eta \nabla_x f(x),$$

with **learning rate** (step size) $\eta > 0$. You can pick $\eta$ as a small constant, try several values and keep the one with the lowest objective (a **line search**), or follow a schedule. Gradient descent converges when every element of the gradient is (close to) zero.

### Critical points: minima, maxima and saddles

Points where $\nabla_x f(x) = 0$ are **critical** or **stationary points**. A **local minimum** is lower than all neighbouring points, a **local maximum** is higher, and a **saddle point** is neither: it is a minimum along some directions and a maximum along others. A **global minimum** has the lowest value anywhere; a function can have several global minima and many local minima that are not global.

For deep networks we rarely find a true global minimum, and we do not need to. We settle for parameters with a low enough loss that generalize well. In high dimensions, saddle points rather than poor local minima are the more common critical points of many random-like functions, and they can slow gradient-based optimizers because the gradient becomes small near them [8]. Figure 1 shows the three kinds of critical point.

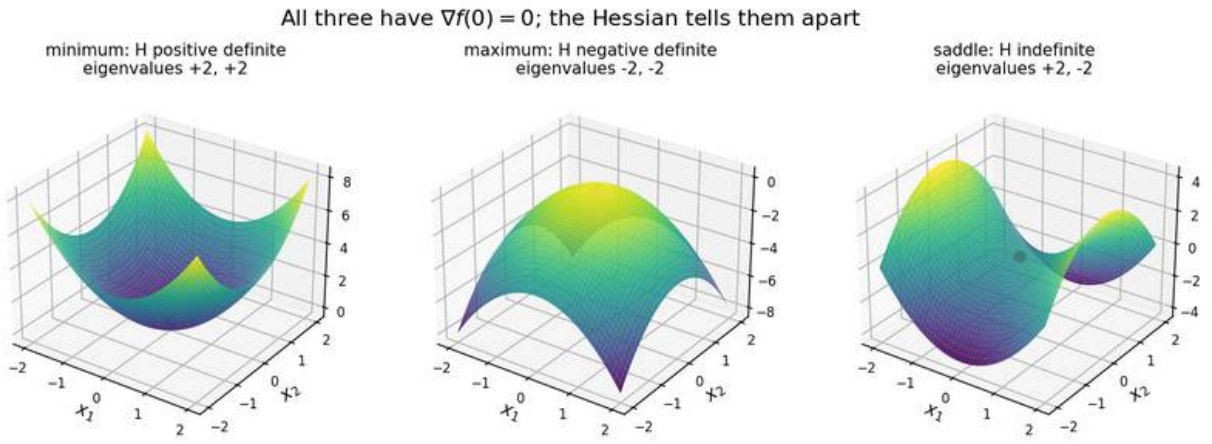



*Figure 1 — Three functions with zero gradient at the origin. The Hessian's eigenvalues distinguish them: all positive (minimum), all negative (maximum), mixed signs (saddle).*

### Jacobian and Hessian matrices

**Plain version.** The Jacobian collects all first derivatives of a vector-valued function. The Hessian collects all second derivatives of a scalar function: it tells you how the slope itself changes, which is curvature.

### Definitions

For a function $f : \mathbb{R}^m \to \mathbb{R}^n$, the **Jacobian** $J \in \mathbb{R}^{n \times m}$ has entries

$$J_{ij} = \frac{\partial f_i(x)}{\partial x_j}, \qquad i = 1, \dots, n,\ j = 1, \dots, m.$$

Row $i$ is the gradient of output $f_i$. Backpropagation is repeated multiplication by Jacobians; reverse-mode automatic differentiation computes vector–Jacobian products $v^\top J$ without ever forming $J$, which is why a gradient costs only a small constant multiple of one function evaluation [11].

For a scalar function $f : \mathbb{R}^n \to \mathbb{R}$, the **Hessian** $H(f)(x) \in \mathbb{R}^{n \times n}$ is the Jacobian of the gradient:

$$H(f)(x)_{ij} = \frac{\partial^2 f(x)}{\partial x_i \, \partial x_j}.$$

Wherever the second partial derivatives are continuous, the order of differentiation does not matter, so $H_{ij} = H_{ji}$: the Hessian is symmetric. A real symmetric matrix has real eigenvalues $\lambda_1, \dots, \lambda_n$ and an orthonormal basis of eigenvectors $v_1, \dots, v_n$. The **second derivative in a unit direction** $d$ is $d^\top H d$. If $d$ is an eigenvector $v_i$, it equals $\lambda_i$; for other directions it is a weighted average $\sum_i \lambda_i (d^\top v_i)^2$ of the eigenvalues, with weights between 0 and 1 that sum to 1. So $\lambda_{\max}$ is the largest curvature in any direction and $\lambda_{\min}$ the smallest.

### What a gradient step really does: second-order Taylor expansion

Write $g = \nabla_x f(x^{(0)})$ and $H = H(f)(x^{(0)})$. The second-order Taylor approximation around $x^{(0)}$ is

$$f(x) \approx f(x^{(0)}) + (x - x^{(0)})^\top g + \tfrac{1}{2}(x - x^{(0)})^\top H(x - x^{(0)}).$$

Substituting a gradient step $x = x^{(0)} - \eta g$ gives

$$f(x^{(0)} - \eta g) \approx f(x^{(0)}) - \eta\, g^\top g + \tfrac{1}{2}\eta^2\, g^\top H g.$$

The three terms are the current value, the improvement predicted by the slope, and a correction for curvature. If $g^\top H g \le 0$, the approximation keeps decreasing as $\eta$ grows (until the approximation itself stops being valid). If $g^\top H g > 0$, the curvature term eventually wins, and the approximation is minimized at

$$\eta^* = \frac{g^\top g}{g^\top H g}.$$

In the worst case, $g$ is aligned with the eigenvector of $\lambda_{\max}$ and $\eta^* = 1/\lambda_{\max}$. Large curvature forces small steps.

### The second derivative test

At a critical point ($g = 0$) the Taylor expansion reduces to the quadratic term, so the Hessian decides what kind of point it is:

- $H$ **positive definite** (all $\lambda_i > 0$): local minimum.
- $H$ **negative definite** (all $\lambda_i < 0$): local maximum.
- $H$ has at least one positive and at least one negative eigenvalue: **saddle point**.
- Otherwise (some $\lambda_i = 0$ and the rest of one sign) the test is inconclusive.

In one dimension a saddle cannot exist; in $n$ dimensions it needs only one eigenvalue of the "wrong" sign, which is why saddles become so common as $n$ grows [8].

### Condition number of the Hessian: why gradient descent zigzags

On the quadratic $f(w) = \frac{1}{2} w^\top H w$ with $H$ symmetric positive definite, gradient descent is linear: $w \leftarrow (I - \eta H)w$. Expanding $w$ in the eigenvectors of $H$, the component along $v_i$ is multiplied by $(1 - \eta\lambda_i)$ at every step. Therefore:

- The iteration is **stable** only if $|1 - \eta\lambda_i| < 1$ for every $i$, that is, $0 < \eta < 2/\lambda_{\max}$.
- The **slowest** direction is the flat one, contracting by $|1 - \eta\lambda_{\min}|$ per step.
- The best fixed step, $\eta = 2/(\lambda_{\max} + \lambda_{\min})$, contracts the error by $(\kappa - 1)/(\kappa + 1)$ per step, with $\kappa = \lambda_{\max}/\lambda_{\min}$ the condition number of the Hessian.

So a large condition number means one direction wants tiny steps (or it diverges) while another direction needs many steps to make progress. Gradient descent wastes its time bouncing across

the steep walls of a narrow valley — the zigzag in Figure 2 — while creeping along the valley floor. Badly scaled input features produce exactly this situation, which is one reason standardization and normalization layers speed up training.

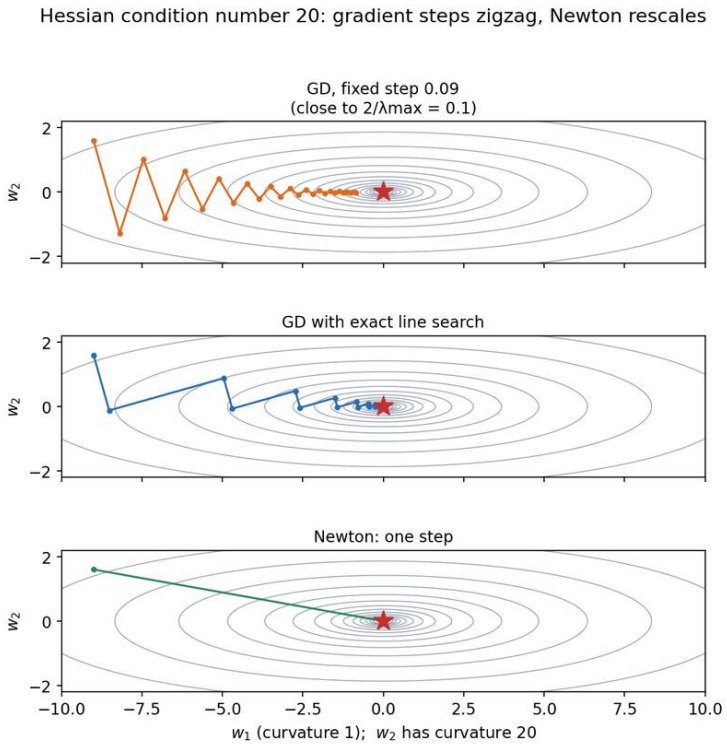



*Figure 2 — Minimizing a quadratic with Hessian condition number 20. Gradient descent with a step close to the stability limit oscillates across the steep direction; exact line search makes successive steps orthogonal; Newton's method rescales by the inverse Hessian and reaches the minimum of a quadratic in one step.*

Figure 3 shows the stability limit numerically. Below $2/\lambda_{\max}$ the loss decreases; just above it the steep component grows by a factor $|1 - \eta\lambda_{\max}| > 1$ every step and the loss explodes.

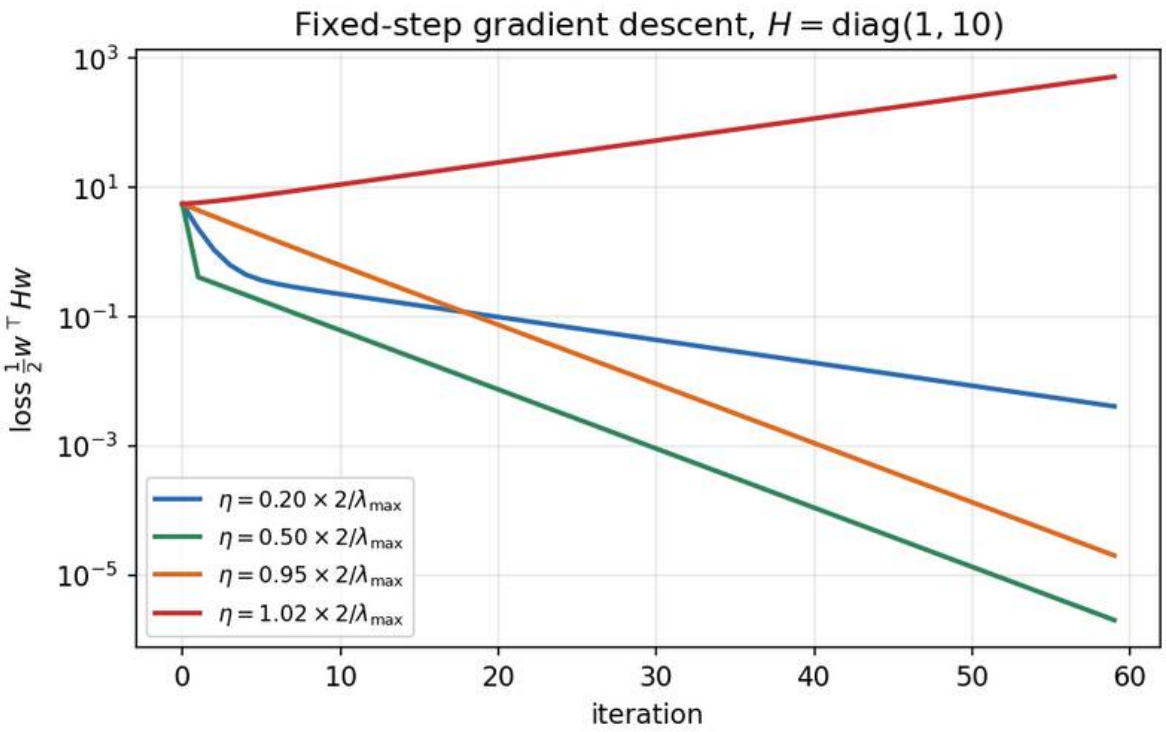



*Figure 3 — Fixed-step gradient descent on a quadratic with eigenvalues 1 and 10. Step sizes below 2/λmax converge (fastest near 2/(λmax+λmin)); a step just 2% above the limit diverges.*

```
import numpy as np

H = np.diag([1.0, 25.0])                    # Hessian: condition number 25
f = lambda w: 0.5 * w @ H @ w
grad = lambda w: H @ w

def run_gd(eta, steps=50):
    w = np.array([10.0, 1.0])
    for _ in range(steps):
        w = w - eta * grad(w)
    return f(w)

lam_max = np.linalg.eigvalsh(H).max()
for eta in (0.01, 0.07, 0.079, 0.081):       # stability limit 2/25 = 0.08
    print(f"eta={eta:<6} loss after 50 steps: {run_gd(eta):.3e}")

# Exact line search along -g for a quadratic: eps* = g'g / g'Hg
w = np.array([10.0, 1.0])
g = grad(w)
eps_star = (g @ g) / (g @ H @ g)
print("optimal step along the gradient:", round(eps_star, 4))

# One Newton step lands on the minimizer of a quadratic
w_newton = w - np.linalg.solve(H, g)
```

```
print("after one Newton step:", w_newton)
```

The output shows the whole story: $\eta = 0.01$ is stable but slow (loss 18.3 after 50 steps), $\eta = 0.07$ reaches 0.035, $\eta = 0.079$ is barely stable and the steep component still oscillates (loss about 1.0), and $\eta = 0.081$ diverges (loss 148). The optimal step along the first gradient is 0.046, and one Newton step lands exactly on $(0, 0)$.

### Second-order methods

**Plain version.** Newton's method uses curvature to rescale the gradient. Near a well-behaved minimum this can help; indefinite curvature can produce a poor direction.

### Newton's method

Setting the gradient of the second-order Taylor approximation to zero gives the **Newton step**

$$x^* = x^{(0)} - H(f)(x^{(0)})^{-1} \nabla_x f(x^{(0)}).$$

On a positive-definite quadratic this jumps to the minimum in one step (Figure 2, bottom). On a general function that is locally close to a positive-definite quadratic, iterating the update converges much faster than gradient descent — quadratically near the solution, under standard assumptions [6]. The cost is the Hessian: storing it takes $O(n^2)$ memory and solving with it $O(n^3)$ time, impossible for a network with $10^8$ parameters.

There is also a more fundamental problem for non-convex objectives. Newton's method finds a point where the *gradient* vanishes, and it treats a saddle the same way as a minimum: near a saddle, the Newton step points *toward* it. In a landscape full of saddles, plain Newton is attracted to exactly the points we want to escape. Dauphin et al. proposed a "saddle-free Newton" method that rescales by the absolute values of the Hessian eigenvalues to avoid this [8]. Gradient descent, by contrast, is not attracted to saddles in the same way, although it can move slowly near them.

### Quasi-Newton methods and L-BFGS

**Quasi-Newton** methods build an approximation $B_k \approx H$ (or of its inverse) from gradients alone. After a step $s_k = x_{k+1} - x_k$ with gradient change $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$, they require the **secant condition** $B_{k+1} s_k = y_k$, which says the model's curvature along the step matches the observed change in slope. BFGS is the most widely used update of this kind; **L-BFGS** (limited-memory BFGS) stores only the last $m$ pairs $(s_k, y_k)$ and applies the inverse-Hessian approximation implicitly in $O(mn)$ time [6], [10]. L-BFGS is excellent for smooth, deterministic, medium-sized problems (logistic regression, style transfer on one image, fitting a few thousand parameters); its curvature estimates suffer when gradients are noisy mini-batch estimates.

### Hessian-vector products

Many curvature-aware methods never need $H$ itself, only products $Hv$. These can be computed exactly at roughly the cost of a couple of gradient evaluations, because $Hv = \nabla_x(\nabla_x f(x)^\top v)$ is the gradient of a scalar; Pearlmutter showed how to do this with automatic differentiation [9]. A finite-difference version, $Hv \approx [\nabla f(x + rv) - \nabla f(x - rv)]/(2r)$, is a useful check. Power iteration with Hessian-vector products estimates $\lambda_{\max}$, which is how researchers track the sharpness of the loss during training.

```
import numpy as np

rng = np.random.default_rng(0)
X = rng.normal(size=(50, 4))
y = (rng.random(50) < 0.5).astype(float)

def grad(w):                                    # logistic regression, mean loss
    p = 1 / (1 + np.exp(-(X @ w)))
    return X.T @ (p - y) / len(y)

def hessian(w):
    p = 1 / (1 + np.exp(-(X @ w)))
    return (X.T * (p * (1 - p))) @ X / len(y)

w = rng.normal(size=4)
v = rng.normal(size=4)
r = 1e-5
hvp_fd = (grad(w + r*v) - grad(w - r*v)) / (2*r)   # never forms H
print("Hv (finite diff):", hvp_fd.round(6))
print("Hv (explicit)   :", (hessian(w) @ v).round(6))
print("eigenvalues of H:", np.linalg.eigvalsh(hessian(w)).round(4))
```

Both products agree to six decimals, and all Hessian eigenvalues are positive: logistic regression is convex, so its Hessian is positive semidefinite everywhere.

### Lipschitz continuity and convexity

Deep-learning objectives come with few guarantees, but two properties explain much of the theory. A function is **Lipschitz continuous** with constant $L$ if $|f(x) - f(y)| \le L\|x - y\|_2$ for all $x, y$: its rate of change is bounded. When the *gradient* is $L$-Lipschitz (the function is "$L$-smooth"), the curvature is bounded by $L$, and a gradient step with $\eta \le 1/L$ is guaranteed not to increase the function:

$$f(x - \eta \nabla f(x)) \le f(x) - \tfrac{\eta}{2}\|\nabla f(x)\|_2^2 \qquad (0 < \eta \le 1/L).$$

This is the general form of the "$\eta < 2/\lambda_{\max}$" rule above, and it is the starting point of most convergence proofs [7], [14].

A function is **convex** if its Hessian is positive semidefinite everywhere (for twice-differentiable functions). Convex functions have no saddles and no spurious local minima: every local minimum is global. Least squares, logistic regression and support vector machines are convex; deep networks are not. Convex optimization is the source of many tools, such as duality and the KKT conditions below, that deep learning borrows [7].

### Constrained optimization

**Plain version.** Constraints restrict the allowed solutions. Projection returns an update to the feasible region; Lagrange multipliers describe how constraints enter the optimality conditions.

### Projected gradient descent

We want to minimize $f(x)$ over a feasible set $\mathbb{S} \subseteq \mathbb{R}^n$. When projecting onto $\mathbb{S}$ is easy, take a normal gradient step and then return to the nearest feasible point:

$$x \leftarrow \text{Proj}_{\mathbb{S}}\left(x - \eta \nabla f(x)\right), \qquad \text{Proj}_{\mathbb{S}}(v) = \arg\min_{x \in \mathbb{S}} \|x - v\|_2.$$

For the Euclidean ball $\{x : \|x\|_2 \leq r\}$ the projection is $v \cdot \min(1, r/\|v\|_2)$: rescale if too long. Max-norm constraints on weights, clipping parameters into a box, and projecting onto the probability simplex all follow this pattern. Another option is to reparameterize so the constraint holds automatically, for example $x = r\,(\cos\theta, \sin\theta)$ for a vector on a circle, or a softmax to produce probabilities.

### The generalized Lagrangian

A more general approach describes the feasible set with $m$ equality constraints $g^{(i)}(x) = 0$ and $k$ inequality constraints $h^{(j)}(x) \leq 0$:

$$\mathbb{S} = \{x \mid g^{(i)}(x) = 0 \ \forall i,\ h^{(j)}(x) \leq 0 \ \forall j\}.$$

Introduce one **KKT multiplier** for each constraint, $\lambda_i$ for equalities and $\alpha_j$ for inequalities, and define the **generalized Lagrangian**

$$L(x, \lambda, \alpha) = f(x) + \sum_{i=1}^{m} \lambda_i g^{(i)}(x) + \sum_{j=1}^{k} \alpha_j h^{(j)}(x).$$

The constrained problem is equivalent to the unconstrained min–max problem

$$\min_{x} \max_{\lambda} \max_{\alpha,\ \alpha \geq 0} L(x, \lambda, \alpha).$$

Why does this work? If $x$ violates a constraint, the inner maximization can drive $L$ to $+\infty$ by choosing a large multiplier, so infeasible points are never chosen. If $x$ is feasible, the best the inner maximization can do is set the terms to zero (equalities are already 0; for $h^{(j)}(x) \leq 0$ the best non-negative $\alpha_j$ makes $\alpha_j h^{(j)}(x) = 0$), so the value is exactly $f(x)$ [1].

### The KKT conditions

At a solution $x^*$ (under mild regularity conditions called constraint qualifications), there exist multipliers satisfying the **Karush–Kuhn–Tucker (KKT) conditions** [1], [7]:

1. **Stationarity:** $\nabla_x L(x^*, \lambda^*, \alpha^*) = 0$.
2. **Primal feasibility:** all constraints hold at $x^*$.
3. **Dual feasibility:** $\alpha^* \geq 0$.
4. **Complementary slackness:** $\alpha_j^* \, h^{(j)}(x^*) = 0$ for every $j$.

Complementary slackness says each inequality is either **active** ($h^{(j)}(x^*) = 0$: the solution sits on the boundary and the constraint pushes back with $\alpha_j \geq 0$) or **inactive** ($h^{(j)}(x^*) < 0$: the constraint does not matter locally and $\alpha_j = 0$). Geometrically, stationarity says that at an active constraint the downhill direction $-\nabla f$ points straight out of the feasible set, parallel to $\nabla h$; any feasible move would go uphill (Figure 4). For convex problems the KKT conditions are also sufficient for optimality [7].

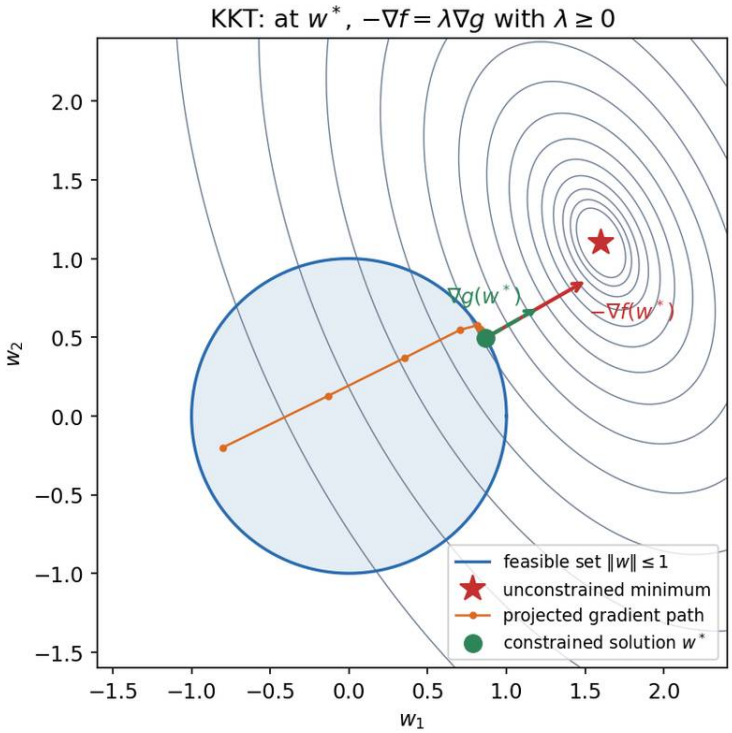



*Figure 4 — Minimizing a quadratic over the unit disk. The unconstrained minimum (star) is infeasible. Projected gradient descent stops on the boundary at w*, where $-\nabla f$ is parallel to the outward normal $\nabla g$ of the constraint: the KKT stationarity condition with a positive multiplier.*

### Worked example: least squares with a norm constraint

Following the example in [1], minimize the squared error $f(w) = \|Aw - b\|_2^2$, with $A \in \mathbb{R}^{N \times n}$ and $b \in \mathbb{R}^N$, subject to $h(w) = w^\top w - r^2 \le 0$. The Lagrangian is

$$L(w, \alpha) = \|Aw - b\|_2^2 + \alpha\,(w^\top w - r^2).$$

Stationarity gives $2A^\top(Aw - b) + 2\alpha w = 0$, so

$$w(\alpha) = (A^\top A + \alpha I)^{-1} A^\top b.$$

This is ridge regression with penalty $\alpha$. If the ordinary least-squares solution $w(0)$ already satisfies $\|w\| \le r$, the constraint is inactive, $\alpha^* = 0$, and we are done. Otherwise the constraint is active, and we increase $\alpha$ until $\|w(\alpha)\| = r$; because $\|w(\alpha)\|$ decreases monotonically in $\alpha$, a bisection finds it. This is also how the inner maximization over $\alpha$ behaves: the multiplier grows while the constraint is violated and stops when it is just satisfied. The code solves the problem both ways and checks all four KKT conditions.

```
import numpy as np

# minimize ||Aw - b||^2  subject to  ||w||^2 <= r^2
rng = np.random.default_rng(1)
A = rng.normal(size=(30, 5))
b = rng.normal(size=30)
r = 0.2

def w_of(lam):                                  # stationarity of the Lagrangian
    return np.linalg.solve(A.T @ A + lam * np.eye(5), A.T @ b)

w_ls = w_of(0.0)
print("unconstrained norm:", np.linalg.norm(w_ls).round(3))

lo, hi = 0.0, 1e6                               # find lambda with ||w|| = r
for _ in range(200):
    mid = 0.5 * (lo + hi)
    if np.linalg.norm(w_of(mid)) > r:
        lo = mid
    else:
        hi = mid
lam = hi
w = w_of(lam)
print("lambda:", round(lam, 3), " ||w||:", np.linalg.norm(w).round(6))

# KKT checks
stationarity = 2 * A.T @ (A @ w - b) + 2 * lam * w
print("stationarity residual:", np.abs(stationarity).max())
print("primal feasible:", w @ w <= r**2 + 1e-9, " dual feasible:", lam >= 0)
print("complementary slackness:", lam * (w @ w - r**2))

# Projected gradient descent reaches the same point
L = 2 * np.linalg.eigvalsh(A.T @ A).max()     # Lipschitz constant of the gradient
v = np.zeros(5)
for _ in range(5000):
```

```
    v = v - (1 / L) * 2 * A.T @ (A @ v - b)
    n = np.linalg.norm(v)
    if n > r:
        v = v * (r / n)                       # projection onto the ball
print("projected GD matches KKT solution:", np.allclose(v, w, atol=1e-6))
```

The unconstrained solution has norm 0.219, larger than $r = 0.2$, so the constraint is active. Bisection finds $\alpha^* \approx 1.81$, the stationarity residual is about $10^{-15}$, complementary slackness holds, and projected gradient descent with step $1/L$ converges to the same vector. Note how $L$, the Lipschitz constant of the gradient, sets a safe step size, exactly as in the previous section.

### Checking derivatives

**Plain version.** Check that the gradient your code computes matches how the loss actually changes when you nudge the parameters.

A central finite difference estimates a partial derivative,

$$\frac{\partial f}{\partial x_j} \approx \frac{f(x + \epsilon e_j) - f(x - \epsilon e_j)}{2\epsilon},$$

where $e_j$ is the $j$-th standard basis vector and $\epsilon > 0$ a small step. Its **truncation error** (from ignoring higher-order Taylor terms) is $O(\epsilon^2)$, better than the $O(\epsilon)$ of the forward difference $[f(x + \epsilon e_j) - f(x)]/\epsilon$. But the numerator subtracts two nearly equal numbers, so its **roundoff error** grows like $u\,|f|/\epsilon$. The total error is smallest at an intermediate step — roughly $\epsilon \approx u^{1/3}$ for central differences and $u^{1/2}$ for forward differences, when $f$ and its derivatives have unit scale. Figure 5 shows the resulting V-shaped curves, and why float32 gradient checks are much less precise than float64 ones.

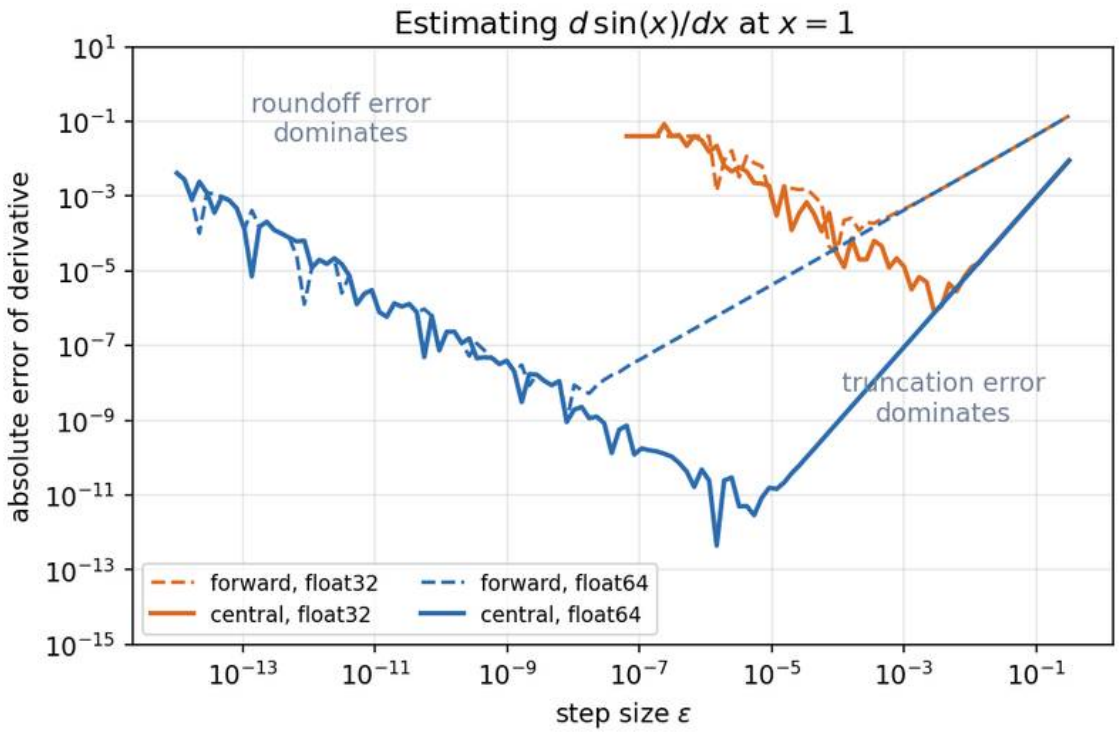



*Figure 5 — Error of finite-difference derivatives of $\sin(x)$ at x=1. Right of each minimum, truncation error dominates (slope 1 for forward, 2 for central differences); left of it, roundoff dominates. Central differences bottom out around 1e−6 in float32 but around 1e−11 or lower in float64.*

Practical rules: use float64 for checks; try several step sizes; compare with a relative error such as $|a-n|/\max(|a|,|n|,\text{tiny})$; and test at smooth points. ReLU exactly at zero is not differentiable, so there is no single derivative to match there. For many parameters, a **directional check** is cheaper: choose a random unit vector $v$ and compare $\nabla f(w)^\top v$ with a finite difference along $v$ — one pair of function evaluations instead of $2n$.

```
import numpy as np

rng = np.random.default_rng(0)
X = rng.normal(size=(20, 5))
y = rng.normal(size=20)
w = rng.normal(size=5)
v = rng.normal(size=5)
v /= np.linalg.norm(v)
def loss(w):
    return .5*np.mean((X@w-y)**2)
grad = X.T@(X@w-y)/len(y)
eps = 1e-5
numeric = (loss(w+eps*v)-loss(w-eps*v))/(2*eps)
analytic = grad@v
assert np.allclose(numeric, analytic, rtol=1e-6, atol=1e-8)
print("directional derivative:", numeric, analytic)
```

The two numbers agree to about nine significant digits. A passing check confirms a derivative, not the intended task. A wrong reduction, accidental broadcasting or target leakage can define a differentiable but incorrect objective whose gradient passes [11]. An accidental `detach` is different: if the forward value still changes with an input but autograd cuts that dependency, finite differences can expose the mismatch [24]. Test the relevant inputs at smooth points, hold randomness fixed, and also check loss values on cases with known answers.

### Mixed precision

**Plain version.** Doing most arithmetic in 16-bit numbers halves memory traffic and uses fast hardware units, but 16 bits are not enough for everything. Keep a few sensitive quantities in 32 bits.

Mixed-precision training, as introduced by Micikevicius et al. [12], combines three ideas:

1. Keep a **float32 master copy of the weights**. An update smaller than about $2^{-11}$ times the weight it is added to rounds away entirely in float16 (the same effect as $1 + \varepsilon_{\text{mach}}/2 = 1$ above), and many updates are that small.
2. Use **loss scaling** for float16: multiply the loss by a factor $S$ before backpropagation so small gradients do not underflow below float16's smallest subnormal ($6 \times 10^{-8}$), then divide the gradients by $S$ before the update. Dynamic loss scaling reduces $S$ when an overflow (inf or NaN) appears and skips that step.
3. **Accumulate** dot products and reductions in float32 even when inputs are float16.

**bfloat16** has the same 8-bit exponent as float32, so it rarely needs loss scaling; Kalamkar et al. found bfloat16 training reached float32 results across a range of models without changing hyperparameters [5]. The price is precision: with $\varepsilon_{\text{mach}} \approx 0.0078$, bfloat16 cannot represent $1 + 0.003$ distinctly from 1, so long sums and weight updates still need float32 accumulation.

Loss scaling protects small float16 gradients; it cannot repair an invalid objective or an overflow elsewhere in the network (an attention logit or a normalization statistic, for example). Changing precision can also change optimization trajectories and outputs, especially for reductions and long training runs. Validate on the target hardware; a smaller dtype does not guarantee that every operation becomes faster [13].

*Online only: Mixed Precision Training — where precision matters — https://blog.shyandram.dev/en/learn/ml/numerical-computation/papers/mixed-precision-training–where-precision-matters/*

### A debugging order for non-finite values

When a loss becomes `nan` or `inf`, resist the urge to clip everything. Find the *first* operation that produced a non-finite value, then ask which of this chapter's failure modes it is. A useful order:

1. **Inputs:** check shapes, ranges, dtypes and missing values. A single NaN pixel or an un-normalized 0–255 image changes everything downstream.
2. **Forward pass:** check that activations and the loss are finite. Look for `exp`, `log`, division, `sqrt` and normalization layers with near-zero variance.

3. **Loss:** use logits-based losses (`cross_entropy`, `binary_cross_entropy_with_logits`) rather than probabilities followed by `log`.
4. **Gradients:** run a gradient check on a tiny model; inspect per-layer gradient norms.
5. **Updates:** compare the update size with the weight size; a learning rate above the curvature limit $2/\lambda_{\max}$ causes oscillation and divergence even with perfect arithmetic.
6. **Precision:** reproduce in float32 (or float64) before adding mixed precision or compilation.

Overfit a tiny batch first to establish that the pipeline can learn. Gradient-norm clipping bounds the gradient passed to the optimizer; it does not directly bound an adaptive optimizer's update, and it does not make NaN gradients meaningful. ML 02 develops optimizers and normalization in more detail.

### Modern view

The core ideas of this chapter are decades old, but they are at the centre of current research because models are larger, training runs are longer, and hardware keeps pushing precision downward.

**Optimization theory for large-scale learning.** Bottou, Curtis and Nocedal's SIAM Review survey [14] is the standard map of the field. It explains why stochastic gradient methods dominate large-scale machine learning (cheap noisy steps beat expensive exact ones when data are redundant), analyses their convergence under smoothness assumptions like the Lipschitz condition above, and organizes improvements into two families: *noise reduction* (larger or growing batches, variance reduction, iterate averaging) and *second-order* methods (Hessian-free Newton, Gauss–Newton, quasi-Newton, natural gradient and diagonal scaling). Nocedal and Wright [6] and Boyd and Vandenberghe [7] remain the reference textbooks for the deterministic and convex theory, and Baydin et al. [11] survey the automatic differentiation that makes gradients cheap.

**What the loss surface actually looks like.** Dauphin et al. [8] argued that saddle points, not poor local minima, are the main obstacle in high-dimensional non-convex optimization. Later, Cohen et al. [15] observed that full-batch gradient descent on neural networks typically operates at the "edge of stability": the largest Hessian eigenvalue rises until it hovers just above $2/\eta$ — precisely the classical stability limit derived above — while the loss keeps decreasing over long time scales, though not monotonically. The classical quadratic analysis predicts divergence there, so explaining this regime remains an open theoretical problem.

**Curvature-aware optimizers.** Full Newton is impossible at scale, but structured approximations are practical. K-FAC [16] approximates the Fisher information matrix (a positive-semidefinite curvature matrix closely related to the Gauss–Newton approximation of the Hessian) with Kronecker products of small per-layer matrices, so inverting it is cheap. Shampoo [17] keeps one preconditioning matrix per tensor dimension. More recently, Muon applies an approximate orthogonalization to each weight matrix's update; Liu et al. [18] report that, with suitable weight decay and per-parameter update scaling, it trains large language models with roughly twice the computational efficiency of their adaptive-optimizer baseline in their experiments. These methods all revisit the question this chapter asks with the Hessian: how should a step be rescaled direction by direction?

**Lower precision.** The format story continues from float16 with loss scaling [12] and bfloat16 [5] to 8-bit floating point. Micikevicius et al. [19] proposed two FP8 encodings, E4M3 (more precision) and E5M2 (more range), and showed training results comparable to 16-bit baselines across image and language models. The range–precision trade-off in the table above is exactly what these designs balance, and the IEEE 754 rules [3] are what they relax. Per-tensor or per-block scaling factors play the role that loss scaling played for float16.

**Stable softmax at scale.** The max-subtraction trick has a streaming version. Milakov and Gimelshein [20] showed that a softmax normalizer can be computed in one pass by keeping a running maximum and rescaling the running sum whenever the maximum changes. FlashAttention [21] builds on this idea: it computes exact attention block by block in fast on-chip memory, never materializing the full attention matrix, and FlashAttention-3 [22] extends it to FP8 with block quantization and "incoherent processing" to reduce quantization error. The core of the online trick fits in a few lines:

```
import numpy as np

def online_logsumexp(chunks):
    m, s = -np.inf, 0.0                 # running max and rescaled running sum
    for c in chunks:
        m_new = max(m, c.max())
        s = s * np.exp(m - m_new) + np.exp(c - m_new).sum()
        m = m_new
    return m + np.log(s)

rng = np.random.default_rng(0)
z = rng.normal(0, 50, size=10_000)
chunks = np.array_split(z, 37)          # seen one block at a time
m = z.max()
reference = m + np.log(np.exp(z - m).sum())
print(online_logsumexp(chunks), reference)
assert np.isclose(online_logsumexp(chunks), reference)
```

Both values print as 174.0918752579 ..., identical to the last digit shown, although the online version never sees all logits at once. ML 05 covers attention itself, and ML 11 covers quantization for deployment.

**Open problems.** Why gradient methods succeed at the edge of stability; how to make low-precision (8-bit and below) training as robust as float32 without hand-tuned scaling; how to make curvature-aware optimizers cheap and robust enough to replace first-order defaults; and how to obtain reproducible results when parallel reductions change summation order.

### Key takeaways

- Exponent bits set range and fraction bits set precision; subnormal numbers have different relative-precision behavior.
- Use stable formulas such as log-sum-exp and logits-based losses before increasing precision.

- Conditioning measures input sensitivity; a stable algorithm cannot eliminate an ill-conditioned problem's sensitivity.
- Curvature controls safe step sizes for quadratic objectives; general nonconvex losses require additional care.
- Projection enforces feasibility; KKT conditions require regularity assumptions and are not a universal certificate of global optimality.
- Check gradients at smooth points with suitable precision and step sizes; validate mixed precision on the actual computation.

## Exercises

1. **Softmax shift.** Prove that softmax is invariant under $z_i \mapsto z_i + c$, and show that with $c = -\max_i z_i$ the denominator lies in $[1, K]$.

   **Hint**

   The shared factor $\exp(c)$ cancels between numerator and denominator. After the shift, every term $\exp(z_j - m)$ is in $(0, 1]$ and the largest equals 1, so the sum of $K$ terms is between 1 and $K$. It can therefore neither overflow nor underflow to 0.

2. **Finite-difference sweep.** For $f(x) = e^x$ at $x = 0$, compute forward and central differences for $\epsilon = 10^{-1}, \ldots, 10^{-12}$ in float32 and float64. At which $\epsilon$ is each error smallest? Compare with $u^{1/2}$ and $u^{1/3}$.

   **Hint**

   For float64, $u \approx 1.1 \times 10^{-16}$, so $u^{1/2} \approx 10^{-8}$ (forward) and $u^{1/3} \approx 5 \times 10^{-6}$ (central). For float32, $u \approx 6 \times 10^{-8}$, giving about $2 \times 10^{-4}$ and $4 \times 10^{-3}$. The observed minima should be within an order of magnitude of these, and the error at the minimum is roughly $u^{1/2}$ and $u^{2/3}$ respectively.

3. **Stability limit.** For $f(w) = \frac{1}{2} w^\top H w$ with $H = \mathrm{diag}(2, 50)$, find the largest stable fixed learning rate and the step size that minimizes the worst-case contraction factor. How many iterations does that best step need to reduce the error by a factor of $10^6$?

   **Hint**

   Stability requires $\eta < 2/50 = 0.04$. The best step is $2/(\lambda_{\max} + \lambda_{\min}) = 2/52 \approx 0.0385$, with contraction $(\kappa - 1)/(\kappa + 1) = 24/26 \approx 0.923$. Solve $0.923^k \leq 10^{-6}$: $k \geq \ln 10^6 / (-\ln 0.923) \approx 173$ iterations. A Newton step would need one.

4. **Classify critical points.** Find and classify the critical points of $f(x, y) = x^3 - 3x + y^2$.

   **Hint**

   $\nabla f = (3x^2 - 3,\ 2y) = 0$ gives $(1, 0)$ and $(-1, 0)$. The Hessian is $\mathrm{diag}(6x,\ 2)$: at $(1, 0)$ it is $\mathrm{diag}(6, 2)$, positive definite, a local minimum; at $(-1, 0)$ it is $\mathrm{diag}(-6, 2)$, indefinite, a saddle. Neither is a global minimum because $f \to -\infty$ as $x \to -\infty$. Starting Newton's method near $(-1, 0)$ converges to the saddle.

5. **KKT on the simplex.** Minimize $f(x) = \frac{1}{2}\|x - c\|_2^2$ subject to $\sum_i x_i = 1$ and $x_i \geq 0$, for $c = (0.8, 0.6, -0.4)$. Write the Lagrangian and the KKT conditions, and find the solution.

   **Hint**

   $L = \frac{1}{2}\|x - c\|^2 + \lambda(\sum_i x_i - 1) - \sum_i \alpha_i x_i$ with $\alpha_i \geq 0$. Stationarity gives $x_i = c_i - \lambda + \alpha_i$; complementary slackness gives $x_i = \max(c_i - \lambda, 0)$. Guess that only the first two are positive: $(0.8 - \lambda) + (0.6 - \lambda) = 1$, so $\lambda = 0.2$ and $x = (0.6, 0.4, 0)$. Check: $c_3 - \lambda = -0.6 < 0$, so $x_3 = 0$ with $\alpha_3 = 0.6 \geq 0$. This is the Euclidean projection onto the probability simplex.

6. **A correct gradient of the wrong loss.** Change a mean reduction into a sum in the gradient-check example. Does the check still pass? Is autograd wrong? What else changes in training?

   **Hint**

   If you change both the loss and the analytic gradient consistently, the check passes: the derivative is correct for the new objective. Nothing is wrong with autograd. But the gradient is now $N$ times larger, so the largest stable learning rate shrinks by $N$, and in a multi-term loss the balance between terms changes. A gradient check cannot tell you that you chose the wrong objective.

## References


1. I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016, ch. 4 (Numerical Computation) and ch. 8. Book
2. D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1991. DOI
3. IEEE, "IEEE Standard for Floating-Point Arithmetic," IEEE Std 754-2019, 2019. DOI
4. N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, 2002. DOI
5. D. Kalamkar et al., "A Study of BFLOAT16 for Deep Learning Training," arXiv:1905.12322, 2019. arXiv
6. J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed., Springer, 2006. DOI
7. S. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge University Press, 2004. Book
8. Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli and Y. Bengio, "Identifying and Attacking the Saddle Point Problem in High-Dimensional Non-Convex Optimization," *NeurIPS*, 2014. Paper
9. B. A. Pearlmutter, "Fast Exact Multiplication by the Hessian," *Neural Computation*, vol. 6, no. 1, pp. 147–160, 1994. DOI
10. D. C. Liu and J. Nocedal, "On the Limited Memory BFGS Method for Large Scale Optimization," *Mathematical Programming*, vol. 45, pp. 503–528, 1989. DOI
11. A. G. Baydin, B. A. Pearlmutter, A. A. Radul and J. M. Siskind, "Automatic Differentiation in Machine Learning: a Survey," *Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018. arXiv
12. P. Micikevicius et al., "Mixed Precision Training," *ICLR*, 2018. arXiv
13. PyTorch documentation, "Numerical Accuracy." Documentation

14. L. Bottou, F. E. Curtis and J. Nocedal, “Optimization Methods for Large-Scale Machine Learning,” *SIAM Review*, vol. 60, no. 2, pp. 223–311, 2018. DOI
15. J. M. Cohen, S. Kaur, Y. Li, J. Z. Kolter and A. Talwalkar, “Gradient Descent on Neural Networks Typically Occurs at the Edge of Stability,” *ICLR*, 2021. arXiv
16. J. Martens and R. Grosse, “Optimizing Neural Networks with Kronecker-factored Approximate Curvature,” *ICML*, 2015. Paper
17. V. Gupta, T. Koren and Y. Singer, “Shampoo: Preconditioned Stochastic Tensor Optimization,” *ICML*, 2018. Paper
18. J. Liu, J. Su, X. Yao et al., “Muon is Scalable for LLM Training,” arXiv:2502.16982, 2025. arXiv
19. P. Micikevicius et al., “FP8 Formats for Deep Learning,” arXiv:2209.05433, 2022. arXiv
20. M. Milakov and N. Gimelshein, “Online Normalizer Calculation for Softmax,” arXiv:1805.02867, 2018. arXiv
21. T. Dao, D. Y. Fu, S. Ermon, A. Rudra and C. Ré, “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness,” *NeurIPS*, 2022. Paper
22. J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani and T. Dao, “FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision,” *NeurIPS*, 2024. Paper
23. NumPy developers, “numpy.linalg.cond,” NumPy documentation, accessed 2026-11-05. Documentation
24. PyTorch contributors, “torch.autograd.gradcheck.gradcheck,” PyTorch documentation, accessed 2026-11-05. Documentation

# ML 02 • Neural Networks from Perceptron to Deep Networks

From a single perceptron to deep networks: activations, normalization, output layers and losses, backpropagation, optimizers, and the habits that make a network train and generalize.

Online edition: https://blog.shyandram.dev/en/learn/ml/neural-networks/

## What you'll learn

- Distinguish perceptrons, logistic regression and multilayer networks.
- Explain the roles of nonlinear activations and normalization.
- Match output layers and losses to prediction tasks.
- Derive backpropagation and check gradients numerically.
- Diagnose training with initialization, optimization and regularization choices.

## The big picture

A neural network is a function with many adjustable knobs (its **parameters**). It takes an input, such as the pixels of an image or two coordinates of a point, and produces an output, such as a class label. Training means turning the knobs, a tiny amount at a time, so the outputs on known examples get closer to the right answers. Everything in this tutorial is about one of three questions: what shape the function has (the architecture), how to measure "closer" (the loss), and how to turn the knobs well (the optimization).

> **Explain like I'm five**
>
> Think of a team of people passing notes down a line. The first person looks at a picture and writes a few simple observations ("there is a curve here", "this part is bright"). The next person reads those notes and writes slightly smarter ones ("this looks like an eye"). The last person says "cat" or "dog". When the answer is wrong, a teacher walks back up the line telling each person how much their note was to blame, and each one changes their habits a little. That walk backward is **backpropagation**, and doing it many thousands of times is **training**.

Why it matters: almost every modern system in vision, speech and language is built from the pieces in this tutorial. Convolutional networks, Transformers and diffusion models all use the same ingredients: linear layers, nonlinear activations, normalization, residual connections, a loss, backpropagation and an adaptive optimizer. Once these are clear, the deep-learning track is about arranging them for images.

**About this tutorial**

This tutorial is expanded from the author's earlier study notes on machine learning and neural networks [1], which are written in Traditional Chinese: 機器學習及類神經網路筆記. It assumes the ideas of ML 01: data, features, a loss, and gradient descent.

## The perceptron

**Plain version.** A perceptron is the simplest artificial neuron. It multiplies each input by a weight, adds them up with a bias, and answers "yes" if the total is positive and "no" otherwise. It learns by nudging its weights only when it gets an example wrong.

### The model

Rosenblatt introduced the perceptron in 1958 as a model of how a brain might store and recognize patterns [4]. For an input vector $\mathbf{x} \in \mathbb{R}^d$ it computes

$$z = \mathbf{w}^\top \mathbf{x} + b, \qquad \hat{y} = \mathbf{1}[z > 0],$$

where $\mathbf{w} \in \mathbb{R}^d$ is the weight vector, $b \in \mathbb{R}$ is the bias, $z$ is the **pre-activation** (a weighted sum), $\mathbf{1}[\cdot]$ is the indicator that equals 1 when its condition is true and 0 otherwise, and $\hat{y} \in \{0, 1\}$ is the predicted class. The indicator is the **step function**. Geometrically, $\mathbf{w}^\top \mathbf{x} + b = 0$ is a hyperplane (a line in 2-D), and the perceptron labels points by which side of it they fall on.

### The perceptron learning rule

Go through the training examples $(\mathbf{x}_i, y_i)$ one at a time. For each one, compute $\hat{y}_i$ and update

$$\mathbf{w} \leftarrow \mathbf{w} + \eta\,(y_i - \hat{y}_i)\,\mathbf{x}_i, \qquad b \leftarrow b + \eta\,(y_i - \hat{y}_i),$$

where $\eta > 0$ is the **learning rate** (step size) and $y_i \in \{0, 1\}$ is the true label. If the prediction is right, $y_i - \hat{y}_i = 0$ and nothing changes. If the perceptron said 0 but the answer was 1, the weights move *toward* $\mathbf{x}_i$, which raises $z$ for this input next time; the opposite mistake moves them away. When the two classes can be separated by a hyperplane (the data are **linearly separable**), this rule stops making mistakes after a finite number of updates. When they cannot, it never settles.

```
import numpy as np

def perceptron_train(X, y, epochs=20, eta=1.0):
    """X: (N, d), y in {0, 1}. Returns w, b."""
    w, b = np.zeros(X.shape[1]), 0.0
    for _ in range(epochs):
        mistakes = 0
        for x_i, y_i in zip(X, y):
            y_hat = float(w @ x_i + b > 0)       # step function
            if y_hat != y_i:                     # update only on a mistake
```

```
                w += eta * (y_i - y_hat) * x_i
                b += eta * (y_i - y_hat)
                mistakes += 1
        if mistakes == 0:
            break
    return w, b

# AND is linearly separable; XOR is not
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=float)
for name, y in [("AND", [0, 0, 0, 1]), ("XOR", [0, 1, 1, 0])]:
    w, b = perceptron_train(X, np.array(y, float))
    print(name, (X @ w + b > 0).astype(int))
# AND [0 0 0 1]
# XOR [1 1 0 0]   <- wrong: no single line can do it
```

### Perceptron versus logistic regression

The two models are easy to confuse because they share the same first step, $z = \mathbf{w}^\top \mathbf{x} + b$. They are different methods:

| | Perceptron | Logistic regression |
|---|---|---|
| Output function | step: $\hat{y} = \mathbf{1}[z > 0]$ | sigmoid: $\hat{p} = \sigma(z) = \dfrac{1}{1+e^{-z}}$ |
| Output meaning | a hard 0/1 decision | a probability $P(y = 1 \mid \mathbf{x})$ |
| Training | perceptron rule, updates only on mistakes | minimize cross-entropy (maximum likelihood) with gradient descent |
| Non-separable data | never converges | converges to a well-defined best fit |

The gradient of the logistic-regression loss for one example is $(\hat{p}_i - y_i)\,\mathbf{x}_i$, so its gradient-descent step is $\mathbf{w} \leftarrow \mathbf{w} + \eta\,(y_i - \hat{p}_i)\,\mathbf{x}_i$. It looks almost identical to the perceptron rule, but $\hat{p}_i$ is a probability between 0 and 1, so *every* example contributes, and confidently correct ones contribute very little. That smoothness is what makes logistic regression trainable by gradient descent and what lets it be stacked into larger differentiable networks. The step function, in contrast, has derivative zero almost everywhere, so gradients cannot flow through it.

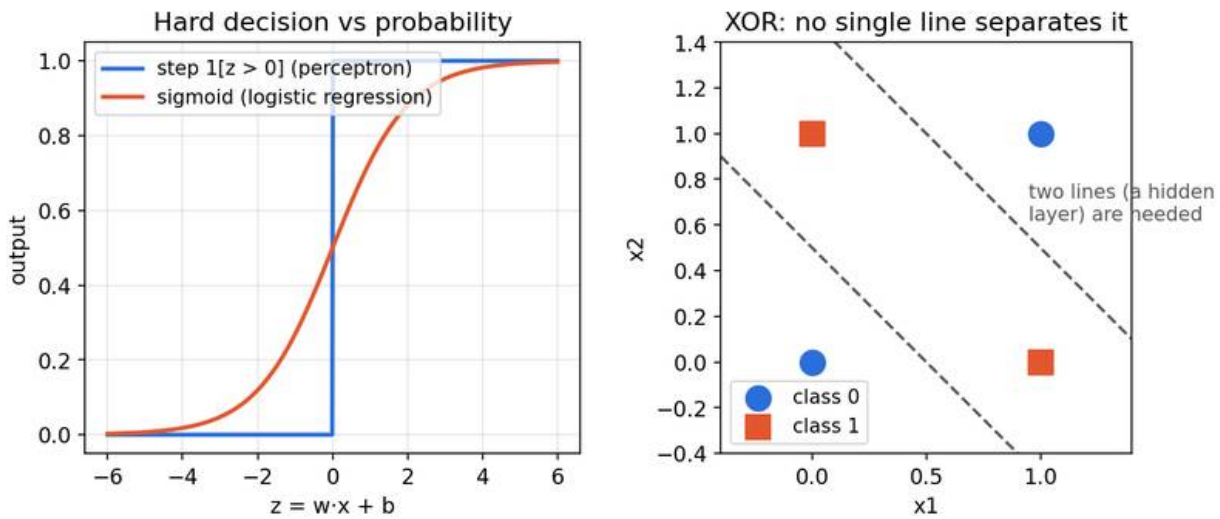



*Figure 1 — Left: the perceptron's step output versus logistic regression's sigmoid probability. Right: XOR, the classic problem that no single linear boundary can solve; two boundaries (a hidden layer) are needed.*

## The multilayer perceptron (MLP)

**Plain version.** Layers combine weighted sums with nonlinear activations to form complex decision regions. Stacking linear maps alone still gives a linear map.

### Stacking layers

An MLP with one hidden layer computes

$$\mathbf{h} = f(W_1\mathbf{x} + \mathbf{b}_1)\,,$$
$$\hat{\mathbf{y}} = g(W_2\mathbf{h} + \mathbf{b}_2)\,,$$

where $\mathbf{x} \in \mathbb{R}^d$ is the input, $W_1 \in \mathbb{R}^{m\times d}$ and $\mathbf{b}_1 \in \mathbb{R}^m$ are the first layer's weights and biases, $m$ is the **width** (number of hidden units), $f$ is a nonlinear **activation function** applied element-wise, $\mathbf{h} \in \mathbb{R}^m$ is the hidden representation, $W_2 \in \mathbb{R}^{K\times m}$ and $\mathbf{b}_2 \in \mathbb{R}^K$ map it to $K$ outputs, and $g$ is the output function chosen for the task (see below). Each hidden unit is one "logistic-regression-like" neuron; the name multilayer *perceptron* is historical, since modern MLPs use smooth activations, not the step function.

The nonlinearity $f$ is essential. Without it,

$W_2(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (W_2W_1)\mathbf{x} + (W_2\mathbf{b}_1 + \mathbf{b}_2)$ is again a single linear map: any number of linear layers collapses to one, and the network can still only draw one hyperplane. For XOR, two hidden units can each draw one of the dashed lines in Figure 1, and the output unit then answers "between the two lines".

### Universal approximation

Cybenko proved in 1989 that a network with **one** hidden layer of sigmoidal units can approximate any continuous function on a bounded box (the unit hypercube) to any desired accuracy, provided the hidden layer is wide enough [5]. Later work extended the result to other activations, including ReLU. The theorem is reassuring, but read it carefully:

- It is about **existence**. It says suitable weights exist; it says nothing about whether gradient descent will find them.
- It says nothing about **how wide** "wide enough" is. For some functions the required width grows extremely fast with the input dimension.
- It says nothing about **generalization**: fitting the training points well does not mean predicting new points well.

So the theorem explains why MLPs are flexible enough in principle. The rest of this tutorial is about the practical questions it leaves open.

## Deep neural networks: depth and width

**Plain version.** "Deep" just means many layers stacked in sequence. There is no official number of layers at which a network becomes deep.

A **deep neural network** (DNN) is an MLP, or any other layered network, with several hidden layers: $\mathbf{h}_\ell = f(W_\ell \mathbf{h}_{\ell-1} + \mathbf{b}_\ell)$ for $\ell = 1, \dots, L$, with $\mathbf{h}_0 = \mathbf{x}$. There is no agreed threshold for "deep". A common informal usage calls any network with more than one hidden layer deep, while other texts count total layers or hidden layers differently, so "three layers" means different things in different books. What matters is the idea, not the count: a deep network builds its answer as a **composition** of many simple steps.

The two ways to add capacity are **width** (more units per layer) and **depth** (more layers). They are not equivalent. A wider layer adds more pieces that are combined once; a deeper network re-uses the features of earlier layers to build more abstract ones. The review by LeCun, Bengio and Hinton [2] explains the success of deep learning through exactly this point: each layer turns the representation from the previous layer into a slightly more abstract one, and composing many such layers lets the network learn very intricate functions from raw data, with the features learned rather than hand-designed.

Historically, the 1990s were dominated by **shallow** methods. Support vector machines, a kernel method rather than a wide neural network, were the main tool, and deep networks were hard to train. From the mid-2000s onward, more data, GPUs and better training techniques made depth practical, and deep networks began to win benchmark after benchmark [2]. Depth alone is not the whole story, though: simply stacking more plain layers eventually makes training *worse*, a problem that residual connections solved (see the last main section).

Figure 2 shows width and depth at work on the two-moons dataset from scikit-learn. With two or eight hidden units in one layer, the boundary is made of only a few straight pieces and cannot

follow the moons. Wider or deeper networks bend the boundary to fit; the widest and deepest one even wraps around single noisy points, a first hint of overfitting.

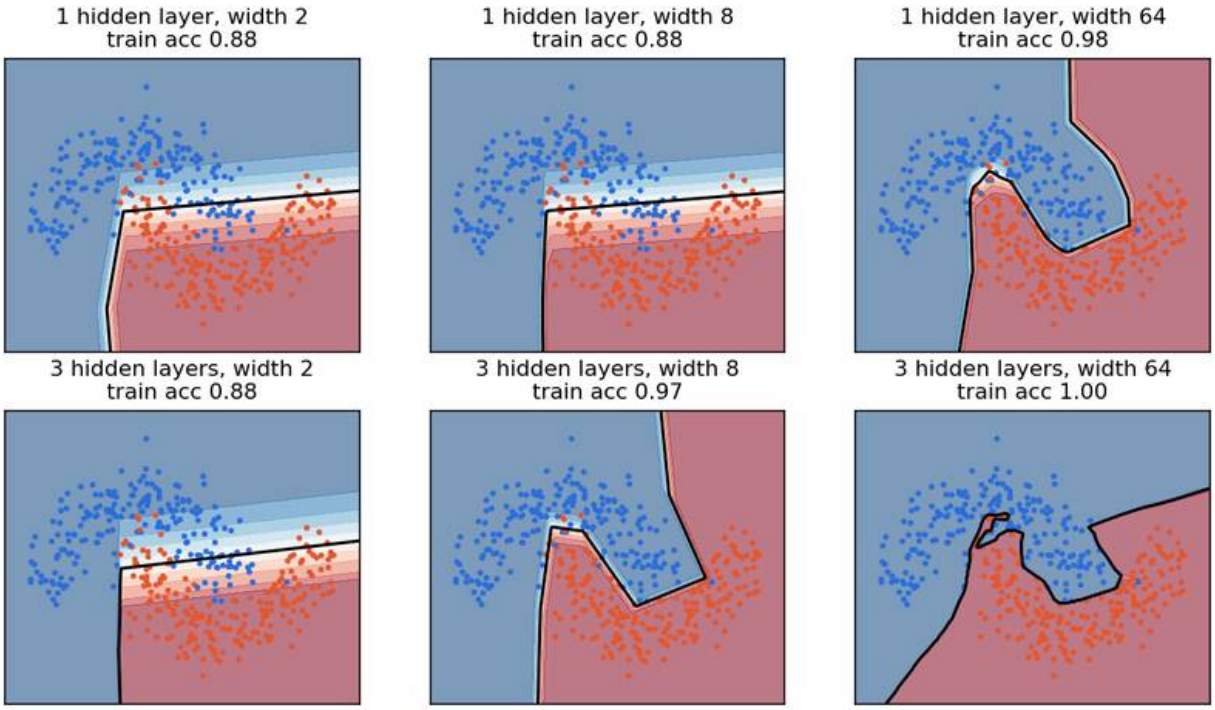



*Figure 2 — Decision boundaries of ReLU MLPs on make_moons (400 points, noise 0.2) as width (columns) and depth (rows) change, each trained with Adam for 1,500 full-batch steps. The black curve is the 0.5 probability contour.*

## Activation functions

**Plain version.** The activation function is the "bend" applied after each weighted sum. Without it, the whole network would be a straight-line model. Different bends trade off smoothness, speed and how well gradients pass through many layers.

Figure 3 plots the common choices and their derivatives. The derivative matters as much as the function, because backpropagation multiplies derivatives layer after layer (see the training section).

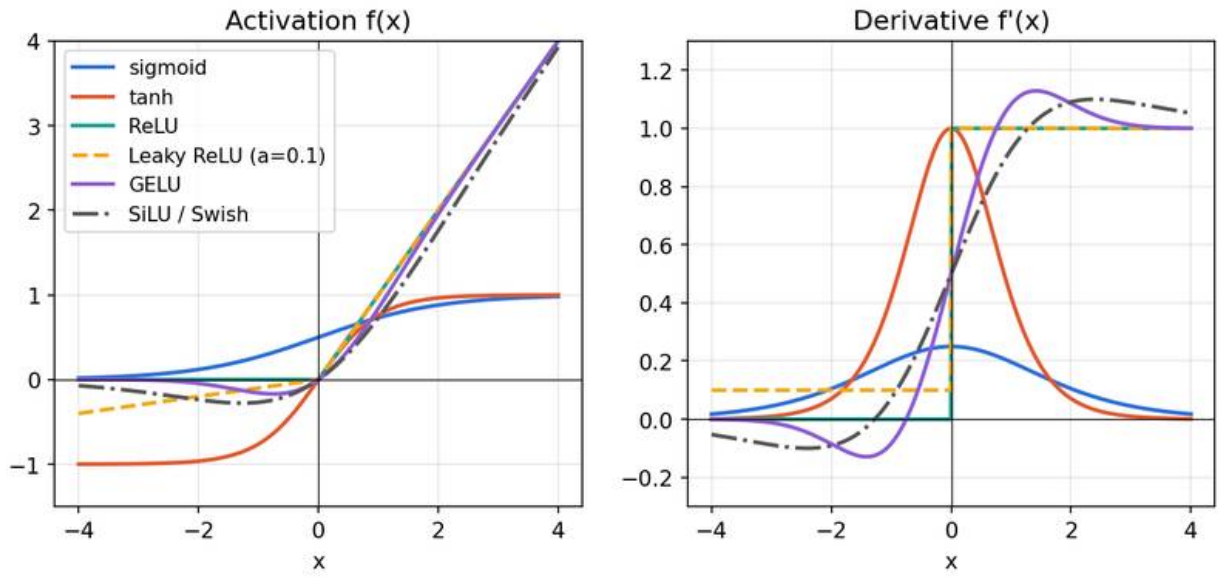



*Figure 3 — Common activation functions (left) and their derivatives (right). Note how small the sigmoid's derivative is everywhere (at most 0.25) and how flat both sigmoid and tanh become for large inputs.*

### Sigmoid and tanh

$$\sigma(x) = \frac{1}{1+e^{-x}}, \quad \sigma'(x) = \sigma(x)(1-\sigma(x)); \qquad \tanh(x) = 2\sigma(2x) - 1, \quad \tanh'(x) = 1 - \tanh^2(x).$$

The sigmoid maps any real number to $(0, 1)$; tanh maps it to $(-1, 1)$ and is centered at zero. Both **saturate**: for large $|x|$ the curve is flat and the derivative is nearly zero, so almost no gradient passes through. The sigmoid's derivative is at most $1/4$ (at $x = 0$). These two facts made deep sigmoid networks hard to train, which Glorot and Bengio analyzed in detail [17]. Today sigmoid is used mainly at the **output** of binary classifiers and inside gates (as in SwiGLU below), and tanh in some recurrent networks.

### Softmax

Softmax turns a vector of $K$ real scores $\mathbf{z}$ (called **logits**) into a probability distribution:

$$p_k = \text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \qquad \frac{\partial p_k}{\partial z_j} = p_k(\delta_{kj} - p_j),$$

where $\delta_{kj}$ is 1 if $k = j$ and 0 otherwise. Each $p_k$ is positive and they sum to 1. Unlike the others in this section, softmax acts on a **whole vector**, not element by element, so it is used almost only in the output layer of multi-class classifiers (and inside attention), not as a hidden activation. In code, subtract $\max_j z_j$ before exponentiating; it does not change the result and avoids overflow.

### ReLU and Leaky ReLU

$$\mathrm{ReLU}(x) = \max(0, x), \quad \mathrm{ReLU}'(x) = \mathbf{1}[x > 0]; \qquad \mathrm{LeakyReLU}_a(x) = \max(ax, x), \quad 0 < a < 1.$$

The rectified linear unit passes positive inputs unchanged and zeroes the rest. Its derivative is exactly 1 for all positive inputs, so gradients do not shrink as they pass through active units, and it is very cheap to compute. ReLU was one of the changes that made deep supervised networks trainable without unsupervised pre-training [2]. Its weakness is the **dying ReLU**: a unit whose input is negative for every example has zero gradient and stops learning. Leaky ReLU keeps a small slope $a$ (for example 0.01) on the negative side so some gradient always flows. The activation survey by Dubey, Singh and Chaudhuri [7] groups these and many later variants into a ReLU-based family.

### GELU and SiLU/Swish

These are smooth versions of ReLU that let small negative values through:

$$\mathrm{GELU}(x) = x\,\Phi(x), \qquad \mathrm{GELU}'(x) = \Phi(x) + x\,\varphi(x),$$

where $\Phi$ is the standard normal cumulative distribution function and $\varphi$ its density [8]. Instead of gating by sign as ReLU does, GELU weights the input by how large it is: $\Phi(x)$ is the probability that a standard normal variable is below $x$. A common fast approximation is $0.5x(1 + \tanh[\sqrt{2/\pi}\,(x + 0.044715x^3)])$ [8]. GELU is the default activation in many Transformer models.

$$\mathrm{Swish}_\beta(x) = x\,\sigma(\beta x), \qquad \mathrm{SiLU}(x) = x\,\sigma(x), \qquad \mathrm{SiLU}'(x) = \sigma(x)(1 + x(1 - \sigma(x))).$$

The sigmoid-weighted linear unit (SiLU) was proposed by Elfwing, Uchibe and Doya for reinforcement learning [9]. Ramachandran, Zoph and Le found the same shape with a learnable or fixed $\beta$ by automated search and called it Swish [10]; with $\beta = 1$, Swish is SiLU. GELU and SiLU look very similar (Figure 3). Both are non-monotonic: they dip slightly below zero for moderate negative inputs.

### SwiGLU: a gated feed-forward block

A **gated linear unit** (GLU) multiplies two linear projections element-wise, one of them passed through a nonlinearity that acts like a gate. Shazeer tested several variants in the Transformer's feed-forward block [11]. The SwiGLU version, written in the paper's row-vector notation and without biases, is

$$\mathrm{FFN}_{\mathrm{SwiGLU}}(\mathbf{x}) = (\,\mathrm{Swish}_1(\mathbf{x}W) \odot \mathbf{x}V)\,W_2,$$

where $\mathbf{x}$ is a row vector of size $d_{\text{model}}$, $W$ and $V$ are $d_{\text{model}} \times d_{\text{ff}}$ matrices, $W_2$ is $d_{\text{ff}} \times d_{\text{model}}$, and $\odot$ is element-wise multiplication. In Shazeer's experiments the GLU variants, SwiGLU and GEGLU among them, gave better results than the ReLU and GELU versions of the block. Because SwiGLU has three weight matrices instead of two, $d_{\text{ff}}$ is reduced by a factor of $2/3$ to keep the parameter

count equal [11]. LLaMA, for example, replaced ReLU with SwiGLU using a hidden size of $\frac{2}{3} \cdot 4d$ [12], and SwiGLU has become a common default in large language models.

| Activation | Range | Smooth | Typical use today |
|---|---|---|---|
| sigmoid | $(0, 1)$ | yes | binary / multi-label output, gates |
| tanh | $(-1, 1)$ | yes | some recurrent networks |
| softmax | probability vector | yes | multi-class output, attention weights |
| ReLU / Leaky ReLU | $[0, \infty)$ / $\mathbb{R}$ | no (kink at 0) | CNNs, MLPs, the safe default |
| GELU, SiLU | about $[-0.17, \infty)$, $[-0.28, \infty)$ | yes | Transformers, modern CNNs |
| SwiGLU | $\mathbb{R}$ | yes | Transformer feed-forward blocks |

## Normalization layers

**Plain version.** As training goes on, the numbers flowing through a network can drift to very large or very small scales, which makes learning unstable. A normalization layer rescales them to a standard size at each step, then lets the network choose the final scale it prefers.

All three common layers share one template. For a set of values $x_1, \ldots, x_n$,

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i, \qquad \sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2, \qquad y_i = \gamma\,\frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta,$$

where $\mu$ and $\sigma^2$ are the mean and variance, $\epsilon$ is a small constant for numerical safety, and $\gamma, \beta$ are a learnable scale and shift (so the layer can undo the normalization if that helps). The layers differ in **which values** form the set (Figure 4).

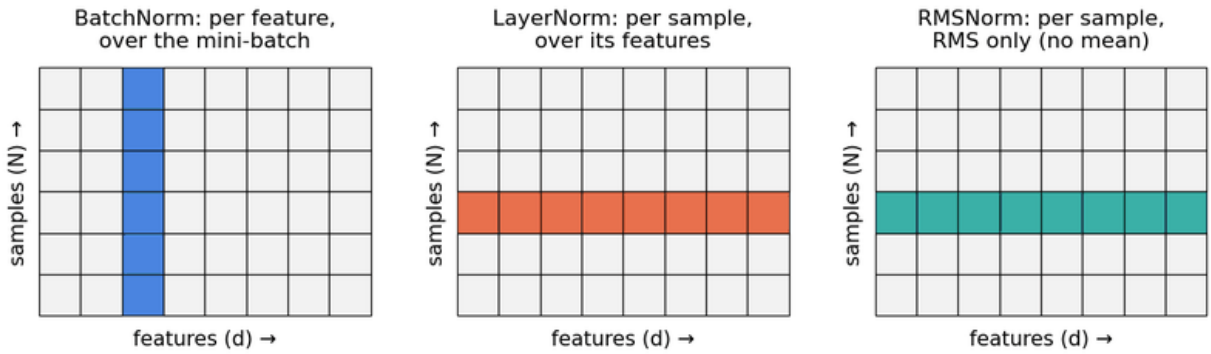



*Figure 4 — Which entries share statistics. BatchNorm normalizes each feature over the mini-batch; LayerNorm and RMSNorm normalize each sample over its own features.*

- **BatchNorm** (Ioffe and Szegedy [13]) computes $\mu, \sigma^2$ for each feature (each channel, in a CNN) **across the mini-batch**. It speeds up training and became standard in CNNs. Because the statistics depend on the batch, it behaves differently in training and inference: at inference it uses running averages collected during training. It also degrades when batches are very small, since the batch statistics become noisy.
- **LayerNorm** (Ba, Kiros and Hinton [14]) computes $\mu, \sigma^2$ **over the features of a single sample**. It does not depend on the batch, behaves identically at training and test time, and works naturally for sequences. It is the standard normalization in Transformers.
- **RMSNorm** (Zhang and Sennrich [15]) drops the mean subtraction and divides by the root mean square, $y_i = \gamma_i \, x_i / \sqrt{\frac{1}{n}\sum_j x_j^2 + \epsilon}$. The authors hypothesize that LayerNorm's re-centering is dispensable and report comparable quality with lower running time [15]. LLaMA uses RMSNorm and applies it to the **input** of each Transformer sub-layer ("pre-normalization") rather than to its output [12].

```
import torch

x = torch.randn(4, 8) * 5 + 3                       # 4 samples, 8 features
ln = torch.nn.LayerNorm(8, elementwise_affine=False)
bn = torch.nn.BatchNorm1d(8, affine=False)
rms = x / torch.sqrt(x.pow(2).mean(dim=1, keepdim=True) + 1e-6)

print(ln(x).mean(dim=1))     # ~0 for every sample (row)
print(bn(x).mean(dim=0))     # ~0 for every feature (column), in train mode
print(rms.pow(2).mean(dim=1))  # ~1 for every sample: unit RMS, mean not removed
```

Huang et al. [16] survey this family and decompose any normalization method into three choices: which values are pooled together (the *normalization area*), what operation is applied (standardizing, centering, scaling or whitening), and how the representation is recovered afterward (the learnable affine step). BatchNorm, LayerNorm and RMSNorm differ mainly in the first two.

## Designing the output layer and choosing the loss

**Plain version.** The last layer must speak the language of the answer: any real number for a quantity, a probability for yes/no, a probability distribution for "which of $K$". Each kind of output has a natural loss, and the pairs below are the ones to use.

| Task | Output layer | Output range | Loss |
|---|---|---|---|
| Regression | linear (no activation) | $\mathbb{R}$ | mean squared error (MSE, L2) or mean absolute error (MAE, L1) |
| Binary classification | 1 unit + sigmoid | $(0, 1)$ | binary cross-entropy |
| Multi-label (several yes/no) | $K$ units + sigmoid each | $(0, 1)^K$ | binary cross-entropy per label |

| Task | Output layer | Output range | Loss |
|---|---|---|---|
| Multi-class (exactly one of $K$) | $K$ units + softmax | probability vector | cross-entropy |

A common mistake is to put a sigmoid on a regression output. The sigmoid's range is the open interval $(0, 1)$, and its job is to output a probability for binary or multi-label classification. A regression head is normally **linear**, so it can produce any real value. A sigmoid on a regression output is reasonable only when the target itself is known to lie between 0 and 1, such as a normalized pixel intensity or a fraction.

The losses, for one example, are

$$\begin{aligned} \text{MSE:} \quad & \ell = \tfrac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|^2, \\ \text{binary cross-entropy:} \quad & \ell = -[\, y \log \hat{p} + (1-y)\log(1-\hat{p}) \,], \\ \text{cross-entropy:} \quad & \ell = -\sum_{k=1}^{K} y_k \log p_k = -\log p_c, \end{aligned}$$

where $\mathbf{y}$ is the target, $\hat{p} = \sigma(z)$, $\mathbf{p} = \text{softmax}(\mathbf{z})$, $y_k$ is the one-hot label and $c$ is the correct class. (The factor $\frac{1}{2}$ in MSE is a convenience; PyTorch's `MSELoss` averages without it.) MAE, $\|\hat{\mathbf{y}} - \mathbf{y}\|_1$, is less sensitive to outliers than MSE. Each pair is a maximum-likelihood estimate: MSE corresponds to Gaussian noise on the target, binary cross-entropy to a Bernoulli label, cross-entropy to a categorical label [3].

The pairs are also chosen for a beautiful practical reason. In all three cases the gradient of the loss with respect to the **pre-activation** of the output layer is simply

$$\frac{\partial \ell}{\partial \mathbf{z}} = \hat{\mathbf{y}} - \mathbf{y} \quad \text{(prediction minus target)},$$

with $\hat{\mathbf{y}}$ meaning the linear output, $\sigma(z)$ or $\text{softmax}(\mathbf{z})$ respectively. The saturating sigmoid or softmax derivative cancels out, so a confidently wrong prediction still produces a large gradient. (Exercise 3 asks you to verify this for softmax.) This is why frameworks fuse the last activation into the loss: PyTorch's `nn.CrossEntropyLoss` and `nn.BCEWithLogitsLoss` take **logits**, so the model itself must not end with a softmax or sigmoid during training.

## Training I: the forward pass and backpropagation

**Plain version.** The forward pass computes predictions and loss. Backpropagation reuses those intermediate values to compute parameter derivatives through the chain rule.

Backpropagation was popularized for training multi-layer networks by Rumelhart, Hinton and Williams [6]. It is nothing more than the chain rule of calculus, organized so that no work is repeated. Figure 5 shows the computation graph for the network we will differentiate.

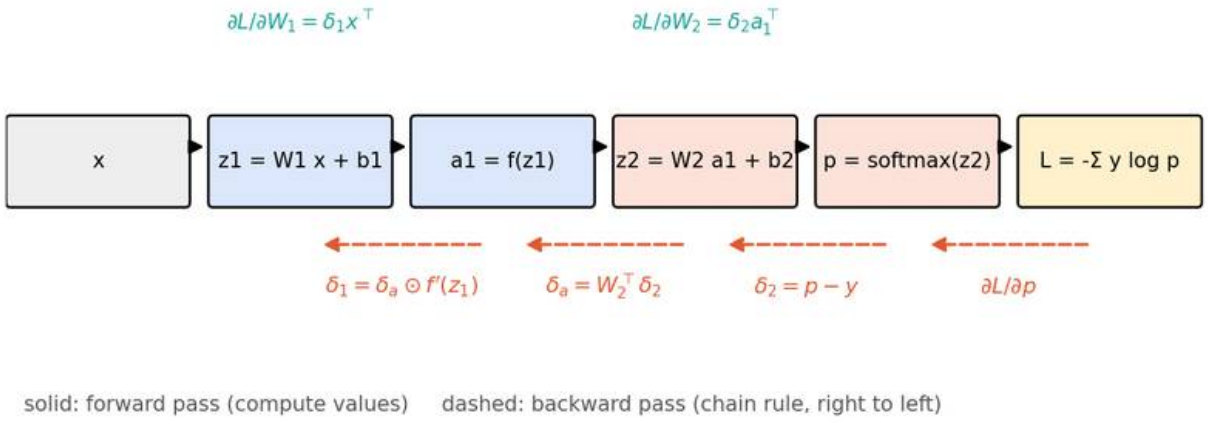



*Figure 5 — Computation graph of a two-layer MLP with softmax cross-entropy. Solid arrows: forward pass. Dashed arrows: the backward pass, which carries the error signal $\delta$ from the loss back toward the input.*

### Derivation for a two-layer MLP

Take one example $(\mathbf{x}, \mathbf{y})$ with $\mathbf{x} \in \mathbb{R}^d$ and a one-hot label $\mathbf{y} \in \{0,1\}^K$. The forward pass is

$$\begin{aligned} \mathbf{z}_1 &= W_1\mathbf{x} + \mathbf{b}_1 \in \mathbb{R}^m, \quad \mathbf{a}_1 = f(\mathbf{z}_1) \in \mathbb{R}^m, \\ \mathbf{z}_2 &= W_2\mathbf{a}_1 + \mathbf{b}_2 \in \mathbb{R}^K, \quad \mathbf{p} = \mathrm{softmax}(\mathbf{z}_2), \qquad L = -\textstyle\sum_k y_k \log p_k, \end{aligned}$$

with $W_1 \in \mathbb{R}^{m\times d}$, $W_2 \in \mathbb{R}^{K\times m}$, and $f$ an element-wise activation. Define the **error signal** of a layer as the gradient of the loss with respect to its pre-activation, $\delta_\ell = \partial L/\partial \mathbf{z}_\ell$.

**Step 1, output layer.** From the previous section, softmax with cross-entropy gives

$$\delta_2 = \frac{\partial L}{\partial \mathbf{z}_2} = \mathbf{p} - \mathbf{y} \in \mathbb{R}^K.$$

**Step 2, output weights.** Since $z_{2,k} = \sum_j W_{2,kj}\, a_{1,j} + b_{2,k}$, we have $\partial z_{2,k}/\partial W_{2,kj} = a_{1,j}$, so

$$\frac{\partial L}{\partial W_2} = \delta_2\, \mathbf{a}_1^\top \in \mathbb{R}^{K\times m}, \qquad \frac{\partial L}{\partial \mathbf{b}_2} = \delta_2.$$

The gradient of a weight is (error at its output) times (activation at its input). The shapes match $W_2$, which is a useful check.

**Step 3, back through the linear layer.** Each hidden activation $a_{1,j}$ affects every output, so its gradient sums over them: $\partial L/\partial a_{1,j} = \sum_k \delta_{2,k} W_{2,kj}$, or in matrix form

$$\frac{\partial L}{\partial \mathbf{a}_1} = W_2^\top \delta_2 \in \mathbb{R}^m.$$

**Step 4, back through the activation.** Because $f$ acts element by element, its Jacobian is diagonal, and the chain rule becomes an element-wise product:

$$\delta_1 = \frac{\partial L}{\partial \mathbf{z}_1} = (W_2^\top \delta_2) \odot f'(\mathbf{z}_1).$$

**Step 5, first-layer weights.** Exactly as in Step 2,

$$\frac{\partial L}{\partial W_1} = \delta_1 \, \mathbf{x}^\top \in \mathbb{R}^{m \times d}, \qquad \frac{\partial L}{\partial \mathbf{b}_1} = \delta_1.$$

For a deeper network, Steps 3 to 5 simply repeat:

$\delta_\ell = (W_{\ell+1}^\top \delta_{\ell+1}) \odot f'(\mathbf{z}_\ell)$ and $\partial L / \partial W_\ell = \delta_\ell \, \mathbf{a}_{\ell-1}^\top$. For a mini-batch of $N$ examples, the loss is the average, so each gradient is the average of the per-example gradients.

### From scratch in NumPy, with a gradient check

The code below stores examples as **rows** (`X` has shape $N \times d$), which is the usual convention in code. Then `X @ W1` with `W1` of shape $d \times m$ plays the role of $W_1 \mathbf{x}$ in the math, and every formula above appears transposed: for example `dW2 = A1.T @ dZ2` is $\delta_2 \mathbf{a}_1^\top$ stacked over the batch. The **gradient check** compares one analytic derivative with a central finite difference, $(L(w+\epsilon) - L(w-\epsilon))/2\epsilon$. If they agree to many digits, the backward pass is correct. Always do this when you write gradients by hand.

```
import numpy as np
from sklearn.datasets import make_moons

rng = np.random.default_rng(0)
X, y = make_moons(n_samples=200, noise=0.2, random_state=0)
N, d, h, K = X.shape[0], 2, 16, 2
Y = np.eye(K)[y]                                  # one-hot targets, (N, K)

# He-style initialization (std = sqrt(2 / fan_in))
W1 = rng.normal(0, np.sqrt(2 / d), (d, h)); b1 = np.zeros(h)
W2 = rng.normal(0, np.sqrt(2 / h), (h, K)); b2 = np.zeros(K)

def forward(X, W1, b1, W2, b2):
    Z1 = X @ W1 + b1                              # (N, h)
    A1 = np.maximum(Z1, 0)                        # ReLU
    Z2 = A1 @ W2 + b2                             # (N, K) logits
    Z2 = Z2 - Z2.max(axis=1, keepdims=True)       # numerical stability
    P = np.exp(Z2) / np.exp(Z2).sum(axis=1, keepdims=True)
    loss = -np.mean(np.sum(Y * np.log(P + 1e-12), axis=1))
    return Z1, A1, P, loss

def backward(X, Z1, A1, P, W2):
    dZ2 = (P - Y) / N                             # delta at the output
    dW2 = A1.T @ dZ2;  db2 = dZ2.sum(axis=0)
    dA1 = dZ2 @ W2.T
    dZ1 = dA1 * (Z1 > 0)                          # ReLU'(z) = 1[z > 0]
```

```
    dW1 = X.T @ dZ1;   db1 = dZ1.sum(axis=0)
    return dW1, db1, dW2, db2

# Gradient check on one entry of W1 (central difference)
Z1, A1, P, loss = forward(X, W1, b1, W2, b2)
dW1, db1, dW2, db2 = backward(X, Z1, A1, P, W2)
eps = 1e-6; E = np.zeros_like(W1); E[0, 3] = eps
num = (forward(X, W1 + E, b1, W2, b2)[3] - forward(X, W1 - E, b1, W2, b2)[3]) / (2 * eps)
print(f"analytic {dW1[0, 3]:.8f}  numeric {num:.8f}")

# Plain gradient descent
lr = 0.5
for step in range(2000):
    Z1, A1, P, loss = forward(X, W1, b1, W2, b2)
    dW1, db1, dW2, db2 = backward(X, Z1, A1, P, W2)
    W1 -= lr * dW1; b1 -= lr * db1; W2 -= lr * dW2; b2 -= lr * db2
acc = (P.argmax(axis=1) == y).mean()
print(f"final loss {loss:.3f}, train accuracy {acc:.3f}")
# analytic -0.01941581  numeric -0.01941581
# final loss 0.063, train accuracy 0.970
```

In practice you never write the backward pass yourself: frameworks such as PyTorch record the forward computation and run this exact procedure automatically when you call `loss.backward()`. Writing it once by hand is still the best way to understand what that call does, and why it fails when something in the graph has a zero derivative.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

## Training II: making optimization work

**Plain version.** Knowing the gradient is not enough. You also need a sensible starting point (initialization), a rule for turning gradients into steps (the optimizer), a step size that changes over time (the schedule), a choice of how many examples to look at per step (the batch size), and a guard against occasional huge steps (clipping).

### Vanishing and exploding gradients

Unroll the recursion $\delta_\ell = (W_{\ell+1}^\top \delta_{\ell+1}) \odot f'(\mathbf{z}_\ell)$ across $L$ layers and the gradient at the first layer contains a **product** of $L$ weight matrices and $L$ activation derivatives. If the typical factor is below 1, the product shrinks exponentially with depth and the early layers barely learn: **vanishing gradients**. With the sigmoid, every $f'$ is at most 0.25, so ten layers can shrink the signal by a factor of up to $0.25^{10} \approx 10^{-6}$. If the typical factor is above 1, for example because the initial weights are too large, the product grows exponentially, updates become huge, and the loss turns into `inf` or `NaN`: **exploding gradients**.

The remedies, each covered below, attack the product from different sides:

- activations whose derivative is 1 over a wide range (ReLU and its smooth relatives);
- initialization that keeps signal size constant from layer to layer;
- normalization layers that re-standardize activations;
- **residual connections**, which give the gradient a path with factor 1 (the last main section);
- **gradient clipping**, the standard emergency brake for explosions.

Other practices sometimes listed, such as fine-tuning a pre-trained model or using a smaller network, make training easier overall, but they do not directly fix the gradient problem.

### Weight initialization

If all weights start equal, every hidden unit computes the same thing and receives the same gradient, and they stay identical forever. So weights start random. The scale matters: we want the variance of activations (forward) and of gradients (backward) to stay roughly constant from layer to layer. For a linear layer with $n_{\text{in}}$ inputs and $n_{\text{out}}$ outputs:

- **Xavier (Glorot) initialization** [17] balances the forward and backward conditions for symmetric activations such as tanh: $\operatorname{Var}(W_{ij}) = \dfrac{2}{n_{\text{in}} + n_{\text{out}}}$, for example uniform on $[-\sqrt{6/(n_{\text{in}} + n_{\text{out}})}, \sqrt{6/(n_{\text{in}} + n_{\text{out}})}]$.
- **He (Kaiming) initialization** [18] accounts for ReLU zeroing about half its inputs, which halves the variance at every layer, by doubling the scale: $\operatorname{Var}(W_{ij}) = \dfrac{2}{n_{\text{in}}}$, for example $W_{ij} \sim \mathcal{N}(0, 2/n_{\text{in}})$.

Biases usually start at zero. He et al. showed that this initialization lets very deep ReLU networks be trained from scratch [18]. The NumPy example above uses it.

### Optimizers

Every optimizer below is a variant of **stochastic gradient descent** (SGD): estimate the gradient on a small random mini-batch, take a step against it, repeat. Write $\mathbf{g}_t = \nabla_\theta L_{\text{batch}}(\theta_t)$ for the mini-batch gradient at step $t$, $\theta$ for all parameters and $\eta$ for the learning rate.

**SGD.** $\theta_{t+1} = \theta_t - \eta\,\mathbf{g}_t$. Simple and memory-free, but it zigzags in narrow valleys and moves slowly along flat directions.

**SGD with momentum.** Keep a running velocity: $\mathbf{v}_{t+1} = \mu\,\mathbf{v}_t + \mathbf{g}_t$, $\theta_{t+1} = \theta_t - \eta\,\mathbf{v}_{t+1}$, with $\mu$ around 0.9. Like a heavy ball rolling downhill, it averages out the zigzag and builds speed along consistent directions.

**Adam** (Kingma and Ba [19]) keeps two running averages per parameter, of the gradient and of its square:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\,\mathbf{g}_t, \quad \mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\,\mathbf{g}_t^2,$$
$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}, \qquad \hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}, \qquad \theta_t = \theta_{t-1} - \eta\,\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon},$$

where squares, square roots and division are element-wise, $\beta_1 \approx 0.9$ and $\beta_2 \approx 0.999$ are decay rates, and $\epsilon$ is a small constant. The first average is momentum. Dividing by the root of the second gives every parameter its own step size, which is the idea of RMSprop that Adam builds on [19]: parameters with consistently large gradients take smaller steps, rarely updated ones take larger steps. The **bias correction** $1/(1-\beta^t)$ fixes the fact that both averages start at zero and would otherwise be too small in the first steps.

**AdamW** (Loshchilov and Hutter [20]). Weight decay shrinks weights toward zero at every step. For plain SGD this is the same as adding an L2 penalty $\frac{\lambda}{2}\|\theta\|^2$ to the loss. For Adam it is not: an L2 term enters $\mathbf{g}_t$ and is then divided by $\sqrt{\hat{\mathbf{v}}_t}$, so weights with large gradients are regularized *less*. AdamW **decouples** the decay from the adaptive step:

$$\theta_t = \theta_{t-1} - \eta\left(\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} + \lambda\,\theta_{t-1}\right).$$

The authors showed this improves Adam's generalization [20], and AdamW is now the most common optimizer for Transformers and large language models. In PyTorch, `torch.optim.Adam(..., weight_decay=λ)` is the coupled L2 version and `torch.optim.AdamW` is the decoupled one.

**Newer optimizers (brief notes).** Two recent alternatives are worth knowing by name. **Lion** was found by an automated program search [21]: it keeps only a momentum buffer (so it uses less memory than Adam) and updates every parameter by the **sign** of an interpolated momentum, so all updates have the same magnitude; it needs a smaller learning rate than AdamW. **Muon** [22] applies to the 2-D weight matrices of hidden layers: it takes the momentum update and approximately orthogonalizes it with a few Newton–Schulz iterations, while embeddings, output layers, biases and other scalar or vector parameters are still trained with AdamW. Liu et al. report that, with weight decay and per-parameter update scaling added, Muon reached about twice the compute efficiency of AdamW in their compute-optimal large-language-model training experiments [23]. These are active research topics; AdamW remains the safe default.

### Learning-rate schedules

The learning rate is usually the most important hyperparameter. A rate that is fine early in training is often too large later, when the model needs small, careful steps, so the rate is varied over time (Figure 6).

- **Step decay**: multiply the rate by a factor such as 0.1 at fixed epochs. Traditional for CNN training with SGD.
- **Cosine annealing**: $\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos(\pi t/T))$, a smooth decay from $\eta_{\max}$ to $\eta_{\min}$ over $T$ steps; SGDR [24] combined it with periodic warm restarts.

- **Warmup**: increase the rate linearly from near zero over the first few hundred or thousand steps. Early in training, Adam's second-moment estimates are noisy and the weights are far from any good region, and a full-size step can be destabilizing.

As a concrete recipe from a published large model, LLaMA was trained with AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.95$), weight decay 0.1, gradient clipping at 1.0, 2,000 warmup steps, and a cosine schedule ending at 10% of the peak rate [12].

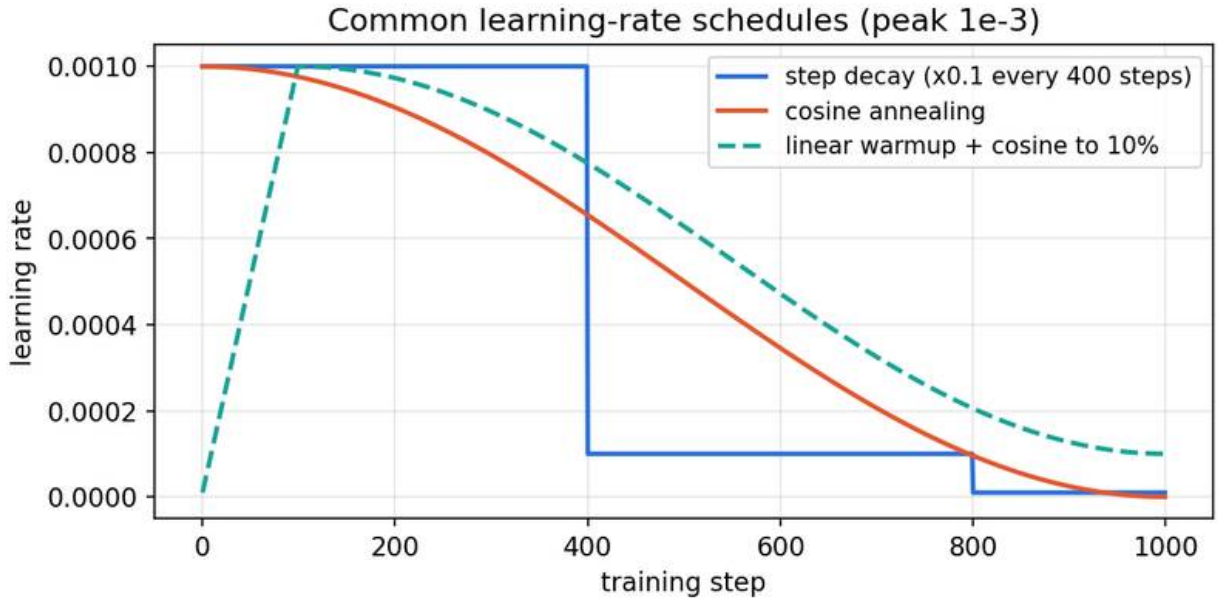



*Figure 6 — Three common learning-rate schedules over 1,000 steps with a peak of 1e-3.*

### Mini-batches, iterations and epochs

A **mini-batch** (often just "batch") is the small random subset of $B$ examples used for one gradient estimate. One **iteration** is one parameter update, using one mini-batch. One **epoch** is one full pass over the training set, which takes $\lceil N/B \rceil$ iterations for $N$ examples. For example, $N = 50{,}000$ and $B = 128$ give 391 iterations per epoch.

The batch size $B$ affects both speed and the result, and bigger is **not** always better:

- **Small batches** give noisy gradient estimates (the variance of the mini-batch mean falls roughly like $1/B$), so the loss curve is jumpy and each epoch takes many steps with poor hardware utilization.
- **Large batches** give stable gradients and run efficiently in parallel on GPUs, but each epoch has fewer updates, and they tend to generalize worse when other settings are kept fixed. Keskar et al. found that large-batch training tends to converge to **sharp minima**, where the loss rises steeply around the solution, while small-batch training, thanks to its gradient noise, tends to find **flat minima**, which generalize better [25].

In practice, choose the largest batch that fits comfortably in memory *and* still trains well, and retune the learning rate when you change $B$; the two interact strongly.

### Gradient clipping

Even in a well-designed network, a rare batch can produce a huge gradient. **Gradient-norm clipping** rescales the whole gradient when its norm exceeds a threshold $c$:

$$\mathbf{g} \leftarrow \mathbf{g} \cdot \min\left(1, \frac{c}{\|\mathbf{g}\|}\right),$$

where $\|\mathbf{g}\|$ is the norm of all parameter gradients concatenated. It keeps the direction but limits the step size, which is the standard defense against exploding gradients, especially in recurrent networks and Transformers [3]. A threshold of 1.0 is a common choice.

### Putting it together in PyTorch

The snippet trains a small MLP on `make_moons` with mini-batches, AdamW, a cosine schedule and gradient clipping. It runs on a CPU in seconds.

```
import torch, torch.nn as nn
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split

torch.manual_seed(0)
X, y = make_moons(n_samples=1000, noise=0.25, random_state=0)
Xtr, Xva, ytr, yva = train_test_split(X, y, test_size=0.3, random_state=0)
Xtr, Xva = torch.tensor(Xtr, dtype=torch.float32), torch.tensor(Xva, dtype=torch.float32)
ytr, yva = torch.tensor(ytr), torch.tensor(yva)

model = nn.Sequential(
    nn.Linear(2, 64), nn.ReLU(),
    nn.Linear(64, 64), nn.ReLU(),
    nn.Linear(64, 2),               # logits: no softmax here
)
loss_fn = nn.CrossEntropyLoss()     # applies log-softmax internally
opt = torch.optim.AdamW(model.parameters(), lr=1e-2, weight_decay=1e-2)
sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, T_max=200)

for epoch in range(200):
    model.train()
    perm = torch.randperm(len(Xtr))
    for i in range(0, len(Xtr), 64):                # mini-batches of 64
        idx = perm[i:i + 64]
        loss = loss_fn(model(Xtr[idx]), ytr[idx])
        opt.zero_grad()
        loss.backward()                              # backpropagation
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        opt.step()
    sched.step()
```

```
model.eval()
with torch.no_grad():
    acc = (model(Xva).argmax(dim=1) == yva).float().mean().item()
print(f"validation accuracy: {acc:.3f}")   # about 0.94
```

Note `model.train()` and `model.eval()`: they switch layers such as dropout and BatchNorm between their training and inference behavior. Forgetting `eval()` at test time is a classic bug.

*Online only: Adam — adaptive updates and the convergence caveat — https://blog.shyandram.dev/en/learn/ml/neural-networks/papers/adam–adaptive-updates-and-the-convergence-caveat/*

## Generalization: underfitting, overfitting and regularization

**Plain version.** Compare training and validation performance. High training error may reflect capacity or optimization; a large validation gap suggests poor generalization.

### Capacity and the two failure modes

A model's **capacity** is, informally, how complex a function it can fit; it grows with the number of parameters, depth and width, and training time. As capacity increases, training error keeps falling, while validation error first falls and then rises again once the model starts fitting noise. The gap between the two is the **generalization gap** [3].

- **Underfitting**: the **training** error itself is high. The model, or the training procedure, cannot capture the pattern.
- **Overfitting**: the training error is low but the validation or test error is much higher.

What to do about underfitting, in order:

1. **Check the code and data first.** A bug, mismatched inputs and labels, wrong preprocessing, or a loss applied to the wrong tensor all look like underfitting.
2. **Increase capacity**: a wider or deeper model, or a better architecture.
3. **Improve optimization**: train longer, tune the learning rate and schedule, switch optimizer, check initialization and normalization.

Making the model **smaller** never fixes underfitting; it lowers capacity further. A smaller model can be a reasonable choice for speed or memory once accuracy is adequate, but that is an efficiency decision, not a cure.

What to do about overfitting: use more or more varied data, add regularization (below), reduce capacity, and stop early. It is also worth checking that the training and validation sets really come from the same distribution; a mismatch between them looks like overfitting but needs a data fix, not a regularizer.

### Reading a loss curve

Plot both the training and the validation loss against iterations or epochs. Four patterns cover most situations:

- **A. Both curves flat from the start.** If the loss is high, the model is not learning at all: suspect a bug, a far too small or large learning rate, or a model/data mismatch. If the loss is already low, the task may be very easy, or each logged point covers so many iterations that the curve's shape is hidden; log more often.
- **B. Training loss keeps falling, validation loss turns upward.** Classic overfitting. Keep the parameters from the point of lowest validation loss (early stopping) and add regularization.
- **C. Both curves fall and level off close together.** Either the model is near its best, which is common when training and validation data are very similar, or both are still high, which is underfitting: train longer or increase capacity.
- **D. Validation loss above training loss, but both still decreasing steadily.** The normal, healthy situation. Keep training and keep the checkpoint with the lowest validation loss.

### Regularization tools

Figure 7 shows overfitting and two cures on purpose. A wide MLP (two hidden layers of 256 units) is trained on only 100 noisy two-moons points. Without regularization the training loss falls toward zero while the validation loss reaches its minimum early and then climbs steeply. Weight decay and dropout both slow the climb dramatically.

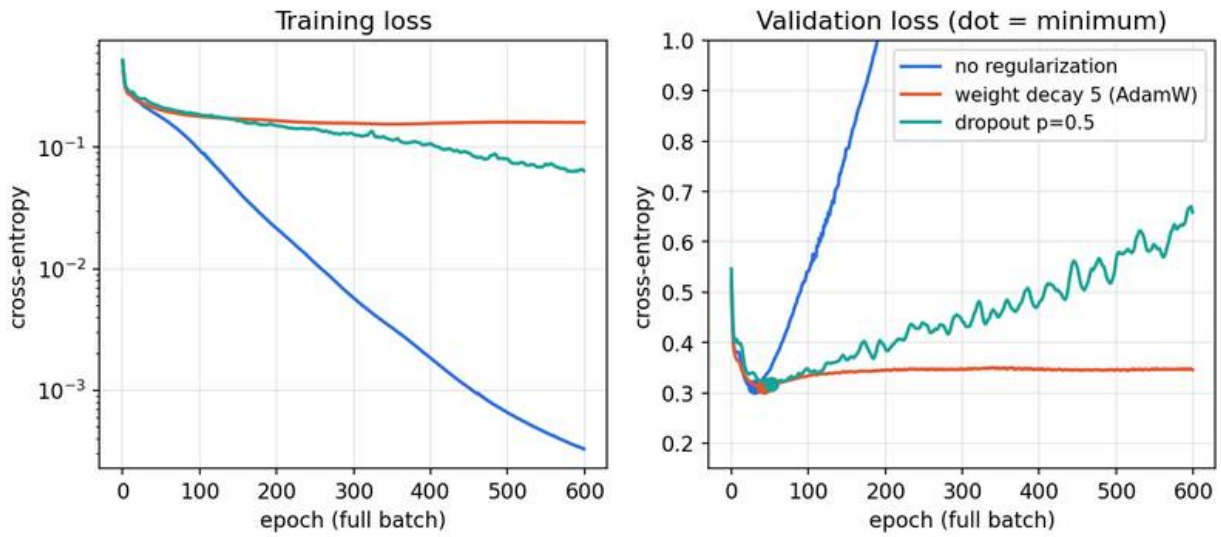



*Figure 7 — Overfitting on 100 training points from make_moons (noise 0.35), evaluated on 2,000 validation points. Full-batch AdamW, learning rate 3e-3. Dots mark each run's minimum validation loss, which is where early stopping would keep the model.*

**Weight decay (L2 regularization).** Penalize large weights by adding $\frac{\lambda}{2}\|\theta\|^2$ to the loss, or, with

Adam, use decoupled decay (AdamW). Large weights let a network make sharp, wiggly boundaries around individual points; keeping them small favors smoother functions. In Figure 7, decoupled decay with $\lambda = 5$ keeps the validation loss almost flat after its minimum.

**Dropout** (Srivastava et al. [27]). During training, set each hidden unit to zero with probability $p$, independently for every example and step, and scale the survivors by $1/(1-p)$ so the expected activation is unchanged ("inverted dropout"). At test time, use all units with no scaling. Each step thus trains a different thinned sub-network; units cannot rely on specific partners and must learn features that are useful on their own, and the full network at test time behaves like an average of many thinned networks [27]. This is why `model.eval()` matters.

**Data augmentation.** Create new training examples by applying transformations that do not change the label: flips, crops, small rotations, color jitter and noise for images. It is often the single most effective regularizer in vision, because it teaches the invariances directly. It must respect the task: a horizontal flip is fine for cats, not for reading text.

**Early stopping.** Monitor the validation loss during training and keep the parameters from the epoch where it was lowest; stop when it has not improved for a set number of epochs (the "patience"). It costs nothing, needs no change to the model, and acts as a regularizer by limiting how far the parameters can travel from their initialization [3].

**More data and better strategy.** The other levers the original notes list are worth repeating: more training data is the most reliable cure for overfitting; a stronger architecture (deeper, or better structured, so that it converges well) helps when the model underfits; and a better training strategy (optimizer, learning-rate schedule, batch size) helps both.

## Residual connections and the road to Transformers

**Plain version.** If you stack many layers, each must pass the signal on intact as well as improve it, which turns out to be hard. A residual connection gives every block an express lane: the block only has to learn a *correction* that is added to its input.

He et al. observed that adding more plain layers to an already deep network made even the **training** error worse, which is an optimization problem rather than overfitting [28]. Their **residual block** computes

$$\mathbf{y} = \mathbf{x} + F(\mathbf{x}),$$

where $\mathbf{x}$ is the block's input and $F$ is a small stack of layers (in ResNet, convolutions with BatchNorm and ReLU). If the best thing a block can do is nothing, it only has to push $F$ toward zero, which is easy. The backward pass shows why it trains well:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = I + \frac{\partial F}{\partial \mathbf{x}},$$

where $I$ is the identity matrix. The identity term passes the gradient straight through every block, so the product over many blocks no longer has to shrink to zero. With residual connections, He et al. trained networks with up to 152 layers on ImageNet and won several ILSVRC 2015 tasks [28].

The **Transformer** (Vaswani et al. [29]) combines every idea in this tutorial. Each block has two sub-layers, multi-head self-attention (which uses softmax to mix information between positions) and a position-wise MLP, and each sub-layer is wrapped in a residual connection with LayerNorm. The original paper normalized after the addition; many recent models, LLaMA among them, normalize the sub-layer **input** instead (pre-normalization) and use RMSNorm and SwiGLU [12]:

$$\mathbf{h} = \mathbf{x} + \text{Attention}\,(\,\text{Norm}(\mathbf{x})), \qquad \mathbf{y} = \mathbf{h} + \text{FFN}\,(\,\text{Norm}(\mathbf{h})).$$

Since 2017 this design has spread from machine translation to language modeling, vision, speech and more, and it is the basis of today's large language models. The trained recipe is the one from this tutorial too: AdamW, warmup and cosine decay, gradient clipping, and weight decay [12]. The deep-learning track picks up from here with convolutional networks and Transformers for images, starting with image classification.

## Modern view

The basic toolkit above has been stable for several years; research continues on why it works and on better versions of each piece. The following reviews are good guides, each from a different angle.

**The big picture: LeCun, Bengio and Hinton [2] (Nature, 2015).** This short review is still the clearest statement of what deep learning is: representation learning with many layers, where each layer transforms the previous representation into a more abstract one and the features are learned from data with backpropagation rather than engineered. It walks through supervised learning with SGD, the role of ReLU, convolutional networks for images, recurrent networks for sequences, and distributed representations, and it closes by predicting that unsupervised learning and the combination of representation learning with reasoning would matter more in the future. For a textbook treatment of every topic in this tutorial, including the maximum-likelihood view of loss functions, regularization and optimization, the book by Goodfellow, Bengio and Courville [3] remains the standard reference.

**Optimization: Sun [26].** Sun's survey asks "when and why can a neural network be successfully trained?" and organizes the answer in three parts: problems with the gradient (explosion, vanishing) and their fixes through careful initialization and normalization; the practical algorithms (SGD, adaptive methods such as Adam, and distributed training) with their convergence theory; and the global picture of the loss landscape, including bad local minima, mode connectivity, the lottery-ticket hypothesis and the infinite-width limit. It is a good bridge from the practice above to theory.

**Normalization: Huang et al. [16] (IEEE TPAMI, 2023).** This survey gives a unified view of normalization methods from the perspective of optimization, and a taxonomy that splits any method into its normalization area, normalization operation and representation recovery. It covers the history from BatchNorm on, the analyses of why normalization speeds training and helps generalization, and applications in different domains. Read it to see BatchNorm, LayerNorm, group-wise and weight-based normalization as points in one design space.

**Activation functions: Dubey, Singh and Chaudhuri [7] (Neurocomputing, 2022).** This survey classifies activation functions into logistic sigmoid/tanh-based, ReLU-based, ELU-based and learning-based families, characterizes them by properties such as output range, monotonicity and smoothness, and benchmarks 18 of them across network types and data. Its practical message is that no single activation is best everywhere; the choice interacts with the architecture and the data.

**Where things stand (2023–2026).** Several convergences are visible in published model recipes:

- **Architecture.** The pre-normalized Transformer block with residual connections is the dominant architecture for language and increasingly for vision; RMSNorm and SwiGLU, as in LLaMA [12], are common defaults in large language models.
- **Optimization.** AdamW [20] with linear warmup, a cosine (or similar) decay, and gradient clipping is the standard recipe. Optimizers that look beyond per-coordinate scaling are an active area: Lion [21] uses sign updates and less memory, and Muon [22] orthogonalizes matrix updates, with reported compute-efficiency gains at scale [23]. How robust such gains are across tasks and scales is still being studied, which is why AdamW remains the safe choice.
- **Generalization.** The sharp-versus-flat-minima view [25] is a useful intuition rather than a complete theory, and why heavily over-parameterized networks generalize as well as they do is still an open question; Sun's survey [26] summarizes the main lines of attack.

**Open problems.** A reliable theory of generalization for deep networks; principled rules for choosing the learning rate, batch size and schedule together, especially when scaling models up; understanding exactly how normalization, residual connections and adaptive optimizers interact; and making training stable at very large scale without the many small tricks that current recipes rely on.

## Key takeaways

- Perceptrons use threshold decisions and mistake-driven updates; logistic regression uses smooth probabilities and a likelihood-based loss.
- Nonlinear hidden layers expand what a network can represent; approximation theorems do not guarantee successful training or generalization.
- Activation and normalization choices change gradient flow; distinguish batch statistics from per-sample statistics.
- Choose the output–loss pair for the task and check reduction factors when deriving gradients.
- Backpropagation reuses intermediate derivatives through the chain rule; numerical checks catch many implementation errors.
- Read training and validation curves together; tune optimization and regularization before attributing failure to model size alone.

## Exercises

1. **Perceptron by hand.** Run the perceptron rule on the AND data with $\eta = 1$, $\mathbf{w} = \mathbf{0}$, $b = 0$, visiting the points in the order $(0,0)$, $(0,1)$, $(1,0)$, $(1,1)$. Write down $\mathbf{w}$ and $b$ after each update until an epoch passes with no mistakes. Then explain in one sentence why the same procedure on XOR never stops.

   **Hint**

   With the strict test $z > 0$, the first point $(0,0)$ has $z = 0$, so it is predicted 0 and is correct. The first mistake is $(1,1)$, which gives $\mathbf{w} = (1,1)$, $b = 1$. Keep going: each mistake on a 0-labeled point subtracts $\mathbf{x}$ and 1, each mistake on the 1-labeled point adds them. With this order there are ten updates over five epochs, and the sixth epoch is error-free with $\mathbf{w} = (2,1)$, $b = -2$, that is, the line $2x_1 + x_2 = 2$. A different visiting order gives a different, equally valid line. For XOR, no $(\mathbf{w}, b)$ classifies all four points correctly, so some point is always wrong and an update always happens.

2. **How fast do sigmoid gradients vanish?** Show that $\sigma'(x) = \sigma(x)(1 - \sigma(x)) \leq 1/4$. Then consider a chain of 20 sigmoid layers with all weights equal to 1. What is the largest factor by which the gradient can be multiplied on its way back? Repeat the reasoning for ReLU with active units.

   **Hint**

   $s(1-s)$ with $s \in (0,1)$ is a downward parabola with maximum $1/4$ at $s = 1/2$. Twenty factors of at most $1/4$ give at most $4^{-20} \approx 9 \times 10^{-13}$. For an active ReLU unit, $f' = 1$, so the activations contribute no shrinkage at all; only the weights do, which is why initialization matters.

3. **The prediction-minus-target gradient.** Using $\partial p_k / \partial z_j = p_k(\delta_{kj} - p_j)$, prove that for $L = -\sum_k y_k \log p_k$ with a one-hot $\mathbf{y}$, $\partial L / \partial z_j = p_j - y_j$. Then show the same result for the sigmoid with binary cross-entropy.

   **Hint**

   $\partial L / \partial z_j = -\sum_k (y_k / p_k)\, p_k(\delta_{kj} - p_j) = -\sum_k y_k \delta_{kj} + p_j \sum_k y_k = -y_j + p_j$, since $\sum_k y_k = 1$. For the sigmoid, $\partial \ell / \partial \hat{p} = -y/\hat{p} + (1-y)/(1-\hat{p})$ and $\partial \hat{p} / \partial z = \hat{p}(1-\hat{p})$; multiply and simplify to $\hat{p} - y$.

4. **Extend the gradient check.** In the NumPy example, check every entry of `W2`, `b1` and `b2` (not just one entry of `W1`) against finite differences, and report the maximum relative error $|a-n| / \max(|a|, |n|, 10^{-12})$. Then deliberately introduce a bug, for example drop the `* (Z1 > 0)` factor, and see how large the error becomes.

   **Hint**

   Loop over `np.ndindex(param.shape)`, perturb one entry by $\pm 10^{-6}$, and recompute the loss. A correct implementation gives relative errors around $10^{-7}$ or smaller (a few entries next to ReLU kinks can be larger). Removing the ReLU derivative makes errors of order 1 appear in `W1` and `b1`, while `W2` and `b2` stay correct, because the bug is only in the backward path below the hidden layer.

5. **Iterations, epochs and batch size.** A dataset has 60,000 training images. (a) How many iterations make one epoch with $B = 32$, and with $B = 1{,}024$? (b) If you train both for 20 epochs, how many parameter updates does each make? (c) Based on the batch-size discussion, give two reasons the $B = 1{,}024$ run might reach a worse validation accuracy with the same learning rate, and one change that could help.

**Hint**

(a) $\lceil 60000/32 \rceil = 1875$ and $\lceil 60000/1024 \rceil = 59$. (b) 37,500 versus 1,180 updates. (c) Far fewer updates in the same number of epochs, and less gradient noise, which the sharp-minima findings associate with worse generalization. Tuning the learning rate for the new batch size (typically larger, with warmup) and training for more epochs are the usual adjustments.

6. **Diagnose the curve.** A colleague's network has a training loss that drops quickly at first and then stays flat at a high value; the validation loss tracks it closely. They propose halving the number of hidden units "to make it converge faster". What is happening, what would you check first, and what would you suggest instead?

**Hint**

Both losses are high and close together: this is underfitting (pattern C with high values), not overfitting. First check for bugs: labels aligned with inputs, the loss applied to logits, data normalized, the learning rate sensible. Then increase capacity or improve optimization (wider or deeper model, a learning-rate schedule, longer training). Shrinking the model lowers capacity and will make underfitting worse.

## References


1. Shyandram, "機器學習及類神經網路筆記 (Notes on machine learning and neural networks)," Medium post (in Traditional Chinese), 2024; updated 2026. link
2. Y. LeCun, Y. Bengio and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. doi
3. I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016. online book
4. F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, pp. 386–408, 1958. doi
5. G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, pp. 303–314, 1989. doi
6. D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986. doi
7. S. R. Dubey, S. K. Singh and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," *Neurocomputing*, vol. 503, pp. 92–108, 2022. arXiv
8. D. Hendrycks and K. Gimpel, "Gaussian error linear units (GELUs)," arXiv:1606.08415, 2016. arXiv
9. S. Elfwing, E. Uchibe and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural Networks*, vol. 107, pp. 3–11, 2018. arXiv

10. P. Ramachandran, B. Zoph and Q. V. Le, “Searching for activation functions,” arXiv:1710.05941, 2017. arXiv
11. N. Shazeer, “GLU variants improve Transformer,” arXiv:2002.05202, 2020. arXiv
12. H. Touvron, T. Lavril, G. Izacard et al., “LLaMA: Open and efficient foundation language models,” arXiv:2302.13971, 2023. arXiv
13. S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *Proc. International Conference on Machine Learning* (*ICML*), PMLR 37, pp. 448–456, 2015. PMLR
14. J. L. Ba, J. R. Kiros and G. E. Hinton, “Layer normalization,” arXiv:1607.06450, 2016. arXiv
15. B. Zhang and R. Sennrich, “Root mean square layer normalization,” in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2019. arXiv
16. L. Huang, J. Qin, Y. Zhou, F. Zhu, L. Liu and L. Shao, “Normalization techniques in training DNNs: Methodology, analysis and application,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 8, pp. 10173–10196, 2023. arXiv
17. X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proc. International Conference on Artificial Intelligence and Statistics* (*AISTATS*), PMLR 9, pp. 249–256, 2010. PMLR
18. K. He, X. Zhang, S. Ren and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification,” in *Proc. IEEE International Conference on Computer Vision* (*ICCV*), pp. 1026–1034, 2015. arXiv
19. D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2015. arXiv
20. I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2019. arXiv
21. X. Chen, C. Liang, D. Huang, E. Real, K. Wang, H. Pham, X. Dong, T. Luong, C.-J. Hsieh, Y. Lu and Q. V. Le, “Symbolic discovery of optimization algorithms,” in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2023. arXiv
22. K. Jordan, “Muon: An optimizer for hidden layers in neural networks,” Medium post, December 2024. link
23. J. Liu, J. Su, X. Yao et al., “Muon is scalable for LLM training,” arXiv:2502.16982, 2025. arXiv
24. I. Loshchilov and F. Hutter, “SGDR: Stochastic gradient descent with warm restarts,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2017. arXiv
25. N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy and P. T. P. Tang, “On large-batch training for deep learning: Generalization gap and sharp minima,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2017. arXiv
26. R. Sun, “Optimization for deep learning: Theory and algorithms,” arXiv:1912.08957, 2019. arXiv
27. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. JMLR
28. K. He, X. Zhang, S. Ren and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), pp. 770–778, 2016. arXiv
29. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*

(*NeurIPS*), 2017. arXiv
30. S. J. Reddi, S. Kale and S. Kumar, “On the Convergence of Adam and Beyond,” *ICLR*, 2018. arXiv

# ML 03 • Convolution, Image Filtering and CNNs

From blur and Sobel kernels to learned filters: convolution versus correlation, border handling, padding, stride, dilation, pooling, receptive fields, backpropagation through a convolution, and the classic CNN architectures.

Online edition: https://blog.shyandram.dev/en/learn/ml/convolutional-networks/

Notebook: https://github.com/Shyandram/cv-blog/blob/main/notebooks/image-processing/convolution.ipynb

## What you'll learn

- Explain filtering kernels and distinguish correlation from convolution.
- Implement filtering with explicit data types and border rules.
- Track channels, padding, stride, dilation and output shapes.
- Relate receptive fields, translation behavior and backpropagation.
- Compare CNN designs through their structural and computational choices.

## The big picture

An image is a grid of numbers (see Images as arrays). Many of the most useful things we can do to it are **local**: whether a pixel sits on an edge, inside a smooth region or in a noisy patch depends mostly on its neighbors. **Convolution** is the operation that formalizes "look at a small neighborhood, combine it with fixed weights, and repeat everywhere." It is the workhorse of classical image processing, and it is also the building block of **convolutional neural networks** (**CNNs**), which learn their weights from data instead of having them designed by hand.

> **Explain like I'm five**
>
> Imagine you have a small cardboard window with nine holes, and behind each hole you write a number. You place the window on a picture, multiply each visible dot of color by the number behind its hole, add everything up, and write the answer on a new sheet. Then you slide the window one step and do it again, until you have covered the whole picture. If all nine numbers are equal, the new picture is a blurry average. If the numbers on the left are negative and those on the right are positive, you get big answers only where the picture changes from dark to light: an edge finder. A CNN is a machine that slides thousands of such windows and **learns** which numbers to write behind the holes.

This tutorial has two halves. The first half (filtering) is the classical toolbox and needs only NumPy. The second half (CNNs) builds on ML 02, where you met layers, losses and backpropagation. If you want more depth on fixed filters afterward, IP 03 covers spatial filtering and IP 04 explains the frequency-domain view through the convolution theorem. For the deep-learning side, DLCV 02

studies modern classification backbones in detail, ML 04 turns to sequences, and ML 05 introduces attention, the main alternative to convolution. Standard textbook treatments are chapter 9 of Goodfellow et al. [1] and chapter 10 of Prince [10].

## Part I • Filtering with a fixed kernel

### The sliding weighted sum

**Plain version.** Every output pixel is a weighted sum of the input pixels around the same position. The small grid of weights is called a **kernel** (also *filter* or *mask*).

**Precise version.** Let $I$ be a grayscale image, with $I(x, y)$ the intensity at column $x$ and row $y$. Let $K$ be a kernel of size $(2a + 1) \times (2b + 1)$, indexed from $-a$ to $a$ vertically and $-b$ to $b$ horizontally, so that $K(0, 0)$ is its center. The filtered image $G$ is

$$G(x, y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} K(i, j)\, I(x + j,\ y + i).$$

Here $i$ indexes kernel rows and $j$ kernel columns, $G(x, y)$ is the output at the same position, and the double sum visits every tap of the kernel once. For a $3 \times 3$ kernel, $a = b = 1$ and each output pixel costs nine multiplications and eight additions.

Figure 1 works through one output value by hand. The orange window covers the top-left $3 \times 3$ neighborhood of the padded input. Multiply element-wise by the kernel and add: $1 \cdot 1 + 0 \cdot 2 + (-1) \cdot 0 + 2 \cdot 0 + 0 \cdot 1 + (-2) \cdot 3 + 1 \cdot 2 + 0 \cdot 0 + (-1) \cdot 1 = -4$. That number becomes $G$ at the window's center. Sliding the window one step to the right and repeating fills in the next output.

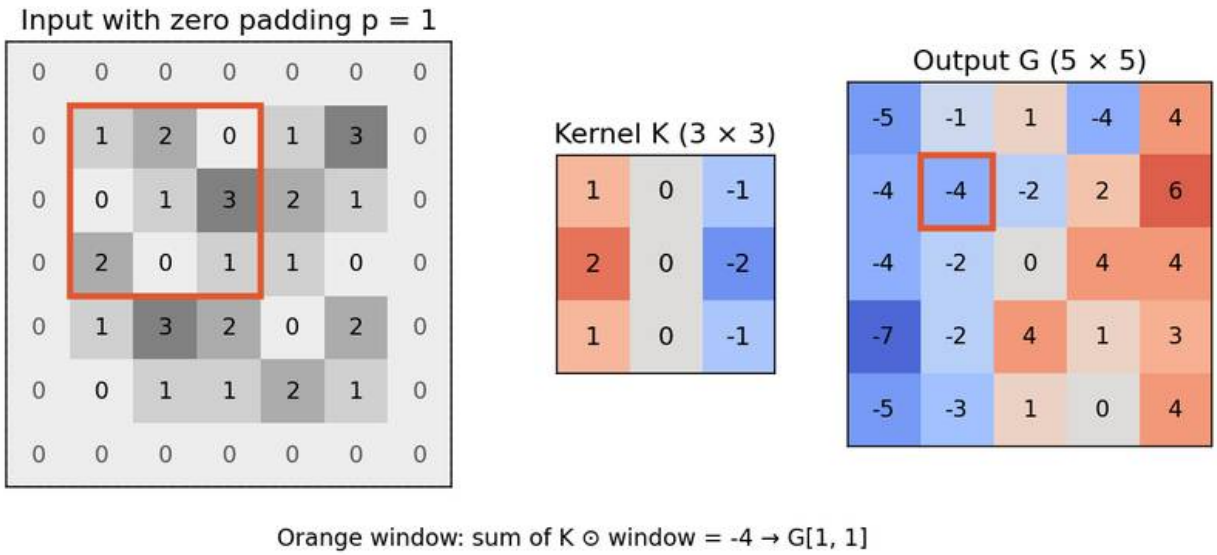



*Figure 1 — One step of the sliding weighted sum. The orange $3 \times 3$ window, multiplied element-wise by $K$ and summed, gives $-4$ at the matching output position. Zero padding (gray) lets the output keep the*

*input size.*

Two observations already matter for everything that follows. First, the same nine weights are used at every position: the operation is **shift-invariant** (also called *space-invariant*). Second, the operation is **linear**: filtering $\alpha I_1 + \beta I_2$ gives $\alpha G_1 + \beta G_2$. Linear shift-invariant (LSI) operations are exactly the ones a convolution can express, which is why convolution is so central in signal processing.

### Correlation versus convolution

**Plain version.** The formula above is technically **cross-correlation**. True **convolution** does the same thing with the kernel rotated by 180 degrees. For symmetric kernels the two agree; for asymmetric ones they differ.

**Precise version.** Mathematical convolution, written $K * I$, is

$$(K * I)(x, y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} K(i, j)\, I(x - j,\ y - i) = \sum_{i,j} K(-i, -j)\, I(x + j,\ y + i).$$

The second form shows that convolution equals correlation with the flipped kernel $K(-i, -j)$. A clean way to see the difference is to filter a single bright pixel (an **impulse**). Correlation leaves a copy of the kernel rotated by 180 degrees; convolution leaves an exact copy of the kernel. That copy is the **impulse response**, and it is why signal processing prefers convolution: the output for any image is the sum of shifted, scaled impulse responses.

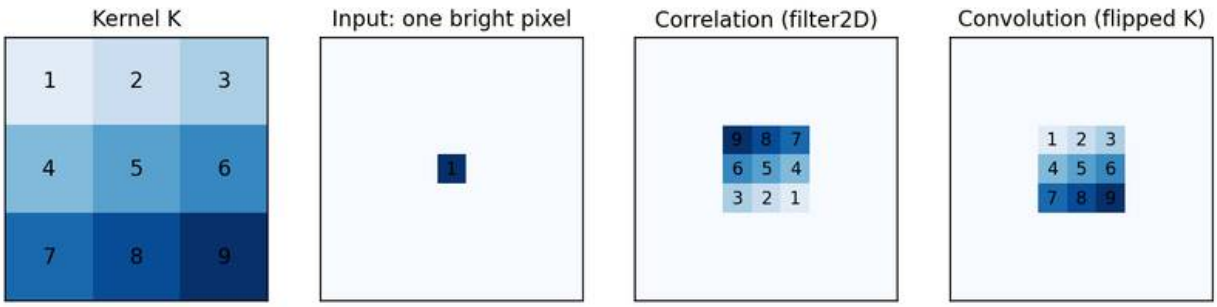



*Figure 2 — Filtering an impulse. Correlation (what cv2.filter2D computes) stamps a 180°-rotated kernel; convolution stamps the kernel itself.*

Convolution has algebraic properties that correlation lacks. It is **commutative** ($K * I = I * K$) and **associative** ($K_2 * (K_1 * I) = (K_2 * K_1) * I$), so two filters applied in a row equal one bigger filter. Applying a $3 \times 3$ box blur twice, for instance, is a single $5 \times 5$ kernel. Associativity also explains **separable** kernels: a $k \times k$ Gaussian equals a $k \times 1$ column filter followed by a $1 \times k$ row filter, cutting the cost per pixel from $k^2$ to $2k$ operations. The **convolution theorem** says convolution in space is multiplication in the Fourier domain; IP 04 develops that view.

Which libraries compute which?

| Library call | Computes | Default border |
|---|---|---|
| `cv2.filter2D` | correlation [11] | reflect-101 (`BORDER_DEFAULT`) |
| `scipy.ndimage.correlate` / `convolve` | correlation / convolution | `reflect` (edge sample repeated) |
| `scipy.signal.convolve2d` | convolution | zero fill |
| `torch.nn.functional.conv2d`, `nn.Conv2d` | correlation [31] | zero padding |

For hand-designed asymmetric kernels (Sobel, emboss, motion blur) the flip changes the sign or direction of the result, so flip deliberately when you need true convolution, for example with `cv2.flip(K, -1)`. For learned kernels the distinction disappears: a network that "should" use the flipped kernel simply learns it. That is why deep-learning frameworks call their correlation "convolution" without harm.

### Blur, edges and sharpening

**Plain version.** Positive weights that sum to one average neighbors and blur. Weights that sum to zero cancel flat regions and keep only changes, so they find edges. Adding a bit of "the image minus its blur" back to the image sharpens it.

A **box blur** gives every neighbor equal weight; the 3×3 version has nine entries of $1/9$. A **Gaussian blur** samples

$$K(i,j) = \frac{1}{Z}\exp\!\left(-\frac{i^2+j^2}{2\sigma^2}\right),$$

where $\sigma$ (in pixels) sets the blur width and $Z$ normalizes the weights to sum to one. Nearby pixels count more than distant ones, which avoids the blocky artifacts of a box. A practical rule is to use a kernel radius of about $3\sigma$.

The **sum of the weights** tells you what happens in a flat region where every pixel equals $c$: the output is $c\sum K$. A blur with sum one preserves brightness; a kernel with sum two doubles it (and saturates an 8-bit image); a kernel with sum zero outputs zero.

The **Sobel** kernels estimate horizontal and vertical derivatives while smoothing in the orthogonal direction:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}, \qquad S_y = S_x^{\top}.$$

$S_x$ responds to intensity changes along $x$, so it highlights **vertical** boundaries; $S_y$ highlights **horizontal** ones. The outputs $G_x$, $G_y$ are signed: dark-to-light and light-to-dark edges have opposite signs. The **gradient magnitude** $\sqrt{G_x^2+G_y^2}$ is a nonnegative edge strength, and $\operatorname{atan2}(G_y, G_x)$ is the edge orientation. The factorization shows that Sobel is separable: a $[1,2,1]^{\top}$ smoothing column times a $[-1,0,1]$ central difference.

The **Laplacian** kernel $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ approximates $\partial^2 I/\partial x^2 + \partial^2 I/\partial y^2$; it responds to thin lines and to both sides of an edge. **Unsharp masking** sharpens with

$$I_{\text{sharp}} = I + \lambda\,(I - G_\sigma * I),$$

where $G_\sigma * I$ is a Gaussian-blurred copy, $I - G_\sigma * I$ is the high-frequency detail, and $\lambda > 0$ controls strength. Large $\lambda$ amplifies noise and creates bright and dark halos along edges.

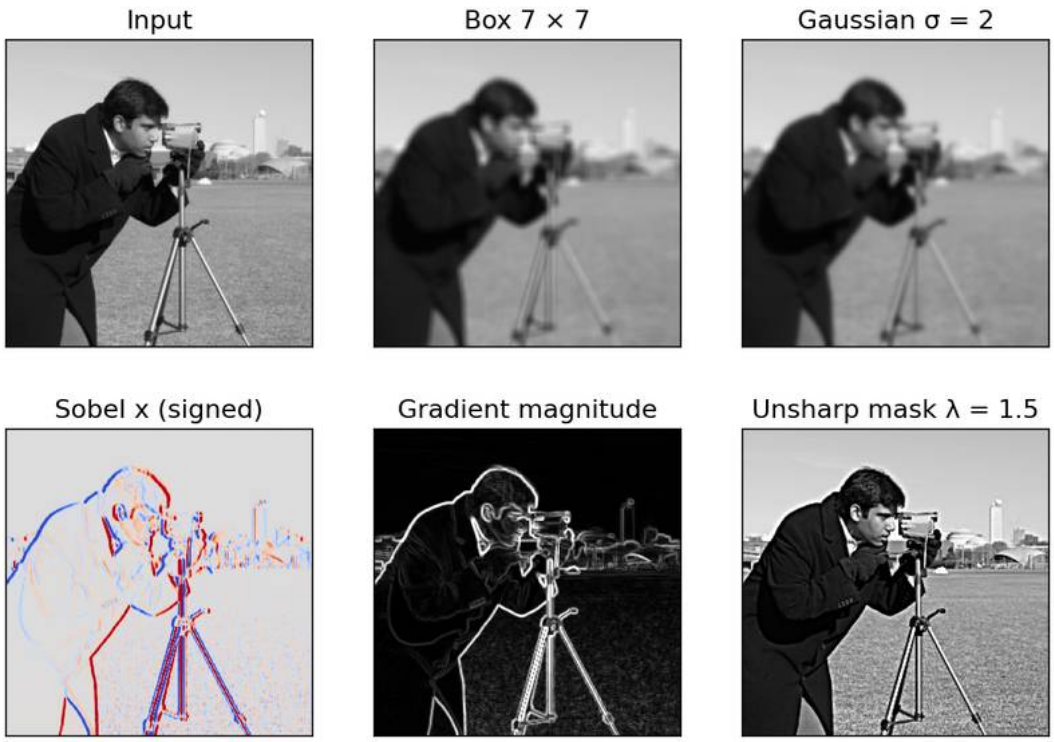



*Figure 3 — The same image through six kernels. Box and Gaussian blur suppress fine detail; Sobel x is signed (red = dark-to-light, blue = light-to-dark); the gradient magnitude outlines objects; unsharp masking adds back amplified detail.*

### Try it: the kernel playground

Edit the kernel cells below and watch the output update. Start from Sobel x, then transpose it to detect horizontal edges instead of vertical ones. Make every weight of the blur preset larger and notice that flat regions brighten, exactly as the sum-of-weights rule predicts. Try the Laplacian and emboss presets and predict their kernel sums before you look.

**Interactive demonstration** (**online**): open the online edition of this tutorial to use this demonstration.Open demonstration

The slider below compares a synthetic test image (a disc, a square, a checkerboard and stripes) with a $5 \times 5$ Gaussian blur, $\sigma = 1.4$. The fine stripes and the checkerboard disappear first because a blur is a **low-pass filter**: it removes fast intensity variations while leaving slow ones.

> **Interactive demonstration** (**online**): open the online edition of this tutorial to use this demonstration.Open demonstration

### Code: from scratch, then OpenCV, SciPy and PyTorch

The loop below is the whole algorithm. Instead of looping over every pixel (slow in Python), it loops over the kernel taps and adds one shifted copy of the padded image per tap, which NumPy vectorizes. The script then checks that four libraries agree once you match the definition (correlation versus convolution) and the border rule. It needs `numpy`, `scipy`, `opencv-python` and `torch`.

```
import numpy as np
import cv2
import torch
import torch.nn.functional as F
from scipy import ndimage, signal

def correlate2d(img, K, mode="edge"):
    """Cross-correlation of a 2-D float image with an odd-sized kernel.
    mode is any np.pad mode: "constant" (zeros), "edge", "reflect", "wrap"."""
    kh, kw = K.shape
    ph, pw = kh // 2, kw // 2
    padded = np.pad(img, ((ph, ph), (pw, pw)), mode=mode)
    H, W = img.shape
    out = np.zeros((H, W), dtype=np.float64)
    for i in range(kh):              # loop over kernel taps, not pixels:
        for j in range(kw):          # each step adds one shifted copy of the image
            out += K[i, j] * padded[i:i + H, j:j + W]
    return out

def convolve2d(img, K, mode="edge"):
    return correlate2d(img, K[::-1, ::-1], mode)     # convolution = correlation with flipped
    ↪ K

rng = np.random.default_rng(0)
img = rng.random((32, 40))
K = rng.standard_normal((3, 5))                      # deliberately asymmetric

ours_corr = correlate2d(img, K, "reflect")
ours_conv = convolve2d(img, K, "reflect")
# OpenCV: filter2D computes correlation; BORDER_REFLECT_101 is np.pad's "reflect"
cv_corr = cv2.filter2D(img, cv2.CV_64F, K, borderType=cv2.BORDER_REFLECT_101)
# SciPy: ndimage.correlate / ndimage.convolve with mode="mirror" (= reflect-101)
sp_corr = ndimage.correlate(img, K, mode="mirror")
sp_conv = ndimage.convolve(img, K, mode="mirror")
# PyTorch: conv2d is correlation; pad first with the same rule
```

```
t = torch.tensor(img)[None, None]
t = F.pad(t, (2, 2, 1, 1), mode="reflect")
pt_corr = F.conv2d(t, torch.tensor(K)[None, None])[0, 0].numpy()

for name, a in [("OpenCV", cv_corr), ("SciPy", sp_corr), ("PyTorch", pt_corr)]:
    print(f"{name:8s} correlation matches:", np.allclose(ours_corr, a))
print("SciPy    convolution matches:", np.allclose(ours_conv, sp_conv))
print("conv == corr for this kernel?", np.allclose(ours_conv, ours_corr))
# zero padding: our convolution equals signal.convolve2d with mode="same"
same = signal.convolve2d(img, K, mode="same", boundary="fill")
print("zero-padded 'same' convolution matches:", np.allclose(convolve2d(img, K, "constant"),
 ↪  same))
```

All checks print `True` except `conv == corr`, which is `False` because the random kernel is asymmetric. Note the naming trap: NumPy's `"reflect"` padding (`dcb|abcd|cba`, edge sample not repeated) is OpenCV's `BORDER_REFLECT_101` and SciPy's `"mirror"`, while SciPy's `"reflect"` repeats the edge sample (`dcba|abcd|dcba`).

The second snippet shows the two bugs that cause most beginner frustration with real images: integer saturation and the kernel-sum rule.

```
import numpy as np
import cv2
from skimage import data

img_u8 = data.camera()                                  # 512 x 512 uint8 test image
img = img_u8.astype(np.float32) / 255.0
sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], np.float32)

gx = cv2.filter2D(img, -1, sobel_x)                     # float in, float out: negatives kept
gy = cv2.filter2D(img, -1, sobel_x.T)
mag = np.hypot(gx, gy)
print("float Sobel x range:", gx.min().round(3), gx.max().round(3))

gx_u8 = cv2.filter2D(img_u8, -1, sobel_x)               # uint8 in, uint8 out: saturated
print("uint8 Sobel x range:", gx_u8.min(), gx_u8.max(),
      "| pixels clipped to 0:", f"{(gx_u8 == 0).mean():.0%}")

blur = cv2.GaussianBlur(img, (5, 5), 1.4)               # 5 x 5 Gaussian, sigma 1.4
print("mean brightness kept by blur:", np.isclose(img.mean(), blur.mean(), atol=1e-3))
K2 = 2 * np.ones((3, 3), np.float32) / 9                # weights sum to 2
print("mean after sum-2 kernel:", round(float(cv2.filter2D(img, -1, K2).mean() / img.mean()),
 ↪  3))
```

In floating point the Sobel response ranges from about $-3.4$ to 3.3. With `uint8` input and output, every negative response is clipped to zero (more than half the pixels here), so all light-to-dark edges vanish. Always filter in `float32` and convert back only for display.

### Border handling

**Plain version.** Near the edge of the image, part of the kernel hangs outside. You must decide what values to pretend are there.

The common choices are: **zero** (constant) padding, which pretends the world outside is black; **replicate**, which repeats the edge pixel; **reflect**, which mirrors the image at its border; and **wrap**, which treats the image as periodic, as the discrete Fourier transform implicitly does. OpenCV's default, `BORDER_REFLECT_101`, mirrors without repeating the edge pixel [11]. PyTorch's `nn.Conv2d` uses zero padding unless you set `padding_mode` to `'reflect'`, `'replicate'` or `'circular'` [31].

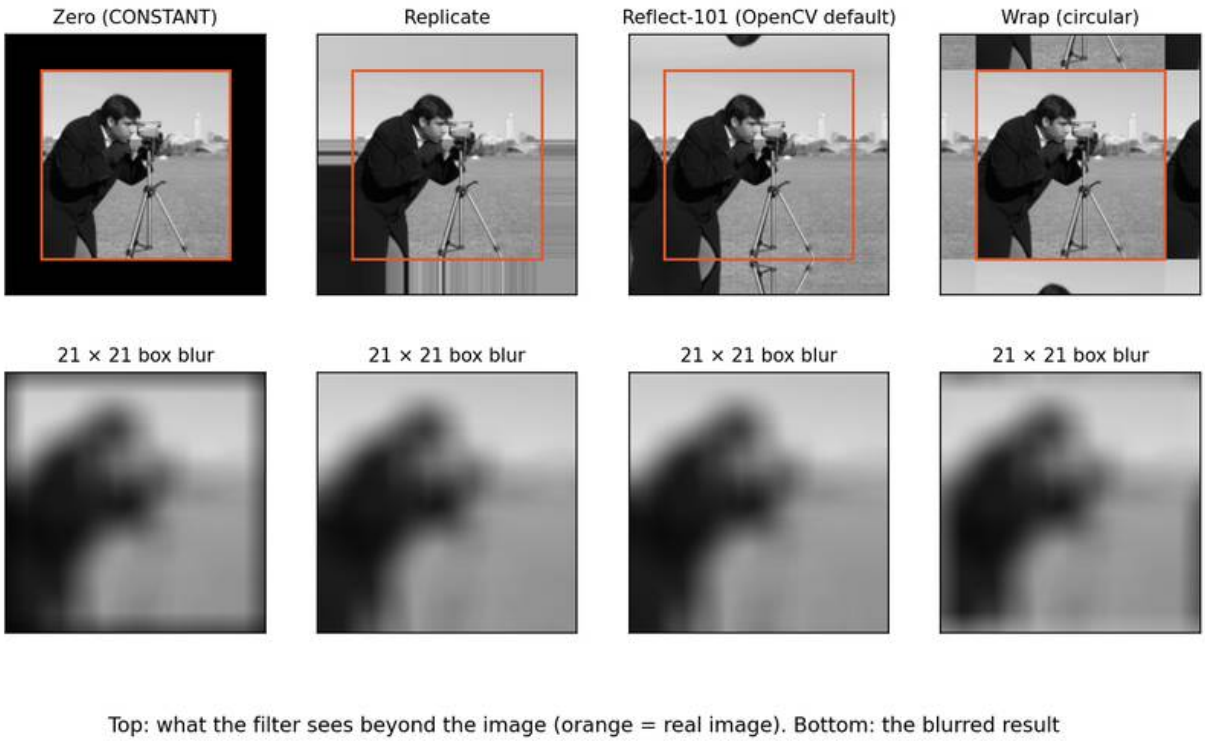



*Figure 4 — Four border rules and their effect on a large 21 × 21 box blur. Zero padding darkens every border; wrap blends opposite edges; replicate and reflect keep flat borders flat.*

Zero padding darkens the borders of a blurred image because black pixels are averaged in. Wrap mixes the top of the image into the bottom. Replicate and reflect are usually the least visible choices for photographs. With small kernels the difference touches only a thin frame, but in a deep CNN every layer adds its own frame: after many zero-padded layers, a large fraction of the feature map has "seen" the artificial border, which lets networks infer absolute position from padding.

# Part II • Convolutional neural networks

## From a fixed filter to a learned layer

**Plain version.** A standard dense convolution learns filters spanning all input channels, each producing one output feature map. Grouped convolutions restrict that channel access.

**Precise version.** Let the input be $X \in \mathbb{R}^{C_{\text{in}} \times H \times W}$ (channels, height, width) and the weights $K \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times k \times k}$ with bias $b \in \mathbb{R}^{C_{\text{out}}}$. A stride-one layer with zero padding $p$ computes

$$Y_{o,u,v} = b_o + \sum_{c=1}^{C_{\text{in}}} \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} K_{o,c,i,j}\, X_{c,\, u+i-p,\, v+j-p}.$$

$o$ indexes output channels (filters), $c$ input channels, $(u, v)$ output positions and $(i, j)$ kernel taps; positions outside the image read the padding value. A nonlinearity such as ReLU is then applied element-wise. An RGB filter has three $k \times k$ slices, one per color channel, and their responses are **summed** into a single output number per position: a filter does not produce three separate images.

Two structural assumptions are built into this formula:

- **Local connectivity**: each output depends only on a $k \times k$ neighborhood, not on the whole image.
- **Weight sharing** (parameter sharing): the same $K$ is used at every position $(u, v)$.

Together they are the CNN's **inductive bias**, an assumption about which functions are likely. Its payoff is efficiency. A layer with $C_{\text{in}} = 3$, $C_{\text{out}} = 32$ and $k = 3$ has $32 \times (3 \cdot 9 + 1) = 896$ parameters, whatever the image size. A fully connected layer from a flattened $224 \times 224 \times 3$ image to just 32 numbers needs $150{,}528 \times 32 \approx 4.8$ million weights, and it would have to learn the same edge detector separately at every position.

Goodfellow et al. [1] summarize the motivation as sparse interactions, parameter sharing and equivariance. Figure 5 shows what a tiny CNN learns in its first layer when trained on the $8 \times 8$ digits dataset used in the classifier code later on. The learned $3 \times 3$ filters look like oriented derivative and blob detectors: the network rediscovered Sobel-like kernels because they are useful, not because anyone told it to. Zeiler and Fergus [30] showed the same phenomenon at scale by visualizing the filters and feature maps of large ImageNet networks.

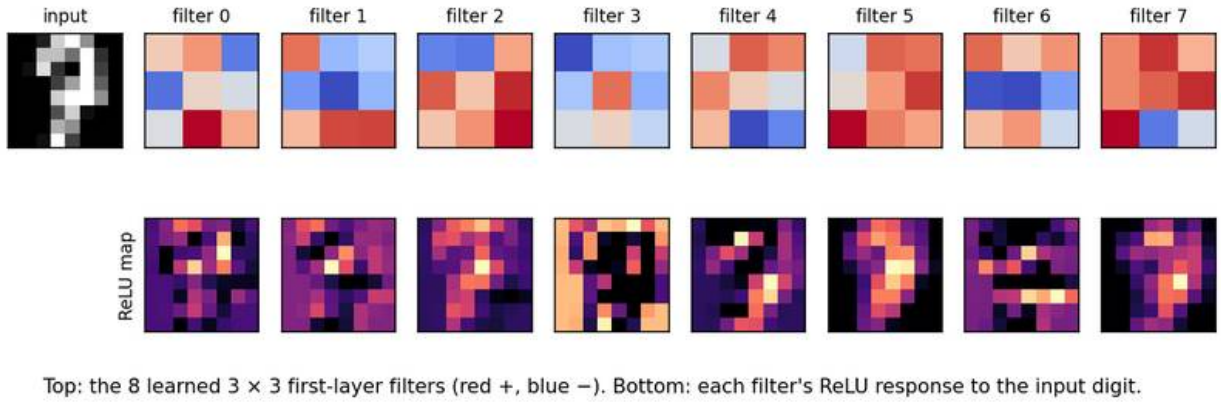



*Figure 5 — First-layer filters learned on 8 × 8 digits (top) and their ReLU responses to one test digit (bottom). Many look like oriented edge or stroke detectors.*

### Padding, stride and dilation

**Plain version.** Padding controls what happens at the border and whether the output keeps its size. Stride skips positions to shrink the output. Dilation spreads the kernel taps apart to see farther without adding weights.

For input size $n$ (height or width), kernel size $k$, padding $p$ on each side, stride $s$ and dilation $d$, the output size is [12]

$$n_{\text{out}} = \left\lfloor \frac{n + 2p - d(k-1) - 1}{s} \right\rfloor + 1.$$

$d(k-1)+1$ is the **effective kernel size**: dilation $d$ inserts $d-1$ gaps between taps. Three padding conventions have names: **valid** ($p = 0$, output shrinks by $k-1$), **same** ($p = (k-1)/2$ for odd $k$ and $s = 1$, output keeps its size) and **full** ($p = k-1$, every partial overlap counted, output grows by $k-1$). Stride two roughly halves each spatial dimension and is a common alternative to pooling for downsampling.

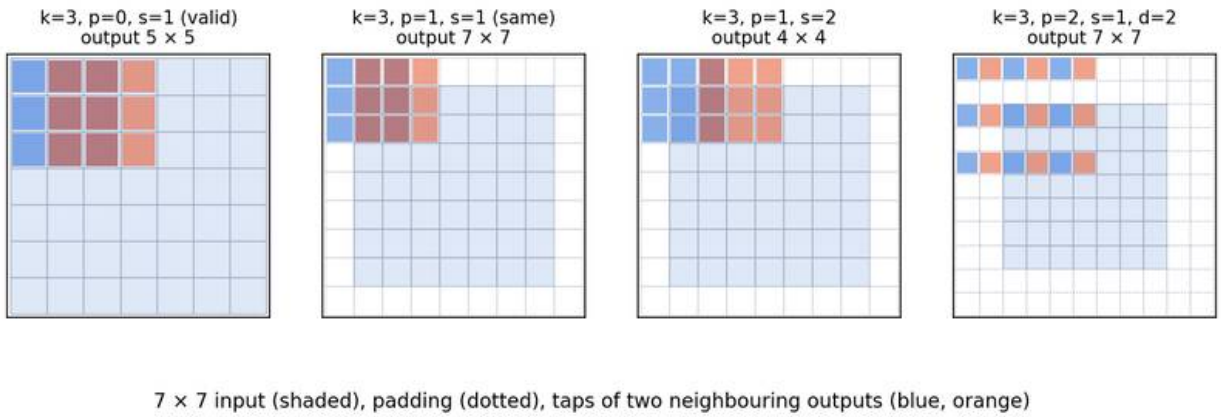


*Figure 6 — Taps of two neighboring outputs (blue and orange; overlap appears brown) under four settings. Stride moves the window by s; dilation spaces the taps d apart.*

**Dilated (atrous) convolutions** were popularized for dense prediction by Yu and Koltun [18]: stacking layers with dilations 1, 2, 4, ... grows the field of view exponentially while keeping full resolution. A single dilated layer sees only a sparse lattice of inputs, so dilations should be mixed with care to avoid "gridding" artifacts. A **transposed convolution** runs the forward map of a strided convolution backward to upsample; it reappears in the backpropagation section below. The convolution arithmetic guide by Dumoulin and Visin [12] has diagrams for every combination of these settings.

The snippet checks the formula against PyTorch, counts parameters, shows that a 1×1 convolution is a per-pixel linear layer, and measures translation equivariance:

```
import torch
from torch import nn

def out_size(n, k, p=0, s=1, d=1):
    return (n + 2 * p - d * (k - 1) - 1) // s + 1

x = torch.randn(1, 3, 64, 64)                          # one RGB 64 x 64 image
for k, p, s, d in [(3, 1, 1, 1), (5, 2, 2, 1), (3, 2, 1, 2), (1, 0, 1, 1)]:
    conv = nn.Conv2d(3, 16, k, stride=s, padding=p, dilation=d)
    y = conv(x)
    n_params = sum(q.numel() for q in conv.parameters())
    assert y.shape[-1] == out_size(64, k, p, s, d)
    assert n_params == 16 * (3 * k * k + 1)
    print(f"k={k} p={p} s={s} d={d}: output {tuple(y.shape[1:])}, parameters {n_params}")

# A 1x1 convolution is the same linear layer applied at every pixel
conv1 = nn.Conv2d(16, 4, kernel_size=1)
lin = nn.Linear(16, 4)
lin.weight.data = conv1.weight.data[:, :, 0, 0].clone()
lin.bias.data = conv1.bias.data.clone()
z = torch.randn(2, 16, 8, 8)
same = torch.allclose(conv1(z), lin(z.permute(0, 2, 3, 1)).permute(0, 3, 1, 2), atol=1e-6)
print("1x1 conv == per-pixel Linear:", same)

# Translation equivariance away from borders (stride 1, no padding effects in the centre)
conv = nn.Conv2d(1, 4, 3, padding=1, bias=False)
img = torch.zeros(1, 1, 32, 32); img[..., 10:14, 8:12] = 1
shift = lambda t: torch.roll(t, shifts=(3, 5), dims=(-2, -1))
with torch.no_grad():
    a, b = conv(shift(img)), shift(conv(img))
print("shift then conv == conv then shift:", torch.allclose(a, b, atol=1e-6))
conv2 = nn.Conv2d(1, 4, 3, padding=1, stride=2, bias=False)
with torch.no_grad():
    y0, y1 = conv2(img), conv2(torch.roll(img, 1, dims=-1))   # shift by one pixel
print("stride 2, 1-pixel shift: max output change =", round((y0 - y1).abs().max().item(), 3))
```

It prints output shapes 16 × 64 × 64, 16 × 32 × 32, 16 × 64 × 64 and 16 × 64 × 64 with 448, 1216, 448 and 64 parameters, confirms the 1 × 1 and equivariance checks, and shows that a one-pixel

shift changes a stride-two output by a clearly nonzero amount; the exact value depends on the randomly initialized weights.

### Channels and 1×1 convolutions

**Plain version.** Channels are the "what" dimension of a feature map; height and width are the "where". A $1 \times 1$ convolution mixes channels at each pixel without looking at neighbors.

A $1 \times 1$ convolution with $C_{\text{in}}$ inputs and $C_{\text{out}}$ outputs multiplies the channel vector at every pixel by the same $C_{\text{out}} \times C_{\text{in}}$ matrix, as the snippet verified. Lin et al.'s Network in Network [17] used this idea as a tiny per-pixel MLP and introduced **global average pooling** as a classifier head. GoogLeNet [16] used $1 \times 1$ layers to reduce channels before expensive $3 \times 3$ and $5 \times 5$ branches, and ResNet's bottleneck block [5] wraps a $3 \times 3$ layer between a $1 \times 1$ reduction and a $1 \times 1$ expansion. In every case the motivation is the same: spatial filtering costs $k^2$ times more than channel mixing, so do the cheap part with $1 \times 1$ layers and keep the expensive part narrow.

### Pooling and downsampling

**Plain version.** Pooling summarizes each small block of a feature map by a single number, usually the maximum or the average. It shrinks the map and makes the representation less sensitive to small shifts.

**Max pooling** with a $2 \times 2$ window and stride two keeps the largest value in each block: "was this feature present anywhere here?". **Average pooling** keeps the mean. **Global average pooling** averages each channel over all positions, producing one number per channel regardless of input size, which is why modern classifiers can accept different image sizes. Pooling has no learnable parameters. Its gradient is simple: max pooling routes the whole upstream gradient to the winning input of each window; average pooling spreads it equally over the window.

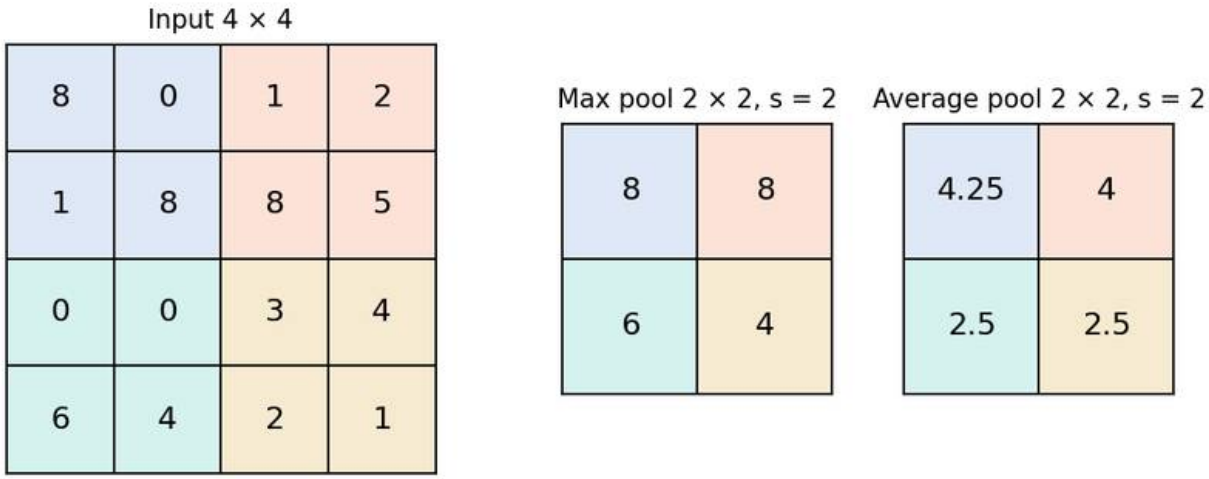


*Figure 7 — Max and average pooling with a 2 × 2 window and stride 2. Each colored block becomes one number.*

Downsampling, by pooling or by strided convolution, saves computation and enlarges later receptive fields, but it is also **subsampling** without a proper low-pass filter. High frequencies alias, and the output can change noticeably when the input moves by a single pixel, as the snippet above measured. Zhang [20] showed that inserting a small blur before subsampling ("anti-aliased" or "blur pooling") makes standard CNNs measurably more shift-invariant: a neat case of a classical signal-processing rule fixing a deep network.

### Equivariance is not invariance

**Plain version.** A stride-one convolution shifts with its input away from boundary effects. This equivariance does not make the prediction invariant to translation.

Let $T_\Delta$ shift an image by $\Delta$ pixels. Away from the borders, a stride-one convolution is **translation-equivariant**: $f(T_\Delta X) = T_\Delta f(X)$. A classifier wants **invariance**: $g(T_\Delta X) = g(X)$. CNNs get approximate invariance by stacking equivariant layers and then pooling, especially global average pooling, which discards position. Padding, strides, cropping and the finite image break exact equivariance. Rotation and scale equivariance are **not** built in; data augmentation or specially designed group-equivariant layers must supply them.

The distinction matters in practice. A detector must be equivariant: moving a person should move the predicted box. A classifier should be invariant: moving the person should not change the label. The same backbone serves both, with different heads.

### Receptive field

**Plain version.** Early layers see small pieces of the image; deeper layers combine those pieces and see larger regions.

The **receptive field** of a unit is the set of input pixels that can influence it. Let $r_\ell$ be its width after layer $\ell$, and $j_\ell$ the **jump**, the distance in input pixels between adjacent units of layer $\ell$. Starting from $r_0 = j_0 = 1$, a layer with kernel size $k_\ell$, dilation $d_\ell$ and stride $s_\ell$ gives

$$r_\ell = r_{\ell-1} + (d_\ell(k_\ell - 1))\, j_{\ell-1}, \qquad j_\ell = s_\ell\, j_{\ell-1}.$$

Three stacked stride-one $3 \times 3$ layers see $7 \times 7$ pixels, the same area as one $7 \times 7$ layer but with $3 \cdot 9C^2 = 27C^2$ instead of $49C^2$ weights and three nonlinearities instead of one; this was the central design argument of VGG [15]. A stride-two layer doubles the jump, so every later layer grows the field twice as fast.

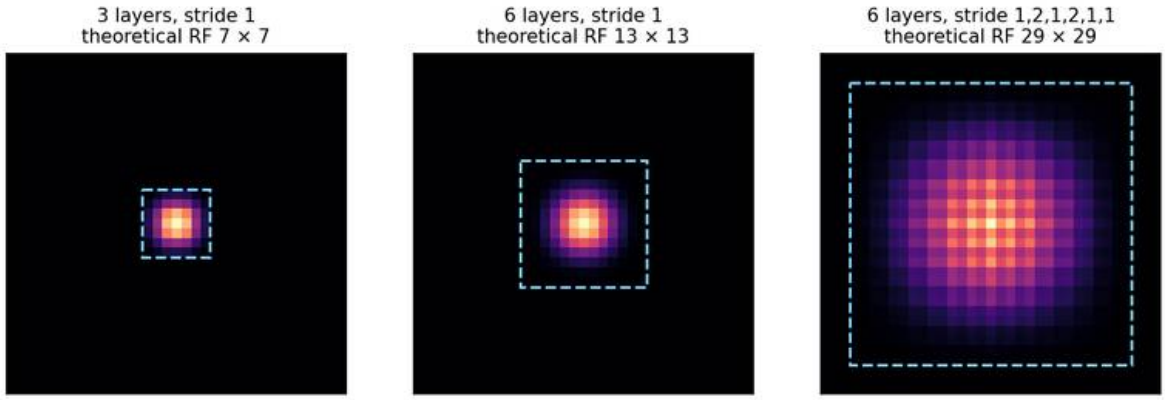



*Figure 8 — Theoretical receptive field (dashed) versus actual influence, measured as the input gradient of one central output through stacks of 3 × 3 averaging layers. Influence is Gaussian-like and concentrated in the center; strides add a checkerboard pattern.*

The theoretical field says which pixels **could** matter. Luo et al. [3] measured the **effective receptive field** via input gradients, as in Figure 8, and found that it is roughly Gaussian and occupies only a fraction of the theoretical field, growing like the square root of depth for uniform layers. Two practical lessons follow: a network may need more context than its theoretical field suggests, and downsampling, dilation and large kernels are ways to enlarge the effective field without unbounded depth.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

### Backpropagation through a convolution

**Plain version.** Kernel gradients combine inputs with upstream gradients. Input gradients use the transpose of the forward linear operator, with matching stride and padding conventions.

**Precise version.** Take one channel, no padding, stride one: $Y_{u,v} = \sum_{i,j} K_{i,j} X_{u+i,v+j}$. Let $L$ be the loss and $G_{u,v} = \partial L / \partial Y_{u,v}$ the **upstream gradient** delivered by backpropagation, as in ML 02. Each weight $K_{i,j}$ is used at every output position, so by the chain rule its gradient sums over them:

$$\frac{\partial L}{\partial K_{i,j}} = \sum_{u,v} G_{u,v}\, X_{u+i,v+j}.$$

That is a valid correlation of the input $X$ with $G$, which is why weight sharing does not complicate training: shared weights simply accumulate gradients from every location. Each input pixel $X_{m,n}$ contributes to every output whose window covers it, at tap $(i,j) = (m-u, n-v)$:

$$\frac{\partial L}{\partial X_{m,n}} = \sum_{i,j} G_{m-i,\,n-j}\, K_{i,j},$$

with $G$ taken as zero outside its range. This is a **full convolution** of $G$ with $K$, equivalently a correlation of the zero-padded $G$ with the flipped kernel. It is exactly what a **transposed convolution** computes, which is how transposed convolutions got their name. Bias gradients are just $\sum_{u,v} G_{u,v}$ per output channel. With multiple channels, the input gradient also sums over output channels, and the kernel gradient also sums over the batch.

```
import torch
import torch.nn.functional as F

torch.manual_seed(0)
X = torch.randn(1, 1, 6, 7, requires_grad=True)     # input
K = torch.randn(1, 1, 3, 3, requires_grad=True)     # kernel
Y = F.conv2d(X, K)                                   # 'valid' correlation: 4 x 5
G = torch.randn_like(Y)                              # pretend upstream gradient dL/dY
(Y * G).sum().backward()                             # so that dL/dY = G exactly

# 1) dL/dK: correlate the input with the upstream gradient
dK = F.conv2d(X.detach(), G)
# 2) dL/dX: 'full' convolution of G with K = correlation of zero-padded G with flipped K
dX = F.conv2d(F.pad(G, (2, 2, 2, 2)), K.detach().flip(-1, -2))
# 3) the same thing is what a transposed convolution computes
dX_t = F.conv_transpose2d(G, K.detach())

print("dL/dK matches autograd:", torch.allclose(dK, K.grad, atol=1e-5))
print("dL/dX matches autograd:", torch.allclose(dX, X.grad, atol=1e-5))
print("conv_transpose2d gives the same:", torch.allclose(dX_t, X.grad, atol=1e-5))
```

All three lines print `True`. The trick `(Y * G).sum().backward()` makes autograd use a chosen upstream gradient $G$, a handy way to test any hand-derived backward pass.

## Classic architectures in brief

**Plain version.** Every famous CNN is a stack of convolutions, nonlinearities and downsampling. What changed over the years was depth, how to keep deep stacks trainable, and how to spend computation wisely.

| Model (year) | Key idea | What it taught |
|---|---|---|
| LeNet-5 (1998) [13] | conv → subsample → conv → subsample → fully connected, trained end to end | Learned convolutional features beat hand-designed ones for digits |
| AlexNet (2012) [14] | Deeper and wider, ReLU, dropout, data augmentation, GPU training | Large labeled data plus GPUs made CNNs win ImageNet |
| VGG (2014) [15] | Only 3 × 3 convolutions, 16–19 layers | Depth with small kernels is simple and effective |

| Model (year) | Key idea | What it taught |
|---|---|---|
| GoogLeNet / Inception (2014) [16] | Parallel $1 \times 1$, $3 \times 3$, $5 \times 5$ branches with $1 \times 1$ reductions | Multi-scale features at controlled cost |
| ResNet (2015) [5] | Residual connections $y = x + F(x)$, BatchNorm | Very deep networks (100+ layers) become trainable |
| U-Net (2015) [4] | Encoder–decoder with skip connections at each resolution | Dense, pixel-aligned outputs from few images |
| MobileNetV2 (2018) [7] | Depthwise-separable convolutions, inverted residuals, linear bottlenecks | Accurate models for phones |
| EfficientNet (2019) [8] | Compound scaling of depth, width and resolution | Scale dimensions together, not one at a time |
| ConvNeXt (2022) [9] | $7 \times 7$ depthwise kernels, LayerNorm, inverted bottlenecks, transformer-era training | A modernized pure CNN competes with vision transformers |

The pattern from LeNet to ResNet is a steady increase in depth. AlexNet [14] showed that a deep CNN trained on ImageNet with GPUs, ReLU and dropout could beat traditional pipelines by a wide margin. VGG [15] argued for uniform stacks of 3×3 layers, as computed in the receptive-field section. GoogLeNet [16] made width cheap with 1×1 reductions. Then depth hit an optimization wall: plain networks with more layers had **higher training error**, which is not overfitting but a failure to optimize. ResNet [5] fixed it with identity shortcuts. DLCV 02 follows this line further, through training recipes, transfer learning and evaluation.

### Residual blocks and normalization

**Plain version.** Instead of asking a block to compute the whole output, ask it to compute a correction to its input.

A residual block computes $y = x + F_\theta(x)$, where $F_\theta$ is typically two or three convolutions with normalization and ReLU. Its Jacobian is $\partial y/\partial x = I + \partial F_\theta/\partial x$: even if $F_\theta$ contributes little, the identity path carries the signal forward and the gradient backward. If $F_\theta \approx 0$ is the best a block can do, it can learn that easily, so adding blocks no longer hurts optimization. When the shape changes (more channels, stride two), the shortcut becomes a projection $P(x)$, usually a strided $1 \times 1$ convolution.

**Batch normalization** [19] standardizes each channel using the batch mean and variance during training, then applies a learned scale and shift; at evaluation it uses running averages. It makes training faster and less sensitive to initialization, but it misbehaves with very small or correlated batches, where **GroupNorm** [6] (statistics over groups of channels within each sample) is more reliable. Always call `model.eval()` before validation so BatchNorm uses its running statistics.

```
import torch
from torch import nn

count = lambda m: sum(p.numel() for p in m.parameters())

C = 64
```

```
dense = nn.Conv2d(C, C, 3, padding=1, bias=False)
separable = nn.Sequential(
    nn.Conv2d(C, C, 3, padding=1, groups=C, bias=False),   # depthwise: one 3x3 per channel
    nn.Conv2d(C, C, 1, bias=False))                        # pointwise: mix channels
print("dense:", count(dense), "| depthwise+pointwise:", count(separable),
      "| ratio:", round(count(separable) / count(dense), 4), "vs 1/C + 1/9 =",
      round(1 / C + 1 / 9, 4))

class BasicBlock(nn.Module):
    """ResNet-style block: y = relu(x + F(x)), with a 1x1 projection when shapes change."""
    def __init__(self, c_in, c_out, stride=1):
        super().__init__()
        self.f = nn.Sequential(
            nn.Conv2d(c_in, c_out, 3, stride, 1, bias=False), nn.BatchNorm2d(c_out),
            ↪  nn.ReLU(),
            nn.Conv2d(c_out, c_out, 3, 1, 1, bias=False), nn.BatchNorm2d(c_out))
        self.proj = (nn.Identity() if stride == 1 and c_in == c_out else
                     nn.Sequential(nn.Conv2d(c_in, c_out, 1, stride, bias=False),
                                   nn.BatchNorm2d(c_out)))
    def forward(self, x):
        return torch.relu(self.proj(x) + self.f(x))

x = torch.randn(2, 64, 32, 32)
print("same shape :", tuple(BasicBlock(64, 64)(x).shape))
print("downsample :", tuple(BasicBlock(64, 128, stride=2)(x).shape))
```

The output reports 36,864 weights for the dense layer against 4,672 for the separable pair, a ratio of 0.1267 that matches the formula below, and block outputs of shape $64 \times 32 \times 32$ and $128 \times 16 \times 16$.

### Efficient CNNs separate spatial filtering from channel mixing

**Plain version.** Looking around (spatial filtering) and combining features (channel mixing) need not happen in one expensive operation.

A dense $k \times k$ convolution has $k^2 C_{\text{in}} C_{\text{out}}$ weights. A **depthwise** convolution filters each channel separately ($k^2 C_{\text{in}}$ weights, `groups=C` in PyTorch); a following **pointwise** $1 \times 1$ convolution mixes channels ($C_{\text{in}} C_{\text{out}}$ weights). With $C_{\text{in}} = C_{\text{out}} = C$,

$$\frac{\text{depthwise} + \text{pointwise}}{\text{dense}} = \frac{k^2 C + C^2}{k^2 C^2} = \frac{1}{C} + \frac{1}{k^2}.$$

For $k = 3$ and large $C$ this approaches $1/9$.

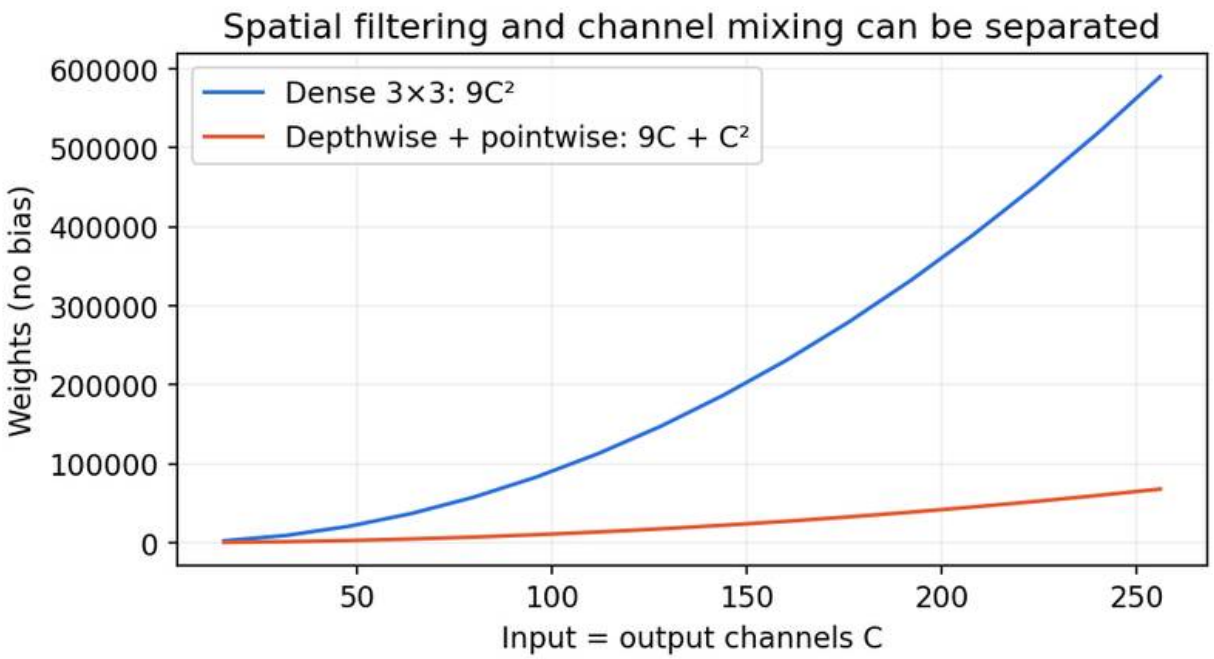



*Figure 9 — Analytical weight counts for equal input and output channels. Fewer weights do not automatically mean lower latency: depthwise layers have low arithmetic intensity and may run slowly on some hardware.*

MobileNetV2 [7] builds blocks that expand channels with a $1 \times 1$ layer, filter with a depthwise $3 \times 3$, then project back with a linear (no ReLU) $1 \times 1$ bottleneck, with a residual connection between the narrow ends. EfficientNet [8] scales depth, width and input resolution together with a single compound coefficient. ConvNeXt [9] started from a ResNet and applied, one at a time, design choices associated with vision transformers: patchify stem, $7 \times 7$ depthwise kernels, inverted bottlenecks, fewer activations, LayerNorm, and longer training with heavy augmentation. The result matched Swin Transformers of similar size, evidence that much of the transformer advantage came from training recipes and macro-design rather than from attention alone.

*Online only: ResNet — learning a residual instead of a whole mapping — https://blog.shyandram.dev/en/learn/ml/convolutional-networks/papers/resnet–learning-a-residual-instead-of-a-whole-mapping/*

## Code: a complete small-image classifier

This example needs `torch`, `numpy` and `scikit-learn`. It uses the bundled $8 \times 8$ digits dataset, so no download or GPU is needed. A stratified split gives separate training, validation and test sets: validation picks the best epoch, and the test set is evaluated once at the end. It is a teaching baseline, not an ImageNet recipe.

```
import copy
import numpy as np
```

```
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

torch.manual_seed(0)
torch.set_num_threads(1)
d = load_digits()
X = torch.tensor(d.images[:, None] / 16.0, dtype=torch.float32)
y = torch.tensor(d.target, dtype=torch.long)
train, test = train_test_split(np.arange(len(y)), test_size=.2,
                               stratify=d.target, random_state=0)
train, val = train_test_split(train, test_size=.2,
                              stratify=d.target[train], random_state=1)
model = nn.Sequential(
    nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(),
    nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(),
    nn.MaxPool2d(2),                      # B × 32 × 4 × 4
    nn.Flatten(), nn.Linear(32 * 4 * 4, 10),
)
opt = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
best_loss, best_state = float("inf"), None
for epoch in range(12):
    model.train()
    order = torch.tensor(train)[torch.randperm(len(train))]
    for idx in order.split(64):
        loss = nn.functional.cross_entropy(model(X[idx]), y[idx])
        opt.zero_grad()
        loss.backward()
        opt.step()
    model.eval()
    with torch.no_grad():
        val_loss = nn.functional.cross_entropy(model(X[val]), y[val]).item()
    if val_loss < best_loss:
        best_loss = val_loss
        best_state = copy.deepcopy(model.state_dict())
model.load_state_dict(best_state)
model.eval()
with torch.no_grad():
    accuracy = (model(X[test]).argmax(1) == y[test]).float().mean().item()
print("parameters:", sum(p.numel() for p in model.parameters()))
print("best validation loss:", round(best_loss, 4))
print("held-out accuracy:", round(accuracy, 4))
```

On our CPU run it reports 9,930 parameters and about 96% held-out accuracy (exact numbers vary slightly across PyTorch versions). Cross-entropy receives **logits**, with no softmax in front. The head flattens the $32 \times 4 \times 4$ map rather than global-average-pooling it, because on tiny digits the position of strokes carries information; for variable-size images, adaptive or global pooling gives a fixed-size head.

Inspect the errors, not just the score. Which digit pairs get confused? Does shifting every test digit by one pixel hurt? Compare against a parameter-matched MLP and a classical SVM, as in IP 13, before crediting every gain to convolution.

## Modern view

**Survey literature.** Several reviews map the CNN landscape, each with a different emphasis. Gu et al. [23] (*Pattern Recognition*, 2018) organize CNN advances by component: convolution variants, pooling, activations, loss functions, regularization and optimization, followed by applications. Khan et al. [22] (*Artificial Intelligence Review*, 2020) propose a taxonomy of architectures by their main innovation: spatial exploitation (kernel sizes), depth, multi-path designs (shortcuts, Inception branches), width, feature-map exploitation (channel attention), and attention-based designs. Li et al. [21] (*IEEE TNNLS*, 2022) cover the history from LeNet onward, compare one-, two- and multi-dimensional convolutions, and survey lightweight networks and applications with rules of thumb for choosing hyperparameters. Alzubaidi et al. [2] (*Journal of Big Data*, 2021) give a broad, application-oriented review of deep learning centered on CNN architectures, with discussion of data scarcity, transfer learning and hardware. These surveys predate ConvNeXt, so read them as maps of the 2012–2020 era.

**CNNs after transformers (2022–2026).** Vision transformers (ML 05) prompted a re-examination of convolution, and the main finding is that the gap was smaller than early comparisons suggested:

- **Modernized CNNs.** ConvNeXt [9] showed that a carefully modernized ResNet matches Swin. ConvNeXt V2 [24] (CVPR 2023) added masked-autoencoder pretraining with sparse convolutions and a global response normalization layer, bringing self-supervised pretraining (see ML 06) to pure CNNs.
- **Large kernels.** RepLKNet [26] (CVPR 2022) used depthwise kernels as large as $31 \times 31$, with reparameterized small kernels to ease optimization, to enlarge the effective receptive field. UniRepLKNet [27] (CVPR 2024) extended large-kernel design to audio, video, point clouds and time series.
- **Dynamic and deformable convolution.** InternImage [25] (CVPR 2023) scaled deformable convolution, whose sampling locations are predicted from the input, into a large foundation backbone for detection and segmentation.
- **Scale.** Smith et al. [29] (2023) pretrained NFNet CNNs on the large JFT-4B image dataset and found that their performance followed similar compute scaling trends to vision transformers under comparable budgets, arguing that compute and data, more than architecture, dominate at scale.
- **Edge devices.** MobileNetV4 [28] (ECCV 2024) designed blocks and a neural architecture search for good latency across CPUs, GPUs, DSPs and accelerators, an acknowledgment that parameter counts alone do not predict speed.

**Where CNNs fit in 2026.** Convolution remains a default for latency-constrained and on-device inference, for dense prediction at high resolution (restoration, segmentation, medical imaging, see DLCV 09), for small datasets where the locality bias helps, and as the stem or early stages of hybrid CNN–transformer models. Large vision foundation models are mostly transformer-based, but they frequently contain convolutions in their patch embeddings, upsampling decoders and diffusion U-Nets.

**Open problems.** Exact shift equivariance under downsampling (aliasing [20]); building rotation

and scale symmetry in without heavy cost; understanding what the effective receptive field really needs to be for a task [3]; and fair comparison across architectures, which requires matching data, augmentation, resolution, schedule and compute before attributing a difference to convolution versus attention.

## Key takeaways

- Convolution flips the kernel; correlation does not. Check the library convention when reproducing a filter.
- Kernel sums determine the response to constant input, but a sum of one alone does not imply smoothing.
- Learned filters share weights spatially; channel grouping determines which inputs each filter reads.
- Padding, stride and dilation determine shapes and sampling; downsampling may introduce aliasing.
- Translation equivariance differs from invariance; boundaries, stride and pooling affect the actual behavior.
- Separate spatial filtering, channel mixing and skip connections when comparing architectures; measure efficiency at matched budgets.

## Exercises

1. **Design and verify.** Write a $3 \times 3$ kernel that detects only horizontal edges, and another that detects only diagonal edges running from top-left to bottom-right. Verify both in the kernel playground or with `cv2.filter2D` on a synthetic image containing a horizontal, a vertical and a diagonal line.

   **Hint**

   A horizontal edge is a change along $y$, so use $S_y = S_x^\top$ (rows $-1,-2,-1$ / $0,0,0$ / $1,2,1$). For a diagonal edge, put the derivative across the diagonal, for example $\begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$. Its weights sum to zero, so flat regions give zero.

2. **Kernel sums.** Why should a blur kernel's weights sum to one? Predict, then check, what happens to a mid-gray `uint8` image filtered with a $3 \times 3$ kernel whose weights all equal $2/9$, and what changes if you filter in `float32` instead.

   **Hint**

   A flat region of value $c$ maps to $c \sum K$. With sum two, a gray of 128 becomes 256, which saturates at 255 in `uint8`; every brighter-than-mid-gray region turns white. In `float32` the values double without clipping, and you can rescale afterward.

3. **Composition.** Show that applying a $3 \times 3$ box blur twice equals one $5 \times 5$ kernel, and write that kernel. Verify numerically away from the borders, and explain why the equality can fail at the borders.

**Hint**

By associativity, the combined kernel is the box convolved with itself: the outer product of $[1, 2, 3, 2, 1]$ with itself, divided by 81. At the border, the first pass already invented padding values, and the second pass pads that result; a single $5 \times 5$ pass pads the original image instead, so the two differ in a frame of width two.

4. **Layer arithmetic.** For a $64 \times 64$ RGB input and a layer with 16 filters, $k = 5$, $p = 2$, $s = 2$: give the output shape and the number of parameters. Then compute $(r, j)$ after three $3 \times 3$ layers with strides 1, 2, 1, and compare with all strides equal to one.

**Hint**

$\lfloor (64 + 4 - 4 - 1)/2 \rfloor + 1 = 32$, so the output is $16 \times 32 \times 32$ with $16(3 \cdot 25 + 1) = 1216$ parameters. Receptive field: $(3, 1) \to (5, 2) \to (9, 2)$. All strides one gives $(7, 1)$.

5. **Gradient by hand.** For a $1 \times 4$ input $X = [x_1, x_2, x_3, x_4]$ and a kernel $K = [k_1, k_2]$ with valid correlation, write $Y$, then derive $\partial L/\partial K$ and $\partial L/\partial X$ in terms of $G = \partial L/\partial Y$. Check your formulas with the autograd trick from the backpropagation snippet.

**Hint**

$Y = [k_1 x_1 + k_2 x_2,\ k_1 x_2 + k_2 x_3,\ k_1 x_3 + k_2 x_4]$. Then $\partial L/\partial k_1 = G_1 x_1 + G_2 x_2 + G_3 x_3$ and $\partial L/\partial k_2 = G_1 x_2 + G_2 x_3 + G_3 x_4$ (a correlation of $X$ with $G$). $\partial L/\partial X = [G_1 k_1,\ G_1 k_2 + G_2 k_1,\ G_2 k_2 + G_3 k_1,\ G_3 k_2]$, the full convolution of $G$ with $K$.

6. **Ablation on digits.** In the classifier, replace `Flatten` + `Linear(512, 10)` by global average pooling + `Linear(32, 10)`. Compare parameter counts, validation loss and test accuracy, then evaluate both models on test digits shifted by one pixel. Explain the trade-off you observe.

**Hint**

`nn.AdaptiveAvgPool2d(1)` then `nn.Flatten()`. Pooling removes most head parameters and position information, which can help shift robustness but hurt on $8 \times 8$ digits where stroke position matters. Keep everything else fixed, select on validation only, and report both accuracy and parameter count.

## References

1. I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016, ch. 9. Book
2. L. Alzubaidi et al., "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions," *Journal of Big Data*, 2021. Review

3. W. Luo, Y. Li, R. Urtasun and R. Zemel, “Understanding the Effective Receptive Field in Deep Convolutional Neural Networks,” NeurIPS, 2016. Paper
4. O. Ronneberger, P. Fischer and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” MICCAI, 2015. Paper
5. K. He, X. Zhang, S. Ren and J. Sun, “Deep Residual Learning for Image Recognition,” CVPR, 2016. Paper
6. Y. Wu and K. He, “Group Normalization,” ECCV, 2018. Paper
7. M. Sandler et al., “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” CVPR, 2018. Paper
8. M. Tan and Q. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,” ICML, 2019. Paper
9. Z. Liu et al., “A ConvNet for the 2020s,” CVPR, 2022. Paper
10. S. J. D. Prince, *Understanding Deep Learning*, MIT Press, 2023, ch. 10. Book and notebooks
11. OpenCV, “Image Filtering” (`filter2D`, border types), documentation, version 4.x. Documentation
12. V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,” arXiv:1603.07285, 2016. arXiv
13. Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, 1998. DOI
14. A. Krizhevsky, I. Sutskever and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” NeurIPS, 2012. Paper
15. K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” ICLR, 2015. arXiv
16. C. Szegedy et al., “Going Deeper with Convolutions,” CVPR, 2015. arXiv
17. M. Lin, Q. Chen and S. Yan, “Network In Network,” ICLR, 2014. arXiv
18. F. Yu and V. Koltun, “Multi-Scale Context Aggregation by Dilated Convolutions,” ICLR, 2016. arXiv
19. S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” ICML, 2015. arXiv
20. R. Zhang, “Making Convolutional Networks Shift-Invariant Again,” ICML, 2019. arXiv
21. Z. Li, F. Liu, W. Yang, S. Peng and J. Zhou, “A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects,” *IEEE Transactions on Neural Networks and Learning Systems*, 2022. DOI
22. A. Khan, A. Sohail, U. Zahoora and A. S. Qureshi, “A Survey of the Recent Architectures of Deep Convolutional Neural Networks,” *Artificial Intelligence Review*, 2020. arXiv
23. J. Gu et al., “Recent Advances in Convolutional Neural Networks,” *Pattern Recognition*, 2018. arXiv
24. S. Woo et al., “ConvNeXt V2: Co-designing and Scaling ConvNets with Masked Autoencoders,” CVPR, 2023. arXiv
25. W. Wang et al., “InternImage: Exploring Large-Scale Vision Foundation Models with Deformable Convolutions,” CVPR, 2023. arXiv
26. X. Ding et al., “Scaling Up Your Kernels to 31x31: Revisiting Large Kernel Design in CNNs” (RepLKNet), CVPR, 2022. arXiv
27. X. Ding et al., “UniRepLKNet: A Universal Perception Large-Kernel ConvNet for Audio, Video,

Point Cloud, Time-Series and Image Recognition," CVPR, 2024. arXiv
28. D. Qin et al., "MobileNetV4: Universal Models for the Mobile Ecosystem," ECCV, 2024. arXiv
29. S. L. Smith, A. Brock, L. Berrada and S. De, "ConvNets Match Vision Transformers at Scale," arXiv:2310.16764, 2023. arXiv
30. M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," ECCV, 2014. arXiv
31. PyTorch, `torch.nn.Conv2d` documentation. Documentation

# ML 04 • Recurrent and Recursive Networks

Learn from sequences and trees: shared recurrent state, backpropagation through time, vanishing and exploding gradients, LSTM and GRU gates, bidirectional and encoder–decoder models, attention as the bridge to Transformers, recursive networks, and their uses in vision.

Online edition: https://blog.shyandram.dev/en/learn/ml/recurrent-recursive-networks/

## What you'll learn

- Unroll recurrent updates and track shared parameters.
- Explain backpropagation through time and long-range gradient difficulties.
- Compare LSTM and GRU memory paths and gates.
- Distinguish bidirectional, sequence-to-sequence and tree-structured models.
- Assess attention and state-space alternatives for context, streaming and memory.

## The big picture

ML 03 used one assumption about images: nearby pixels are related, and the same pattern can appear anywhere, so the same filter is reused across **space**. Sequences invite the same trick across **time**. A video is a sequence of frames, a caption is a sequence of words, a pen stroke is a sequence of positions. A **recurrent neural network** (**RNN**) reads such a sequence one element at a time, carries a summary called the **hidden state** from step to step, and uses the same weights at every step [1]. A **recursive neural network** generalizes the idea from a chain to a tree: it combines parts into wholes by repeatedly applying one shared composition rule.

> **Explain like I'm five**
>
> When you listen to a story, you do not remember every word. You keep a little picture in your head of what has happened so far, and each new sentence updates that picture. A recurrent network does exactly this: it has a small notebook (the hidden state), reads one word or one video frame at a time, and rewrites the notebook after each one using the same rules every time. An LSTM is a notebook with an eraser and a pen that the network learns to use carefully, so important notes from the beginning are not smudged away by the end. A recursive network is like building with LEGO: snap two small pieces into a bigger piece, then snap that into an even bigger piece, using the same kind of snap each time.

Why study RNNs when Transformers dominate large-scale language and vision? Three reasons. First, every concept in modern sequence modeling — teacher forcing, encoder–decoder, attention, masking, causal generation — was developed on RNNs, and the Transformer is easiest to understand as the answer to RNN limitations. Second, recurrence is efficient for **streaming**: a

constant-size state per step suits robots, sensors and on-device video. Third, recurrence is back: modern state-space models such as Mamba are recurrences with clever parameterizations, and ML 05 applies them to images.

## A recurrent state

**Plain version.** At every step, combine the current input with the previous state to make a new state, using the same weights each time. Read outputs off the state whenever you need them.

**Precise version.** Let the input sequence be $x_1, \dots, x_T$ with $x_t \in \mathbb{R}^d$, and let the hidden state be $h_t \in \mathbb{R}^m$ with a fixed starting state $h_0$ (usually zeros). The simple (Elman [9]) RNN is

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b), \qquad \hat{y}_t = g\big(W_y h_t + c\big).$$

$W_x \in \mathbb{R}^{m \times d}$ maps the input into the state space, $W_h \in \mathbb{R}^{m \times m}$ is the **recurrent** weight matrix that carries information from the previous step, $b$ and $c$ are biases, $W_y$ maps the state to outputs, and $g$ is an output function such as softmax. Crucially, none of these parameters depends on $t$. That is **parameter sharing across time**, the temporal version of a CNN's weight sharing: the model has the same number of parameters whether the sequence has 5 or 5,000 steps, and a pattern learned at step 3 can be recognized at step 300.

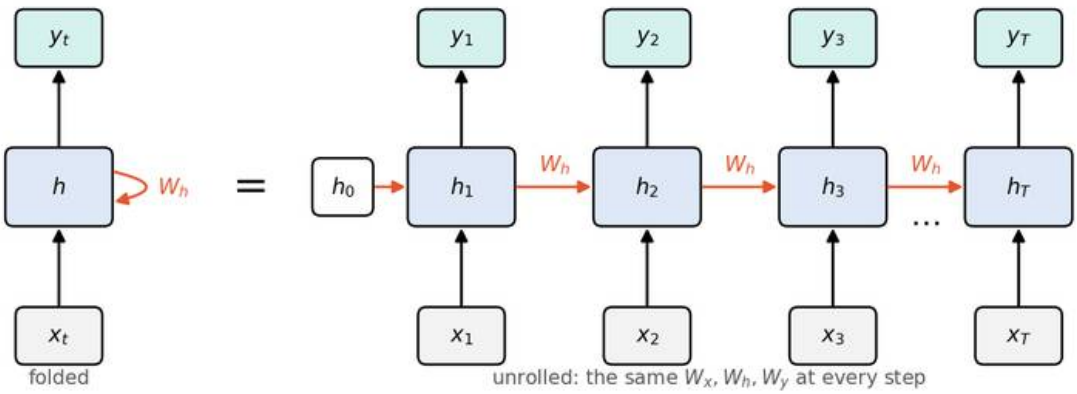



*Figure 1 — Folded and unrolled views of an RNN. Unrolling turns the loop into a deep feed-forward network whose layers all share the same weights.*

**Unrolling** (Figure 1) is the key mental move. Writing out the loop for $T$ steps produces an ordinary feed-forward network that is $T$ layers deep, in which every layer has the same weights. Everything you learned about backpropagation in ML 02 applies to this unrolled graph.

The same recurrence supports several input–output patterns:

| Pattern | Example | Output read from |
|---|---|---|
| Many-to-one | Classify a video clip; sentiment of a sentence | The last state $h_T$ (or a pooled state) |

| Pattern | Example | Output read from |
|---|---|---|
| Many-to-many, aligned | Label every video frame; tag every word | Every $h_t$ |
| One-to-many | Generate a caption from one image vector | A decoder that feeds its own outputs back in |
| Many-to-many, unaligned | Translate a sentence; describe a video | An **encoder–decoder** (covered below) |

An RNN that predicts $x_{t+1}$ from $x_{\le t}$ is an **autoregressive** model, and sampling from it generates new sequences; but an RNN used as an encoder is not automatically a generator. The head and the loss decide the task.

### Code: an RNN and its backward pass in NumPy

The snippet below implements the forward pass and backpropagation through time by hand for a loss $L = v^\top h_T$, then checks one gradient entry against a finite difference. Notice the line `dWh += ⋯`: because $W_h$ is used at every step, its gradient is the **sum** of contributions from all steps.

```
import numpy as np

rng = np.random.default_rng(0)
d, m, T = 3, 5, 7                                  # input size, state size, length
Wx = rng.normal(0, 0.5, (m, d))
Wh = rng.normal(0, 0.5, (m, m))
b = np.zeros(m)

def rnn_forward(xs, h0):
    hs = [h0]
    for x in xs:                                    # the SAME Wx, Wh, b at every step
        hs.append(np.tanh(Wx @ x + Wh @ hs[-1] + b))
    return hs

def rnn_backward(xs, hs, dhT):
    """BPTT for the loss L = dhT . h_T; returns dL/dWh and dL/dx_t for every t."""
    dWh, dxs, dh = np.zeros_like(Wh), [None] * len(xs), dhT
    for t in reversed(range(len(xs))):
        da = dh * (1 - hs[t + 1] ** 2)              # through tanh
        dWh += np.outer(da, hs[t])                  # shared weight: gradients ADD up
        dxs[t] = Wx.T @ da
        dh = Wh.T @ da                              # pass the gradient one step back
    return dWh, dxs

xs = rng.normal(size=(T, d))
hs = rnn_forward(xs, np.zeros(m))
dhT = rng.normal(size=m)
dWh, dxs = rnn_backward(xs, hs, dhT)

# finite-difference check of one entry of dL/dWh
eps, (i, j) = 1e-6, (2, 3)
Wh[i, j] += eps; up = dhT @ rnn_forward(xs, np.zeros(m))[-1]
```

```
Wh[i, j] -= 2 * eps; down = dhT @ rnn_forward(xs, np.zeros(m))[-1]
Wh[i, j] += eps
print("BPTT vs finite difference:", np.isclose(dWh[i, j], (up - down) / (2 * eps)))
print("|dL/dx_t| from first to last step:", np.round([np.linalg.norm(g) for g in dxs], 3))
```

It prints `True`, and the input-gradient norms are $[0.009, 0.015, 0.006, 0.029, 0.383, 0.212, 0.433]$: even in a seven-step sequence, the early inputs receive gradients roughly ten to fifty times smaller than the late ones. That is the vanishing-gradient problem in miniature.

## Backpropagation through time

**Plain version.** Unroll the network, run ordinary backpropagation through the unrolled graph, and add up the gradients for each shared weight.

**Precise version.** Write the pre-activation as $a_t = W_x x_t + W_h h_{t-1} + b$, so $h_t = \tanh(a_t)$. Suppose the loss is a sum over steps, $L = \sum_t L_t(h_t)$. Working backward from $t = T$, the total gradient arriving at $h_t$ is

$$\frac{\partial L}{\partial h_t} = \frac{\partial L_t}{\partial h_t} + \left(\frac{\partial h_{t+1}}{\partial h_t}\right)^{\top} \frac{\partial L}{\partial h_{t+1}}, \qquad \frac{\partial h_{t+1}}{\partial h_t} = \operatorname{diag}(1 - h_{t+1}^2)\, W_h.$$

The first term is the local loss at step $t$; the second is everything later that $h_t$ influenced, passed back one step. With $\delta_t = (1 - h_t^2) \odot \partial L / \partial h_t$ (the gradient at the pre-activation), the parameter gradients are sums over time:

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^{T} \delta_t\, h_{t-1}^{\top}, \qquad \frac{\partial L}{\partial W_x} = \sum_{t=1}^{T} \delta_t\, x_t^{\top}, \qquad \frac{\partial L}{\partial b} = \sum_{t=1}^{T} \delta_t.$$

This algorithm, **backpropagation through time** (**BPTT**), was laid out clearly by Werbos [8]. Its cost is linear in $T$, but so is its memory: every $h_t$ must be stored for the backward pass.

### Truncated BPTT

For a very long stream (an hour of video, a continuous sensor log) storing all states is impossible. **Truncated BPTT** processes the stream in windows of $k$ steps. It carries the hidden state forward from one window to the next, so the model still *sees* the whole past, but it **detaches** the state from the graph at each window boundary, so gradients flow back at most $k$ steps. The model can therefore use long-range information only if it learns to preserve it from the gradient signal within windows.

```
import torch
from torch import nn

torch.manual_seed(0)
rnn, head = nn.GRU(1, 16, batch_first=True), nn.Linear(16, 1)
```

```
params = list(rnn.parameters()) + list(head.parameters())
opt = torch.optim.Adam(params, lr=1e-2)
t = torch.arange(0, 2000) * 0.05
stream = torch.sin(t) + 0.1 * torch.randn_like(t)   # one long noisy signal
x, y = stream[:-1, None][None], stream[1:, None][None]   # predict the next sample

h, window = None, 50
for start in range(0, x.shape[1] - window, window):
    out, h = rnn(x[:, start:start + window], h)
    loss = nn.functional.mse_loss(head(out), y[:, start:start + window])
    opt.zero_grad()
    loss.backward()                        # gradients flow back at most `window` steps
    nn.utils.clip_grad_norm_(params, 1.0)
    opt.step()
    h = h.detach()                         # keep the state, cut the graph
print("final window MSE:", round(loss.item(), 4), "(noise variance is 0.01)")
```

After a single pass over the stream, the final-window mean squared error is about 0.014, close to the 0.01 floor set by the added noise. Forgetting `h.detach()` makes the second window try to backpropagate into a graph that was already freed, and PyTorch raises an error; resetting `h = None` at each window instead would throw the context away.

## Vanishing and exploding gradients

**Plain version.** Backpropagation multiplies state Jacobians across time. These products can shrink or grow sharply, making distant dependencies hard to learn.

**Precise version.** Chaining the one-step Jacobians gives

$$\frac{\partial h_t}{\partial h_s} = \prod_{k=s+1}^{t} \frac{\partial h_k}{\partial h_{k-1}} = \prod_{k=s+1}^{t} \operatorname{diag}(1 - h_k^2)\, W_h,$$

an ordered product of $t - s$ matrices (matrices do not commute, so the order matters). Because $|\tanh'| \le 1$,

$$\left\| \frac{\partial h_t}{\partial h_s} \right\| \le \left(\sigma_{\max}(W_h)\right)^{t-s},$$

where $\sigma_{\max}$ is the largest singular value. If $\sigma_{\max}(W_h) < 1$, the gradient from distant steps **vanishes** exponentially. If $W_h$ has large singular values and the units are not saturated, the gradient can **explode**. Bengio, Simard and Frasconi [6] showed in 1994 that this is not a bug of a particular training method: a system that robustly stores information in the presence of noise must be in a regime where gradients of long-range dependencies shrink. Pascanu, Mikolov and Bengio [7] revisited the analysis with dynamical-systems tools and gave a practical remedy for explosions.

The intuition is clearest for a scalar linear RNN, $h_t = \lambda h_{t-1} + x_t$: then $\partial h_t / \partial h_s = \lambda^{t-s}$, which is $0.9^{80} \approx 2 \times 10^{-4}$ for $\lambda = 0.9$ and $1.1^{80} \approx 2 \times 10^{3}$ for $\lambda = 1.1$. Figure 2 shows the same behavior measured in real networks.

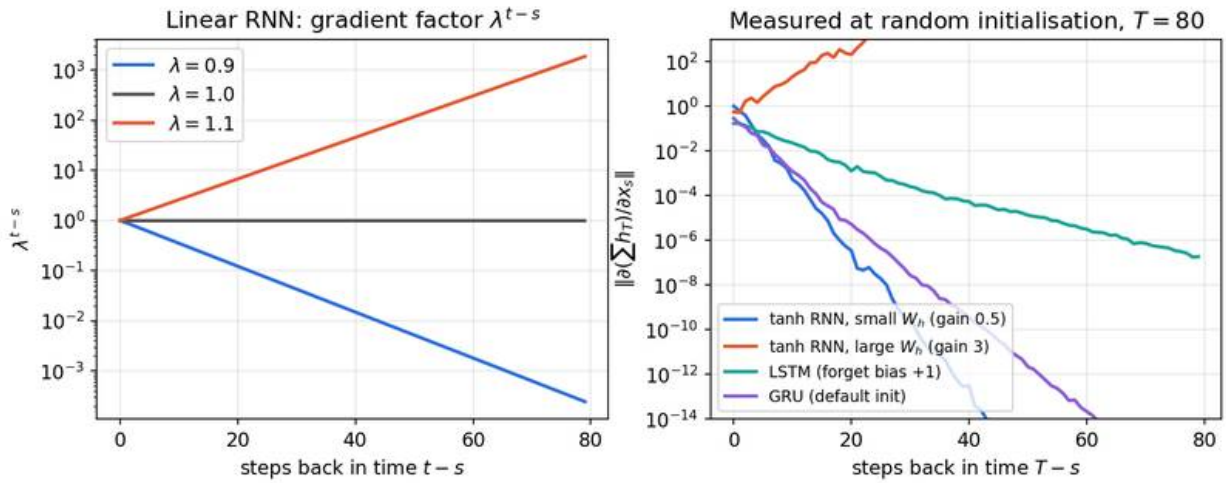



*Figure 2 — Left: the scalar factor $\lambda^{(t-s)}$. Right: gradient of the final state with respect to each input, measured at random initialization ($T = 80$). The plain tanh RNN vanishes or explodes; the LSTM's additive memory path decays far more slowly.*

**Remedies.** Explosions are handled by **gradient clipping** [7]: if the gradient norm $\|g\|$ exceeds a threshold $\tau$, rescale it to $g \leftarrow \tau\, g/\|g\|$. This keeps the update direction and caps its length. PyTorch's `clip_grad_norm_` does exactly this. Vanishing gradients are harder: they require a different architecture, one whose state update is close to the identity so that gradients can pass through many steps. That architecture is the LSTM.

## Long short-term memory (LSTM)

**Plain version.** Give the network a separate memory lane, the **cell state**, that is updated by addition rather than by repeated matrix multiplication, and let learned **gates** decide what to erase, what to write and what to reveal.

**Precise version.** With $[x_t, h_{t-1}]$ denoting concatenation and $\sigma$ the logistic sigmoid, a standard LSTM step is

$$
\begin{aligned}
f_t &= \sigma\big(W_f[x_t, h_{t-1}] + b_f\big) && \text{forget gate}\\
i_t &= \sigma(W_i[x_t, h_{t-1}] + b_i) && \text{input gate}\\
o_t &= \sigma(W_o[x_t, h_{t-1}] + b_o) && \text{output gate}\\
\tilde{c}_t &= \tanh(W_c[x_t, h_{t-1}] + b_c) && \text{candidate memory}\\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t && \text{cell update}\\
h_t &= o_t \odot \tanh(c_t) && \text{exposed state}
\end{aligned}
$$

Each gate is a vector of numbers between 0 and 1, and $\odot$ is element-wise multiplication, so each gate acts like a bank of dimmer switches, one per memory unit. The forget gate $f_t$ decides how much of the old memory to keep, the input gate $i_t$ how much of the candidate $\tilde{c}_t$ to write, and the output gate $o_t$ how much of the memory to reveal as $h_t$.

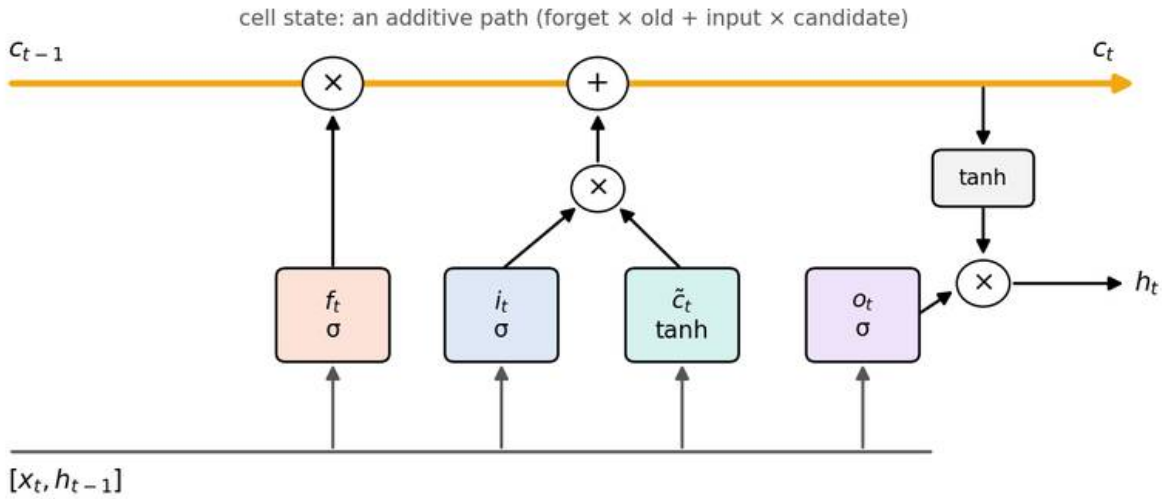



*Figure 3 — Inside an LSTM cell. The cell state (gold) changes only by an element-wise scaling and an addition, which gives gradients a protected path through time.*

**Why it works.** Along the cell-state path, $\partial c_t / \partial c_{t-1} = \mathrm{diag}(f_t)$ (ignoring the smaller indirect paths through the gates). There is no recurring weight matrix and no squashing nonlinearity on this path. If the forget gate stays near one, the gradient passes through many steps almost unchanged. That is why the LSTM curve in Figure 2 decays so slowly. It is also why a common initialization sets the forget-gate bias to a positive value such as +1, so the network starts out remembering. This is not a guarantee of unlimited memory: the forget gate can and does learn to close, and long-range credit assignment still gets harder with distance.

**History and attribution.** Hochreiter and Schmidhuber introduced the LSTM in 1997 [2] with a constant-error "carousel": the cell state passed on unchanged and there was **no forget gate**. Gers, Schmidhuber and Cummins added the forget gate in 2000 [5] so that the cell could learn to reset its memory, for example between independent segments of a continuous stream. The equations above, with the forget gate, are the version used in practice and in every major framework. Greff et al. [19] compared eight LSTM variants on speech, handwriting and music tasks and found that none significantly beat the standard architecture, while the forget gate and the output activation were its most critical components.

**Parameters.** Each of the four gate blocks has an $m \times (d + m)$ weight matrix and a bias, so an LSTM layer has about $4m(d + m + 1)$ parameters, four times a simple RNN with the same state size. (PyTorch keeps separate input and recurrent biases, giving $4m(d + m + 2)$.)

### Code: an LSTM step from scratch, checked against PyTorch

PyTorch stores the four gate blocks stacked in the order input, forget, cell candidate ($g$), output. Implementing the step by hand and comparing with `nn.LSTMCell` is the best way to be sure you read the equations correctly:

```
import torch
from torch import nn

torch.manual_seed(0)
d, m = 4, 6
cell = nn.LSTMCell(d, m)                # PyTorch stacks gate weights in order i, f, g, o

def lstm_step(x, h, c):
    z = x @ cell.weight_ih.T + cell.bias_ih + h @ cell.weight_hh.T + cell.bias_hh
    i, f, g, o = z.chunk(4, dim=-1)
    i, f, o = torch.sigmoid(i), torch.sigmoid(f), torch.sigmoid(o)
    g = torch.tanh(g)                   # candidate memory, often written c~_t
    c = f * c + i * g                   # additive cell update
    h = o * torch.tanh(c)
    return h, c

x = torch.randn(2, d)
h0, c0 = torch.randn(2, m), torch.randn(2, m)
h_ref, c_ref = cell(x, (h0, c0))
h_ours, c_ours = lstm_step(x, h0, c0)
print("matches nn.LSTMCell:", torch.allclose(h_ours, h_ref, atol=1e-6),
      torch.allclose(c_ours, c_ref, atol=1e-6))
print("parameters:", sum(p.numel() for p in cell.parameters()), "= 4 * m * (d + m + 2) =",
      4 * m * (d + m + 2))
```

It prints `True True` and `288 = ⋯ = 288`.

*Online only: LSTM — protecting a state across long delays — https://blog.shyandram.dev/en/learn/ml/recurrent-recursive-networks/papers/lstm–protecting-a-state-across-long-delays/*

## Gated recurrent unit (GRU)

**Plain version.** A GRU is a slimmer LSTM: it merges the cell and hidden states into one, and uses two gates instead of three.

**Precise version.** Following Cho et al. [3] and PyTorch's `nn.GRU`,

$$
\begin{aligned}
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) && \text{reset gate} \\
z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) && \text{update gate} \\
\tilde{h}_t &= \tanh(W x_t + r_t \odot (U h_{t-1}) + b) && \text{candidate} \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t && \text{interpolation}
\end{aligned}
$$

The reset gate $r_t$ decides how much of the old state to use when forming the candidate; the update gate $z_t$ interpolates between keeping the old state ($z_t \approx 1$) and replacing it ($z_t \approx 0$). The interpolation plays the role of the LSTM's additive cell path. Two caveats are worth knowing. Some papers and textbooks swap the roles of $z_t$ and $1 - z_t$; the models are equivalent, but the gate means the opposite. And the original paper applies the reset gate before the recurrent matrix ($U(r_t \odot h_{t-1})$),

whereas PyTorch and cuDNN apply it after, as written above. Always check the framework's documentation before reading meaning into a gate.

GRUs have three gate blocks instead of four, so about 25% fewer parameters than an LSTM of the same width. Neither consistently wins across tasks; the GRU is a reasonable default when data or compute is limited.

### Code: a recurrent classifier that generalizes to fresh sequences

This complete CPU example learns to classify the sign of a sequence's sum, which requires accumulating information over all twelve steps. A fresh test set checks that the model did not merely memorize its training sequences.

```
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(0)
torch.set_num_threads(1)
rnn = nn.GRU(input_size=1, hidden_size=24, batch_first=True)
head = nn.Linear(24, 2)
opt = torch.optim.Adam(list(rnn.parameters())+list(head.parameters()), lr=.003)
def batch(n):
    x = torch.randn(n, 12, 1)
    y = (x.sum(dim=(1, 2)) > 0).long()
    return x, y
for step in range(400):
    x, y = batch(64)
    _, h = rnn(x)
    loss = F.cross_entropy(head(h[-1]), y)
    opt.zero_grad(); loss.backward()
    nn.utils.clip_grad_norm_(list(rnn.parameters())+list(head.parameters()), 1.)
    opt.step()
rnn.eval(); head.eval()
with torch.no_grad():
    x, y = batch(512)
    _, h = rnn(x)
    acc = (head(h[-1]).argmax(1) == y).float().mean()
    print("fresh-sequence accuracy:", acc.item())
```

It reaches about 99% accuracy on fresh length-12 sequences. The returned hidden tensor has shape `layers × batch × hidden` even with `batch_first=True`, which affects only the input and output sequence layout. Testing on longer sequences asks a different question, length generalization, and a model can succeed at length 12 and fail at length 100 (Exercise 2).

## Deep, bidirectional and streaming RNNs

**Stacking.** A **deep RNN** feeds the sequence of hidden states of one recurrent layer into another (`num_layers` in PyTorch). Lower layers can capture short-range patterns and higher layers longer ones. Two to four layers are typical; deeper stacks usually need residual connections between layers.

**Bidirectional.** Schuster and Paliwal [10] proposed running one RNN forward in time and another backward, then concatenating their states, $h_t = [\vec{h}_t; \overleftarrow{h}_t]$. Each position then summarizes both its past and its future. This is ideal for **offline** tasks where the whole input is available (tagging a recorded video, encoding a sentence), but it is **not causal**: it cannot be used for a live prediction that must be made before future frames arrive. Reporting offline bidirectional accuracy for an online application overstates what the deployed system can know.

**Variable lengths.** Batches contain sequences of different lengths, padded to a common length. Without care, the padding leaks into states and losses. Use per-sequence lengths (`pack_padded_sequence` in PyTorch) or masks so that padded steps neither update the state nor contribute to the loss.

**Streaming.** For a live stream, keep the state between calls and reset it only between independent streams. Accidentally carrying state from one user, document or video into the next leaks information and produces irreproducible results.

## Encoder–decoder models and teacher forcing

**Plain version.** A basic encoder–decoder reads an input sequence into a state, then generates an output sequence from it. The two sequences can have different lengths.

**Precise version.** In the **sequence-to-sequence** (seq2seq) model of Sutskever, Vinyals and Le [11] and the closely related encoder–decoder of Cho et al. [3], an **encoder** RNN reads $x_1, \dots, x_T$ and its final state becomes a context vector $c = h_T$. A **decoder** RNN, initialized from $c$, models

$$p(y_1, \dots, y_U \mid x) = \prod_{u=1}^{U} p(y_u \mid y_{<u}, c), \qquad s_u = \mathrm{RNN}(s_{u-1}, y_{u-1}),$$

where $s_u$ is the decoder state and $p(y_u \mid \cdot)$ is a softmax over the vocabulary. Generation stops when the decoder emits an end-of-sequence token, so input and output lengths are independent.

**Teacher forcing.** During training, the decoder input at step $u$ is the **ground-truth** previous token $y_{u-1}$, not the model's own prediction. This makes all steps trainable in parallel against the known target and greatly stabilizes learning. At test time, however, the model must consume its own predictions, which may contain errors it never saw during training. This train–test mismatch is called **exposure bias**. Scheduled sampling [25] gradually replaces ground-truth inputs with model samples during training to reduce it. Decoding usually uses greedy or beam search rather than sampling.

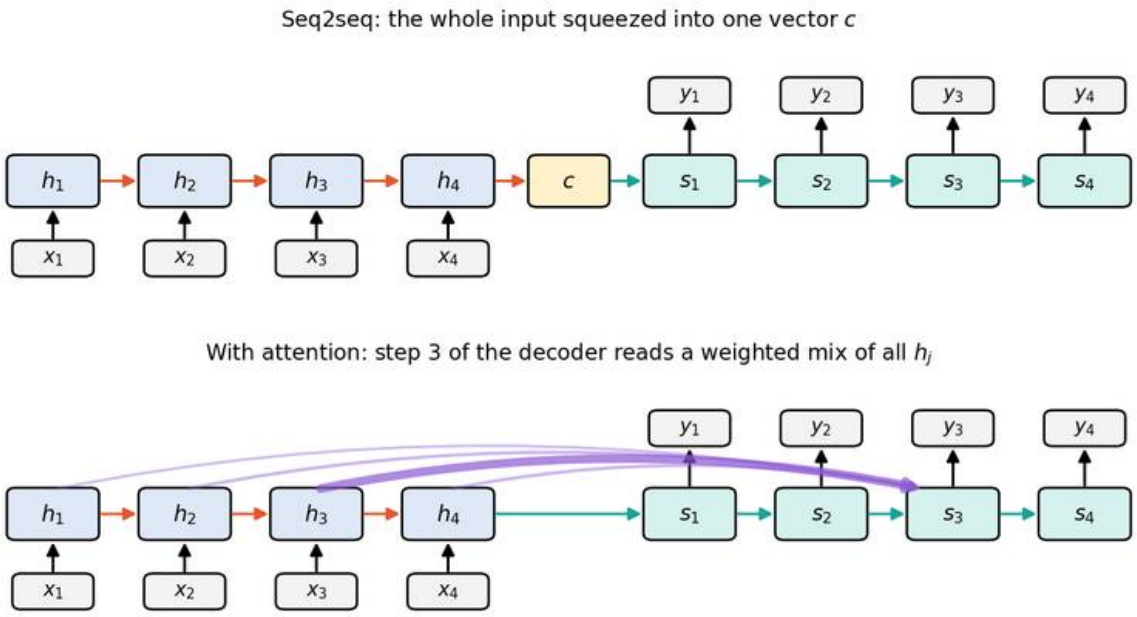



*Figure 4 — Top: plain seq2seq squeezes the entire input into one vector c. Bottom: with attention, every decoder step reads a learned, weighted mix of all encoder states.*

The weakness is visible in Figure 4 (top): everything the decoder knows about the input must pass through one fixed-size vector $c$, no matter how long the input. Translation quality of plain seq2seq models dropped on long sentences. Sutskever et al. found that simply **reversing the source sentence** helped, because it shortened the distance between corresponding words, a hint that the bottleneck and long paths were the problem [11].

## Attention: the bridge to Transformers

**Plain version.** Attention lets the decoder select relevant encoder states at each step instead of relying on one fixed summary.

**Precise version.** Bahdanau, Cho and Bengio [12] kept every encoder state $h_1, \dots, h_T$ (from a bidirectional encoder) and, at each decoder step $u$, computed

$$e_{u,j} = v^\top \tanh\big(W s_{u-1} + U h_j\big), \qquad \alpha_{u,j} = \frac{\exp(e_{u,j})}{\sum_{k=1}^{T} \exp(e_{u,k})}, \qquad c_u = \sum_{j=1}^{T} \alpha_{u,j}\, h_j.$$

$e_{u,j}$ is a learned **alignment score** between the decoder's current need ($s_{u-1}$) and encoder position $j$; $W$, $U$ and $v$ are learned parameters. The softmax turns scores into weights $\alpha_{u,j}$ that are positive and sum to one, and the **context vector** $c_u$ is their weighted average of encoder states. The decoder then uses $c_u$ together with its own state. This form is called **additive attention**. Because $c_u$ changes at every step and has direct access to every input position, the path length between any input and any output becomes one step, regardless of the sequence length.

```
import torch
from torch import nn

class AdditiveAttention(nn.Module):
    """Bahdanau-style scores e_j = v^T tanh(W s + U h_j), then softmax over j."""
    def __init__(self, d_dec, d_enc, d_att):
        super().__init__()
        self.W = nn.Linear(d_dec, d_att, bias=False)
        self.U = nn.Linear(d_enc, d_att, bias=False)
        self.v = nn.Linear(d_att, 1, bias=False)

    def forward(self, s, H, mask=None):
        # s: B x d_dec (decoder state), H: B x T x d_enc (all encoder states)
        e = self.v(torch.tanh(self.W(s)[:, None] + self.U(H))).squeeze(-1)   # B x T
        if mask is not None:                        # True where the input is padding
            e = e.masked_fill(mask, float("-inf"))
        alpha = e.softmax(dim=-1)                   # attention weights, rows sum to 1
        context = (alpha[:, :, None] * H).sum(1)    # B x d_enc weighted average
        return context, alpha

torch.manual_seed(0)
att = AdditiveAttention(d_dec=16, d_enc=32, d_att=8)
H = torch.randn(2, 5, 32)
s = torch.randn(2, 16)
pad = torch.tensor([[False] * 5, [False, False, False, True, True]])   # 2nd sequence has
 ↪  length 3
ctx, alpha = att(s, H, pad)
print("context:", tuple(ctx.shape), "| weights sum to 1:",
      torch.allclose(alpha.sum(-1), torch.ones(2)))
print("weights on padding:", alpha[1, 3:].tolist())
```

It prints a context of shape `(2, 32)`, confirms that weights sum to one, and shows exactly zero weight on padded positions: masking with $-\infty$ before the softmax is the standard way to make attention ignore padding (and, in ML 05, the future).

Attention weights are also interpretable as a soft **alignment**. Figure 5 shows a small GRU encoder–decoder with this attention trained for under a minute on CPU to reverse random digit sequences (about 96% per-token accuracy with teacher forcing on fresh sequences). Without being told, it learned to look at input position $T + 1 - u$ when writing output $u$: the anti-diagonal.

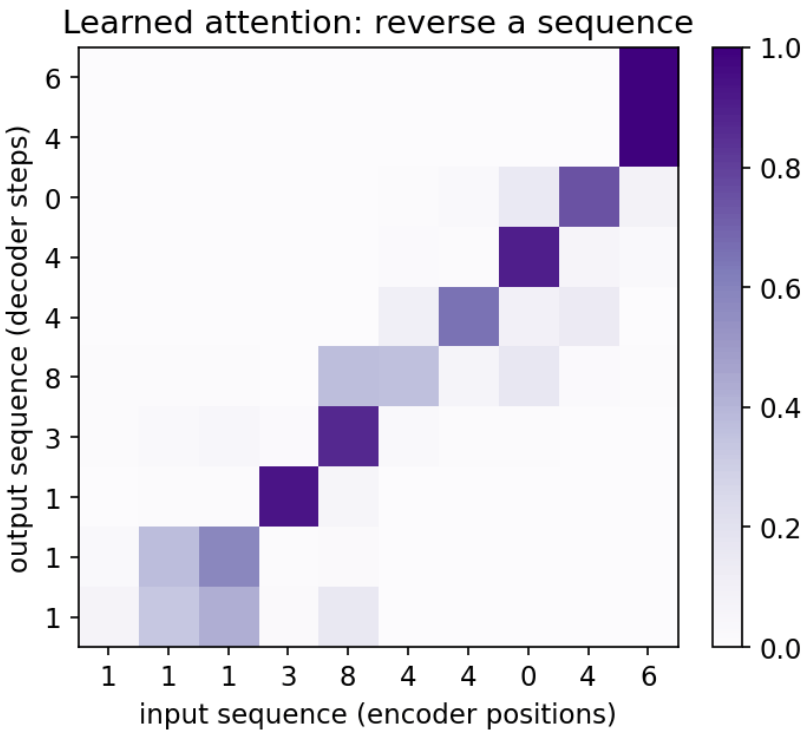



*Figure 5 — Attention weights of a toy encoder–decoder trained to reverse a sequence (generated by the figure script). Each output step concentrates on the mirrored input position.*

**From here to Transformers.** Attention fixed the bottleneck, but the encoder and decoder were still recurrent, so computation over a sequence of length $T$ required $T$ sequential steps that cannot be parallelized during training. Vaswani et al. [28] asked what happens if recurrence is removed entirely and **attention is used inside the encoder and decoder as well** (self-attention), with positional encodings to restore order. The answer, the Transformer, trains in parallel across positions and became the dominant architecture for language and, with ViT, for vision. ML 05 picks up the story with scaled dot-product attention, which replaces the additive score with $q^\top k / \sqrt{d_k}$.

## Recurrent networks in vision

**Image captioning.** Show and Tell [13] (Vinyals et al.) is a direct image-to-sequence model: a CNN (ML 03) encodes the image into a vector that initializes an LSTM language model, which then generates the caption word by word, trained with teacher forcing on image–caption pairs. Show, Attend and Tell [14] (Xu et al.) kept the CNN's spatial feature grid instead of a single vector and let the LSTM attend over image regions at each word, producing attention maps that often highlight the object being named. This was one of the first widely seen demonstrations of visual attention.

**Video understanding.** Long-term Recurrent Convolutional Networks (LRCN) [15] (Donahue et al.) apply a CNN to each frame and an LSTM across frames, trained end to end, for activity

recognition, image description and video description. The CNN handles space, the RNN handles time: a common pattern for any per-frame feature sequence, such as pose tracks or per-frame detections in object tracking.

**Spatiotemporal forecasting.** When the state itself should be an image, fully connected gates waste the spatial structure. The **ConvLSTM** of Shi et al. [16] replaces the matrix products in the LSTM gates with convolutions, $i_t = \sigma(W_{xi} * X_t + W_{hi} * H_{t-1} + \dots)$, so inputs, states and gates are all 3-D tensors (channels × height × width). It was introduced for precipitation nowcasting from radar maps and is widely used for video prediction and other forecasting on grids.

**What changed.** For captioning and video classification at scale, Transformers and vision–language models (see ML 10) have largely replaced LSTMs, because they train in parallel and scale better with data. Recurrent and convolutional-recurrent models remain common where frames arrive one at a time and memory must stay constant: online video processing, tracking, robotics and low-power devices.

## Recursive networks: composing over trees

**Plain version.** A recursive network applies one shared combination rule to merge two child representations into a parent, following a tree, until it reaches the root.

**Precise version.** For a binary tree whose leaves carry vectors (word embeddings, region features), each internal node computes

$$h_{\text{parent}} = \tanh\!\left(W\,[h_{\text{left}}; h_{\text{right}}] + b\right),$$

with the **same** $W \in \mathbb{R}^{D \times 2D}$ and $b$ at every node. The root vector represents the whole structure and can be classified; every node can also have its own label. Training uses **backpropagation through structure**: the same idea as BPTT, but along the tree's edges. A chain of length $n$ is the special case of a completely unbalanced tree; a balanced tree reduces the path from any leaf to the root from $n-1$ to about $\log_2 n$ steps (Figure 6), which shortens the route for gradients.

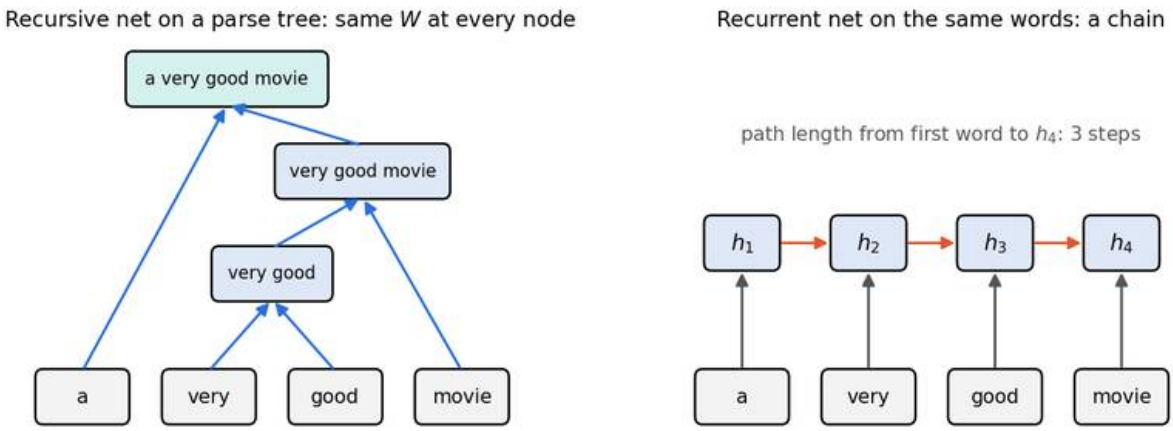


*Figure 6 — A recursive network follows a tree (left); a recurrent network follows a chain (right). Both share weights, but the tree mirrors the structure of the input.*

Socher et al. [18] used recursive networks in a vision setting first: they segmented an image into superpixels, then learned both to merge adjacent regions into a scene parse tree and to compute region representations, using the same method to parse sentences. Their later Recursive Neural Tensor Network and Stanford Sentiment Treebank [17] showed that composing along parse trees captures phenomena such as negation ("not very good"). Tai, Socher and Manning's **Tree-LSTM** [4] added LSTM-style gates to tree composition, with one forget gate per child, so a parent can selectively keep information from each subtree.

```
import torch
from torch import nn

torch.manual_seed(0)
D = 8
vocab = {"a": 0, "very": 1, "good": 2, "movie": 3}
emb = nn.Embedding(len(vocab), D)
compose = nn.Linear(2 * D, D)            # ONE weight matrix shared by every internal node
classify = nn.Linear(D, 2)               # e.g. negative / positive sentiment

def encode(tree):
    """tree is a word (leaf) or a pair (left, right)."""
    if isinstance(tree, str):
        return emb(torch.tensor(vocab[tree]))
    left, right = tree
    return torch.tanh(compose(torch.cat([encode(left), encode(right)])))

tree = ("a", (("very", "good"), "movie"))          # (a ((very good) movie))
root = encode(tree)
loss = nn.functional.cross_entropy(classify(root)[None], torch.tensor([1]))
loss.backward()                                     # backpropagation through structure
print("root vector:", tuple(root.shape), "| loss:", round(loss.item(), 4))
print("gradient reached the shared W:", compose.weight.grad.abs().sum().item() > 0)
swapped = encode((("movie", ("good", "very")), "a"))
print("swapping children changes the root:", not torch.allclose(root, swapped))
```

The recursion is ordinary Python, and autograd builds the tree-shaped graph on the fly. Swapping children changes the result because concatenation gives the left and right slots different roles.

The main practical limitation is the tree itself: someone has to provide it (a parser, a segmentation hierarchy), and batching variable trees efficiently is awkward. Learning structure and representation jointly is possible but harder. In current practice, self-attention can learn soft, input-dependent groupings without an explicit tree, which is one reason recursive networks are now uncommon; the idea of hierarchical composition lives on in hierarchical vision backbones and scene graphs.

## Mamba: selective state spaces and scan

A useful memory should hold a relevant observation through distractions and overwrite it when the situation changes. Mamba [21] makes the memory update depend on the current input, while keeping that update affine in the previous state. This distinction lets us retain a recurrent interpretation and compute whole sequences with a parallel prefix scan.

### From continuous dynamics to selective updates

Start with a small continuous-time memory. A stable transition lets stored information decay; an input projection writes new information; an output projection reads it:

$$\frac{dh(\tau)}{d\tau} = Ah(\tau) + Bu(\tau), \qquad y(\tau) = Ch(\tau).$$

Here $\tau$ is continuous time, $u(\tau) \in \mathbb{R}$ is one input channel, $h(\tau) \in \mathbb{R}^N$ is its $N$-dimensional state, $A \in \mathbb{R}^{N\times N}$ is the transition matrix, $B \in \mathbb{R}^{N\times 1}$ writes the input, $C \in \mathbb{R}^{1\times N}$ reads the state, and $y(\tau) \in \mathbb{R}$ is the output. A full neural block mixes multiple channels with learned projections and includes additional local convolution and gating [21]. This equation isolates its memory mechanism.

Hold the input and its write matrix constant during each interval. Exact zero-order-hold discretization gives

$$\bar{A}_t = \exp(\Delta_t A), \qquad \bar{B}_t = \left( \int_0^{\Delta_t} \exp(sA)\, ds \right) B_t,$$
$$h_t = \bar{A}_t h_{t-1} + \bar{B}_t u_t, \qquad y_t = C_t h_t.$$

Here $t$ indexes discrete tokens, $\Delta_t > 0$ is the interval length, $s$ is an integration variable, $u_t$ is the held input, $B_t, C_t$ are the current write/read matrices, and bars mark discrete coefficients. The integral remains valid when $A$ is singular; an inverse-based formula needs additional assumptions. A diagonal $A$ makes these operations elementwise.

Selectivity means predicting $\Delta_t, B_t, C_t$ from the current token. For example, a learned affine projection followed by $\mathrm{softplus}(z) = \log(1 + e^z)$ supplies a positive interval. The underlying learned $A$ can remain fixed: $\bar{A}_t$ nevertheless changes through $\Delta_t$. With negative diagonal entries, a larger interval reduces the fraction of old state retained. Selective writes and reads also determine which content enters and leaves memory. The resulting map is nonlinear in its inputs even though each state update is affine. Figure 7 isolates the retention effect with synthetic coefficients; it does not measure a trained model.

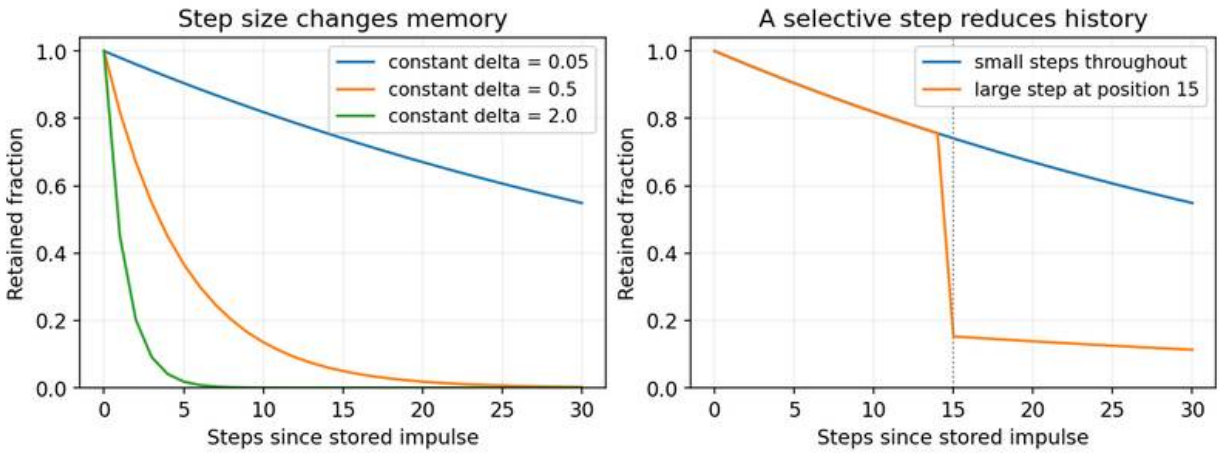



*Figure 7 — For a scalar transition $A = -0.4$, retained history is the product of $\exp(-0.4\ \text{delta})$. A large selected interval sharply reduces old memory; small intervals preserve it longer.*

### Why this recurrence can be scanned

Once input-dependent coefficients are computed, composing two affine updates requires only another affine update:

$$(a_2, b_2) \circ (a_1, b_1) = (a_2 a_1,\ a_2 b_1 + b_2).$$

Here pair $(a_i, b_i)$ represents $h \mapsto a_i h + b_i$, $i = 1, 2$ identifies consecutive updates, and $\circ$ means applying the first update and then the second. Composition is associative; for a diagonal transition, products are elementwise. A prefix scan can therefore regroup the operations into a tree while preserving their chronological effect. Ordinary LSTM/GRU gates depend on the previous hidden state, so their coefficients cannot generally be prepared this way before the state is known.

The following standalone CPU example uses exact diagonal discretization and hand-chosen selective coefficients. It checks causality and state continuity across chunks. It is a teaching recurrence, not a pretrained Mamba implementation or a speed benchmark.

```
import numpy as np

rng = np.random.default_rng(44)
u = rng.normal(size=24)
A = -np.array([0.1, 0.8])  # two stable diagonal entries

def selective_scan(values, state=None):
    h = np.zeros(2) if state is None else state.copy()
    outputs = []
    for x in values:
        delta = np.logaddexp(0.0, 0.5 * x)
        B = np.array([1.0, np.tanh(x)])
        C = np.array([1.0, 0.5 + 0.1 * np.tanh(x)])
        a = np.exp(delta * A)
```

```
        b = np.expm1(delta * A) / A * B * x
        h = a * h + b
        outputs.append(C @ h)
    return np.array(outputs), h

y, final_state = selective_scan(u)
changed = u.copy()
changed[12:] += 10.0
changed_y, _ = selective_scan(changed)
assert np.allclose(y[:12], changed_y[:12])
left, boundary = selective_scan(u[:12])
right, resumed_state = selective_scan(u[12:], boundary)
assert np.allclose(y, np.r_[left, right])
assert np.allclose(final_state, resumed_state)
reset_right, _ = selective_scan(u[12:])
assert not np.allclose(right, reset_right)
print('causal prefix and carried-state chunk checks passed')
```

The output confirms both checks. Resetting the state at a batch boundary changes the computation. Conversely, sharing state between independent examples leaks information. A streaming implementation must track which sequence owns each state and reset it at genuine sequence boundaries.

### Mamba-2 and the attention connection

Mamba-2 [29] changes both the inner sequence mixer and the surrounding block. Its structured state-space duality uses a scalar-times-identity transition within each head and organizes computation into chunks whose dense matrix multiplications suit accelerator hardware. It also restructures parameter projections and normalization. This is more specific than adding attention to the original Mamba block.

With a zero initial state, the scalar-transition recurrence can be expanded as

$$y_t = \sum_{j=1}^{t} \left[ C_t \left( \prod_{k=j+1}^{t} a_k \right) B_j \right] u_j.$$

Here $j$ indexes earlier inputs, $k$ indexes intervening transitions, $a_k$ is the scalar discrete transition multiplying the identity, $B_j$ is the discrete write column, $C_t$ is the read row, and an empty product equals one. The bracketed quantity is a structured causal mixing coefficient. This exposes an attention-like matrix view without requiring ordinary softmax attention or storing that matrix. Hu et al. [30] (ICML 2026) extend the duality to diagonal transitions and clarify why it does not extend directly to standard softmax attention.

At fixed width and state dimension, recurrent inference carries a fixed-size state, whereas ordinary cached causal attention retains keys and values whose storage grows with context length. Compression saves memory but puts pressure on exact recall of many unrelated details. Linear sequence scaling also does not imply constant training memory: activations, gradients and recom-

putation policies matter. Evaluate retrieval quality, context length and measured device latency together. The image-specific ordering problem appears next in ML 05.

## Modern view

**Survey literature.** Goodfellow et al.'s chapter 10 [1] remains the clearest unified treatment of recurrent and recursive networks, echo-state ideas, and long-term dependencies. Lipton, Berkowitz and Elkan [20] wrote a critical review of RNNs for sequence learning in 2015: a self-contained, historically ordered account of three decades of work leading to LSTMs and bidirectional RNNs, with conflicting notation reconciled and applications to captioning, translation and handwriting recognition. Greff et al. [19] is less a survey than a large empirical study, but it is the reference for which LSTM components matter. Two recent reviews by Tiezzi et al. reframe recurrence for the Transformer era: an extensive survey [26] places Transformer–RNN hybrids, attention variants with recurrent form, and deep state-space models (a special class of linear RNNs) under the single umbrella of recurrence, and highlights research opportunities in learning online from potentially infinite streams rather than fixed-length sequences; and a shorter perspective in *Nature Machine Intelligence* [27] revisits recurrent processing at the crossroad of Transformers and state-space models.

**Recurrence returns (2023–2026).** Transformers process all pairs of positions, which costs memory and computation quadratic in length. A family of models revisits recurrence to get linear-time inference with a constant-size state:

- **Linear recurrent units.** Orvieto et al. [24] (ICML 2023) showed that a carefully parameterized **linear** recurrence with diagonal complex-valued state matrices, stable initialization and normalization matches deep state-space models on long-range benchmarks, and that the nonlinearity can live outside the recurrence.
- **Selective state spaces.** Mamba [21] makes the state-space parameters functions of the input, so the model can choose what to store or ignore, as LSTM gates do, while a hardware-aware parallel scan keeps training efficient. Vision variants and the debate about whether they are necessary for images are covered in ML 05.
- **Revived LSTMs and GRUs.** xLSTM [22] (NeurIPS 2024) adds exponential gating and a matrix-valued memory to the LSTM and scales it to language-model sizes. Feng et al. [23] (2024) remove the hidden-state dependence from the gates of LSTMs and GRUs ("minLSTM", "minGRU"), which allows a parallel scan during training, and report results competitive with recent sequence models.

The shared lesson is that the **gated, additive state update** invented for the LSTM was right; what limited classical RNNs was mostly sequential training. Making the recurrence linear in the state (so it can be computed with a parallel scan) recovers parallel training while keeping cheap, streaming inference.

**Open problems.** Length generalization beyond the training horizon; credit assignment over very long streams without full BPTT; how much a fixed-size state can remember compared with attention's growing cache, a real limitation for tasks that need exact recall of distant details; and

choosing a scan order for 2-D images and 3-D video, where no single natural sequence order exists.

## Key takeaways

- An RNN reuses one state-update rule across time; parameter sharing does not guarantee useful long-term memory.
- BPTT sums gradients of shared parameters; truncation limits gradient history while allowing state to continue.
- Clipping limits exploding gradients; additive gated paths help gradient flow without eliminating every long-range learning difficulty.
- Bidirectional encoders use future inputs; causal streaming requires different access and state handling.
- Attention lets a decoder read multiple encoder states; recursive networks instead compose representations along a tree.
- Selective state spaces trade a compact streaming state for compressed history; compare recall and computation under the intended workload.

## Exercises

1. **Causality.** Your team reports 92% frame-level accuracy for an action-recognition model with a bidirectional LSTM and proposes deploying it for live alerts. What is wrong with this plan, and how would you evaluate the deployable alternative fairly? Run the selective-scan example, then add a backward scan: which causality check should fail?

   **Hint**

   The backward LSTM at frame $t$ has read frames after $t$, which a live system has not seen yet. Retrain a unidirectional (causal) model, or allow a fixed look-ahead delay and evaluate with exactly that delay. Report accuracy as a function of latency. A backward scan reads the altered future, so the combined prefix generally changes even when the forward prefix does not.

2. **Length generalization.** Train the GRU sum classifier at length 12, then evaluate on fresh sequences of length 6, 24, 100 and 400 without retraining. Plot accuracy versus length. Then repeat with an LSTM and with training on mixed lengths 6–24.

   **Hint**

   Generate fresh data per length. The sum's scale grows like $\sqrt{T}$, so the state may saturate on long inputs even if the logic is right. Mixed-length training usually helps more than switching cell type. Success at length 12 says nothing by itself about length 400.

3. **Gradient bound.** For the scalar RNN $h_t = \tanh(wh_{t-1} + x_t)$, show that $|\partial h_t / \partial h_s| \leq |w|^{t-s}$. For what values of $w$ can the gradient explode, and why does saturation of tanh work against explosion?

   **Hint**

   Each factor is $w(1 - h_k^2)$ with $0 \leq 1 - h_k^2 \leq 1$. Explosion requires $|w| > 1$, but a large $|w|$ also drives $h_k$ toward $\pm 1$, which makes $1 - h_k^2$ small. The gradient explodes only on trajectories that stay in the non-saturated region, which is why explosions are bursty in practice.

4. **LSTM memory.** In the LSTM equations, suppose $f_t = 1$ and $i_t = 0$ for all $t$ from $s$ to $t$. What is $c_t$ in terms of $c_s$, and what is $\partial c_t / \partial c_s$ along the cell path? What would happen if the forget gate were fixed at 1 forever, as in the original 1997 design, on a never-ending input stream?

   **Hint**

   $c_t = c_s$ and $\partial c_t / \partial c_s = I$: perfect memory and perfect gradient flow. With $f_t \equiv 1$ and nonzero writes, the cell state accumulates without bound over an endless stream, which is exactly the problem Gers et al. solved by letting the network learn to forget.

5. **Attention shapes and masks.** In the additive-attention code, a batch has encoder states of shape $8 \times 40 \times 256$ (with some sequences padded) and a decoder state of shape $8 \times 512$. Give the shapes of $e$, $\alpha$ and the context, and explain what would go wrong if padded positions were masked with 0 instead of $-\infty$.

   **Hint**

   $e$ and $\alpha$ are $8 \times 40$; the context is $8 \times 256$. A score of 0 is an ordinary score: $e^0 = 1$ would still give padding positive weight, and the context would mix in meaningless padded states.

6. **Recursive versus recurrent.** For a balanced binary tree over $n = 1024$ leaves, how many composition steps lie between a leaf and the root? Compare with a left-to-right RNN over the same 1024 inputs, and relate the answer to the vanishing-gradient bound.

   **Hint**

   $\log_2 1024 = 10$ steps versus up to 1023 for the chain. The gradient bound decays like $(\text{factor})^{\text{path length}}$, so a shorter path from each input to the output helps optimization. Attention takes this to the extreme: path length one.

## References

1. I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016, ch. 10. Book
2. S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, 1997. DOI

3. K. Cho et al., “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation,” EMNLP, 2014. arXiv
4. K. S. Tai, R. Socher and C. D. Manning, “Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks,” ACL, 2015. arXiv
5. F. A. Gers, J. Schmidhuber and F. Cummins, “Learning to Forget: Continual Prediction with LSTM,” *Neural Computation*, 2000. DOI
6. Y. Bengio, P. Simard and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Transactions on Neural Networks*, 1994. DOI
7. R. Pascanu, T. Mikolov and Y. Bengio, “On the difficulty of training Recurrent Neural Networks,” ICML, 2013. arXiv
8. P. J. Werbos, “Backpropagation through time: what it does and how to do it,” *Proceedings of the IEEE*, 1990. DOI
9. J. L. Elman, “Finding Structure in Time,” *Cognitive Science*, 1990. DOI
10. M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Transactions on Signal Processing*, 1997. DOI
11. I. Sutskever, O. Vinyals and Q. V. Le, “Sequence to Sequence Learning with Neural Networks,” NeurIPS, 2014. arXiv
12. D. Bahdanau, K. Cho and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” ICLR, 2015. arXiv
13. O. Vinyals, A. Toshev, S. Bengio and D. Erhan, “Show and Tell: A Neural Image Caption Generator,” CVPR, 2015. arXiv
14. K. Xu et al., “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention,” ICML, 2015. arXiv
15. J. Donahue et al., “Long-term Recurrent Convolutional Networks for Visual Recognition and Description,” CVPR, 2015. arXiv
16. X. Shi et al., “Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting,” NeurIPS, 2015. arXiv
17. R. Socher et al., “Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank,” EMNLP, 2013. Paper
18. R. Socher, C. C.-Y. Lin, A. Y. Ng and C. D. Manning, “Parsing Natural Scenes and Natural Language with Recursive Neural Networks,” ICML, 2011. Paper
19. K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink and J. Schmidhuber, “LSTM: A Search Space Odyssey,” *IEEE Transactions on Neural Networks and Learning Systems*, 2017. arXiv
20. Z. C. Lipton, J. Berkowitz and C. Elkan, “A Critical Review of Recurrent Neural Networks for Sequence Learning,” arXiv:1506.00019, 2015. arXiv
21. A. Gu and T. Dao, “Mamba: Linear-Time Sequence Modeling with Selective State Spaces,” COLM, 2024. arXiv
22. M. Beck et al., “xLSTM: Extended Long Short-Term Memory,” NeurIPS, 2024. Paper
23. L. Feng et al., “Were RNNs All We Needed?,” arXiv:2410.01201, 2024. arXiv
24. A. Orvieto et al., “Resurrecting Recurrent Neural Networks for Long Sequences,” ICML, 2023. arXiv
25. S. Bengio, O. Vinyals, N. Jaitly and N. Shazeer, “Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks,” NeurIPS, 2015. arXiv
26. M. Tiezzi, M. Casoni, A. Betti, T. Guidi, M. Gori and S. Melacci, “On the Resurgence of Recur-

rent Models for Long Sequences: Survey and Research Opportunities in the Transformer Era," arXiv:2402.08132, 2024. arXiv
27. M. Tiezzi, M. Casoni, A. Betti, T. Guidi, M. Gori and S. Melacci, "Back to recurrent processing at the crossroad of transformers and state-space models," *Nature Machine Intelligence*, 2025. DOI
28. A. Vaswani et al., "Attention Is All You Need," NeurIPS, 2017. arXiv
29. T. Dao and A. Gu, "Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality," ICML, PMLR 235, 2024. Paper
30. J. Y.-C. Hu, X. Zhang, A. Elsheikh, W. Wu and H. Liu, "On Structured State-Space Duality," ICML, PMLR 306, 2026. Paper

# ML 05 • Attention and Vision Transformers

Build self-attention from queries, keys and values; add multiple heads, positional encodings and the Transformer block; turn images into patch tokens with ViT; then study DeiT, Swin, hybrid CNN–Transformer models, efficient attention and state-space alternatives.

Online edition: https://blog.shyandram.dev/en/learn/ml/vision-transformers/

## What you'll learn

- Trace queries, keys, values and attention weights.
- Construct multi-head, masked and cross-attention.
- Explain positional information and the Transformer block.
- Compare patch-based, windowed and hybrid vision models.
- Separate attention's arithmetic, memory traffic and context trade-offs.

## The big picture

A CNN (ML 03) moves information through fixed local neighborhoods: a pixel learns about a distant pixel only after many layers. **Attention** lets every location decide, based on its current content, which other locations to read from, in a single step. To recognize a bird, a patch containing its head can directly consult a distant patch containing its wing.

> **Explain like I'm five**
>
> Imagine a classroom where each child holds one piece of a jigsaw puzzle. Each child asks a question about their piece ("I have a blue edge, who has sky?"), and every other child holds up a card describing their own piece. Each child listens most closely to the classmates whose cards best match the question, and updates their idea of what their piece shows. Then everyone does it again with the improved ideas. Attention is the listening rule. A Transformer repeats many rounds of listening, with a little thinking time in between.

The attention idea came from machine translation with recurrent networks: ML 04 shows how it removed the single-vector bottleneck of encoder–decoder RNNs. The Transformer of Vaswani et al. [1] went further and removed recurrence entirely, building both encoder and decoder from attention and small per-token networks. The Vision Transformer (ViT) of Dosovitskiy et al. [2] applied the Transformer encoder almost unchanged to sequences of image patches. Bishop and Bishop [10] give a modern textbook treatment, and the surveys discussed in the Modern view [3], [11], [12] map the many vision variants. Afterwards, DLCV 02 compares CNN and Transformer backbones for classification in practice, and ML 06 shows how such backbones are pretrained without labels.

## Self-attention: queries, keys and values

**Plain version.** A query scores the available keys; normalized scores weight the values. This lets each token select content from the tokens it can access.

**Precise version.** Stack $N$ tokens as the rows of $X \in \mathbb{R}^{N\times D}$, where $D$ is the model width. Three learned matrices project them:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V,$$

with $W_Q, W_K \in \mathbb{R}^{D\times d_k}$ and $W_V \in \mathbb{R}^{D\times d_v}$. **Scaled dot-product attention** [1] is

$$\text{Attention}(Q,K,V) = \text{softmax}_{\text{row}}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V = AV.$$

$QK^\top$ is an $N\times N$ matrix of **scores**: entry $(i,j)$ is the dot product of query $i$ with key $j$, a measure of how relevant token $j$ is to token $i$. The row-wise softmax turns each row into nonnegative weights that sum to one, giving the **attention matrix** $A$. The output $Y = AV \in \mathbb{R}^{N\times d_v}$ has one row per token: row $i$ is $\sum_j A_{ij}v_j$, a weighted average of value vectors.

**Why three different projections?** If queries and keys were the same vector, every token would match itself most strongly, and relevance would be forced to be symmetric. Separate $W_Q$ and $W_K$ let "what I seek" differ from "what I offer" (an eye patch can seek a nose without being one), and a separate $W_V$ lets "what I offer" differ from "what I pass on".

**Why divide by $\sqrt{d_k}$?** If the components of a query and a key are independent with mean zero and variance one, their dot product $q\cdot k = \sum_{l=1}^{d_k} q_l k_l$ has mean zero and variance $d_k$. Without scaling, larger heads produce larger scores, the softmax saturates (almost all weight on one token), and its gradients become tiny. Dividing by $\sqrt{d_k}$ restores unit variance. It is a scale convention, not a normalization of each query to unit length.

**Content, not position, decides who talks to whom.** Unlike a convolution, whose weights are tied to relative positions, attention weights are computed from the data at run time. A related consequence: self-attention is **permutation-equivariant**. Shuffle the input tokens, and the outputs are shuffled in the same way, because nothing in the formula refers to token order. The snippet checks both the shapes and this symmetry.

```
import torch

torch.manual_seed(0)
X = torch.randn(5, 8)                    # N=5, D=8
Wq, Wk, Wv = [torch.randn(8, 4) for _ in range(3)]
def attention(x):
    q, k, v = x @ Wq, x @ Wk, x @ Wv
    a = (q @ k.T / q.shape[-1]**.5).softmax(dim=-1)
    return a @ v, a
y, a = attention(X)
perm = torch.tensor([3, 0, 4, 1, 2])
```

```
yp, _ = attention(X[perm])
assert torch.allclose(a.sum(-1), torch.ones(5), atol=1e-6)
assert torch.allclose(yp, y[perm], atol=1e-5)
print("output shape:", tuple(y.shape))
```

This symmetry suits sets, but an image is not a set of patches: a face with its eyes below its mouth is not a face. Positional information, introduced below, breaks the symmetry.

## Multi-head attention

**Plain version.** Run several attention operations in parallel, each with its own smaller projections, so different heads can track different relationships (nearby texture, the same object, symmetric parts). Then concatenate their outputs and mix them.

**Precise version.** With $h$ heads, head $r$ has its own $W_Q^{(r)}, W_K^{(r)}, W_V^{(r)} \in \mathbb{R}^{D\times d_k}$, usually with $d_k = d_v = D/h$:

$$\text{head}_r = \text{Attention}\big(XW_Q^{(r)}, XW_K^{(r)}, XW_V^{(r)}\big), \qquad \text{MHA}(X) = [\,\text{head}_1, \dots, \text{head}_h\,]\,W_O,$$

where the brackets concatenate the $h$ outputs of width $D/h$ back to width $D$, and $W_O \in \mathbb{R}^{D\times D}$ mixes them. Because each head has width $D/h$, multi-head attention costs about the same as one full-width head, and its projections have $4D^2$ weights in total (plus biases). Heads are not guaranteed to learn human-readable roles; some are redundant and can be pruned.

The implementation below splits the projections into heads, computes the scores per head, concatenates and projects, and matches `nn.MultiheadAttention` exactly. It also shows a **causal mask**: setting forbidden scores to $-\infty$ before the softmax gives them exactly zero weight.

```
import torch
from torch import nn

torch.manual_seed(0)
B, N, D, h = 2, 6, 16, 4
mha = nn.MultiheadAttention(D, num_heads=h, batch_first=True, bias=False)
X = torch.randn(B, N, D)

def multi_head(x):
    Wq, Wk, Wv = mha.in_proj_weight.chunk(3)          # each D x D (rows = outputs)
    q, k, v = x @ Wq.T, x @ Wk.T, x @ Wv.T              # B x N x D
    split = lambda t: t.view(B, N, h, D // h).transpose(1, 2)   # B x h x N x d_k
    q, k, v = split(q), split(k), split(v)
    scores = q @ k.transpose(-2, -1) / (D // h) ** 0.5  # B x h x N x N
    A = scores.softmax(dim=-1)
    heads = A @ v                                       # B x h x N x d_k
    concat = heads.transpose(1, 2).reshape(B, N, D)     # put the heads side by side
    return concat @ mha.out_proj.weight.T, A

ours, A = multi_head(X)
ref, _ = mha(X, X, X)
```

```
print("matches nn.MultiheadAttention:", torch.allclose(ours, ref, atol=1e-5))
print("attention tensor:", tuple(A.shape), "| parameters:",
      sum(p.numel() for p in mha.parameters()), "= 4 * D^2 =", 4 * D * D)

# Causal mask: query i may only read keys j <= i
causal = torch.triu(torch.ones(N, N, dtype=torch.bool), diagonal=1)   # True = forbidden
out_c, w_c = mha(X, X, X, attn_mask=causal)
print("no weight on the future:", torch.all(w_c[:, causal] == 0).item())
```

It prints `True`, an attention tensor of shape `(2, 4, 6, 6)` (batch, heads, queries, keys), `1024 = 4 * D^2 = 1024`, and `True` for the causal check. Production code can use `torch.nn.functional.scaled_dot_product_attention`, which dispatches to fused kernels when supported [7]. **Do not copy the Boolean mask unchanged:** `nn.MultiheadAttention` uses `True` for forbidden positions, while `scaled_dot_product_attention` uses `True` for allowed positions. Also pass `dropout_p=0.0` to the functional call during evaluation; it does not inspect a module's training flag. Check mask semantics and dropout explicitly when replacing the teaching implementation.

## Three ways to connect tokens

**Plain version.** Which tokens a query is allowed to read is part of the model's design, expressed as a mask on the score matrix.

| Mechanism | A query reads | Typical use |
|---|---|---|
| Bidirectional self-attention | All tokens of the same sequence | ViT and other image encoders |
| Causal (masked) self-attention | Itself and earlier positions only | Next-token generation, autoregressive image models |
| Cross-attention | Tokens from a different sequence | A decoder reading an encoder; image tokens reading text |

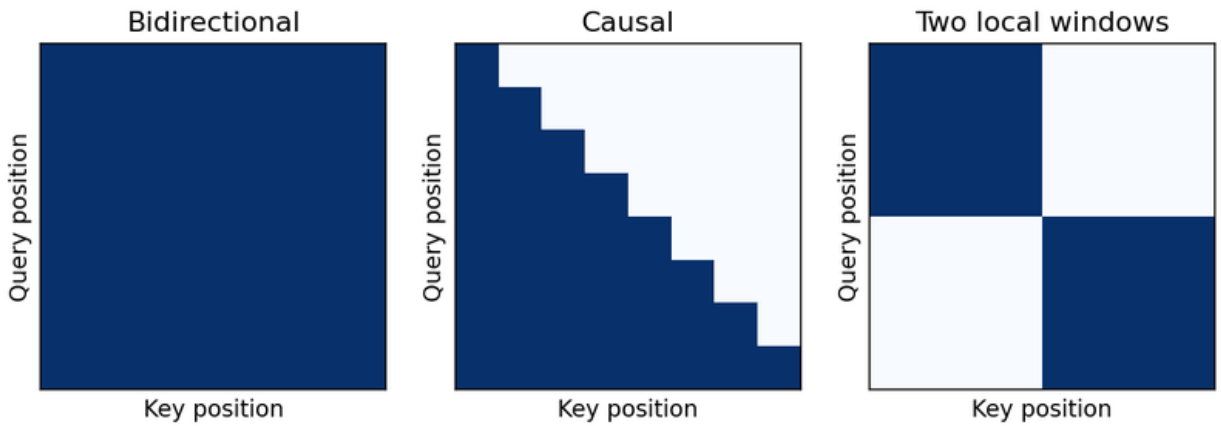



*Figure 1 — Attention masks: blue entries are allowed, white entries are forbidden. Causal generation also shifts its inputs by one position so a target cannot simply be copied.*

In **cross-attention**, queries come from one sequence $X$ and keys and values from another sequence $Z$: $Q = XW_Q$, $K = ZW_K$, $V = ZW_V$. The score matrix has shape $N_X \times N_Z$, and the output has one row per query. This is how the original Transformer decoder reads its encoder, how image captioners read image features, and how text-to-image diffusion models read a text prompt (ML 08) without being autoregressive over image patches.

Attention weights are part of a computation, not automatically an explanation. A large weight can multiply an uninformative value, and the residual and MLP paths also shape the output.

## Positional information

**Plain version.** Content-only self-attention has no built-in position labels. Position embeddings, relative information or position-dependent masks supply structure.

**Sinusoidal encodings.** The original Transformer [1] adds a fixed vector $p_{pos} \in \mathbb{R}^D$ to the token at position *pos*:

$$p_{pos,2i} = \sin\left(\frac{pos}{10000^{2i/D}}\right), \qquad p_{pos,2i+1} = \cos\left(\frac{pos}{10000^{2i/D}}\right), \qquad i = 0, \dots, \frac{D}{2} - 1.$$

Each pair of dimensions $(2i, 2i+1)$ is a point rotating around a circle at angular frequency $\omega_i = 10000^{-2i/D}$, from fast (one radian per position) to very slow. Two properties follow from the angle-addition formulas. First, for any fixed offset $k$, $p_{pos+k} = p_{pos} R_k$ for a block-diagonal rotation $R_k$ that does not depend on *pos*; a linear layer can therefore learn to "look $k$ positions ahead". Second, $p_m \cdot p_n = \sum_i \cos(\omega_i (m-n))$ depends only on the offset $m - n$ (Figure 2, middle).

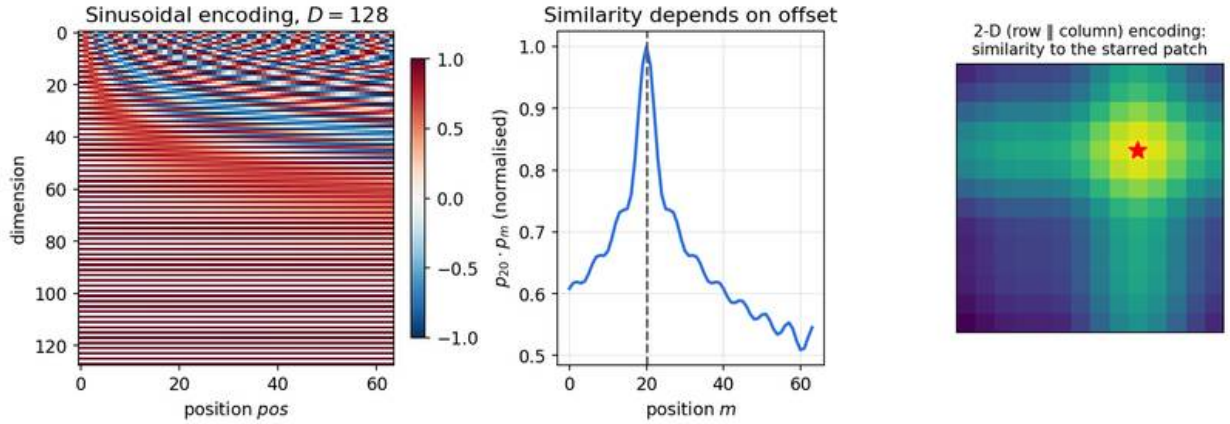



*Figure 2 — Sinusoidal positional encodings. Left: each dimension pair oscillates at its own frequency. Middle: similarity between encodings depends on the offset. Right: concatenating row and column encodings gives a 2-D encoding for a 14 × 14 patch grid.*

```
import numpy as np

def sinusoidal(n_pos, D):
```

```
    pos = np.arange(n_pos)[:, None]
    omega = 1.0 / 10000 ** (np.arange(0, D, 2) / D)    # one frequency per sin/cos pair
    pe = np.zeros((n_pos, D))
    pe[:, 0::2] = np.sin(pos * omega)
    pe[:, 1::2] = np.cos(pos * omega)
    return pe, omega

pe, omega = sinusoidal(100, 16)
k = 7                                                  # a fixed offset
# block-diagonal rotation that maps p_pos to p_{pos+k}, the same for every pos
R = np.zeros((16, 16))
for i, w in enumerate(omega):
    c, s = np.cos(w * k), np.sin(w * k)
    R[2 * i:2 * i + 2, 2 * i:2 * i + 2] = [[c, -s], [s, c]]
print("p[pos+k] == p[pos] @ R for all pos:", np.allclose(pe[:-k] @ R, pe[k:]))
print("dot products depend only on offset:",
      np.isclose(pe[10] @ pe[17], pe[50] @ pe[57]))
```

Both checks print `True`.

**Other schemes.**

- **Learned absolute embeddings**: a trainable vector per position, as in ViT [2]. ViT's 1-D learned embeddings, indexed by raster order, worked about as well as explicitly 2-D ones; after training, embeddings of patches in the same row or column became similar, so the model learned the grid by itself. At a new resolution, the grid of embeddings is resized by 2-D interpolation before fine-tuning.
- **Relative position representations** (Shaw et al. [17]) add a learned term that depends on the offset $j - i$ to the score between tokens $i$ and $j$. Swin [5] adds a learned **relative position bias** table to each window's scores.
- **Rotary position embedding (RoPE)** (Su et al. [16]) rotates each query and key by an angle proportional to its position, using the same rotations as the sinusoidal code above. The score $q_m^\top k_n$ then depends on the positions only through $m - n$. RoPE is standard in large language models and increasingly used in vision with separate rotations for the two image axes.
- **No explicit encoding**: convolutions inside the network, as in CvT [19], or zero padding at borders can leak enough position information on their own.

## The Transformer block

**Plain version.** A block alternates two jobs. Attention lets tokens exchange information; a small MLP then processes each token on its own. Residual connections and normalization keep the stack stable.

**Precise version.** The **pre-norm** block used by ViT and most modern models is

$$U = X + \mathrm{MHA}(\mathrm{LN}(X)), \qquad Y = U + \mathrm{MLP}(\mathrm{LN}(U)), \qquad \mathrm{MLP}(u) = W_2\,\mathrm{GELU}(W_1 u + b_1) + b_2.$$

LN is **layer normalization** [14]: it standardizes each token's $D$ features to zero mean and unit variance, then applies a learned scale and shift; unlike BatchNorm, it does not depend on other

samples in the batch. The MLP expands each token from $D$ to typically $4D$ hidden units, applies the GELU nonlinearity [26] ($\mathrm{GELU}(z) = z\,\Phi(z)$, where $\Phi$ is the standard normal CDF), and projects back. The residual additions mean each sub-layer only learns a correction, exactly as in ResNet.

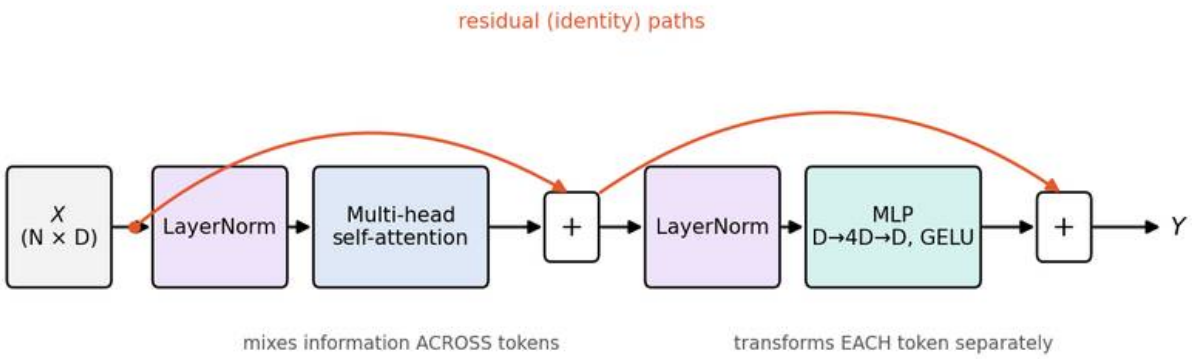



*Figure 3 — A pre-norm Transformer block. Attention mixes information across tokens; the MLP transforms each token separately; residual paths (orange) carry the input around each branch.*

The original Transformer placed LayerNorm **after** each residual addition (post-norm). Xiong et al. [15] showed that with post-norm the gradients near the output are large at initialization, which is why it needs a learning-rate warm-up, whereas pre-norm keeps gradients well behaved and can train without one. So "a Transformer" is not fully specified until you say where the normalization goes. Counting weights per block: $4D^2$ for attention and $8D^2$ for the MLP, so about $12D^2$, two thirds of it in the MLP.

## The Vision Transformer (ViT)

**Plain version.** Cut the image into square patches, turn each patch into a vector, add position information, and feed the sequence to a standard Transformer encoder. Read the answer from a special extra token.

**Patchify.** For an image with $H \times W$ pixels and $C$ channels, and patch size $P$ dividing both $H$ and $W$, there are $N = HW/P^2$ patches. Flatten each into $x_i^{\text{patch}} \in \mathbb{R}^{P^2C}$ and project linearly:

$$z_0 = [\, x_{\text{cls}};\; x_1^{\text{patch}}E;\; \dots;\; x_N^{\text{patch}}E \,] + E_{\text{pos}},$$

where $E \in \mathbb{R}^{P^2C \times D}$ is the learned **patch embedding**, $x_{\text{cls}} \in \mathbb{R}^D$ is a learned **class token** prepended to the sequence, and $E_{\text{pos}} \in \mathbb{R}^{(N+1)\times D}$ holds learned position embeddings. After $L$ blocks, a LayerNorm and a linear head applied to the class token's output give the logits. Averaging all patch outputs (mean pooling) is an equally common alternative to the class token.

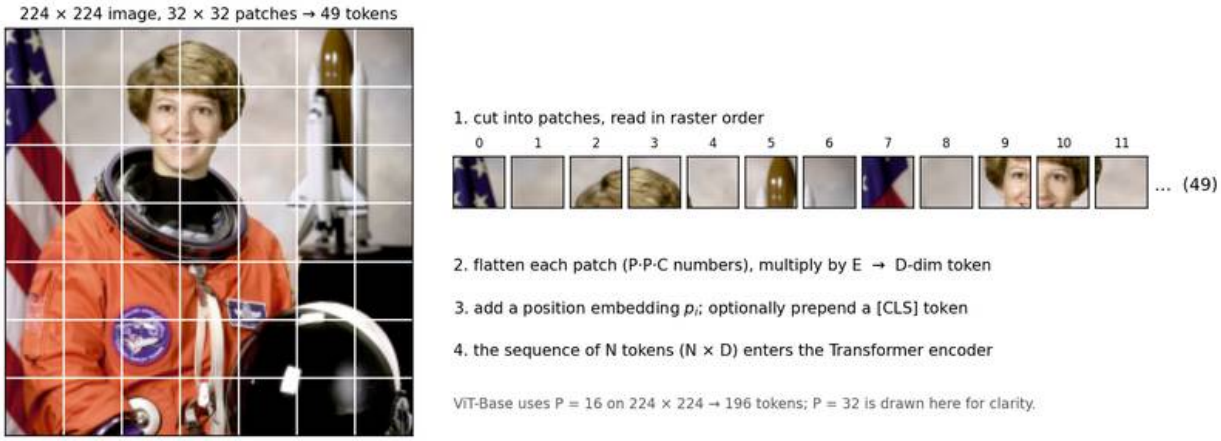



*Figure 4 — Patchify. A 32-pixel grid is drawn for readability; ViT-Base uses 16 × 16 patches, which turn a 224 × 224 image into 196 tokens.*

A linear map applied to non-overlapping $P \times P$ patches is exactly a convolution with kernel size and stride both equal to $P$, which is how implementations write it:

```
import torch
from torch import nn

torch.manual_seed(0)
B, C, H, W, P, D = 2, 3, 32, 32, 8, 48
x = torch.randn(B, C, H, W)

# 1) explicit: cut into P x P patches, flatten each, multiply by E
patches = x.unfold(2, P, P).unfold(3, P, P)                 # B, C, H/P, W/P, P, P
patches = patches.permute(0, 2, 3, 1, 4, 5).reshape(B, -1, C * P * P)   # B, N, P*P*C
E = torch.randn(C * P * P, D) * 0.02
tokens_explicit = patches @ E                                # B, N, D

# 2) the same linear map written as a convolution with kernel = stride = P
conv = nn.Conv2d(C, D, kernel_size=P, stride=P, bias=False)
conv.weight.data = E.T.reshape(D, C, P, P)
tokens_conv = conv(x).flatten(2).transpose(1, 2)             # B, N, D

print("tokens:", tuple(tokens_conv.shape), "| identical:",
      torch.allclose(tokens_explicit, tokens_conv, atol=1e-5))
```

It prints 16 tokens of width 48 and `True`.

**Sizes.** The standard configurations are ViT-Base (12 blocks, $D = 768$, 12 heads, about 86 million parameters), ViT-Large (24 blocks, $D = 1024$, 16 heads) and ViT-Huge (32 blocks, $D = 1280$, 16 heads) [2]. "ViT-B/16" means Base with $16 \times 16$ patches. **Patch size is the main compute knob**: halving $P$ quadruples $N$ and multiplies the attention-score count by sixteen. Small patches keep fine detail; large patches are cheap but lose anything finer than the patch, which no later layer can recover.

**The data lesson.** Dosovitskiy et al. [2] found that ViT trained only on ImageNet-1k (about 1.3 million images) was somewhat worse than comparable ResNets, but when pretrained on ImageNet-21k (14 million) or JFT-300M (300 million) and then fine-tuned, it matched or exceeded them at lower pretraining cost. Their explanation is **inductive bias**: a CNN assumes locality and translation equivariance in every layer, whereas a ViT is local only within the patch embedding and must **learn** spatial relationships from data. Raghu et al. [18] compared the internal representations of ViTs and ResNets and found that ViTs have more uniform representations across layers, that their lower layers attend both locally and globally, and that learning local attention in early layers requires large-scale pretraining.

### Code: a tiny ViT trained on digits

This complete example trains a small ViT on the same $8 \times 8$ digits as the CNN in ML 03, with $2 \times 2$ patches (16 tokens), a class token, learned positions and four pre-norm blocks. It runs on CPU in about half a minute.

```
import copy
import numpy as np
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

torch.manual_seed(0)
torch.set_num_threads(1)

class TinyViT(nn.Module):
    def __init__(self, img=8, patch=2, chans=1, D=64, depth=4, heads=4, classes=10):
        super().__init__()
        N = (img // patch) ** 2
        self.patch = nn.Conv2d(chans, D, kernel_size=patch, stride=patch)   # patchify + E
        self.cls = nn.Parameter(torch.zeros(1, 1, D))                       # [CLS] token
        self.pos = nn.Parameter(torch.randn(1, N + 1, D) * .02)             # learned
         ↪  positions
        layer = nn.TransformerEncoderLayer(D, heads, dim_feedforward=4 * D, dropout=0.1,
                                           activation="gelu", batch_first=True,
                                           norm_first=True)                 # pre-norm block
        self.blocks = nn.TransformerEncoder(layer, depth, enable_nested_tensor=False)
        self.norm, self.head = nn.LayerNorm(D), nn.Linear(D, classes)

    def forward(self, x):
        t = self.patch(x).flatten(2).transpose(1, 2)                         # B, N, D
        t = torch.cat([self.cls.expand(len(t), -1, -1), t], dim=1) + self.pos
        return self.head(self.norm(self.blocks(t))[:, 0])                    # read [CLS]

d = load_digits()
X = torch.tensor(d.images[:, None] / 16.0, dtype=torch.float32)
y = torch.tensor(d.target)
train, test = train_test_split(np.arange(len(y)), test_size=.2, stratify=d.target,
                               random_state=0)
train, val = train_test_split(train, test_size=.2, stratify=d.target[train], random_state=1)
```

```
model = TinyViT()
opt = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=0.05)
best = (float("inf"), None)
for epoch in range(30):
    model.train()
    for idx in torch.tensor(train)[torch.randperm(len(train))].split(64):
        loss = nn.functional.cross_entropy(model(X[idx]), y[idx])
        opt.zero_grad(); loss.backward(); opt.step()
    model.eval()
    with torch.no_grad():
        v = nn.functional.cross_entropy(model(X[val]), y[val]).item()
    if v < best[0]:
        best = (v, copy.deepcopy(model.state_dict()))
model.load_state_dict(best[1]); model.eval()
with torch.no_grad():
    acc = (model(X[test]).argmax(1) == y[test]).float().mean().item()
print("parameters:", sum(p.numel() for p in model.parameters()))
print("best validation loss:", round(best[0], 4), "| held-out accuracy:", round(acc, 4))
```

On our run it uses about 202,000 parameters and reaches about 95% held-out accuracy, while the CNN in ML 03 reached about 96% with fewer than 10,000 parameters and the same split. On a small dataset, the CNN's built-in locality is worth a factor of twenty in parameters. This is the data lesson in miniature, not a verdict on large-scale performance.

*Online only: ViT — image patches and the role of pretraining scale — https://blog.shyandram.dev/en/learn/ml/vision-transformers/papers/vit–image-patches-and-the-role-of-pretraining-scale/*

## Training recipes: DeiT and beyond

**Plain version.** Much of the early gap between ViTs and CNNs on ImageNet-sized data turned out to be about training, not architecture.

DeiT (Touvron et al. [4]) trained ViTs competitively **on ImageNet-1k alone** using strong data augmentation (RandAugment, Mixup, CutMix, random erasing), heavy regularization (stochastic depth, label smoothing) and long schedules. It also introduced a **distillation token**: an extra learned token, like the class token, whose output is trained to match the predictions of a pretrained CNN teacher. The CNN's inductive bias is thus transferred through its predictions. DeiT III [27] later showed that a simpler, carefully tuned supervised recipe (a lightweight "3-Augment" scheme, simple cropping and a binary cross-entropy loss) makes plain ViTs stronger still. Steiner et al. [25] ran a large study of data, augmentation and regularization ("AugReg") and found that the right combination of augmentation, regularization and compute can match models trained on roughly ten times more data.

The general lesson, repeated in ML 03's discussion of ConvNeXt: compare architectures only under matched data, augmentation, resolution, schedule and compute.

## Swin and hierarchical Transformers

**Plain version.** Global attention over thousands of tokens is expensive, and plain ViT keeps one coarse resolution throughout, which is awkward for detection and segmentation. Swin restricts attention to local windows, shifts the windows between layers so information still spreads, and merges patches to build a CNN-like pyramid.

**Windows.** Swin [5] partitions the token grid into non-overlapping windows of $M \times M$ tokens (typically $M = 7$) and computes self-attention inside each window only. For an $h \times w$ token grid with $C$ channels, the paper gives the costs

$$\Omega(\text{global MSA}) = 4hwC^2 + 2(hw)^2C, \qquad \Omega(\text{window MSA}) = 4hwC^2 + 2M^2hwC,$$

where $4hwC^2$ counts the four projections and the second term counts the score and weighting products. Global cost is quadratic in the number of tokens $hw$; window cost is linear for fixed $M$.

**Shifted windows.** With fixed windows, tokens in different windows would never interact. Swin alternates regular windows with windows displaced by $\lfloor M/2 \rfloor$ tokens in each direction. A token near a window boundary at layer $\ell$ shares a window with tokens from neighboring windows at layer $\ell + 1$, so information crosses boundaries (Figure 5). An efficient implementation cyclically shifts the feature map and uses a mask so that tokens wrapped from opposite edges do not attend to each other.

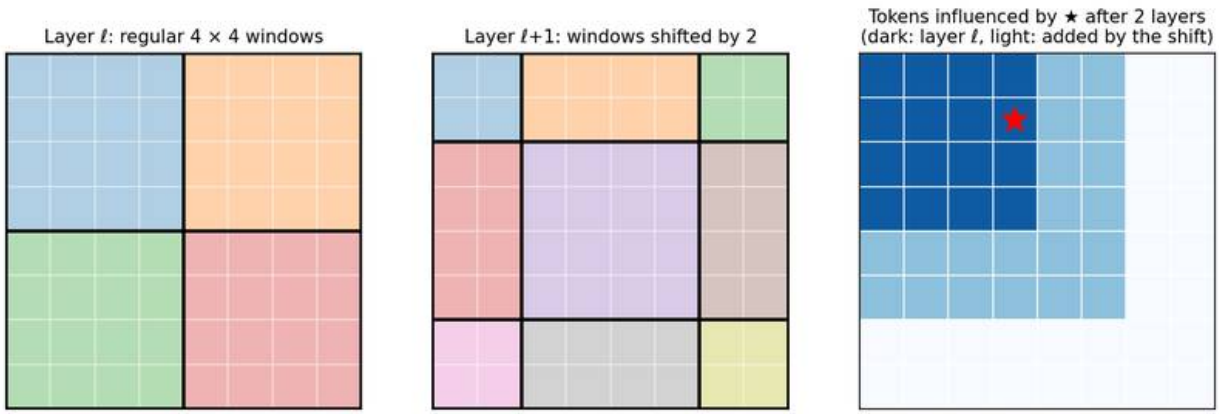



*Figure 5 — Swin's regular and shifted windows on an 8 × 8 token grid. After two layers, information from the starred token reaches the dark window (layer $\ell$) plus the light band added by the shifted windows (layer $\ell$+1).*

**Hierarchy.** Between stages, **patch merging** concatenates each 2 × 2 group of neighboring tokens and projects them linearly, halving the resolution and doubling the channels, as a strided convolution would. The four stages produce feature maps at 1/4, 1/8, 1/16 and 1/32 of the input resolution, the same pyramid a ResNet produces, so Swin can replace a CNN backbone in detection and segmentation frameworks. Each window also adds a learned **relative position bias** to its scores. Swin V2 [32] scaled the design up with residual post-normalization, scaled cosine attention and a continuous, log-spaced position bias that transfers better across window sizes and resolutions.

## Hybrid CNN–Transformer models

**Plain version.** Convolutions are good at local detail with little data; attention is good at long-range interaction. Hybrids use each where it helps.

- **Convolutional stem.** The original ViT paper already tested a "hybrid" in which the patches come from a ResNet feature map instead of raw pixels [2]; it helped at smaller compute budgets.
- **Convolutions inside attention.** CvT [19] uses a convolutional token embedding at each stage and depthwise-convolutional projections for queries, keys and values, which builds in locality and lets it drop explicit position embeddings.
- **Stacking by stage.** CoAtNet [20] places depthwise-convolution stages early, where resolution is high and local patterns dominate, and relative-attention stages later; it reports strong results across data scales because the convolutional stages supply inductive bias when data is limited.
- **Mobile hybrids.** MobileViT [28] inserts small Transformer blocks into a MobileNet-style network to add global context at mobile cost.
- **The reverse direction.** ConvNeXt (see ML 03) imported Transformer-era design choices into a pure CNN and matched Swin.

## Cost and efficient attention

**Plain version.** Global attention compares token pairs, so its pairwise work grows quadratically with token count. Efficiency methods reduce memory traffic, interactions or token count.

For $N$ tokens of width $D$, the score matrix costs about $N^2D$ multiply–adds per layer and the projections and MLP about $12ND^2$. A naive implementation stores an $N \times N$ matrix per head. Doubling the image height and width at a fixed patch size multiplies $N$ by four and the pairwise scores by sixteen.

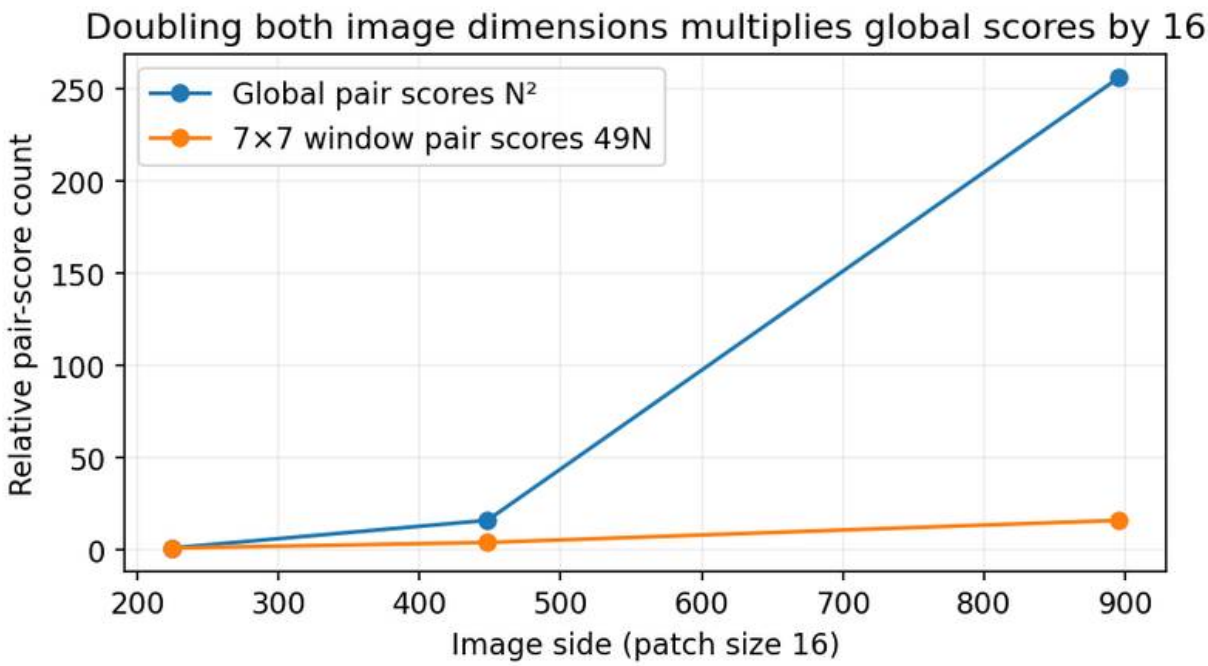



*Figure 6 — Pair-score scaling, not a measured runtime. Projections, MLPs, memory traffic and kernel quality also determine latency.*

The main families:

- **Exact but faster.** FlashAttention [6] computes **exact** attention by tiling the computation so that blocks of queries, keys and values stay in fast on-chip memory, and never writes the full $N \times N$ matrix to slow GPU memory; FlashAttention-2 [29] improves the work partitioning further. The arithmetic is still quadratic; the memory is linear and wall-clock time drops substantially. PyTorch's `scaled_dot_product_attention` selects such kernels automatically when it can [7].
- **Restricted connectivity.** Windowed (Swin) or other sparse patterns reduce the number of pairs, at the price of needing several layers for global communication.
- **Fewer tokens.** Token Merging (ToMe) [21] gradually merges similar tokens inside a pretrained ViT using a fast matching step, increasing throughput without retraining. Larger patches, token pruning and pooling stages do the same more bluntly.
- **Smaller models.** Papa et al.'s survey [13] organizes efficient ViTs into compact architectures, pruning, knowledge distillation and quantization, and proposes an "Efficient Error Rate" metric to compare models across parameters, bits, FLOPs and size.

## State-space models and hybrids

**Plain version.** Instead of comparing every pair of tokens, carry a compact state through the sequence, as a recurrent network does (ML 04). The cost grows linearly with the number of tokens.

A simplified state-space recurrence over a token sequence is

$$h_t = \bar{A}_t\, h_{t-1} + \bar{B}_t\, x_t, \qquad y_t = C_t\, h_t,$$

where $h_t$ is a hidden state of fixed size, and $\bar{A}_t$, $\bar{B}_t$, $C_t$ are (discretized) state-space matrices. In **selective** models such as Mamba, the write/read coefficients and step sizes depend on input $x_t$; the underlying continuous transition can remain fixed while its discretization varies. Input selection and an affine state update permit a parallel scan [33]. An image has no natural one-dimensional order, so vision variants must choose **scan orders**: Vision Mamba (Vim) [30] scans the patch sequence forward and backward with position embeddings, and VMamba [31] scans along four directions over the 2-D grid. The Visual Mamba survey [8] reviews over 200 papers and identifies scanning as the critical design decision for vision.

A useful counterpoint is MambaOut [9] (CVPR 2025). Its authors argue that Mamba suits tasks with long sequences and causal (autoregressive) structure, and build MambaOut models by removing the state-space component while keeping the gated convolutional blocks. Their models outperform visual Mamba models on ImageNet classification, suggesting that Mamba is unnecessary there, while for detection and segmentation, which involve long token sequences, they find state-space models may still be worth exploring. The broader point is methodological: isolate the mechanism (attention, scan, convolution) from gating, normalization, training recipe and parameter budget before crediting a result to it.

### An image scan is an inductive bias

Mamba [33] supplies an efficient sequence operator; it does not choose an image geometry for us. Vim [30] uses a patch sequence, position embeddings and forward/backward processing. VMamba [31] instead places a four-route selective scan inside a hierarchical visual backbone. Neither label alone specifies receptive-field strength, resolution behavior or runtime. The state dimensions, routes, local operators and downsampling stages all matter.

For a fixed scan order, two adjacent pixels in the image may be many steps apart in the sequence. Conversely, the last pixel of one raster row and the first pixel of the next row become consecutive despite being far apart spatially. A local convolution can supply nearby evidence before the scan carries context farther. Multiple routes offer additional paths, but do not restore every geometric symmetry automatically. Position embeddings can also encode useful location while making transfer to a different grid a separate design problem.

A mathematical way to expose the choice is

$$Y^{(r)} = P_r^{-1}\, \mathcal{S}_r(P_r X), \qquad Y = \frac{1}{m}\sum_{r=1}^{m} Y^{(r)}.$$

Here $X$ contains flattened image tokens, $r$ indexes one of $m$ routes, $P_r$ permutes tokens into that route, $\mathcal{S}_r$ applies its sequence recurrence, $P_r^{-1}$ restores image coordinates, and $Y$ averages the resulting features. This is a conceptual route-fusion model. A learned vision block can use different projections or fusion rules; averaging is chosen here to keep the experiment interpretable.

### CPU experiment: measure directional bias

The next example uses a single scalar state with the same decay on every route. It compares forward/backward raster scans with four raster/column routes, then rotates the input by 90 degrees, processes it and rotates the output back. Everything uses a synthetic 6 × 6 impulse; no weights or images are downloaded. This isolates ordering rather than reproducing a trained Vim or VMamba.

```
import numpy as np

def scan_image(image, four=False):
    ids = np.arange(image.size).reshape(image.shape)
    row, col = ids.ravel(), ids.T.ravel()
    orders = [row, row[::-1]]
    if four:
        orders += [col, col[::-1]]
    outputs = []
    for order in orders:
        h = 0.0
        out = np.zeros(image.size)
        for i in order:
            h = 0.8 * h + 0.2 * image.ravel()[i]
            out[i] = h
        outputs.append(out.reshape(image.shape))
    return np.mean(outputs, axis=0)

image = np.zeros((6, 6))
image[1, 4] = 1.0
errors = []
for four in [False, True]:
    original = scan_image(image, four=four)
    rotated_back = np.rot90(scan_image(np.rot90(image), four=four), -1)
    error = np.max(np.abs(original - rotated_back))
    errors.append(error)
    print('four routes:', four, 'rotation discrepancy:', round(error, 6))
assert 0.0 < errors[1] < errors[0]
```

The maximum discrepancies are `0.063622` for two routes and `0.026738` for four. More routes reduce directional bias in this example, but the remaining discrepancy is nonzero. Rotating a raster path reverses one traversal axis; that path need not belong to the original four-route collection. Even shared coefficients and averaging therefore do not guarantee rotation equivariance. Learned directional projections, position information and local operations introduce further choices. Test the property you need instead of inferring it from "2-D scan."

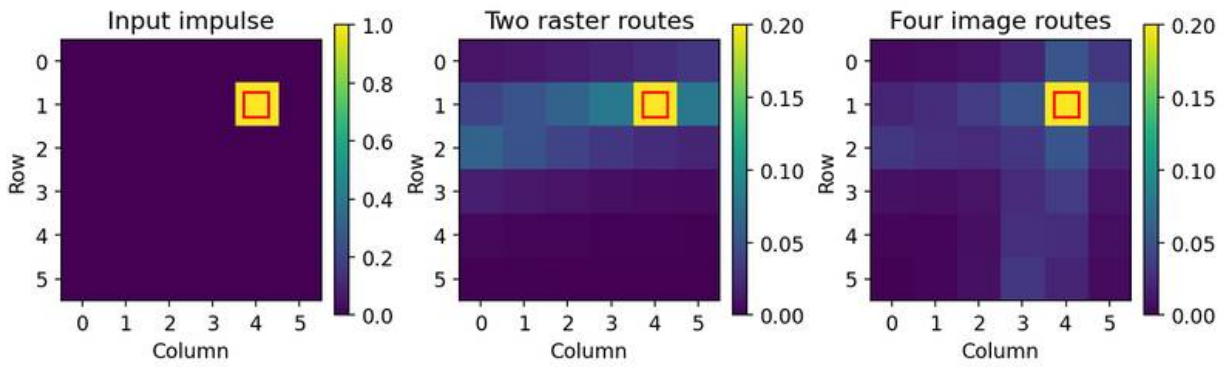



*Figure 7 — The red outline marks the input impulse. Two raster routes spread context unevenly; adding column routes changes the influence map. These synthetic responses use the exact recurrence in the example, not learned attention maps or trained VMamba activations.*

The recurrence can have a global receptive field because information travels through the entire scan. This says a dependence is possible, not that its influence remains strong at every distance. With scalar decay 0.8, the carried contribution decays by another factor of 0.8 per intervening step. Selective transitions can change this retention, as derived in ML 04, but training must still learn which information to preserve. Fine boundaries and small objects can be lost through state compression or downsampling even when distant context is available.

### What Mamba-2 changes, and how to evaluate it

Mamba-2 [34] reorganizes the selective sequence mixer around structured state-space duality and efficient chunk computation. It changes the transition structure and block parameter projections, rather than making an arbitrary image patch sequence equivalent to a Transformer. The causal structured coefficient matrix has an attention-like interpretation, but standard softmax attention retains different expressive and computational properties. An image application must still specify directions and fusion. The duality therefore answers a computation question; it does not remove the geometry question.

A fair vision comparison should separate three experiments: replace only the mixer inside a shared block; compare complete backbones under a matched training budget; and measure the deployed models on the target hardware. Keep data, augmentation, token resolution and downstream heads controlled. Record state size as well as parameter count, and measure latency and peak memory across image resolutions and batch sizes. A linear token-count term can coexist with expensive projections, route permutations and small-kernel overheads. GPU throughput claims do not follow from the Python loop above.

For dense tasks, inspect errors near boundaries, on small structures and across aspect ratios; for video, distinguish an offline bidirectional encoder from causal live processing. Include rotation/crop sensitivity and longer-grid evaluations rather than only in-distribution classification. MambaOut [9] supplies an important mixer-removal ablation. Its results support a task-dependent comparison, not a theorem that scans are unnecessary for every image problem. Xu et al. [8]

organize visual Mamba work by backbones, scan mechanisms and image/video/point-cloud/multimodal applications. Their survey remains a 2024 preprint on the verified record. Its taxonomy is useful for selecting controlled experiments; the number of proposed variants does not establish a universal winner.

## Modern view

**Survey literature.** Khan et al. [3] (*ACM Computing Surveys*, 2022) start from the foundations (self-attention, large-scale pretraining, bidirectional encoding) and then survey Transformers across recognition, generative modeling, multimodal tasks such as visual question answering, video, low-level vision and 3-D analysis, closing with open problems. Han et al. [11] (*IEEE TPAMI*, 2023) organize vision Transformers by task (backbones, high- and mid-level vision such as detection and segmentation, low-level vision, video), and also cover efficient Transformer methods. Liu et al. [12] (*IEEE TNNLS*, 2024) review over a hundred visual Transformers for classification, detection and segmentation on images, point clouds and vision–language data, and suggest three future directions. For efficiency, Papa et al. [13] (*IEEE TPAMI*, 2024) is the most systematic. For state-space vision models, see Xu et al. [8]. Bishop and Bishop [10] provide the textbook derivations.

**State of the art, 2023–2026.** The plain ViT has become the default **foundation** backbone. ViT-22B [24] (ICML 2023) scaled a ViT to 22 billion parameters, with architectural changes (such as normalizing queries and keys) needed for stable training at that size. DINOv2 [23] (TMLR 2024) trained ViTs with self-supervision on a large curated image collection and produced general-purpose features that work frozen across classification, segmentation and depth estimation (ML 06). Darcet et al. [22] (ICLR 2024) found that large ViTs repurpose uninformative background patches as internal scratch space, producing high-norm artifact tokens in attention maps, and that adding a few extra **register tokens** removes the artifacts and improves dense predictions. Vision–language foundation models (ML 10) and diffusion Transformers (ML 08) build on the same block. At the same time, hierarchical and hybrid designs remain common for dense, high-resolution tasks, modernized CNNs remain competitive under matched budgets (ML 03), and state-space and linear-attention models continue to be explored for very long token sequences such as video.

**Open problems.** Quadratic cost at high resolution and long video; the large data and compute appetite of plain ViTs; position encodings that generalize across resolutions and aspect ratios; interpreting attention without over-reading it; and fair, compute-matched comparisons between attention, convolution and state-space mechanisms.

## Key takeaways

- Queries and keys determine weights; values provide the content mixed by attention.
- Score scaling controls magnitude under variance assumptions; it does not guarantee unit variance for arbitrary learned inputs.

- Masks constrain information access, while positional signals distinguish arrangements that content-only attention cannot.
- Transformer blocks alternate token mixing and per-token transformations, with residual paths and normalization.
- ViT, shifted windows and convolutional hybrids make different choices about locality, resolution and training requirements.
- FlashAttention reduces memory traffic for exact attention; windows, token reduction and state-space scans alter computation or context access.

## Exercises

1. **Token budget.** Calculate the number of tokens and attention-score entries per head for a $512 \times 512$ image with patches of size 16 and 8. By what factor does the score count grow when the patch size halves?

   **Hint**

   $N = (512/16)^2 = 1024$ and $(512/8)^2 = 4096$ tokens; $N^2 = 1{,}048{,}576$ and $16{,}777{,}216$ scores. Halving the patch size multiplies $N$ by 4 and the scores by 16.

2. **Scaling check.** Draw random queries and keys with independent standard-normal components for $d_k = 4, 64, 1024$. Measure the variance of $q \cdot k$ with and without division by $\sqrt{d_k}$, and the largest softmax weight over 100 keys in each case. What happens to the softmax without scaling?

   **Hint**

   The variance of the raw dot product is close to $d_k$; after scaling it is close to 1. Without scaling, at $d_k = 1024$ the softmax puts nearly all weight on a single key (it becomes almost one-hot), so gradients through the other keys vanish.

3. **Remove position.** In the TinyViT code, set the position embeddings to zero, drop the class token and use mean pooling, then permute the patch tokens before the blocks and compare the logits in evaluation mode. Explain the result, and what it implies for an image classifier without positional information.

   **Hint**

   The logits agree up to floating-point error, because every block is permutation-equivariant and mean pooling is permutation-invariant. Use dropout off (`eval()`) and permute tokens, not pixels inside a patch. Such a model cannot tell a face from a scrambled face built from the same patches.

4. **Cross-attention shapes.** In a text-to-image model, 256 image tokens of width 320 read 77 text tokens of width 768 with 8 heads of width 40. Give the shapes of $W_Q$, $W_K$, $W_V$, one head's score matrix, and the output.

**Hint**

$W_Q \in \mathbb{R}^{320\times320}$ (8 heads × 40), $W_K, W_V \in \mathbb{R}^{768\times320}$. Each head's scores are $256 \times 77$. The output follows the queries: $256 \times 320$ after the output projection. Key and value lengths must match each other, not the query length.

5. **Swin arithmetic.** Using the cost formulas, compare global and window attention for a $56 \times 56$ token grid with $C = 96$ and $M = 7$. Which term dominates in each case?

**Hint**

$hw = 3136$. Global: $4 \cdot 3136 \cdot 96^2 \approx 1.16 \times 10^8$ plus $2 \cdot 3136^2 \cdot 96 \approx 1.89 \times 10^9$, so the quadratic term dominates. Window: the same $1.16 \times 10^8$ plus $2 \cdot 49 \cdot 3136 \cdot 96 \approx 2.95 \times 10^7$, so the projections dominate. The window version is about 14 times cheaper here.

6. **Fair comparison.** You have 2,000 labeled images and access to pretrained CNN, ViT and visual Mamba backbones. Design an experiment that tells you which backbone to deploy, and list what must be held fixed. Include a mixer-removal ablation and a resolution/rotation test.

**Hint**

Compare linear probes on frozen features first, then fine-tuning with matched budgets and tuned learning rates per model on a validation split. Record pretraining data, input resolution, parameters, FLOPs and measured latency on the target device. A pretrained ViT against a CNN trained from scratch tests pretraining, not architecture. For the scan backbone, record state size and test whether benefits survive removing the mixer while retaining gating and local convolution. Keep the same preprocessing in the resolution and rotation tests.

## References


1. A. Vaswani et al., "Attention Is All You Need," NeurIPS, 2017. arXiv
2. A. Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," ICLR, 2021. arXiv
3. S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan and M. Shah, "Transformers in Vision: A Survey," *ACM Computing Surveys*, 2022. arXiv
4. H. Touvron et al., "Training data-efficient image transformers & distillation through attention," ICML, 2021. arXiv
5. Z. Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows," ICCV, 2021. arXiv
6. T. Dao, D. Y. Fu, S. Ermon, A. Rudra and C. Ré, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," NeurIPS, 2022. arXiv
7. PyTorch, `scaled_dot_product_attention` and `TransformerEncoderLayer` documentation. Attention, Encoder layer MultiheadAttention mask semantics
8. R. Xu et al., "Visual Mamba: A Survey and New Outlooks," arXiv:2404.18861, 2024. arXiv
9. W. Yu and X. Wang, "MambaOut: Do We Really Need Mamba for Vision?," CVPR, 2025. Paper

10. C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts*, Springer, 2024, ch. 12. Book
11. K. Han et al., "A Survey on Vision Transformer," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023. DOI
12. Y. Liu et al., "A Survey of Visual Transformers," *IEEE Transactions on Neural Networks and Learning Systems*, 2024. DOI
13. L. Papa, P. Russo, I. Amerini and L. Zhou, "A Survey on Efficient Vision Transformers: Algorithms, Techniques, and Performance Benchmarking," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024. DOI
14. J. L. Ba, J. R. Kiros and G. E. Hinton, "Layer Normalization," arXiv:1607.06450, 2016. arXiv
15. R. Xiong et al., "On Layer Normalization in the Transformer Architecture," ICML, 2020. arXiv
16. J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo and Y. Liu, "RoFormer: Enhanced Transformer with Rotary Position Embedding," *Neurocomputing*, 2024. DOI
17. P. Shaw, J. Uszkoreit and A. Vaswani, "Self-Attention with Relative Position Representations," NAACL, 2018. arXiv
18. M. Raghu, T. Unterthiner, S. Kornblith, C. Zhang and A. Dosovitskiy, "Do Vision Transformers See Like Convolutional Neural Networks?," NeurIPS, 2021. arXiv
19. H. Wu et al., "CvT: Introducing Convolutions to Vision Transformers," ICCV, 2021. arXiv
20. Z. Dai, H. Liu, Q. V. Le and M. Tan, "CoAtNet: Marrying Convolution and Attention for All Data Sizes," NeurIPS, 2021. arXiv
21. D. Bolya et al., "Token Merging: Your ViT But Faster," ICLR, 2023. arXiv
22. T. Darcet, M. Oquab, J. Mairal and P. Bojanowski, "Vision Transformers Need Registers," ICLR, 2024. arXiv
23. M. Oquab et al., "DINOv2: Learning Robust Visual Features without Supervision," *Transactions on Machine Learning Research*, 2024. arXiv
24. M. Dehghani et al., "Scaling Vision Transformers to 22 Billion Parameters," ICML, 2023. arXiv
25. A. Steiner et al., "How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers," *Transactions on Machine Learning Research*, 2022. arXiv
26. D. Hendrycks and K. Gimpel, "Gaussian Error Linear Units (GELUs)," arXiv:1606.08415, 2016. arXiv
27. H. Touvron, M. Cord and H. Jégou, "DeiT III: Revenge of the ViT," ECCV, 2022. arXiv
28. S. Mehta and M. Rastegari, "MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer," ICLR, 2022. arXiv
29. T. Dao, "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning," ICLR, 2024. arXiv
30. L. Zhu et al., "Vision Mamba: Efficient Visual Representation Learning with Bidirectional State Space Model," ICML, 2024. arXiv
31. Y. Liu et al., "VMamba: Visual State Space Model," NeurIPS, 2024. arXiv
32. Z. Liu et al., "Swin Transformer V2: Scaling Up Capacity and Resolution," CVPR, 2022. arXiv
33. A. Gu and T. Dao, "Mamba: Linear-Time Sequence Modeling with Selective State Spaces," COLM, 2024. Author version
34. T. Dao and A. Gu, "Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality," ICML, PMLR 235, 2024. Paper

# ML 06 • Representation Learning and Self-Supervision

What a representation is, why distributed features transfer, and how SimCLR, MoCo, BYOL, SimSiam, DINO, MAE, JEPA and CLIP learn them without labels, with honest linear-probe and fine-tuning evaluation.

Online edition: https://blog.shyandram.dev/en/learn/ml/representation-learning/

## What you'll learn

- Define useful representations and task-dependent invariances.
- Reuse features through nearest neighbors, linear probes or fine-tuning.
- Compare contrastive, teacher–student, masked and feature-prediction objectives.
- Explain representation collapse and mechanisms that discourage it.
- Evaluate transfer and image–text alignment under explicit protocols.

## The big picture

Suppose you have a million unlabeled images but only 500 labeled examples for the task you care about. Training a large network from scratch on those 500 examples ignores almost all the available data. **Representation learning** splits the job in two. First, learn an encoder $h = f_\theta(x)$ from a broad signal that does not need your labels. Second, train a small task-specific head on top of $h$ using the few labels you have.

> **Explain like I'm five**
>
> Before a child learns the names of every bird, they already notice wings, beaks, colors and how things move. Later, a parent points at three robins and two crows, and that is enough: the child can tell robins from crows from then on. The "noticing" was learned for free, just by looking at the world. A representation is that reusable way of noticing things, and self-supervised learning is the "just looking" part.

Why it matters: almost every modern vision system starts from a pretrained encoder. Image classifiers, detectors, segmentation models, image–text models and the image encoders inside generative models are built on representations learned this way. The quality of that starting point often matters more than the details of the task head.

This chapter starts with what a representation is and why transfer works, then walks through the self-supervised objectives that dominate practice, and ends with how to measure whether a representation is any good. It assumes you know neural networks (ML 02), CNNs (ML 03) and vision transformers (ML 05). Linear factor models and autoencoders, the classical route to representations, have their own chapter: ML 06A.

## What is a representation?

**Plain version.** A representation is the list of numbers a model uses to describe an input. A good one makes the things you care about easy to read off, and hides the things you do not care about.

**Precise version.** An encoder is a function $f_\theta : \mathcal{X} \to \mathbb{R}^D$ from inputs $x$ (here, images) to feature vectors $h = f_\theta(x)$, where $\theta$ are the learnable parameters and $D$ is the feature dimension. A task head $g$ maps $h$ to a prediction, so the full model is $g \circ f_\theta$. We call $h$ a good representation for a task when a *simple* head, such as a linear classifier, can solve the task from $h$ using *few* labeled examples. Bengio, Courville and Vincent's review [1] is the classic statement of this view; it lists the "priors" that make representations useful, such as smoothness, multiple explanatory factors, a hierarchy of increasingly abstract features, and shared factors across tasks.

### Local versus distributed representations

The simplest representation of a category is **one-hot**: one unit per concept, with exactly one unit on. It is easy to read but it cannot generalize. Knowing about "red car" says nothing about "red truck", because they share no active units. A **distributed** representation describes each input with many features that are each shared by many inputs: "red", "blue", "car", "truck". Now "red truck" reuses the "red" unit learned from red cars.

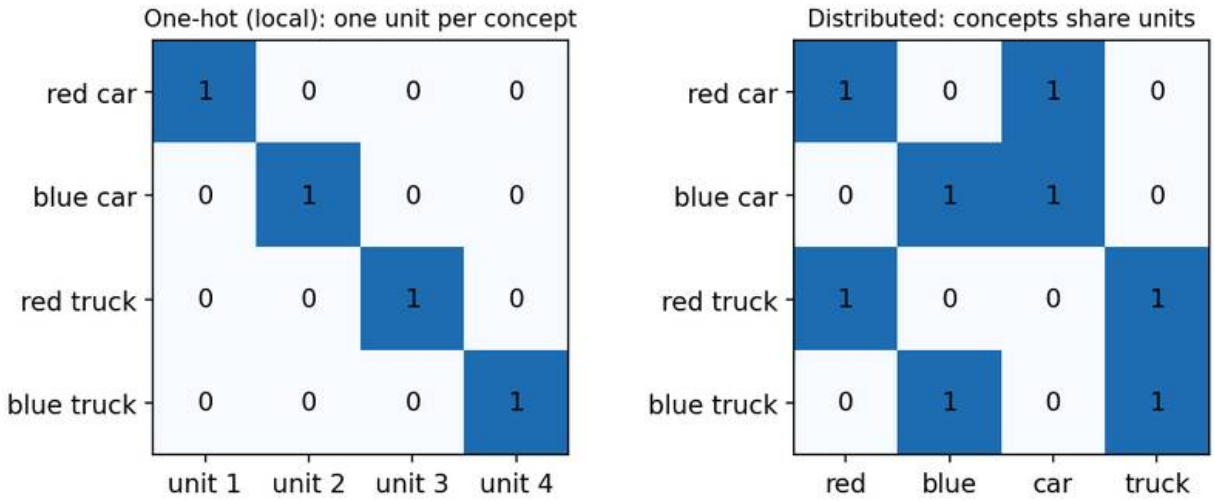



*Figure 1 — Local (one-hot) versus distributed codes for four concepts. With distributed codes, a new combination such as a red truck reuses features learned from other examples.*

The counting argument is striking. With $D$ binary features, a one-hot code distinguishes $D$ concepts, while a distributed code can in principle distinguish $2^D$ configurations [1]. Deep networks learn distributed codes automatically: each hidden unit responds to many inputs, and each input activates many units. Most of the practical value of representation learning comes from this sharing.

### Invariance and equivariance

A useful representation keeps differences relevant to the task and suppresses differences that distract from it.

- A feature is **invariant** to a transformation $T$ if $f(T(x)) = f(x)$: shifting a cat by two pixels should not change "this is a cat".
- A feature is **equivariant** if it changes in a predictable way, $f(T(x)) = T'(f(x))$: shifting the cat should shift its segmentation mask by the same amount.

"Useful" always depends on the downstream task. Color invariance helps object recognition but hurts fruit-ripeness estimation. Rotation invariance helps for textures but confuses 6 and 9. Dense prediction (segmentation, depth) needs spatially equivariant patch features, not a single invariant vector per image. We will see that self-supervised objectives choose their invariances mostly through the **data augmentations** they use, so this choice is a modeling decision, not a detail.

### Learning distances: metric learning

PCA, introduced in ML 01 and developed in ML 06A, keeps the directions of highest variance. But high variance may come from lighting rather than object identity. Supervised learning shapes features around labels; **metric learning** shapes distances around pairs that should match. For an anchor $a$, a positive $p$ (same identity) and a negative $n$ (different identity), the triplet loss is

$$\mathcal{L}_{\text{triplet}} = \max\left(0,\ \|h_a - h_p\|^2 - \|h_a - h_n\|^2 + m\right),$$

where $h_a, h_p, h_n$ are the three feature vectors and $m > 0$ is a margin. The loss is zero once the positive is closer than the negative by at least $m$. An easy negative gives zero loss and teaches nothing, so choosing informative ("hard") negatives matters. Contrastive self-supervised learning, below, is metric learning in which the positives are created by augmentation instead of by labels.

## Transfer learning: pretrain once, reuse many times

**Plain version.** Train a network on a big task, keep its feature extractor, and reuse it for a small task. You can freeze the features and only train a new head, or let the whole network keep learning.

**Precise version.** Pretraining finds encoder parameters $\theta^\star$ on a source dataset or objective. Transfer to a target task with labeled set $\{(x_i, y_i)\}_{i=1}^n$ uses one of three protocols:

1. **$k$-nearest neighbors ($k$-NN):** no training. Classify a test feature $f_{\theta^\star}(x)$ by the labels of its $k$ most similar training features.
2. **Linear probe:** keep $\theta^\star$ frozen and fit a linear classifier $\hat{y} = \text{softmax}(Wh + b)$ on $h = f_{\theta^\star}(x)$, where $W$ and $b$ are the only trained parameters.
3. **Fine-tuning:** start from $\theta^\star$ and update the encoder together with the head, usually with a small learning rate.

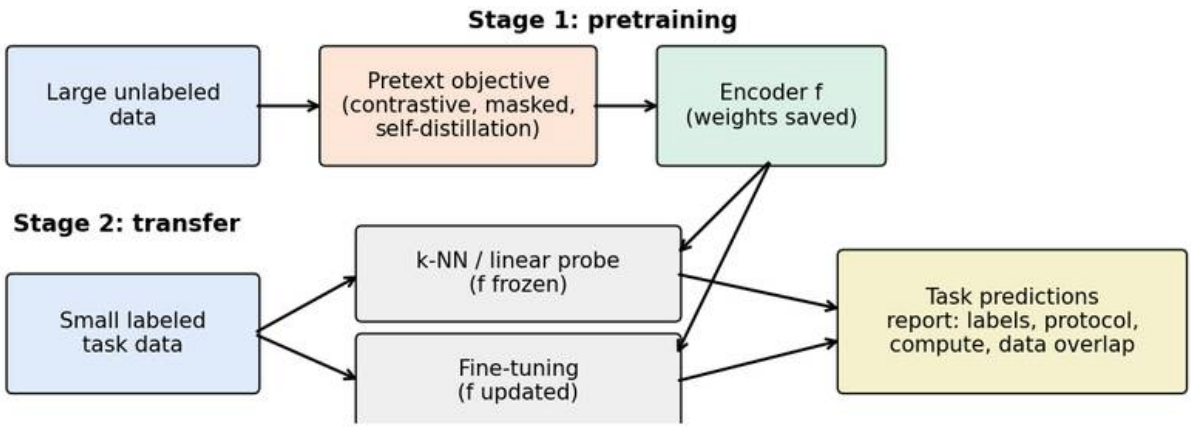



*Figure 2 — The two-stage pattern of representation learning. Pretraining is expensive and done once; transfer is cheap and done per task.*

Why does transfer work at all? Yosinski et al. [2] measured it layer by layer on ImageNet-trained CNNs. First-layer features (edge and color detectors much like Gabor filters) were general: they transferred well between tasks. Features became progressively more specific to the source task in higher layers, and transferability decreased as the source and target tasks became more different. They also found that initializing from transferred features and fine-tuning improved generalization even on the target task, compared with training from random initialization.

For years, the default pretraining source was supervised ImageNet classification. Labels are expensive, though, and a label like "dog" tells the network nothing about the dog's pose, the grass or the lighting. Self-supervised learning removes the label bottleneck by making the data supervise itself.

## Self-supervised learning: let the data make the labels

**Plain version.** Build prediction targets from the data, such as matching two views or recovering masked content. Whether the learned features transfer still needs testing.

**Precise version.** A self-supervised objective is a loss $\mathcal{L}(\theta; x)$ computed from unlabeled inputs alone (or from naturally paired data such as image–caption pairs). The surveys by Liu et al. [3] and Gui et al. [4] and the practical "cookbook" by Balestriero et al. [5] organize the field into the families summarized here:

| Family | Training signal | Representative methods | Main risk |
|---|---|---|---|
| Contrastive | Pull positive pairs together, push other samples apart | SimCLR, MoCo | Needs many negatives; false negatives |
| Self-distillation (no negatives) | Student matches a teacher on another view | BYOL, SimSiam, DINO | Collapse without asymmetry or centering |
| Redundancy reduction | Agreement plus decorrelated, non-constant feature dimensions | Barlow Twins, VICReg | Extra hyperparameters for the regularizers |

| Family | Training signal | Representative methods | Main risk |
|---|---|---|---|
| Masked modeling | Predict hidden pixels or tokens | BEiT, MAE | Features may focus on low-level detail |
| Feature prediction | Predict features of hidden regions | I-JEPA, V-JEPA 2 | Needs anti-collapse design like distillation |
| Language supervision | Match images with their captions | CLIP, SigLIP | Inherits web-data biases and noise |

The families are learning objectives, not architectures: each can train a CNN or a transformer. Modern systems often combine several of them. The sections below take them in turn.

## Contrastive learning and the InfoNCE loss

**Plain version.** Make two augmented versions of the same image agree, and make them disagree with every other image in the batch. It is a multiple-choice question: "out of these $N$ candidates, which one is my partner?"

### InfoNCE

Van den Oord, Li and Vinyals introduced the **InfoNCE** loss in Contrastive Predictive Coding (CPC) [6]. Take a query $q$, one positive key $k^+$ that "belongs" to it, and $N-1$ negative keys $k_1^-$, $\dots, k_{N-1}^-$. With a similarity score $s(\cdot,\cdot)$ and a temperature $\tau > 0$,

$$\mathcal{L}_{\text{InfoNCE}} = -\log \frac{\exp\left(s(q,k^+)/\tau\right)}{\exp\left(s(q,k^+)/\tau\right) + \sum_{j=1}^{N-1} \exp\left(s(q,k_j^-)/\tau\right)}.$$

This is exactly softmax cross-entropy for an $N$-way classification problem whose correct class is the positive. Two consequences follow.

- **A lower bound on mutual information.** CPC shows that $I(q;k^+) \geq \log N - \mathcal{L}_{\text{InfoNCE}}$, where $I$ is mutual information. Minimizing the loss raises this lower bound, and more negatives (larger $N$) allow a higher bound. In practice this bound is loose, and the quality of the features depends heavily on the encoder and augmentations, so read it as motivation rather than as the explanation of why the features are good.
- **The temperature controls hardness.** Dividing by a small $\tau$ sharpens the softmax, so the loss concentrates on the negatives most similar to the query (the hard negatives).

### SimCLR

SimCLR [7] is InfoNCE in its simplest form. Take a batch of $B$ images and produce two augmented views of each (random crop and resize, color distortion, blur), giving $2B$ views. A backbone $f$ maps each view to $h$, a small MLP **projection head** $g$ maps $h$ to $z = g(h)$, and the similarity is cosine similarity $s(z_i, z_j) = z_i^\top z_j / (\|z_i\|\|z_j\|)$. For view $i$ with partner $j$,

$$\ell_{i,j} = -\log \frac{\exp(z_i^\top z_j/\tau)}{\sum_{k\neq i}\exp(z_i^\top z_k/\tau)},$$

where the $z$ are $\ell_2$-normalized and the sum runs over all $2B-1$ other views. Every other image in the batch is a negative, so no memory bank is needed, but the method benefits from large batches. The total loss averages $\ell_{i,j}$ over both orderings of every positive pair; the paper calls this the NT-Xent loss (normalized temperature-scaled cross-entropy).

Three findings from the SimCLR paper are worth remembering [7]. The *composition* of augmentations matters a lot, with random cropping plus color distortion being especially important. The projection head helps: features *before* the head ($h$) transfer better than those after it ($z$), so the head is discarded after pretraining. And contrastive learning benefits from larger batches and longer training than supervised learning.

### Code: check the loss and its collapse value

This standalone PyTorch example uses synthetic paired vectors to verify the loss, the self-mask and the gradients. It does not train an encoder.

```
import torch
from torch.nn import functional as F

torch.manual_seed(0)
B, D = 8, 16
base = torch.randn(B, D)
views = torch.cat([base + .1*torch.randn_like(base),
                   base + .1*torch.randn_like(base)]).requires_grad_()
z = F.normalize(views, dim=-1)
logits = z @ z.T / .2
self_mask = torch.eye(2*B, dtype=torch.bool)
logits = logits.masked_fill(self_mask, float("-inf"))
positive = (torch.arange(2*B) + B) % (2*B)
loss = F.cross_entropy(logits, positive)
loss.backward()
assert torch.isfinite(views.grad).all()
print("paired loss:", round(loss.item(), 4))

collapsed = F.normalize(torch.ones(2*B, D), dim=-1)
scores = (collapsed @ collapsed.T / .2).masked_fill(self_mask, float("-inf"))
collapsed_loss = F.cross_entropy(scores, positive)
print("collapsed loss:", round(collapsed_loss.item(), 4))
print("expected:", round(torch.log(torch.tensor(2*B-1.)).item(), 4))
```

It prints a paired loss of 0.1423 and a collapsed loss of 2.7081, equal to $\log 15$. If every representation is identical, the positive gets probability $1/(2B-1)$ and the loss is $\log(2B-1)$, the worst value a uniform guess can get. Contrastive discrimination therefore makes complete collapse a bad solution, although optimization and negative selection can still be difficult.

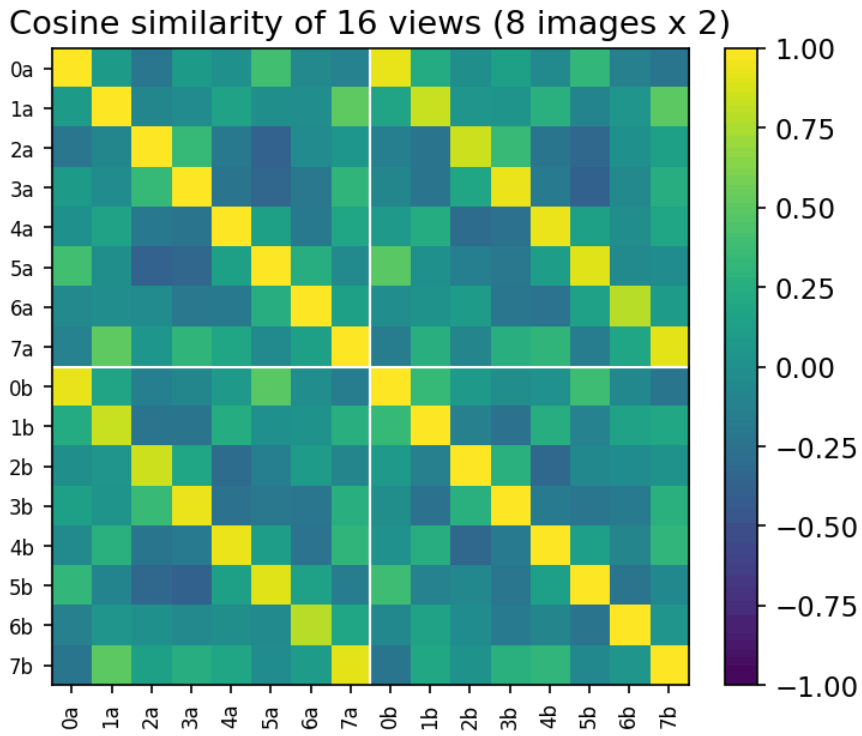



*Figure 3 — Cosine similarities between 16 views (8 digit images, two augmentations each) after training the small SimCLR encoder used later in this chapter. The bright off-diagonal stripes are the positive pairs; InfoNCE pushes them up and everything else down.*

### MoCo: a queue of negatives and a momentum encoder

InfoNCE prefers many negatives, but SimCLR gets them only from the current batch. **Momentum Contrast (MoCo)** [8] decouples the number of negatives from the batch size with two ideas.

1. **A queue.** Keys encoded in recent batches are stored in a first-in, first-out queue of length $K$ and reused as negatives. The query sees $K$ negatives no matter how small the batch is.
2. **A momentum encoder.** Keys in the queue were produced by slightly older networks. If the key encoder changed quickly, old and new keys would be inconsistent. MoCo therefore computes keys with a separate encoder whose parameters $\theta_k$ follow the query encoder's parameters $\theta_q$ slowly:

$$\theta_k \leftarrow m\,\theta_k + (1 - m)\,\theta_q,$$

where $m \in [0, 1)$ is the momentum, close to 1 (the paper's default is 0.999). Only $\theta_q$ receives gradients. This **exponential moving average (EMA)** teacher idea reappears in BYOL, DINO and JEPA.

```
import torch
```

```
from torch.nn import functional as F

torch.manual_seed(0)
B, D, K = 8, 16, 32          # batch, embedding size, queue length

# --- MoCo-style InfoNCE: one positive key per query, negatives from a queue
q = F.normalize(torch.randn(B, D), dim=1)                 # queries (online encoder)
k = F.normalize(q + 0.1 * torch.randn(B, D), dim=1)       # keys (momentum encoder)
queue = F.normalize(torch.randn(K, D), dim=1)             # keys from past batches
l_pos = (q * k).sum(1, keepdim=True)                      # B x 1
l_neg = q @ queue.T                                       # B x K
logits = torch.cat([l_pos, l_neg], dim=1) / 0.07
loss_moco = F.cross_entropy(logits, torch.zeros(B, dtype=torch.long))
queue = torch.cat([k, queue])[:K]                         # enqueue new, drop oldest
print("MoCo logits:", tuple(logits.shape), "loss:", round(loss_moco.item(), 3))

# momentum (EMA) update of the key encoder's parameters
theta_q, theta_k, m = torch.randn(5), torch.zeros(5), 0.999
theta_k = m * theta_k + (1 - m) * theta_q
print("key params moved by:", round((theta_k.norm() / theta_q.norm()).item(), 4))
```

Each query gets a $1 + K$-way classification problem whose correct answer is always column 0, hence the all-zero targets. The output shows logits of shape `(8, 33)`, and the EMA step moves the key encoder only 0.1% of the way toward the query encoder (from zero, after one step). In a real implementation, $q$ and $k$ come from the two encoders applied to two augmentations of the same images.

### Alignment and uniformity

Wang and Isola [9] gave a clean geometric reading of what contrastive losses do on the unit sphere. With normalized features, the loss asymptotically optimizes two properties:

$$\mathcal{L}_{\text{align}} = \mathbb{E}_{(x,x^+)} \|f(x) - f(x^+)\|_2^2, \qquad \mathcal{L}_{\text{unif}} = \log \mathbb{E}_{x,y}\, e^{-t\|f(x)-f(y)\|_2^2},$$

where $(x, x^+)$ is a positive pair, $x, y$ are independent samples, and $t > 0$ (commonly $t = 2$). **Alignment** is low when positive pairs map to nearby points. **Uniformity** is low when features spread out over the sphere, so they keep as much information about the input as possible. A collapsed encoder has perfect alignment (0) and the worst uniformity (0); a useful encoder has both small.

```
import torch
from torch.nn import functional as F

def alignment(z1, z2):            # mean squared distance between positive pairs
    return (z1 - z2).square().sum(1).mean()

def uniformity(z, t=2.0):         # log of mean Gaussian potential over all pairs
    return torch.pdist(z).square().mul(-t).exp().mean().log()

torch.manual_seed(0)
```

```
n, D = 512, 16
collapsed = F.normalize(torch.ones(n, D), dim=1)
spread = F.normalize(torch.randn(n, D), dim=1)
noisy_view = F.normalize(spread + 0.1 * torch.randn(n, D), dim=1)
print("collapsed: align", alignment(collapsed, collapsed).item(),
      "uniform", round(uniformity(collapsed).item(), 3))
print("spread:    align", round(alignment(spread, noisy_view).item(), 3),
      "uniform", round(uniformity(spread).item(), 3))
```

The collapsed set scores alignment 0.0 and uniformity 0.0; the spread set scores about 0.138 and −3.51. Figure 4 tracks the same two quantities while training the digit encoder below. At step 0 the random network maps every view to almost the same point, so alignment is near zero only because everything has collapsed. Training first spreads the features out (uniformity drops sharply), then gradually pulls positives back together.

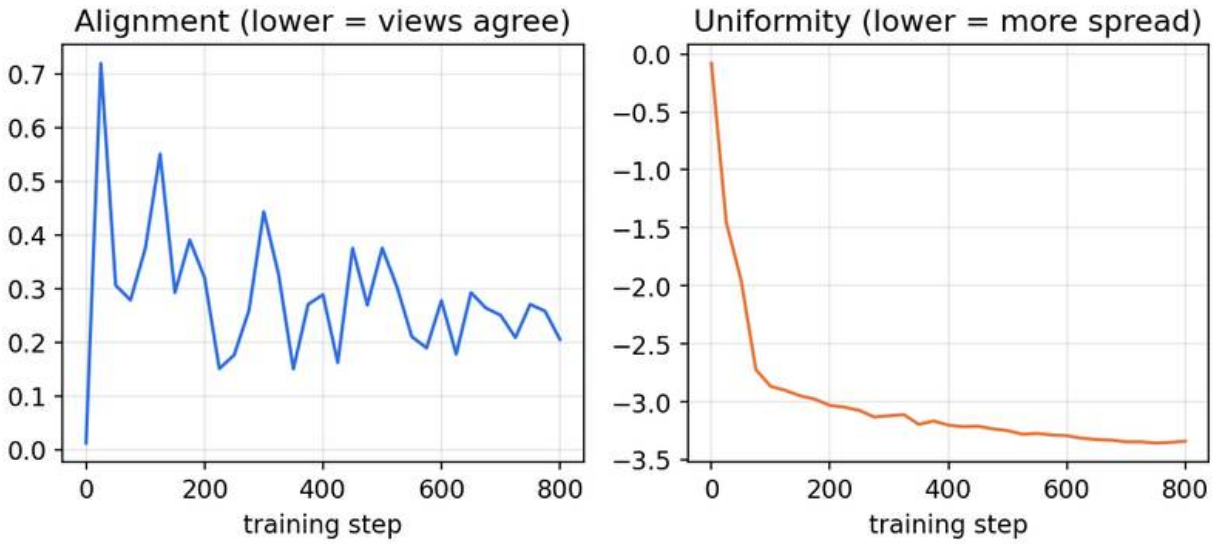



*Figure 4 — Alignment and uniformity of the projection features during SimCLR training on 8×8 digits. Low alignment alone is meaningless (a constant encoder has it); the useful regime is low alignment together with low uniformity.*

## A worked example: SimCLR on 8×8 digits

**Plain version.** Train a tiny SimCLR encoder on scikit-learn's digits without using any labels, then check whether its features make a nearest-neighbor classifier more robust.

The augmentation here is a random shift of up to one pixel, a random brightness change and pixel noise. We therefore expect the learned features to be roughly invariant to small shifts, something raw pixels are not. The evaluation uses the same labeled training set for every feature type and tests on clean digits and on digits shifted by up to one pixel.

```
import copy
import numpy as np
import torch
from torch import nn
from torch.nn import functional as F
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

torch.manual_seed(0); np.random.seed(0); torch.set_num_threads(1)
d = load_digits()
Xtr, Xte, ytr, yte = train_test_split(d.data / 16., d.target, test_size=.3,
                                      random_state=0, stratify=d.target)
Xtr_t = torch.tensor(Xtr, dtype=torch.float32)

def augment(x):                       # the augmentations define the invariances
    img = F.pad(x.view(-1, 1, 8, 8), (1, 1, 1, 1))
    i, j = np.random.randint(0, 3, size=2)               # shift by up to 1 px
    img = img[:, :, i:i + 8, j:j + 8]
    img = img * (0.7 + 0.6 * torch.rand(len(img), 1, 1, 1))  # brightness
    return (img + 0.1 * torch.randn_like(img)).reshape(len(x), 64)  # noise

backbone = nn.Sequential(nn.Unflatten(1, (1, 8, 8)),
                         nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(),
                         nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(),
                         nn.AdaptiveAvgPool2d(2), nn.Flatten())  # h: 256-d
random_backbone = copy.deepcopy(backbone)                    # baseline
head = nn.Sequential(nn.ReLU(), nn.Linear(256, 64))          # projection g
opt = torch.optim.Adam([*backbone.parameters(), *head.parameters()], lr=1e-3)

def nt_xent(z1, z2, tau=0.2):
    z = F.normalize(torch.cat([z1, z2]), dim=-1)
    n = len(z1)
    logits = (z @ z.T / tau).fill_diagonal_(float("-inf"))
    return F.cross_entropy(logits, (torch.arange(2 * n) + n) % (2 * n))

for step in range(800):                                  # about a minute on CPU
    x = Xtr_t[torch.randint(len(Xtr_t), (256,))]
    loss = nt_xent(head(backbone(augment(x))), head(backbone(augment(x))))
    opt.zero_grad(); loss.backward(); opt.step()

def shift(x, dy, dx):                    # move one 8x8 image, fill with zeros
    img = np.zeros((10, 10)); img[1:9, 1:9] = x.reshape(8, 8)
    return img[1 - dy:9 - dy, 1 - dx:9 - dx].ravel()

moves = np.random.default_rng(0).integers(-1, 2, size=(len(Xte), 2))
Xte_shift = np.stack([shift(x, *m) for x, m in zip(Xte, moves)])
enc = lambda net: (lambda A: net(torch.tensor(A, dtype=torch.float32)).detach().numpy())
pca = PCA(32, random_state=0).fit(Xtr)
for name, f in [("raw pixels", lambda A: A), ("PCA-32", pca.transform),
                ("random CNN", enc(random_backbone)), ("SimCLR CNN", enc(backbone))]:
    knn = KNeighborsClassifier(5, metric="cosine").fit(f(Xtr), ytr)
    print(f"{name:11s} clean {knn.score(f(Xte), yte):.3f}  "
```

```
          f"shifted {knn.score(f(Xte_shift), yte):.3f}")
```

On our machine this prints:

```
raw pixels  clean 0.978  shifted 0.524
PCA-32      clean 0.978  shifted 0.507
random CNN  clean 0.974  shifted 0.524
SimCLR CNN  clean 0.987  shifted 0.919
```

Exact numbers vary slightly across PyTorch versions and hardware. The pattern is the lesson. On clean digits, raw pixels are already excellent: this dataset is easy, and a fancy encoder adds little. On shifted digits, raw pixels, PCA and an untrained CNN of the same architecture all drop to about 50%, while the self-supervised features keep most of their accuracy. The network learned the invariance we asked for through the augmentations, and nothing more: had we tested on rotated digits, which the augmentations never showed, we should not expect the same robustness.

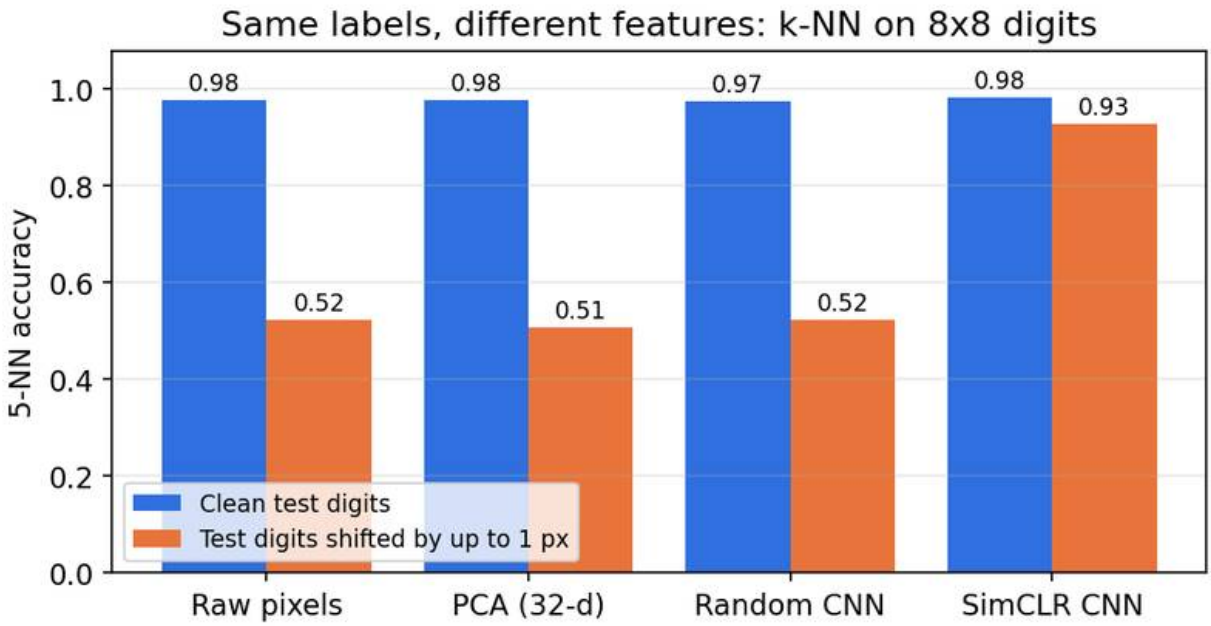



*Figure 5 — Same labels, same classifier, different features. The figure script uses a separate random stream, so its numbers differ from the printed output by up to 0.01. The random CNN rules out the explanation that convolution alone produces the robustness.*

*Online only: SimCLR — what a positive pair teaches a representation — https://blog.shyandram.dev/en/learn/ml/representation-learning/papers/simclr–what-a-positive-pair-teaches-a-representation/*

## Without negatives: why matching alone can collapse

**Plain version.** If the only instruction is "make the two outputs agree", returning the same constant for every image solves it perfectly. Methods without negatives need some other mechanism to stop this.

Minimizing $\|f_\theta(v_1) - f_\theta(v_2)\|^2$ over two views $v_1, v_2$ has the trivial solution $f_\theta(x) = c$ for a constant $c$. Agreement alone does not require the features to carry any information. **Complete collapse** maps everything to one point; **dimensional collapse** is subtler: the features vary, but only along a few directions, wasting most of the $D$ dimensions.

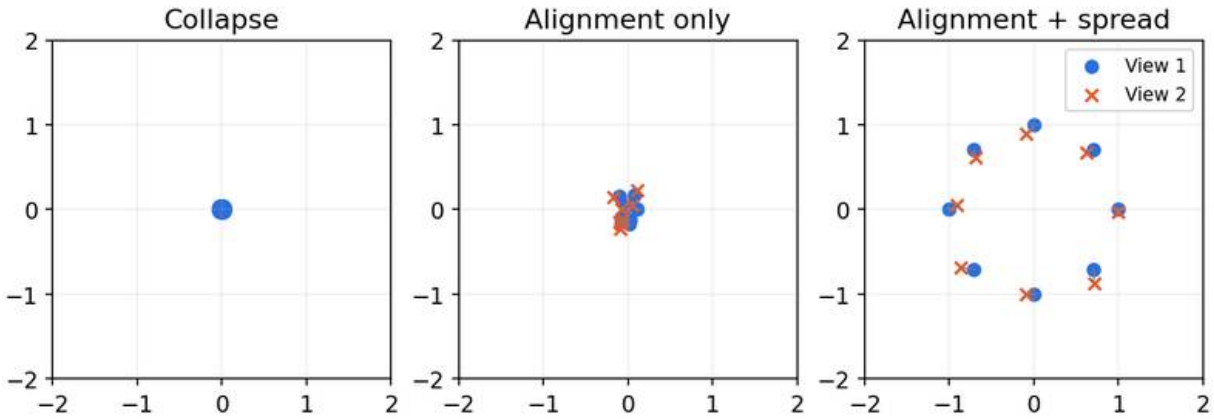



*Figure 6 — Schematic of collapse, alignment without spread, and alignment with spread. The points are constructed illustrations, not embeddings learned by a published model.*

### BYOL: an online network chases a slow target

BYOL [10] uses two networks. The **online** network (encoder, projector and an extra **predictor** $q_\phi$) is trained to predict the **target** network's projection of a different view of the same image. The target is not trained by gradients; it is an EMA of the online weights, as in MoCo:

$$\theta_{\text{target}} \leftarrow m\,\theta_{\text{target}} + (1-m)\,\theta_{\text{online}}.$$

The loss is the squared distance between the normalized prediction and the normalized target projection, symmetrized over the two views. The target output is treated as a constant (**stop-gradient**). There are no negatives, yet BYOL does not collapse in practice. The asymmetry (predictor on one side, stop-gradient and EMA on the other) and the full training recipe matter; EMA alone does not prove collapse impossible.

### SimSiam: stop-gradient is the key ingredient

Chen and He's SimSiam [11] stripped BYOL down further: one shared encoder for both views, a predictor $h$ on one branch, **no** momentum encoder, **no** negatives and no large batches. With

$p_1 = h(f(v_1))$ and $z_2 = f(v_2)$,

$$\mathcal{L} = \tfrac{1}{2}\,\mathcal{D}(p_1, \mathrm{sg}(z_2)) + \tfrac{1}{2}\,\mathcal{D}(p_2, \mathrm{sg}(z_1)), \qquad \mathcal{D}(p, z) = -\frac{p^\top z}{\|p\|\,\|z\|},$$

where sg is stop-gradient and $\mathcal{D}$ is negative cosine similarity. The paper's experiments show that removing the stop-gradient makes the model collapse quickly, while the predictor is also needed. The authors interpret the method as an alternating, expectation-maximization-like optimization. The practical lesson: very small architectural asymmetries decide whether these methods learn or collapse.

### DINO: self-distillation with centering and sharpening

DINO [12] turns the agreement problem into classification with soft labels. Teacher and student both output a probability vector over $K$ "prototype" dimensions via a temperature-scaled softmax. The student, seeing a view $x'$, is trained to match the teacher's distribution on another view $x$:

$$\mathcal{L} = -\sum_{k=1}^{K} P_t(x)_k \,\log P_s(x')_k, \qquad P_t(x) = \mathrm{softmax}\Big(\frac{g_t(x) - c}{\tau_t}\Big),$$

where $g_t$ is the teacher network (an EMA of the student), $c$ is a running mean of teacher outputs (**centering**) and $\tau_t$ is a low teacher temperature (**sharpening**). Centering stops one dimension from dominating; sharpening stops the output from becoming uniform. Each alone collapses in a different way; together they balance. DINO also uses **multi-crop**: the teacher sees only large global crops, while the student also sees small local crops, so the student must infer global content from local evidence. With vision transformers, DINO's attention maps were notable for highlighting object boundaries without any segmentation labels.

### Barlow Twins and VICReg: regularize the statistics directly

A different route is to forbid collapse explicitly in the loss. **Barlow Twins** [13] computes the $D \times D$ cross-correlation matrix $C$ between the (batch-standardized) embeddings of the two views and pushes it toward the identity:

$$\mathcal{L}_{\mathrm{BT}} = \sum_i (1 - C_{ii})^2 + \lambda \sum_i \sum_{j \neq i} C_{ij}^2,$$

where the first term makes each dimension agree across views and the second, weighted by $\lambda > 0$, decorrelates different dimensions so they do not carry redundant information. **VICReg** [14] splits the same idea into three named terms: **invariance** (mean squared distance between the two embeddings), **variance** (a hinge loss keeping the standard deviation of every dimension above a threshold, which directly forbids constant outputs) and **covariance** (penalizing off-diagonal covariance entries). "No negatives" does not mean "no anti-collapse design"; these methods simply put the design in the loss instead of in the architecture.

## Masked modeling: learn by filling gaps

**Plain version.** Hide part of the input and predict what is missing. This is how BERT learned language, and it works for images too, once you choose the right amount of hiding.

**BEiT** [15] brought BERT-style pretraining to vision transformers. An image is split into patches; some are masked, and the network predicts discrete **visual tokens** for the masked positions, where the tokens come from a separately trained image tokenizer.

**MAE** [16] predicts raw pixels instead. It masks a large fraction of the patches, sends only the visible ones through the encoder, and reconstructs the missing pixels with a lightweight decoder. With the set of masked patch indices $\mathcal{M}$,

$$\mathcal{L}_{\text{mask}} = \frac{1}{|\mathcal{M}|} \sum_{i \in \mathcal{M}} \|\hat{x}_i - x_i\|^2,$$

where $x_i$ is the true pixel vector of patch $i$ and $\hat{x}_i$ is the prediction. MAE's design choices are deliberate. A high masking ratio (75% by default) makes the task hard enough that the network cannot simply interpolate from neighbors. The encoder only processes visible patches, which cuts pretraining cost. Normalizing each target patch by its own mean and variance improves the features. After pretraining, the decoder is discarded.

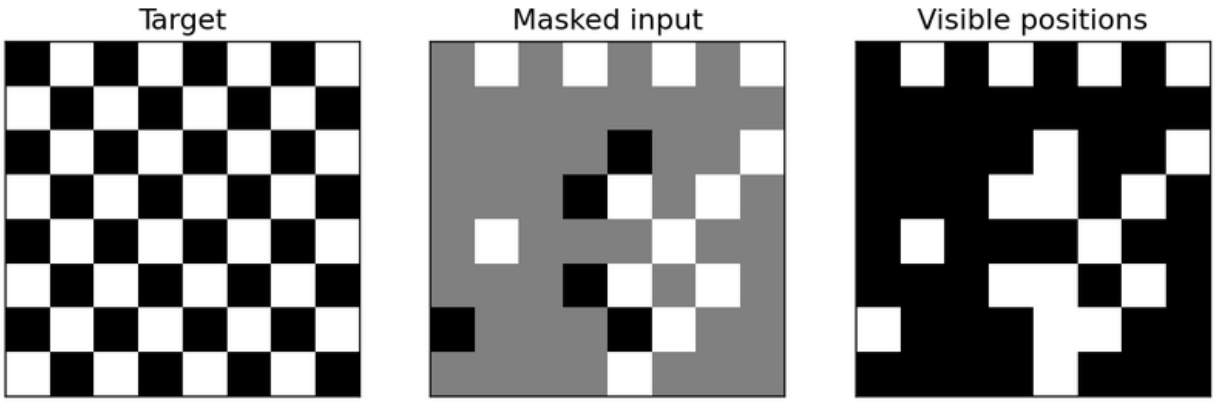



*Figure 7 — Masked prediction creates its own targets from the input. The checkerboard illustrates the masking operation, not semantic understanding.*

Reconstruction is not the same as generation. A model that predicts masked pixels from visible ones is not trained to sample complete images from scratch. Low reconstruction loss does not guarantee the best semantic features either: background texture can dominate pixel error. Masked-modeling features are typically weaker than contrastive or distillation features under a linear probe but strong after fine-tuning, which is one reason the evaluation protocol must be stated.

### Predicting features instead of pixels: JEPA

**Plain version.** Instead of painting in every uncertain pixel, predict a *description* of the hidden region.

I-JEPA [17] (Image Joint-Embedding Predictive Architecture) encodes a visible **context** block and predicts the encoder features of several **target** blocks from it. The target encoder is an EMA of the context encoder, and a predictor receives the context features plus the positions of the targets. A simplified objective is

$$\mathcal{L}_{\text{pred}} = \left\| g_\phi\left(f_\theta(x_{\text{context}}), \text{positions}\right) - \text{sg}\left(f_{\bar{\theta}}(x_{\text{target}})\right)\right\|^2,$$

where $f_\theta$ is the context encoder, $f_{\bar{\theta}}$ the EMA target encoder, $g_\phi$ the predictor and sg stop-gradient. The formula summarizes the idea; the masking strategy (large, semantic-scale target blocks) and the architecture are part of the method, not optional details.

V-JEPA 2 [18] extends feature prediction to video at scale. Its report separates action-free pretraining on video from an action-conditioned model, trained afterward on a small amount of robot data, that is used for planning. Predicting future latent states, recognizing actions and generating photorealistic frames are different capabilities, and a "world model" claim should say which one it makes.

## Language as supervision: CLIP

**Plain version.** Align image and caption embeddings, then compare an image with text descriptions of candidate classes. Prompt wording affects the result.

CLIP [19] trains on image–text pairs collected from the web. In a batch of $N$ pairs, the image encoder gives normalized embeddings $u_1, \dots, u_N$ and the text encoder gives $v_1, \dots, v_N$. The $N \times N$ similarity matrix $S_{ij} = u_i^\top v_j / \tau$, with a learned temperature $\tau$, has the correct pairs on its diagonal. The loss is the average of two InfoNCE losses, one over rows (which caption fits this image?) and one over columns (which image fits this caption?):

$$\mathcal{L}_{\text{CLIP}} = \frac{1}{2}\Big[\frac{1}{N}\sum_i -\log\frac{e^{S_{ii}}}{\sum_j e^{S_{ij}}} + \frac{1}{N}\sum_j -\log\frac{e^{S_{jj}}}{\sum_i e^{S_{ij}}}\Big].$$

**Zero-shot classification** follows directly. Embed one prompt per class, such as "a photo of a dog", and assign an image to the class whose prompt embedding is most similar. No task-specific labels are needed, although prompt wording affects accuracy.

SigLIP [20] replaces the softmax with an independent **sigmoid** loss on every image–text pair: each entry of the matrix becomes a yes/no question, "do these match?". This removes the need to normalize over the whole batch, which simplifies large-scale distributed training.

```
import torch
from torch.nn import functional as F
```

```
torch.manual_seed(0)
D = 16

# --- CLIP-style symmetric loss on N matched image-text pairs
N = 6
img = F.normalize(torch.randn(N, D), dim=1)
txt = F.normalize(img + 0.3 * torch.randn(N, D), dim=1)   # caption i matches image i
logit_scale = torch.tensor(1 / 0.07)
sim = logit_scale * img @ txt.T                           # N x N, diagonal = matches
labels = torch.arange(N)
loss_clip = (F.cross_entropy(sim, labels) + F.cross_entropy(sim.T, labels)) / 2
print("CLIP loss:", round(loss_clip.item(), 3))

# zero-shot classification: embed one prompt per class, pick the most similar
class_prompts = txt[:3]                                   # stand-ins for "a photo of a
 ↪ {class}"
pred = (img @ class_prompts.T).argmax(1)
print("zero-shot predictions for images 0-2:", pred[:3].tolist())

# SigLIP-style alternative: an independent yes/no decision for every pair
t, b = 10.0, -10.0                                        # learnable in practice
signs = 2 * torch.eye(N) - 1                              # +1 on matches, -1 elsewhere
loss_sig = -F.logsigmoid(signs * (t * img @ txt.T + b)).sum() / N
print("sigmoid loss:", round(loss_sig.item(), 3))
```

The synthetic captions are noisy copies of the image embeddings, so the CLIP loss is small (0.054) and the zero-shot step recovers the right "classes" `[0, 1, 2]`. In real CLIP the two encoders are a vision transformer or CNN and a text transformer, and the hard part is the data: what the captions describe, and what they leave out, determines what the representation knows. The sigmoid loss value (3.473) is not comparable to the softmax loss; it uses different units and a bias term $b$.

## DINOv2 and DINOv3: features as a product

**Plain version.** With enough curated data and compute, pretraining can produce an encoder that you freeze and reuse for many different tasks, from classification to depth estimation.

DINOv2 [21] combines image-level self-distillation (as in DINO) with a patch-level masked objective, and pays close attention to **data curation**: building a large, diverse, deduplicated pretraining set from uncurated images by retrieval against curated ones. The result is a family of frozen encoders whose global and patch features work well with simple heads. DINOv3 [22], released in 2025, scales this further and adds **Gram anchoring** to counter the degradation of dense (patch) features during long training. Rather than only matching each feature independently, Gram anchoring constrains the matrix of similarities among patch features toward that of an earlier model. The report also distills a very large teacher into smaller models for deployment.

The practical question is whether frozen global or dense features solve **your** task. A model strong at semantic retrieval may miss a small scratch on a manufactured part. Increasing the input resolution and using patch features can help, but it changes the compute budget and the comparison protocol.

## Evaluate transfer honestly

**Plain version.** Test features on held-out tasks or data with a stated adaptation protocol. Keep evaluation separate from feature and hyperparameter selection.

| Protocol | What is trained | What it measures |
|---|---|---|
| $k$-NN | Nothing; similarity to labeled reference features | Local structure of the representation |
| Linear probe | Linear head; encoder frozen | Information accessible through a simple readout |
| Fine-tuning | Head plus some or all encoder layers | Adaptability under a stated label and compute budget |
| Dense transfer | Spatial head on patch maps | Suitability for pixel or region outputs |

### Linear probing versus fine-tuning

Fine-tuning has more freedom, so it usually wins when the target data is plentiful and similar to the pretraining data. It is not always better. Kumar et al. [23] showed theoretically and empirically that full fine-tuning can **distort** good pretrained features: while a randomly initialized head is still wrong, its large gradients flow into the encoder and change features in directions that suit the training distribution but hurt **out-of-distribution** (OOD) inputs. A linear probe cannot distort the features. Their remedy, **LP-FT**, is two-step: first train the linear head with the encoder frozen, then fine-tune everything starting from that head.

We reproduced the qualitative effect on the digits example. The encoder is the SimCLR CNN from above; the labeled training digits are clean; the test digits are shifted, a mild distribution shift. For each label budget, we average three random draws of the labeled examples.

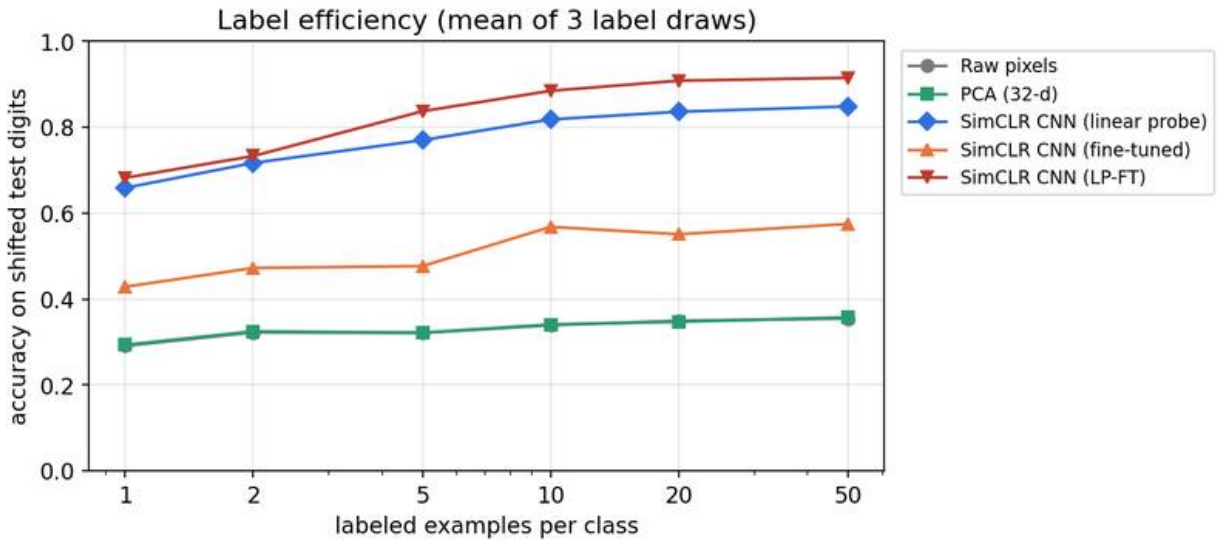



*Figure 8 — Label efficiency on shifted test digits. Self-supervised features are far more label-efficient than*

*pixels or PCA. With one fixed recipe, plain fine-tuning from a random head hurt the shifted-test accuracy, while LP-FT improved on the probe. A toy result: with a tuned learning rate or more labels, plain fine-tuning can do better.*

Read Figure 8 for its shape, not its exact values. The frozen SSL features with a linear head need only a handful of labels per class to beat raw pixels with 50 labels per class. Plain fine-tuning with a randomly initialized head did worse than the frozen probe here, and LP-FT did better than both, the same ordering Kumar et al. report under distribution shift. Fine-tuning needs its own hyperparameter search, and the head initialization matters.

### Rules against leakage

- Normalize features consistently for similarity comparisons (cosine $k$-NN needs normalized vectors).
- Fit scalers, PCA and classifier hyperparameters on training and validation data only.
- A frozen encoder still has a training history: overlap between pretraining data and evaluation data can inflate results, especially for web-scale models.
- Report the number of labels, backbone, checkpoint, input resolution, augmentations, head, tuning budget and protocol. Choosing the best of many encoders on the test set is test leakage, even if none of them is fine-tuned.

## Modern view

**Surveys and review papers.** Four reviews map the field from complementary angles.

- **Bengio, Courville and Vincent (TPAMI 2013) [1]** frame the whole enterprise. They ask what makes one representation better than another and answer with a list of generic priors: distributed codes, depth and abstraction, disentangling of explanatory factors, manifolds and natural clustering. They review probabilistic models, autoencoders and manifold learning. It predates modern SSL but remains the best statement of *why* we learn representations.
- **Liu et al. (TKDE 2023) [3]** classify self-supervised methods into **generative** (autoencoding and autoregressive reconstruction), **contrastive** (context–instance and instance–instance) and **generative-contrastive** (adversarial) groups across vision, language and graphs, and discuss why contrastive objectives were winning on vision benchmarks at the time.
- **Gui et al. (TPAMI 2024) [4]** survey algorithms more recently, covering contrastive, self-distillation, redundancy-reduction and masked image modeling methods, and discuss applications and open problems such as theory, the choice of pretext task and multimodal SSL.
- **Balestriero et al. (2023) [5]** is a practitioner's guide written by many of the field's authors. It groups methods into a deep-metric-learning family (SimCLR and relatives), a self-distillation family (BYOL, SimSiam, DINO), a canonical-correlation-analysis family (VICReg, Barlow Twins) and masked image modeling, and collects the "tricks" (augmentations, projectors, EMA schedules, batch-size effects) that the papers often bury. Prince's textbook [24] gives a gentle, figure-rich introduction to the same ideas with notebooks.

**State of practice (2023–2026).** The trends visible in the papers cited above are:

- **Combined objectives.** Leading image encoders mix image-level distillation with patch-level masked prediction (DINOv2 [21], DINOv3 [22]) rather than relying on one pretext task.
- **Data as a first-class design choice.** DINOv2's curation pipeline and the web-scale pair collection behind CLIP [19] and SigLIP [20] show that what goes into pretraining matters as much as the loss.
- **Dense features.** Segmentation, depth and correspondence need good patch features, which motivated objectives like Gram anchoring [22].
- **Prediction in latent space and video.** I-JEPA [17] and V-JEPA 2 [18] predict features instead of pixels, and extend to video and action-conditioned prediction.
- **Language-aligned encoders as the default backbone** for multimodal models (see ML 10).

**Open problems.** We still lack a predictive theory of which pretext task suits which downstream task; collapse is prevented by recipes more than by guarantees; evaluation depends heavily on protocol and on overlap between pretraining and test data; and web-scale pretraining inherits biases and gaps from its data. Our reading of [3], [4], [5], [21], [22] is that data and target design deserve as much attention as architecture. That is an interpretation of the literature, not a theorem that one objective wins everywhere.

The next chapters change the question: instead of *describing* an image, how can a model describe and **sample** its whole distribution? See ML 06A for autoencoders and ML 07 for generative models.

## Key takeaways

- Useful features make the target task easier; distributed codes alone do not guarantee generalization.
- Augmentations choose which differences to ignore; preserve differences needed by the downstream task.
- Contrastive objectives distinguish positive matches from alternatives; agreement alone admits constant-output collapse.
- Teacher asymmetry, variance constraints and redundancy penalties address collapse in different ways.
- Reconstructing pixels, predicting features and aligning captions train different capabilities.
- Nearest-neighbor tests, linear probes and fine-tuning answer different questions; evaluate adaptation under the expected distribution shift.

## Exercises

1. **Temperature and collapse.** In the first code block, change the temperature from 0.2 to 0.05 and to 1.0. Does the collapsed loss change? Does the paired loss?

**Hint**

The collapsed loss stays $\log 15$: all non-self logits are equal, so dividing them by any $\tau$ leaves the softmax uniform. The paired loss changes, because temperature rescales the *differences* between scores, and with real differences a smaller $\tau$ gives a sharper softmax.

2. **Augmentations are assumptions.** You want features for (a) counting cells in microscopy, (b) recognizing bird species, (c) reading street-sign text. For each, name one standard SimCLR augmentation you would remove and why.

**Hint**

(a) Random cropping can change the number of cells in view, so the "same" label is no longer the same. (b) Strong color jitter can remove plumage color, a key species cue. (c) Horizontal flips turn text into mirror text. Decide what must be preserved before choosing invariances.

3. **MAE compute.** An image has 196 patches and MAE masks 75%. How many patches enter the encoder? By what factor does the number of pairwise attention scores in the encoder shrink?

**Hint**

49 visible patches enter the encoder. Attention scores scale with the square of the token count: $(49/196)^2 = 1/16$. The decoder still processes all 196 positions, but it is small.

4. **Break the digits experiment.** Modify the SimCLR digits code so that `augment` only adds noise (no shift). Retrain and compare the shifted-test $k$-NN accuracy with the original. What does this tell you?

**Hint**

Replace the random `i, j` with `i, j = 1, 1`. On our machine the shifted-test accuracy fell from 0.919 to 0.743, while clean accuracy stayed at 0.981. Without the shift augmentation, nothing in the objective asks for shift invariance; the robustness that remains comes from what a trained convolution-and-pooling encoder learns anyway (the untrained one in the main example had none). The augmentations, not the word "self-supervised", decide most of what is learned.

5. **SimSiam ablation, on paper.** In SimSiam, suppose you remove the stop-gradient so both branches receive gradients. Explain in two sentences why a constant output becomes an easy minimizer, and why the predictor alone is not enough to prevent it.

**Hint**

With gradients on both sides, the loss $-\cos(p_1, z_2)$ is minimized whenever $p_1$ and $z_2$ point the same way, which a constant encoder plus a predictor that outputs that same constant direction achieves for every input. The stop-gradient changes the optimization dynamics so that the target is held fixed during each step; the SimSiam paper reports rapid collapse without it.

6. **Fair probe.** Design an experiment comparing a frozen pretrained encoder with PCA plus logistic regression at 10, 50 and 100 labels per class. List three things you must hold fixed and two things you must report.

**Hint**

Hold fixed: the labeled subsets (same examples for both), the held-out test split, and the procedure for choosing regularization (on validation data only). Report: the pretrained encoder's training data (and any overlap with your test set) and the number of random label draws with their spread.

## References


1. Y. Bengio, A. Courville and P. Vincent, "Representation Learning: A Review and New Perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, 2013. Paper ·arXiv
2. J. Yosinski, J. Clune, Y. Bengio and H. Lipson, "How transferable are features in deep neural networks?" NeurIPS, 2014. Paper
3. X. Liu et al., "Self-supervised Learning: Generative or Contrastive," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, 2023 (online 2021). Paper ·arXiv
4. J. Gui et al., "A Survey on Self-Supervised Learning: Algorithms, Applications, and Future Trends," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024. Paper ·arXiv
5. R. Balestriero et al., "A Cookbook of Self-Supervised Learning," arXiv:2304.12210, 2023. Paper
6. A. van den Oord, Y. Li and O. Vinyals, "Representation Learning with Contrastive Predictive Coding," arXiv:1807.03748, 2018. Paper
7. T. Chen, S. Kornblith, M. Norouzi and G. Hinton, "A Simple Framework for Contrastive Learning of Visual Representations," ICML, 2020. Paper
8. K. He, H. Fan, Y. Wu, S. Xie and R. Girshick, "Momentum Contrast for Unsupervised Visual Representation Learning," CVPR, 2020. Paper
9. T. Wang and P. Isola, "Understanding Contrastive Representation Learning through Alignment and Uniformity on the Hypersphere," ICML, 2020. Paper
10. J.-B. Grill et al., "Bootstrap Your Own Latent: A New Approach to Self-Supervised Learning," NeurIPS, 2020. Paper
11. X. Chen and K. He, "Exploring Simple Siamese Representation Learning," CVPR, 2021. Paper
12. M. Caron et al., "Emerging Properties in Self-Supervised Vision Transformers," ICCV, 2021. Paper
13. J. Zbontar, L. Jing, I. Misra, Y. LeCun and S. Deny, "Barlow Twins: Self-Supervised Learning via Redundancy Reduction," ICML, 2021. Paper
14. A. Bardes, J. Ponce and Y. LeCun, "VICReg: Variance-Invariance-Covariance Regularization for Self-Supervised Learning," ICLR, 2022. Paper
15. H. Bao, L. Dong, S. Piao and F. Wei, "BEiT: BERT Pre-Training of Image Transformers," ICLR, 2022. Paper
16. K. He et al., "Masked Autoencoders Are Scalable Vision Learners," CVPR, 2022. Paper
17. M. Assran et al., "Self-Supervised Learning from Images with a Joint-Embedding Predictive Architecture," CVPR, 2023. Paper

18. M. Assran et al., “V-JEPA 2: Self-Supervised Video Models Enable Understanding, Prediction and Planning,” technical report, 2025. Report
19. A. Radford et al., “Learning Transferable Visual Models From Natural Language Supervision,” ICML, 2021. Paper
20. X. Zhai, B. Mustafa, A. Kolesnikov and L. Beyer, “Sigmoid Loss for Language Image Pre-Training,” ICCV, 2023. Paper
21. M. Oquab et al., “DINOv2: Learning Robust Visual Features without Supervision,” *Transactions on Machine Learning Research*, 2024. Paper
22. O. Siméoni et al., “DINOv3,” technical report, 2025. Report
23. A. Kumar, A. Raghunathan, R. Jones, T. Ma and P. Liang, “Fine-Tuning can Distort Pretrained Features and Underperform Out-of-Distribution,” ICLR, 2022. Paper
24. S. J. D. Prince, *Understanding Deep Learning*, MIT Press, 2023. Book and notebooks

# ML 06A • Linear Factor Models and Autoencoders

PCA, probabilistic PCA, factor analysis, ICA, sparse coding, NMF and autoencoders (undercomplete, sparse, denoising, contractive): how compression, noise assumptions and the manifold view lead to VAEs.

Online edition: https://blog.shyandram.dev/en/learn/ml/factor-models-autoencoders/

## What you'll learn

- Explain how latent factors describe correlated measurements.
- Compare PCA, probabilistic PCA and factor-analysis noise assumptions.
- Distinguish independence, sparsity and nonnegativity constraints.
- Relate linear and nonlinear autoencoders to reconstruction objectives.
- Understand manifold assumptions and why reconstruction alone does not define sampling.

## The big picture

An $8 \times 8$ digit image has 64 pixel values, and a phone photo has millions. Yet the meaningful variation in a collection of such images depends on far fewer underlying quantities: which digit it is, the stroke thickness, the slant, the position, the lighting. A **factor model** tries to explain each observation through a handful of hidden (latent) coordinates. An **autoencoder** is a neural network that learns to squeeze an input into such coordinates and expand it back again.

> **Explain like I'm five**
>
> Imagine describing a friend's face over the phone. You don't read out every freckle; you say "round face, short hair, big smile, light from the left." Those few words are factors: they let the listener rebuild a good picture. A factor model looks for the words that explain the most about many faces at once. An autoencoder is a pair of helpers: one who turns a picture into a few words, and one who turns the words back into a picture. They practise together until the rebuilt pictures look right.

Why spend a whole chapter on this? Because almost every idea in modern representation learning and generative modeling has an ancestor here. Dimensionality reduction for visualization, denoising, compression, anomaly detection, the "tokenizers" that turn images into compact codes for diffusion models, and the probabilistic machinery of VAEs all start from the same question: *what small set of causes could have produced these observations?* The progression from linear factor models to autoencoders follows Goodfellow, Bengio and Courville [1, ch. 13–14] and Murphy [2]; Bengio, Courville and Vincent's review [3] gives the broader motivation for why learned representations matter.

## One template, many models

Every method in this chapter fits the same template. There is an **encoder** that maps an observation $x \in \mathbb{R}^d$ to a code $z \in \mathbb{R}^k$, and a **decoder** that maps a code back to a reconstruction $\hat{x}$. The models

differ in three choices:

| Choice | Options in this chapter |
|---|---|
| Shape of encoder/decoder | linear (PCA, factor analysis, ICA, NMF, sparse coding) or nonlinear (deep autoencoders) |
| Size of the code | **undercomplete** ($k < d$: the bottleneck forces compression) or **overcomplete** ($k \geq d$: something else must stop the model from copying) |
| Extra assumption or penalty | Gaussian noise (PPCA, FA), independence and non-Gaussianity (ICA), sparsity (sparse coding, sparse autoencoders), nonnegativity (NMF), robustness to corruption (denoising), insensitivity to small input changes (contractive) |

Keep this table in mind. When two methods disagree, it is almost always because they made a different choice in one of these rows, not because one of them is "wrong."

### PCA: the best linear reconstruction subspace

**Plain version.** Find a few orthogonal directions such that, if you keep only the coordinates of each example along those directions, you lose as little as possible.

Let $X \in \mathbb{R}^{n \times d}$ hold $n$ centered examples as rows (the column means have been subtracted). PCA chooses a matrix $W \in \mathbb{R}^{d \times k}$ with orthonormal columns that minimizes the squared reconstruction error

$$\min_{W} \ \|X - XWW^{\top}\|_F^2 \qquad \text{subject to} \qquad W^{\top}W = I_k.$$

Here $\|\cdot\|_F$ is the Frobenius norm (the square root of the sum of squared entries), $Z = XW \in \mathbb{R}^{n \times k}$ are the **codes** (also called scores), and $\hat{X} = ZW^{\top}$ is the reconstruction. The encoder is $x \mapsto W^{\top}x$ and the decoder is $z \mapsto Wz$: PCA is already an autoencoder, just a linear one with tied weights.

### Where the solution comes from

Two equivalent views lead to the same answer.

**Maximum variance.** Take a single unit vector $w$. The variance of the projected data $Xw$ is $w^{\top}Sw$, where $S = \frac{1}{n-1}X^{\top}X$ is the sample covariance. Maximize $w^{\top}Sw$ subject to $w^{\top}w = 1$ with a Lagrange multiplier $\lambda$:

$$\nabla_w(w^{\top}Sw - \lambda(w^{\top}w - 1)) = 0 \quad \Longrightarrow \quad Sw = \lambda w.$$

So $w$ must be an eigenvector of $S$, and the variance it captures is its eigenvalue $\lambda$. The best direction is the eigenvector with the largest eigenvalue; the next best direction, constrained to be orthogonal to the first, is the second eigenvector, and so on.

**Minimum reconstruction error.** Because $W$ has orthonormal columns, the error splits into "total variance minus kept variance." Minimizing the error is therefore the same as maximizing the

kept variance, and the residual equals the sum of the discarded eigenvalues (times $n-1$). The Eckart–Young theorem states the same result in matrix language: truncating the singular value decomposition $X = U\Sigma V^\top$ to its $k$ largest singular values gives the best rank-$k$ approximation of $X$ in Frobenius norm. The PCA directions are the first $k$ right singular vectors (columns of $V$), and the covariance eigenvalues are $\sigma_j^2/(n-1)$.

### Code: PCA from the SVD

The snippet fits PCA on the training split of scikit-learn's bundled digits, evaluates reconstruction on held-out images, and checks the result against scikit-learn's own implementation.

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split

X = load_digits().data / 16.
train, test = train_test_split(X, test_size=.2, random_state=0)

mean = train.mean(axis=0)
U, s, Vt = np.linalg.svd(train - mean, full_matrices=False)
var = s**2 / (len(train) - 1)              # eigenvalues of the covariance
ratio = np.cumsum(var) / var.sum()
for k in (2, 10, 20):
    W = Vt[:k].T                            # d x k, orthonormal columns
    rec = (test - mean) @ W @ W.T + mean
    print(f"k={k:2d}  explained={ratio[k-1]:.3f}  "
          f"test MSE={np.mean((test - rec)**2):.4f}")

sk = PCA(n_components=10).fit(train)
print("matches sklearn:", np.allclose(np.abs(sk.components_), np.abs(Vt[:10])))
```

Output:

```
k= 2  explained=0.291  test MSE=0.0546
k=10  explained=0.740  test MSE=0.0202
k=20  explained=0.896  test MSE=0.0084
matches sklearn: True
```

The `np.abs` in the last line is not a hack: each principal direction is only defined up to sign, because $w$ and $-w$ span the same line. Note that the mean is estimated on the training split only and reused for the test split; estimating it on test data would leak information.

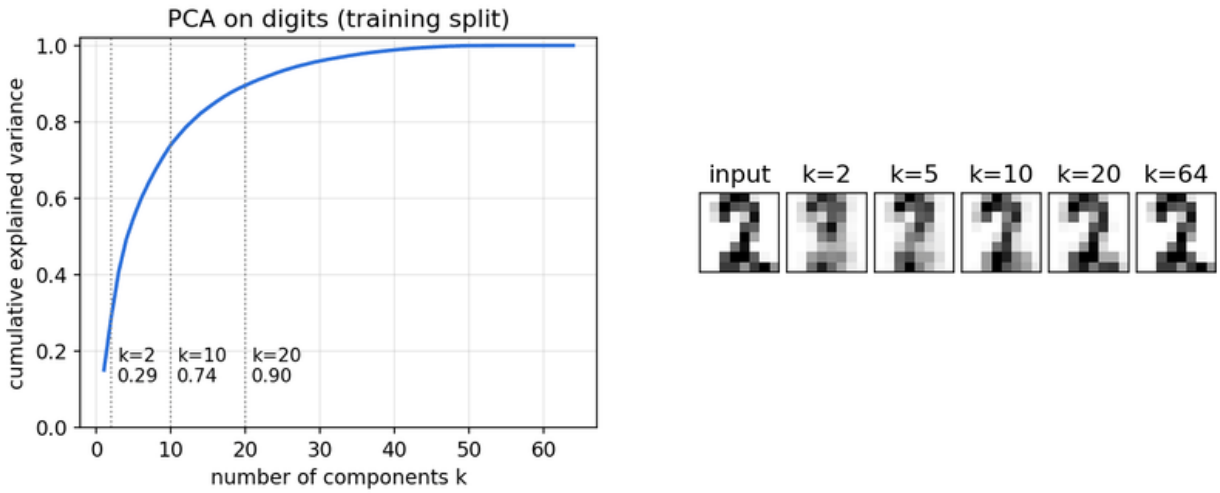



*Figure 1 — PCA on the bundled digits. Left: cumulative explained variance (fit on the training split). Right: one held-out digit reconstructed from k components; k = 64 reproduces the input exactly.*

**Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

### What PCA does not promise

Three cautions follow directly from the objective.

- **Centering and scaling change the answer.** Without centering, the first "component" mostly points at the mean. Standardizing every feature to unit variance gives each pixel equal weight, so raw-pixel PCA and standardized PCA solve different problems.
- **Variance is not relevance.** The direction of largest variance can be lighting or background rather than the class you care about. PCA is unsupervised; it maximizes retained variance, not class separation (that is the job of methods such as linear discriminant analysis).
- **PCA is linear.** If the data lie on a curved surface, a flat subspace may need many dimensions to describe what is really a low-dimensional shape. We return to this in the manifold section.

### Probabilistic PCA and factor analysis

**Plain version.** Instead of just finding a subspace, write down a story for how the data were generated: draw a few hidden factors, mix them linearly, then add noise. Then fit that story by maximum likelihood.

Both models share the generative equation

$$x = \mu + Wz + \epsilon, \qquad z \sim \mathcal{N}(0, I_k),$$

where $\mu \in \mathbb{R}^d$ is the mean, $W \in \mathbb{R}^{d \times k}$ is the **loading matrix** (how each factor shows up in each measured coordinate), $z$ is the vector of latent factors, and $\epsilon$ is noise independent of $z$. They differ only in the noise.

### Probabilistic PCA (PPCA)

In PPCA [4], the noise is isotropic, $\epsilon \sim \mathcal{N}(0, \sigma^2 I_d)$: every coordinate gets the same noise variance. Integrating out $z$ gives a Gaussian marginal

$$x \sim \mathcal{N}(\mu,\, C), \qquad C = WW^\top + \sigma^2 I_d.$$

The covariance has a clear interpretation: a low-rank part $WW^\top$ that the factors explain, plus a constant "floor" $\sigma^2$ that they do not. Tipping and Bishop showed that the maximum-likelihood solution is available in closed form [4]:

$$W_{\mathrm{ML}} = U_k(\Lambda_k - \sigma^2 I)^{1/2} R, \qquad \sigma^2_{\mathrm{ML}} = \frac{1}{d-k} \sum_{j=k+1}^{d} \lambda_j,$$

where $U_k$ holds the top $k$ eigenvectors of the sample covariance, $\Lambda_k$ their eigenvalues $\lambda_1, \dots, \lambda_k$, and $R$ is any $k \times k$ orthogonal matrix. In words: the loadings span exactly the PCA subspace, and the noise variance is the average of the variance PCA throws away. PPCA therefore *is* PCA, plus a likelihood. That likelihood is useful: you can compare models with different $k$ on held-out data, handle missing values with expectation–maximization (EM), and build mixtures of PPCA models.

Given an observation, the posterior over its factors is Gaussian with mean

$$\mathbb{E}[z \mid x] = M^{-1} W^\top (x - \mu), \qquad M = W^\top W + \sigma^2 I_k.$$

This is the probabilistic encoder. As $\sigma^2 \to 0$ it approaches the ordinary PCA projection; for $\sigma^2 > 0$ it shrinks codes toward zero, because part of what we see is attributed to noise.

### Factor analysis (FA)

**Factor analysis** keeps the same structure but allows a diagonal noise covariance, $\epsilon \sim \mathcal{N}(0, \Psi)$ with $\Psi = \mathrm{diag}(\psi_1, \dots, \psi_d)$. Each observed coordinate has its own residual noise level $\psi_i$, while the latent factors explain the **shared** covariance between coordinates. The marginal is $x \sim \mathcal{N}(\mu, WW^\top + \Psi)$, and the posterior mean of the factors is $\mathbb{E}[z \mid x] = W^\top (WW^\top + \Psi)^{-1}(x - \mu)$.

There is no closed-form solution, so FA is fitted with EM or direct likelihood optimization. The extra flexibility matters when measurements come from sensors with unequal noise, or when some pixels are almost constant (such as the always-blank border pixels of the digits). PCA happily spends components on any high-variance coordinate; FA can instead explain an individually noisy coordinate as "noise," and reserve its factors for correlations.

### The rotation problem

Both models share an ambiguity. For any orthogonal $R$, replacing $W$ by $WR$ and $z$ by $R^\top z$ leaves $WW^\top$, and hence the distribution of $x$, unchanged, because $R^\top z$ is still standard normal. The *subspace* is identified; the individual axes are not. That is why the factor-analysis literature uses rotations such as varimax to pick more interpretable axes, and why a factor is not automatically a physical cause. To pin down individual factors we must assume something beyond Gaussianity, which is precisely what ICA does.

*Online only: Probabilistic PCA — a density model behind a subspace — https://blog.shyandram.dev/en/learn/ml/factor-models-autoencoders/papers/probabilistic-pca–a-density-model-behind-a-subspace/*

## Independent component analysis (ICA)

**Plain version.** Several people talk at once in a room, and several microphones record different mixtures of their voices. ICA tries to recover each voice by finding mixtures that are as statistically independent, and as non-Gaussian, as possible.

The noise-free ICA model is

$$x = As, \qquad s = (s_1, \dots, s_k) \text{ with independent components,}$$

where $x \in \mathbb{R}^d$ is the observed mixture, $A \in \mathbb{R}^{d\times k}$ is an unknown **mixing matrix**, and the sources $s_j$ are independent and **non-Gaussian**. The goal is an unmixing matrix $B \approx A^{-1}$ (or a pseudo-inverse) so that $\hat{s} = Bx$ recovers the sources [5].

### Why non-Gaussianity is the key

The rotation problem from the previous section explains why ICA needs non-Gaussian sources. First **whiten** the data: transform $x$ so that its covariance is the identity (PCA followed by rescaling each component to unit variance does this). After whitening, the remaining unknown is just a rotation. If the sources were Gaussian, every rotation of independent unit Gaussians would again be independent unit Gaussians, so no rotation could be preferred. ICA's identifiability theorem states the converse: if **at most one** source is Gaussian, the sources can be recovered up to three harmless ambiguities [5]:

- **order** (we cannot know which source is "first"),
- **sign**, and
- **scale** (a louder source with a quieter mixing column looks the same).

The central limit theorem gives the intuition for how to find the right rotation: a sum of independent variables tends to be *more* Gaussian than its parts. So the projections of $x$ that are *least* Gaussian are the ones most likely to be single sources rather than mixtures. ICA algorithms therefore maximize a measure of non-Gaussianity. A principled one is **negentropy**,

$$J(y) = H(y_{\text{gauss}}) - H(y) \;\geq 0,$$

where $H$ is differential entropy and $y_{\text{gauss}}$ is a Gaussian with the same variance as $y$. Among all distributions with a given variance, the Gaussian has the largest entropy, so $J(y) = 0$ exactly for Gaussian $y$. Because entropy is hard to estimate, the **FastICA** algorithm approximates negentropy with contrast functions such as $G(u) = \log\cosh u$, and finds each unmixing direction with a fast fixed-point iteration [5]. Maximizing non-Gaussianity is closely related to minimizing the mutual information between the estimated components and to maximum-likelihood estimation of the ICA model.

### Code: unmix three signals

The toy problem uses a sine wave, a square wave and a sawtooth, all non-Gaussian, mixed by a fixed matrix.

```
import numpy as np
from sklearn.decomposition import FastICA, PCA

rng = np.random.default_rng(0)
t = np.linspace(0, 8, 2000)
S = np.c_[np.sin(2*t),                  # smooth sine
          np.sign(np.sin(3*t)),         # square wave
          2*(t % 1) - 1]                # sawtooth
S += 0.05 * rng.normal(size=S.shape)
A = np.array([[1.0, 1.0, 1.0],
              [0.5, 2.0, 1.0],
              [1.5, 1.0, 2.0]])         # unknown mixing matrix
X = S @ A.T                             # three "microphones"

def best_match(est):
    C = np.abs(np.corrcoef(S.T, est.T)[:3, 3:])
    return np.round(C.max(axis=1), 3)

ica = FastICA(n_components=3, whiten="unit-variance", random_state=0)
print("ICA |corr| with each source:", best_match(ica.fit_transform(X)))
print("PCA |corr| with each source:", best_match(PCA(3).fit_transform(X)))
```

Output:

```
ICA |corr| with each source: [0.997 0.999 0.997]
PCA |corr| with each source: [0.689 0.811 0.735]
```

We compare with absolute correlation and take the best match because ICA returns the sources in arbitrary order, sign and scale. PCA only decorrelates; it finds orthogonal directions of high variance, which here are still mixtures.

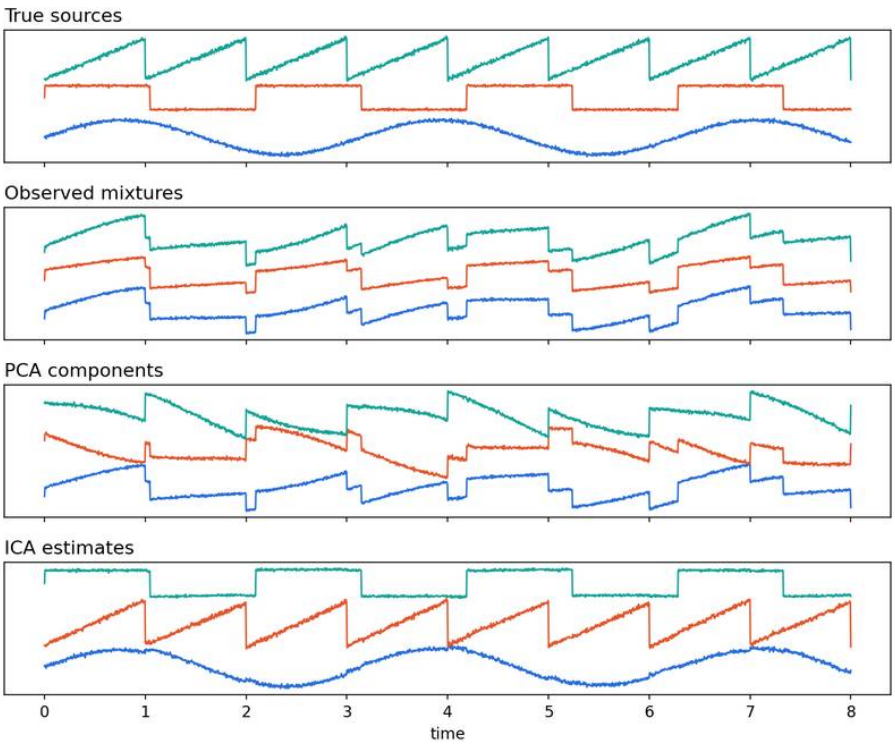



*Figure 2 — Blind source separation. PCA components remain mixtures; FastICA recovers the sine, square and sawtooth waves up to order, sign and scale.*

ICA is widely used for separating artifacts in EEG and other sensor recordings. Its assumptions also mark its limits: it needs genuinely independent, non-Gaussian sources and a mixing process that is close to linear and instantaneous. On image patches, ICA tends to learn localized, oriented edge-like filters, which brings us to a closely related idea.

### Sparse coding

**Plain version.** Keep a large dictionary of small patterns ("atoms") and describe each input using only a few of them at a time.

Sparse coding learns a dictionary $D = [d_1, \dots, d_m] \in \mathbb{R}^{d \times m}$ and, for each example $x_i$, a sparse code $\alpha_i \in \mathbb{R}^m$ by solving

$$\min_{D,\,\{\alpha_i\}} \sum_{i=1}^{n} \left( \tfrac{1}{2} \|x_i - D\alpha_i\|_2^2 + \lambda \|\alpha_i\|_1 \right) \qquad \text{subject to} \qquad \|d_j\|_2 \le 1.$$

The first term asks for accurate reconstruction; the $\ell_1$ norm $\|\alpha\|_1 = \sum_j |\alpha_j|$ pushes most code entries to exactly zero; $\lambda > 0$ sets the trade-off. The norm constraint on atoms stops the model from cheating by making atoms huge and codes tiny. The dictionary is often **overcomplete** ($m > d$): there are more atoms than pixel dimensions, but only a few are active for any one input.

Olshausen and Field [6] trained this kind of model on patches of natural images and found that the learned atoms were localized, oriented and band-pass, much like the receptive fields of simple cells in the primary visual cortex. The result was influential because it suggested that sparsity alone, without any labels, can explain a known property of biological vision.

Two practical points distinguish sparse coding from the autoencoders later in this chapter:

- **Inference is an optimization.** Given $D$, finding $\alpha$ for a new input means solving a lasso problem. There is no fixed encoder network; the "encoder" is an algorithm. Training alternates between updating the codes (with $D$ fixed) and updating the dictionary (with codes fixed).
- **Sparsity is a prior.** The $\ell_1$ penalty corresponds to a Laplace prior on the codes, so sparse coding can be read as maximum a posteriori inference in a linear factor model with a sparse, non-Gaussian prior, which is the same family that ICA belongs to.

### Code: sparsity versus fidelity

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.decomposition import DictionaryLearning
from sklearn.model_selection import train_test_split

X = load_digits().data / 16.
train, test = train_test_split(X, test_size=.2, random_state=0)
for alpha in (0.1, 1.0):
    dl = DictionaryLearning(n_components=32, alpha=alpha, max_iter=30,
                            transform_algorithm="lasso_lars",
                            transform_alpha=alpha, random_state=0)
    dl.fit(train[:600])
    codes = dl.transform(test)
    rec = codes @ dl.components_
    active = np.mean(np.abs(codes) > 1e-8, axis=1).mean()
    print(f"alpha={alpha}: active atoms per image={active*32:.1f}/32, "
          f"test MSE={np.mean((test - rec)**2):.4f}")
```

Output (about half a minute on a laptop CPU):

```
alpha=0.1: active atoms per image=17.9/32, test MSE=0.0075
alpha=1.0: active atoms per image=3.0/32, test MSE=0.0401
```

Raising $\lambda$ (here `alpha`) buys a much sparser description, about three active atoms per image instead of eighteen, at the price of higher reconstruction error. Neither setting is "correct": a compression codec, a denoiser and an interpretability tool would each choose differently. Note that this dictionary does not subtract a mean, so its error is not directly comparable to the PCA numbers above.

### Nonnegative matrix factorization (NMF)

**Plain version.** Describe each example as an *additive* mixture of nonnegative parts: you may add strokes, but you may never subtract one.

For data $X \geq 0$ (pixel intensities, word counts, spectra), NMF finds $Z \geq 0$ and $H \geq 0$ that minimize a reconstruction criterion such as

$$\min_{Z \geq 0,\ H \geq 0} \|X - ZH\|_F^2,$$

where the rows of $H \in \mathbb{R}^{k \times d}$ are nonnegative parts and the rows of $Z \in \mathbb{R}^{n \times k}$ are nonnegative mixing weights. Because nothing can cancel, the parts tend to look like pieces of the object rather than the positive-and-negative "ghost" patterns of PCA. Lee and Seung [7] demonstrated this parts-based behavior on faces and text and popularized simple multiplicative update rules for fitting it.

Two cautions. First, parts-based results are a tendency, not a guarantee: on some datasets the factors are not clean parts. Second, the objective is nonconvex in $Z$ and $H$ jointly, so initialization matters and different runs can give different factors. Scikit-learn's `init="nndsvda"` uses an SVD-based start to make results more stable.

### Code: compare factor reconstructions

This CPU example fits each model on training digits only and evaluates reconstruction on held-out images. It compares different model assumptions; it is not a claim that one method wins every task.

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA, FactorAnalysis, NMF

X = load_digits().data / 16.
train, test = train_test_split(X, test_size=.2, random_state=0)
models = {
    "PCA": PCA(n_components=12),
    "factor analysis": FactorAnalysis(n_components=12, random_state=0),
    "NMF": NMF(n_components=12, init="nndsvda", max_iter=1000,
               random_state=0),
}
for name, model in models.items():
    model.fit(train)
    z = model.transform(test)
    if name == "factor analysis":
        reconstructed = z @ model.components_ + model.mean_
    else:
        reconstructed = model.inverse_transform(z)
    mse = np.mean((test-reconstructed)**2)
    print(name, "held-out reconstruction MSE:", round(mse, 5))
```

Output:

```
PCA held-out reconstruction MSE: 0.01652
factor analysis held-out reconstruction MSE: 0.02501
NMF held-out reconstruction MSE: 0.02207
```

PCA has the lowest held-out reconstruction MSE in this run. Its optimality theorem concerns the centered **training matrix** used to fit the subspace; it does not guarantee the lowest error on every new test set. Another fitted model can generalize better when its assumptions suit that population. The factor-analysis reconstruction here is a posterior mean that attributes part of each pixel to noise, and FA's probability model is not summarized by MSE alone. NMF instead imposes nonnegative, additive factors. Compare their held-out errors empirically rather than treating the observed ranking as a theorem.

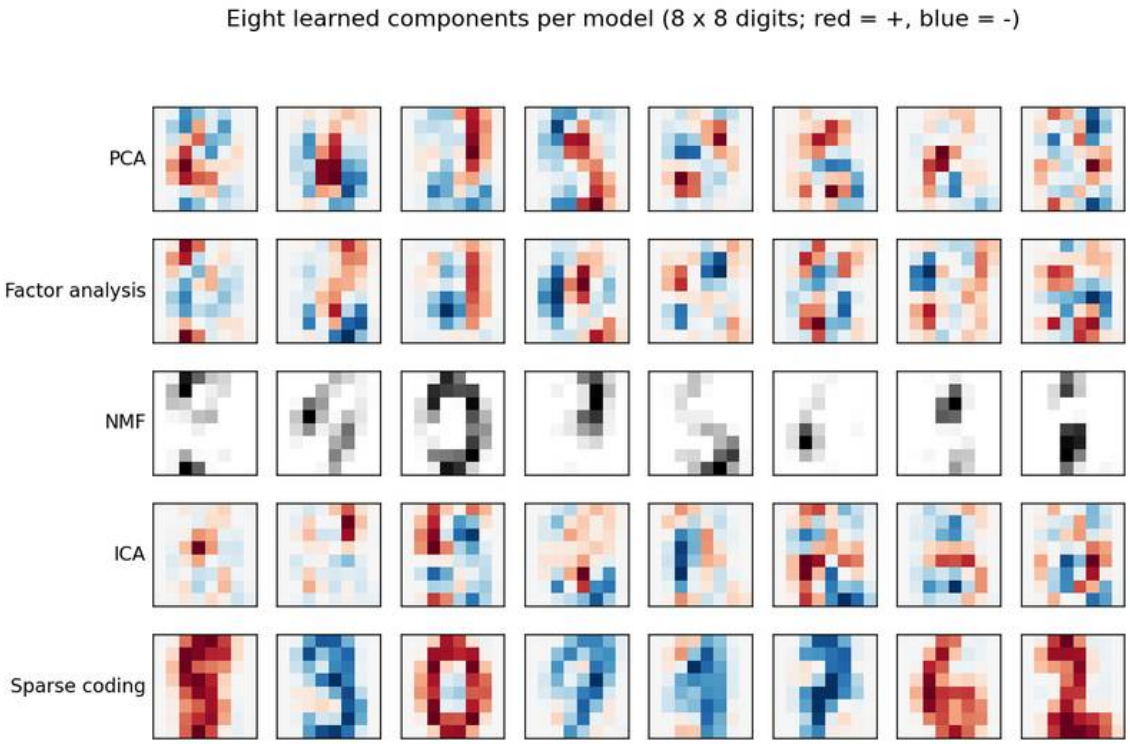



*Figure 3 — Eight components from each linear factor model on the same digits. PCA and FA give signed, global patterns; NMF gives nonnegative strokes; ICA gives signed but more localized patterns; sparse-coding atoms resemble prototypes that are combined a few at a time.*

### Linear autoencoders and PCA

**Plain version.** With centered data and squared reconstruction error, a linear bottleneck autoencoder can recover a PCA subspace at a global optimum. Training need not reach it.

A linear autoencoder with centered input computes

$$z = W_e x, \qquad \hat{x} = W_d z, \qquad \mathcal{L}(W_e, W_d) = \frac{1}{n} \sum_i \|x_i - W_d W_e x_i\|^2,$$

with encoder $W_e \in \mathbb{R}^{k \times d}$ and decoder $W_d \in \mathbb{R}^{d \times k}$, $k < d$. The product $W_d W_e$ has rank at most $k$, so by Eckart–Young the best possible product is the projection onto the top-$k$ principal subspace. Baldi and Hornik [8] proved more: for this squared-error linear network, every local minimum is a global one, and at the global minimum the network projects onto the principal subspace. The individual weight matrices, however, are not unique. For any invertible $k \times k$ matrix $G$, replacing $W_e$ by $GW_e$ and $W_d$ by $W_d G^{-1}$ gives the same product, so the code coordinates can be any invertible linear transform of the PCA scores.

### Code: a linear autoencoder rediscovers the PCA subspace

```
import numpy as np, torch
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
from scipy.linalg import subspace_angles

torch.manual_seed(0)
X = load_digits().data / 16.
X = X - X.mean(axis=0)
Xt = torch.tensor(X, dtype=torch.float32)
k = 5
enc = torch.nn.Linear(64, k, bias=False)
dec = torch.nn.Linear(k, 64, bias=False)
opt = torch.optim.Adam([*enc.parameters(), *dec.parameters()], lr=1e-2)
for step in range(3000):
    loss = (dec(enc(Xt)) - Xt).square().mean()
    opt.zero_grad(); loss.backward(); opt.step()

Wd = dec.weight.detach().numpy()            # 64 x k decoder columns
P = PCA(n_components=k).fit(X).components_.T
angles = np.degrees(subspace_angles(Wd, P))
print("largest angle to PCA subspace (deg):", round(angles.max(), 3))
print("decoder columns orthonormal?",
      np.allclose(Wd.T @ Wd, np.eye(k), atol=1e-2))
```

Output:

```
largest angle to PCA subspace (deg): 0.003
decoder columns orthonormal? False
```

The subspace angle (the largest angle between the two $k$-dimensional subspaces) is essentially zero: the network found the PCA subspace. Yet its decoder columns are not orthonormal, so they are not the principal components themselves. This is the coordinate ambiguity in action, and it is

the first hint of a theme that recurs through the chapter: reconstruction alone pins down *what is kept*, not *how it is labeled*.

### Nonlinear autoencoders

**Plain version.** Replace the linear maps with neural networks, so the code can describe curved structure that no flat subspace can capture.

A general autoencoder has an encoder $f_\phi$ and a decoder $g_\theta$, both neural networks:

$$z = f_\phi(x), \qquad \hat{x} = g_\theta(z), \qquad \min_{\phi,\theta} \frac{1}{n} \sum_i \ell(x_i,\, g_\theta(f_\phi(x_i))),$$

where $\ell$ is a reconstruction loss, such as squared error for real values or binary cross-entropy for values in $[0, 1]$. The choice of $\ell$ is itself an assumption: squared error corresponds to a Gaussian likelihood around $\hat{x}$, cross-entropy to a Bernoulli likelihood.

Hinton and Salakhutdinov [9] showed in 2006 that deep autoencoders, initialized with greedy layer-wise pretraining and then fine-tuned end to end, could find low-dimensional codes that reconstructed images much better than PCA with the same number of dimensions. Today, better initializations, normalization and optimizers make the pretraining step unnecessary, but the core lesson stands: nonlinear encoders can compress much more aggressively than linear ones when the data are curved.

### Undercomplete is not enough, overcomplete needs a rule

An **undercomplete** autoencoder ($k < d$) is forced to compress. That is useful, but not a full guarantee of good features. A sufficiently powerful encoder and decoder could, in principle, map each training example to its own arbitrary code (say, its index) and memorize the decoding, achieving low training error without learning any reusable structure [1, ch. 14]. An **overcomplete** autoencoder ($k \geq d$) can simply learn the identity function. In both cases, what makes the representation useful is not the bottleneck by itself but the *inductive bias*: limited capacity, the architecture (convolutions, for instance), and explicit penalties. The next section covers the three classic penalties, which together are called **regularized autoencoders**.

### Regularized autoencoders: sparse, denoising and contractive

Each penalty answers the question "what should the code ignore?" in a different way.

### Sparse autoencoders

**Plain version.** Allow many code units, but require that only a few of them be active for any given input.

The loss adds a sparsity penalty on the code $h = f_\phi(x)$:

$$\mathcal{L} = \ell(x,\, g_\theta(h)) + \lambda\, \Omega(h), \qquad \Omega(h) = \sum_j |h_j|.$$

Alternatives penalize the deviation of each unit's average activation from a small target rate, or keep only the $k$ largest activations (a "top-$k$" or $k$-sparse autoencoder). A sparse autoencoder is the amortized cousin of sparse coding: the encoder network predicts the sparse code in one forward pass, instead of solving a lasso problem for every input. Goodfellow et al. interpret the penalty as an approximate prior over codes rather than an ordinary weight regularizer [1, ch. 14]. As the Modern view explains, sparse autoencoders have recently become a key tool for interpreting the internals of large neural networks.

### Denoising autoencoders

**Plain version.** Show the network a damaged input and ask it to rebuild the clean one. To succeed, it must learn what clean data look like.

A **denoising autoencoder (DAE)** draws a corrupted copy $\tilde{x} \sim C(\tilde{x} \mid x)$ and minimizes

$$\mathcal{L} = \mathbb{E}_x\, \mathbb{E}_{\tilde{x}\sim C(\cdot|x)}\big[\, \ell(x,\, g_\theta(f_\phi(\tilde{x})))\, \big],$$

where $C$ is a chosen corruption process: additive Gaussian noise, masking noise (setting a random fraction of inputs to zero), or salt-and-pepper noise [10], [11]. Vincent et al. introduced the idea in 2008 [10] and showed in 2010 that stacking DAE layers gave useful features for classifiers [11]. Because the input differs from the target, copying is no longer a solution, even for an overcomplete code.

The corruption defines what the model learns to be robust to. A DAE trained on Gaussian noise has no reason to fill in missing patches, and one trained with masking noise learns to complete missing parts. Modern masked image modeling, such as the masked autoencoder (MAE) in ML 06, is a direct descendant of masking-noise DAEs.

There is also a deeper geometric reading. For small Gaussian corruption, the vector $g(f(\tilde{x})) - \tilde{x}$ points from a corrupted point back toward regions of high data density; it approximately estimates the **score** $\nabla_x \log p(x)$, the gradient of the log-density [1, ch. 14]. This connection is one of the roots of score-based and diffusion generative models, covered in ML 08.

### Code: a denoising autoencoder on digits

This standalone PyTorch example trains a small nonlinear DAE on training digits and evaluates clean reconstruction from noisy held-out input.

```
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

torch.manual_seed(0); torch.set_num_threads(1)
```

```
train, test = train_test_split(load_digits().data/16., test_size=.2, random_state=0)
X = torch.tensor(train, dtype=torch.float32)
Xt = torch.tensor(test, dtype=torch.float32)
encoder = nn.Sequential(nn.Linear(64, 32), nn.ReLU(), nn.Linear(32, 12))
decoder = nn.Sequential(nn.Linear(12, 32), nn.ReLU(), nn.Linear(32, 64), nn.Sigmoid())
model = nn.Sequential(encoder, decoder)
opt = torch.optim.Adam(model.parameters(), lr=.003)
for step in range(2000):
    clean = X[torch.randint(len(X), (64,))]
    noisy = (clean + .2*torch.randn_like(clean)).clamp(0, 1)
    loss = (model(noisy)-clean).square().mean()
    opt.zero_grad(); loss.backward(); opt.step()
model.eval()
with torch.no_grad():
    noisy = (Xt + .2*torch.randn_like(Xt)).clamp(0, 1)
    print("noisy-input MSE:", (noisy-Xt).square().mean().item())
    print("reconstructed MSE:", (model(noisy)-Xt).square().mean().item())
```

Output (about 0.0248 for the noisy input and 0.0197 after the DAE; the last digits depend on the PyTorch build):

```
noisy-input MSE: 0.024781296029686928
reconstructed MSE: 0.01968955807387829
```

The improvement is real but modest, and Figure 4 shows why: with a 12-dimensional code, the DAE removes the speckle but also smooths away some stroke detail. Denoising and faithful compression pull in different directions; a wider code or a convolutional architecture would shift the balance.

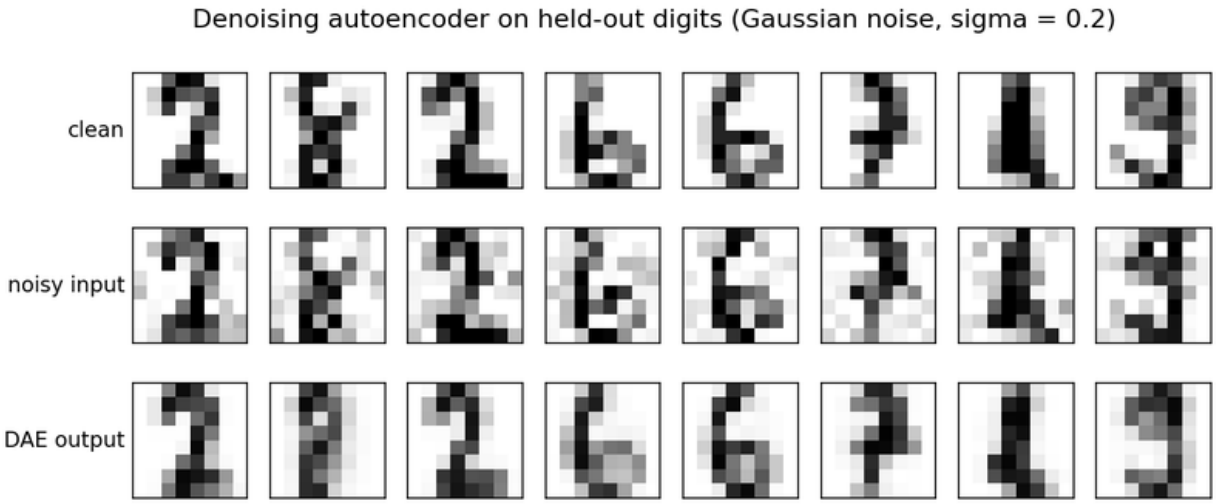



*Figure 4 — Denoising autoencoder outputs on held-out digits. The background speckle is removed; some fine stroke detail is lost through the 12-unit bottleneck.*

### Contractive autoencoders

**Plain version.** A contractive penalty discourages large code changes under small input perturbations. It does not itself know which changes matter to the task.

Rifai et al. [12] add the squared Frobenius norm of the encoder's Jacobian:

$$\mathcal{L} = \ell(x,\, g_\theta(f_\phi(x))) + \lambda \left\| \frac{\partial f_\phi(x)}{\partial x} \right\|_F^2, \qquad \left\| \frac{\partial f_\phi(x)}{\partial x} \right\|_F^2 = \sum_{j,i} \left( \frac{\partial h_j}{\partial x_i} \right)^2 .$$

Here $h = f_\phi(x)$ and the Jacobian entry $\partial h_j / \partial x_i$ measures how much code unit $j$ moves when input coordinate $i$ moves. For a single sigmoid layer $h = \sigma(Wx + b)$, the entry is $h_j(1 - h_j)W_{ji}$, so the penalty is cheap to compute:

$$\left\| \frac{\partial h}{\partial x} \right\|_F^2 = \sum_j \left( h_j(1 - h_j) \right)^2 \sum_i W_{ji}^2 .$$

The reconstruction term wants the code to change with the input (otherwise it could not reconstruct different inputs), while the penalty wants it to change as little as possible. The compromise is a code that is sensitive only along the few directions in which the training data actually vary. A DAE achieves a related effect implicitly: for small Gaussian noise, denoising training penalizes the sensitivity of the whole reconstruction function [1, ch. 14], whereas the CAE penalizes only the encoder, explicitly.

### Code: the contractive penalty in action

```
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

train, test = train_test_split(load_digits().data/16., test_size=.2, random_state=0)
X = torch.tensor(train, dtype=torch.float32)
Xt = torch.tensor(test, dtype=torch.float32)

def fit(lam):
    torch.manual_seed(0)
    W = nn.Linear(64, 16)                    # encoder h = sigmoid(Wx + b)
    dec = nn.Linear(16, 64)
    opt = torch.optim.Adam([*W.parameters(), *dec.parameters()], lr=3e-3)
    for step in range(1500):
        x = X[torch.randint(len(X), (64,))]
        h = torch.sigmoid(W(x))
        rec = (dec(h) - x).square().sum(1).mean()
        # ||J||_F^2 = sum_j (h_j(1-h_j))^2 * ||W_j||^2 for a sigmoid layer
        jac = ((h*(1-h))**2 * W.weight.square().sum(1)).sum(1).mean()
        loss = rec + lam*jac
        opt.zero_grad(); loss.backward(); opt.step()
    with torch.no_grad():
```

```
        h = torch.sigmoid(W(Xt))
        mse = (dec(h) - Xt).square().mean().item()
        jac = ((h*(1-h))**2 * W.weight.square().sum(1)).sum(1).mean().item()
    return mse, jac

for lam in (0.0, 0.5):
    mse, jac = fit(lam)
    print(f"lambda={lam}: test MSE={mse:.4f}, mean ||J||_F^2={jac:.2f}")
```

Output:

```
lambda=0.0: test MSE=0.0141, mean ||J||_F^2=4.79
lambda=0.5: test MSE=0.0194, mean ||J||_F^2=0.70
```

The penalty cuts the encoder's average sensitivity by a factor of almost seven, at the cost of a somewhat higher reconstruction error. Whether that trade is worthwhile depends on the downstream use: a smoother code often helps nearest-neighbor retrieval or classification with few labels, while a codec would prefer the unpenalized model.

### The manifold view

**Plain version.** The manifold hypothesis models data near a lower-dimensional surface, like a rolled sheet in a room. Useful coordinates describe position along that surface.

The **manifold hypothesis** says that natural data, such as images, concentrate near a low-dimensional manifold embedded in the high-dimensional input space [3]. A random 64-pixel array almost never looks like a digit; the set of plausible digits is a tiny, structured region. Locally, around any data point, the manifold looks like a flat plane, the **tangent space**, spanned by the few directions you can move while staying realistic (thicker stroke, slight slant, small shift). Directions orthogonal to the tangent space lead off the manifold into noise.

This view gives a single lens for everything in the chapter:

- **PCA** fits one global flat sheet. It works when the manifold is nearly flat and fails when it is curved.
- **Mixtures of PPCA or FA** fit many local flat sheets.
- **Nonlinear autoencoders** learn a curved chart of the manifold: the encoder gives coordinates on the sheet, and the decoder maps them back into the room.
- **Denoising** teaches the model to move points that fell off the manifold back onto it.
- **Contractive** penalties make the encoder insensitive to directions orthogonal to the manifold, while reconstruction keeps it sensitive along the tangent directions [1, ch. 14], [12].

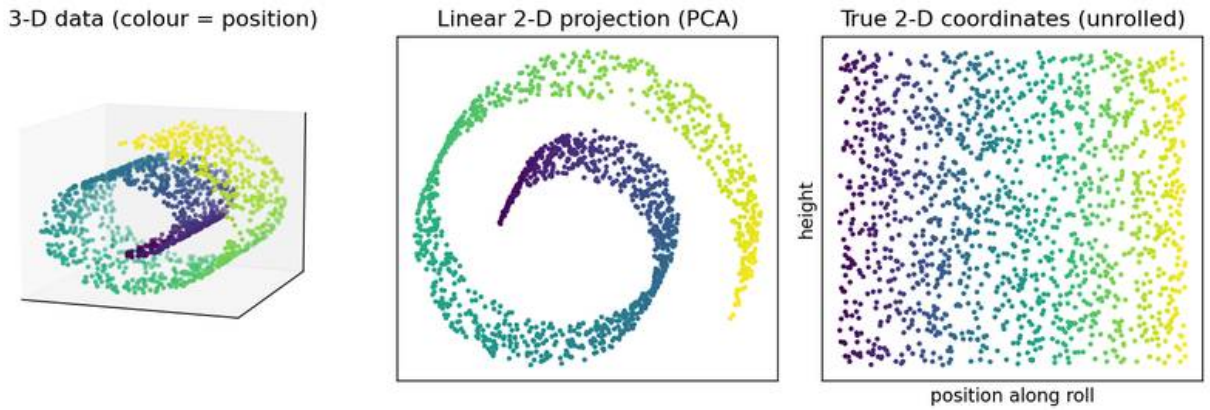



*Figure 5 — The Swiss roll is a two-dimensional sheet rolled up in three dimensions. A linear projection (PCA) flattens it into a spiral that does not preserve position along the sheet; the true intrinsic coordinates form a flat rectangle.*

Figure 5 makes the limitation of linear methods concrete. The Swiss roll has two intrinsic dimensions, position along the roll and height, but no two-dimensional linear projection can unroll it: the PCA view keeps the spiral, so points far apart along the sheet are not laid out in order. A nonlinear encoder can, in principle, learn the unrolled coordinates. In practice, whether it does depends on capacity, regularization and the amount of data; manifold learning in high dimensions remains hard.

### From autoencoders to VAEs

**Plain version.** An ordinary autoencoder learns to compress and decompress, but it gives no rule for *inventing* new codes. A VAE adds that rule.

Suppose a trained autoencoder maps every training digit to a code in $\mathbb{R}^{12}$. To generate a new digit, we would like to pick a random code and decode it. But which random code? The training codes may occupy a strange, clumpy region, and a random point between them may decode to garbage. Nothing in the reconstruction loss asked the code space to be smooth, filled, or shaped like any known distribution. Excellent reconstruction therefore does not imply good generation.

The **variational autoencoder** [13] fixes this by returning to the probabilistic factor-model story of PPCA and FA, but with a neural decoder:

$$z \sim p(z) = \mathcal{N}(0, I), \qquad x \sim p_\theta(x \mid z),$$

where $p_\theta(x \mid z)$ is a distribution whose parameters are produced by a decoder network. Because the true posterior $p_\theta(z \mid x)$ is now intractable, a second network, the encoder, outputs an approximate posterior $q_\phi(z \mid x)$, typically a diagonal Gaussian. Training maximizes the **evidence lower bound (ELBO)** on $\log p_\theta(x)$:

$$\log p_\theta(x) \;\geq\; \underbrace{\mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z)\big]}_{\text{reconstruction}} \;-\; \underbrace{D_{\mathrm{KL}}\big(q_\phi(z \mid x) \,\|\, p(z)\big)}_{\text{keep codes near the prior}}.$$

Read the two terms against everything above:

- The **reconstruction** term is exactly an autoencoder loss, now with noise injected into the code. With a Gaussian decoder it becomes squared error, with a Bernoulli decoder binary cross-entropy.
- The **KL** term is a regularizer, like the sparse and contractive penalties, but chosen so that codes from all training examples together fill the prior. That is what makes sampling $z \sim \mathcal{N}(0, I)$ meaningful.
- With a **linear** Gaussian decoder, the model *is* PPCA. Its exact posterior is Gaussian and can be matched by a linear Gaussian encoder, so the bound becomes tight and the optimum coincides with the PPCA maximum-likelihood solution. The VAE is the nonlinear generalization of the factor models we began with.

Kingma and Welling's introduction [14] develops this connection at length. Weighting the KL term by a factor $\beta > 1$ gives the **β-VAE** [15], which was proposed to encourage disentangled factors at the cost of reconstruction quality; note that with $\beta \neq 1$ the objective is no longer the ELBO. The full derivation of the ELBO, the reparameterization trick and failure modes such as posterior collapse are developed in ML 07.

### What reconstruction error can and cannot tell you

Before leaving autoencoders, a practical warning. Reconstruction error is often used as an anomaly score: an autoencoder trained on normal data should reconstruct normal inputs well and unusual ones poorly. This can work, but it fails when the autoencoder generalizes too well and reconstructs unusual inputs too, or when anomalies differ only in fine detail that the bottleneck discards anyway. Similarly, a code that reconstructs well may be poor for classification, and a code that classifies well may discard the pixel detail a restoration task needs. Evaluate compression, denoising, downstream usefulness and generation separately.

## Modern view

**Plain version.** The classical models of this chapter did not disappear. They became components: compression stages for generative models, pretraining objectives for vision, and microscopes for looking inside large networks.

### Surveys worth reading

- **Bengio, Courville and Vincent (2013)** [3] is the standard review of representation learning. It frames good representations through a list of general priors (smoothness, multiple explanatory factors, a hierarchy of concepts, shared factors across tasks, the manifold hypothesis, sparsity and others), and connects autoencoders, probabilistic models and manifold learning under that lens. It is a good map for *why* the penalties in this chapter exist.

- **Bank, Koenigstein and Giryes (2020)** [16] is a compact survey chapter on autoencoders. It walks through regularized variants (sparse, denoising, contractive), VAEs and disentangled variants, applications (generative modeling, classification, clustering, anomaly detection, recommendation and dimensionality reduction) and hybrids with adversarial training, which makes it a useful companion to this tutorial.
- **Kingma and Welling (2019)** [14] is the authors' own extended introduction to VAEs. It covers the ELBO, the reparameterization trick, richer posteriors and hierarchical models, and places VAEs among other deep latent-variable models.
- **Goodfellow, Bengio and Courville** [1, ch. 13–14] and **Murphy** [2] remain the clearest textbook treatments of linear factor models, the DAE–score connection and the manifold interpretation.

### Disentanglement: a sober result

A recurring hope has been that unsupervised factor models would recover *the* true underlying factors, one per code unit ("disentanglement"). Locatello et al. [17] showed theoretically that unsupervised learning of disentangled representations is fundamentally impossible without inductive biases on both models and data, which mirrors the rotation ambiguity of factor analysis at a much larger scale. In a large reproducible study they also found that well-disentangled models could not be identified without supervision, and that more disentanglement did not reliably reduce the sample complexity of downstream tasks. The practical lesson echoes ICA: identifiability always comes from extra assumptions, so state them.

### Autoencoders as tokenizers

The biggest modern role of autoencoders is as the first stage of large generative models. **VQ-VAE** [18] replaces the continuous code with a learned codebook of discrete vectors, so images become grids of discrete tokens that an autoregressive model can predict (see ML 09). **Latent diffusion** [19] trains an autoencoder for perceptual compression and then runs the diffusion model in its latent space instead of pixel space, which greatly reduces compute. In both cases, the autoencoder's reconstruction quality sets a ceiling: the generator cannot restore detail the encoder discarded. Designing these tokenizers, balancing compression, reconstruction and how easy the latent space is for the generator to model, is an active research area as of 2026.

### Sparse autoencoders for interpretability (2023–2026)

Sparse autoencoders have found a new and very active application: **mechanistic interpretability** of large language and vision models. The motivating observation is that individual neurons are often polysemantic, responding to several unrelated concepts. Cunningham et al. [20] trained overcomplete sparse autoencoders to reconstruct a language model's internal activations, and found sparsely activating features that automated measures rated more interpretable and monosemantic than the neurons or other decompositions. Gao et al. [21] used $k$-sparse (top-$k$) autoencoders to control sparsity directly, reported clean scaling laws in autoencoder size and sparsity, proposed new metrics for feature quality, and trained a 16-million-latent autoencoder on GPT-4 activations.

Conceptually this is sparse coding (Olshausen and Field's idea) applied to a network's activations instead of image patches.

### Open problems

- **Evaluation.** Reconstruction error, sparsity and downstream accuracy disagree, and there is no agreed measure of whether learned features are the "right" ones, either for disentanglement or for interpretability.
- **Identifiability.** When can learned factors be trusted to correspond to real causes? Nonlinear ICA and causal representation learning study which extra assumptions (time structure, auxiliary variables, interventions) make this possible.
- **Tokenizer design.** How much to compress, whether codes should be discrete or continuous, and how to make latents both reconstructive and easy to model remain open engineering and theory questions.

## Key takeaways

- Factor models explain observations through latent variables; their assumptions determine which representations are identifiable.
- PCA optimizes a linear reconstruction subspace; PPCA and factor analysis add different noise models.
- ICA, sparse coding and NMF use independence, sparsity and nonnegativity to favor different decompositions.
- A squared-error linear autoencoder can recover the PCA subspace at a suitable global optimum; its weights need not be PCA axes.
- Bottlenecks and sparse, denoising or contractive objectives constrain codes differently; reconstruction quality is not semantic quality.
- A VAE adds a probabilistic latent model and approximate inference; linear Gaussian choices connect it to PPCA.

## Exercises

1. **Noise assumptions.** You record 20 sensors, five of which are much noisier than the rest, but all respond to the same few underlying causes. Would you start with PPCA or factor analysis? What would PCA tend to do with the noisy sensors?

   **Hint**

   Factor analysis: its diagonal $\Psi$ can assign large residual variance to the five noisy sensors without spending factors on them. PCA (and PPCA, with a single shared $\sigma^2$) may devote components to the high-variance noisy sensors, because high variance is all PCA sees.

2. **Latent ambiguity.** For an invertible $k \times k$ matrix $G$, show that replacing $W_e$ by $GW_e$ and $W_d$ by $W_d G^{-1}$ leaves a linear autoencoder's reconstruction unchanged. Which matrices $G$ also preserve the PPCA likelihood?

   **Hint**

   The product is $W_d G^{-1} G W_e = W_d W_e$. For PPCA, the likelihood depends on $WW^\top$; replacing $W$ by $WG$ gives $WGG^\top W^\top$, which is unchanged when $GG^\top = I$, that is, when $G$ is orthogonal.

3. **Why ICA fails on Gaussians.** Generate two independent standard Gaussian sources, mix them with any invertible $2 \times 2$ matrix, and run FastICA several times with different random seeds. What do you observe, and why?

   **Hint**

   After whitening, every rotation of two independent unit Gaussians is again two independent unit Gaussians, so there is no "right" rotation to find. Different seeds give different, equally valid unmixing directions, and the correlation with the true sources varies from run to run.

4. **Sparse codes versus PCA codes.** Using the sparse-coding snippet, compare the number of nonzero code entries per test image with PCA codes of the same dimension (32). Then plot the test MSE of sparse coding as `alpha` varies over 0.05, 0.1, 0.3, 1.0.

   **Hint**

   PCA codes are dense: essentially all 32 entries are nonzero. As `alpha` grows, the number of active atoms drops and the error rises, tracing a sparsity–fidelity curve. Remember that the dictionary was not given a mean term, so compare curves rather than single numbers with PCA.

5. **Denoising mismatch.** Train the DAE from the tutorial on Gaussian noise, then test it on inputs where a random $3 \times 3$ block is set to zero. Then train a second DAE on that masking corruption and test both on both corruptions.

   **Hint**

   These are different corruption processes. The Gaussian-trained model has never been asked to fill in missing structure, so it typically leaves holes; the masking-trained model learns completion. Report a $2 \times 2$ table (train corruption × test corruption) rather than inferring robustness to one from success on the other.

6. **Contractive penalty by hand.** For a single sigmoid unit $h = \sigma(w^\top x + b)$, derive $\|\partial h / \partial x\|^2$. When is it largest, and what does that say about where a contractive autoencoder allows its code to be sensitive?

   **Hint**

   $\partial h / \partial x = h(1-h)w$, so the squared norm is $h^2(1-h)^2\|w\|^2$. It is largest at $h = 1/2$ (the steep middle of the sigmoid) and nearly zero when the unit is saturated. The penalty pushes units to be saturated for most inputs, keeping them in the sensitive regime only where the data require variation.

## References


1. I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016, ch. 13–14. Book
2. K. P. Murphy, *Probabilistic Machine Learning: An Introduction*, MIT Press, 2022. Book
3. Y. Bengio, A. Courville and P. Vincent, "Representation Learning: A Review and New Perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, 2013. Paper
4. M. E. Tipping and C. M. Bishop, "Probabilistic Principal Component Analysis," *Journal of the Royal Statistical Society: Series B*, vol. 61, no. 3, 1999. Paper
5. A. Hyvärinen and E. Oja, "Independent component analysis: algorithms and applications," *Neural Networks*, vol. 13, no. 4–5, 2000. Paper
6. B. A. Olshausen and D. J. Field, "Emergence of simple-cell receptive field properties by learning a sparse code for natural images," *Nature*, vol. 381, 1996. Paper
7. D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, 1999. Paper
8. P. Baldi and K. Hornik, "Neural networks and principal component analysis: Learning from examples without local minima," *Neural Networks*, vol. 2, no. 1, 1989. Paper
9. G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science*, vol. 313, 2006. Paper
10. P. Vincent, H. Larochelle, Y. Bengio and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," ICML, 2008. Paper
11. P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio and P.-A. Manzagol, "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion," *JMLR*, vol. 11, 2010. Paper
12. S. Rifai, P. Vincent, X. Muller, X. Glorot and Y. Bengio, "Contractive Auto-Encoders: Explicit Invariance During Feature Extraction," ICML, 2011. Paper
13. D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," ICLR, 2014. Paper
14. D. P. Kingma and M. Welling, "An Introduction to Variational Autoencoders," *Foundations and Trends in Machine Learning*, vol. 12, no. 4, 2019. Paper
15. I. Higgins et al., "β-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework," ICLR, 2017. Paper
16. D. Bank, N. Koenigstein and R. Giryes, "Autoencoders," survey chapter, arXiv:2003.05991, 2020. Survey
17. F. Locatello et al., "Challenging Common Assumptions in the Unsupervised Learning of Disentangled Representations," ICML, 2019. Paper
18. A. van den Oord, O. Vinyals and K. Kavukcuoglu, "Neural Discrete Representation Learning," NeurIPS, 2017. Paper
19. R. Rombach, A. Blattmann, D. Lorenz, P. Esser and B. Ommer, "High-Resolution Image Synthesis with Latent Diffusion Models," CVPR, 2022. Paper
20. H. Cunningham, A. Ewart, L. Riggs, R. Huben and L. Sharkey, "Sparse Autoencoders Find Highly Interpretable Features in Language Models," arXiv:2309.08600, 2023. Paper
21. L. Gao et al., "Scaling and evaluating sparse autoencoders," arXiv:2406.04093, 2024. Paper

# ML 07 • Generative Models: Autoencoders, VAEs, GANs and Flows

Learn distributions rather than labels: likelihood-based versus implicit models, the ELBO and reparameterization, adversarial training and Wasserstein GANs, normalizing flows, and how generative models are evaluated.

Online edition: https://blog.shyandram.dev/en/learn/ml/generative-models/

## What you'll learn

- Distinguish distribution modeling from label prediction.
- Compare likelihood-based models with implicit samplers.
- Derive the VAE bound and understand reparameterization.
- Explain adversarial training and invertible density transformations.
- Evaluate likelihood, realism, diversity and conditioning separately.

## The big picture

A classifier asks, "What is in this image?" A **generative model** asks, "What kinds of images could occur?" If it learns a useful distribution, we can draw new samples, fill missing regions, compress data, detect unusual inputs, or condition generation on text, a class label or another image.

> **Explain like I'm five**
>
> A child who recognizes drawings of houses can say "house." A child who understands how houses are drawn can invent a new one with a door, windows and a roof. Generative modeling tries to learn the recipe behind the examples, so it can cook up new ones — and a good cook makes a different, but still tasty, dish every time instead of the same one over and over.

Learning a distribution is harder than learning a label. A label is one number; an image has thousands of pixels whose values depend on one another in complicated ways. Every family in this chapter is a different answer to the same engineering question: how do you represent a distribution over very high-dimensional data so that you can both **train** it from examples and **sample** from it afterwards?

This chapter uses the probabilistic viewpoint developed in Bishop and Bishop [1] and Prince [2]. The comparative review by Bond-Taylor et al. [3] organizes the same families along the trade-offs we will meet — training speed, sample quality, diversity, and whether a likelihood is available — and the flow review by Papamakarios et al. [4] is particularly helpful for distinguishing a learned distribution from a convenient sampling procedure. Diffusion and autoregressive models get

their own chapters (ML 08 and ML 09) because their training and sampling deserve careful treatment; here we place them on the map.

## Modeling a distribution

**Plain version.** Fit a probability model so that the observed examples look likely under it, then use the model to produce new examples that look like they came from the same source.

### Maximum likelihood

Suppose the data $x_1, \dots, x_n$ are independent draws from an unknown distribution $p_{\text{data}}$, and $p_\theta$ is a model with parameters $\theta$. Maximum likelihood minimizes the average **negative log-likelihood (NLL)**:

$$\mathcal{L}_{\text{NLL}}(\theta) = -\frac{1}{n}\sum_{i=1}^{n} \log p_\theta(x_i).$$

Here $n$ is the number of training examples and $\log p_\theta(x_i)$ is the log-density the model assigns to example $i$.

Why is this a sensible goal? As $n$ grows, the average approaches an expectation over the data distribution, and

$$\mathbb{E}_{x\sim p_{\text{data}}}[-\log p_\theta(x)] = D_{\text{KL}}(p_{\text{data}} \,\|\, p_\theta) + H(p_{\text{data}}).$$

$D_{\text{KL}}(p\|q) = \mathbb{E}_p[\log p - \log q]$ is the Kullback–Leibler divergence, which is zero only when the two distributions agree, and $H(p_{\text{data}})$ is the entropy of the data, which does not depend on $\theta$. So **maximum likelihood is the same as minimizing the forward KL divergence** from data to model.

The direction of the KL matters. The forward KL heavily punishes a model that assigns near-zero density where real data lives, so likelihood training is "mass-covering": it prefers to spread probability over every mode, even if that means putting some probability between modes. Keep this in mind; we will see that GANs behave differently.

### Densities, discrete pixels and bits per dimension

For continuous values, $p_\theta(x)$ is a **density**, not the probability of observing exactly $x$; a density can exceed 1. Real images, however, store integers such as 0–255. Fitting a continuous density directly to integers is ill-posed: the model can place infinitely tall spikes on the integer grid and drive the likelihood to infinity. The standard fix is **dequantization** — add uniform noise in $[0, 1)$ to each integer value before fitting a continuous density. Theis et al. [5] show that the continuous model's log-likelihood on dequantized data is then a lower bound on the log-likelihood of a corresponding discrete model, which makes the numbers comparable across papers. Alternatively one uses a discrete likelihood directly (a softmax over 256 values, or a Bernoulli for binary pixels, as in our code below).

Image likelihoods are usually reported in **bits per dimension**:

$$\text{bpd} = \frac{\mathcal{L}_{\text{NLL}}}{D \ln 2},$$

Here $D$ is the number of stored values (height × width × channels). For a discrete model, $\mathcal{L}_{\text{NLL}}$ is the negative log-probability in nats per image, and bpd is the ideal coding cost per stored value. A continuous density needs the dequantization convention too: if $y = (x + u)/256$ for 8-bit integers $x$ and independent uniform noise $u \in [0, 1)^D$, its dequantization bound in bits per dimension is $8 - \mathbb{E}_u[\log p_\theta(y)]/(D \ln 2)$. The added 8 accounts for the volume of a quantization bin after rescaling. This is an upper bound on the discrete NLL per dimension, not an exact discrete likelihood [5]. State the units, noise distribution and any further change-of-variables correction before comparing scores.

### Conditional and unconditional generation

A discriminative classifier models $p(y \mid x)$. A conditional generator models $p(x \mid c)$, where $c$ can be text, a class, a low-resolution image or observed pixels. Bayes' rule links related quantities, but a classifier does not by itself provide an image sampler.

Conditioning also explains a familiar failure. Minimizing pixel MSE predicts the **conditional mean** $\mathbb{E}[x \mid c]$. If an ambiguous input has several valid outputs — a gray-scale photo whose shirt could be red or blue — the mean blends them into a brownish average. A generative model instead tries to represent the whole distribution $p(x \mid c)$ and sample one plausible answer at a time.

### Two families: likelihood-based and implicit

Every model below needs two abilities: a way to **train** from data and a way to **sample**. They differ in whether they can evaluate $p_\theta(x)$.

- **Likelihood-based (explicit) models** write down a density, or a bound on it, and maximize it. Autoregressive models and normalizing flows give exact likelihoods; VAEs (and diffusion models, in their variational view) optimize a lower bound; energy-based models define a density only up to an unknown normalizing constant.
- **Implicit models** define a sampler — "push noise through a network" — without any formula for $p_\theta(x)$. GANs are the main example. Because there is no likelihood to maximize, they must be trained by comparing samples with data in some other way.

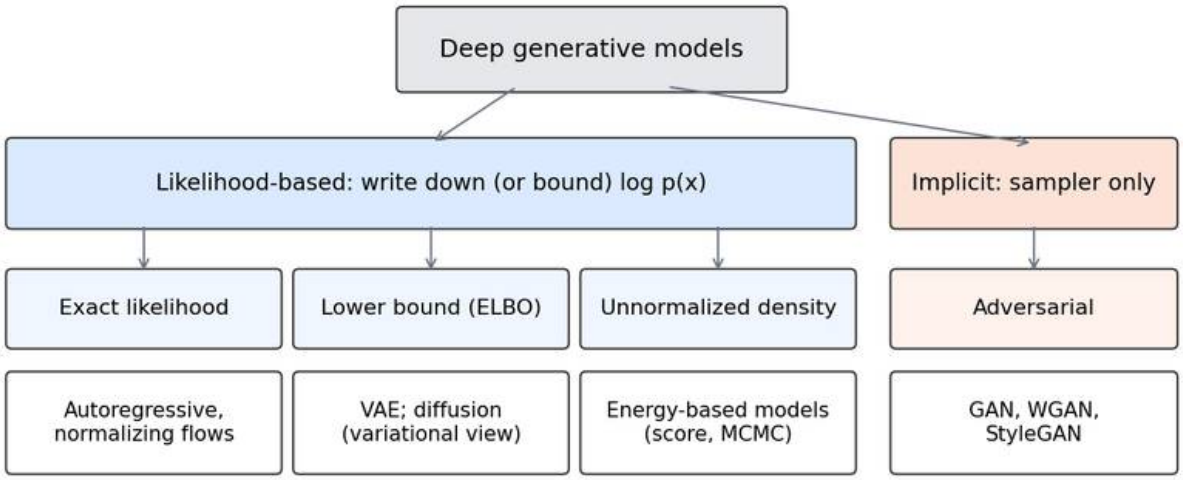



*Figure 1 — A teaching taxonomy of generative models, following the explicit/implicit split used in comparative reviews* [3]. *Real systems often combine several branches.*

## Ordinary autoencoders: compression without a prior

**Plain version.** Encode an example into a compact code and decode it back. That is excellent for compression and representation learning, but it does not by itself tell you how to invent new codes.

An encoder $z = f_\phi(x)$ and decoder $\hat{x} = g_\theta(z)$ minimize a reconstruction loss such as $\|x - \hat{x}\|^2$. Autoencoders, their linear relatives (PCA, factor analysis) and their regularized variants are developed in ML 06A. The point for this chapter is narrow: an ordinary autoencoder places no requirement on **where** codes live. The training codes may occupy a thin, oddly shaped region of latent space. If you then sample $z \sim \mathcal{N}(0, I)$ and decode it, you may land in an empty region the decoder never saw, and get garbage. Excellent reconstruction therefore does not guarantee good unconditional generation.

A VAE fixes this by turning the autoencoder into a probability model with a prior over codes.

## VAEs: a latent-variable probability model

**Plain version.** A VAE models observations through random latent codes and a decoder. Its encoder approximates the posterior over codes for each observation.

### The model

Choose a prior $p(z)$ over a latent vector $z \in \mathbb{R}^k$, usually a standard Gaussian $\mathcal{N}(0, I)$, and a decoder $p_\theta(x \mid z)$, a neural network that outputs the parameters of a distribution over images

(Bernoulli probabilities for binary pixels, or Gaussian means). The model's density over images is the marginal

$$p_\theta(x) = \int p_\theta(x \mid z)\, p(z)\, dz.$$

Sampling is easy: draw $z \sim p(z)$, then draw $x \sim p_\theta(x \mid z)$. Evaluating $p_\theta(x)$ is hard: the integral runs over every possible code, and the decoder is a nonlinear network, so there is no closed form. The true posterior $p_\theta(z \mid x) = p_\theta(x \mid z)p(z)/p_\theta(x)$ is intractable for the same reason.

### Deriving the evidence lower bound

Kingma and Welling [6] introduce an **encoder** (or inference network) $q_\phi(z \mid x)$ that approximates the posterior. Here is the derivation step by step. Since $\log p_\theta(x)$ does not depend on $z$, we can take its expectation under any distribution over $z$ without changing it:

$$\begin{aligned}
\log p_\theta(x) &= \mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x)\big] \\
&= \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{p_\theta(z \mid x)}\right] \\
&= \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z \mid x)}\right] + \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{q_\phi(z \mid x)}{p_\theta(z \mid x)}\right] \\
&= \mathrm{ELBO}(x) + D_{\mathrm{KL}}(q_\phi(z \mid x) \,\|\, p_\theta(z \mid x)).
\end{aligned}$$

The second line uses $p_\theta(x) = p_\theta(x,z)/p_\theta(z \mid x)$; the third multiplies and divides by $q_\phi(z \mid x)$ inside the log. Because a KL divergence is never negative, the first term is a lower bound on $\log p_\theta(x)$: the **evidence lower bound (ELBO)**. The **gap** between the bound and the true log-likelihood is exactly the KL divergence between the approximate and true posteriors. A better encoder tightens the bound; a perfect one closes it.

Expanding $p_\theta(x,z) = p_\theta(x \mid z)p(z)$ gives the form used in code:

$$\mathrm{ELBO}(x) = \underbrace{\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)]}_{\text{reconstruction}} - \underbrace{D_{\mathrm{KL}}(q_\phi(z \mid x) \,\|\, p(z))}_{\text{stay close to the prior}}.$$

The first term rewards codes that let the decoder reproduce $x$; the second penalizes codes whose distribution wanders away from the prior. The same bound can be reached in one line with Jensen's inequality ($\log \mathbb{E} \geq \mathbb{E} \log$), but the decomposition above is more useful because it names the gap. Training maximizes the average ELBO over the data with respect to both $\theta$ and $\phi$; equivalently, it minimizes the **negative ELBO**.

For a diagonal Gaussian encoder $q_\phi = \mathcal{N}(\mu, \mathrm{diag}(\sigma^2))$ and a standard Gaussian prior, the KL term has a closed form:

$$D_{\mathrm{KL}}(q_\phi \| p) = \frac{1}{2}\sum_{j=1}^{k}\left(\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1\right),$$

where $\mu_j$ and $\sigma_j^2$ are the encoder's mean and variance for latent coordinate $j$. Each summand is zero exactly when $\mu_j = 0$ and $\sigma_j^2 = 1$.

### Reparameterization

The reconstruction term is an expectation over $z \sim q_\phi(z \mid x)$, and we need its gradient with respect to $\phi$. Sampling is not differentiable as written: a random draw is a black box. The **reparameterization trick** [6] moves the randomness into a fixed noise variable:

$$\epsilon \sim \mathcal{N}(0, I), \qquad z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon.$$

Here $\odot$ is elementwise multiplication. Now $z$ is a deterministic, differentiable function of $\phi$ and an independent $\epsilon$, so ordinary backpropagation gives low-variance gradient estimates through $\mu$ and $\sigma$. In code, the encoder predicts `logvar` $= \log \sigma^2$ and uses `exp(.5 * logvar)` as the standard deviation, which guarantees positivity without constraints.

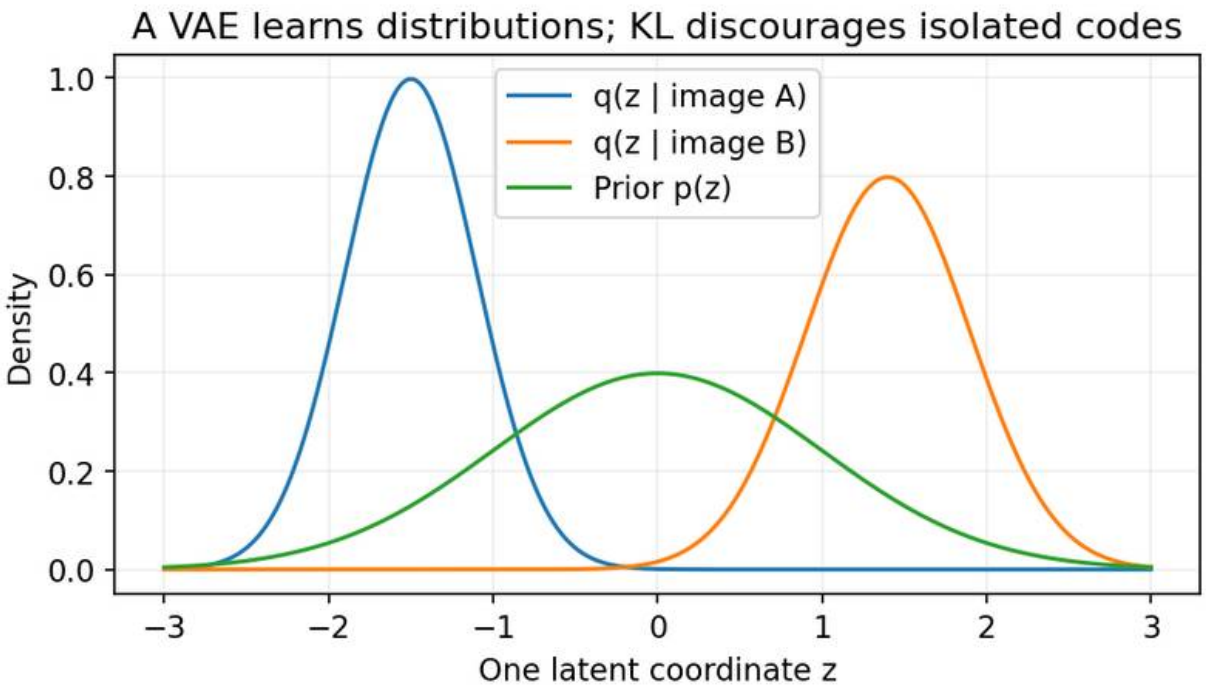



*Figure 2 — A VAE represents uncertainty in its code and encourages compatibility with a shared prior. The curves are illustrative.*

### Code: a small VAE for binary digits

This complete CPU example uses scikit-learn's bundled 8 × 8 digits, binarized so that a Bernoulli decoder is a stated likelihood model. Reconstruction is summed over pixels and KL over latent dimensions, then both are averaged over examples; silently averaging only one term would change their balance.

```
import torch
```

```
from torch import nn
from torch.nn import functional as F
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

torch.manual_seed(0)
torch.set_num_threads(1)
d = load_digits()
train, test = train_test_split(d.data, test_size=.2, random_state=0,
                               stratify=d.target)
X = torch.tensor((train > 8).astype("float32"))
Xt = torch.tensor((test > 8).astype("float32"))
enc = nn.Sequential(nn.Linear(64, 64), nn.ReLU(), nn.Linear(64, 16))
dec = nn.Sequential(nn.Linear(8, 64), nn.ReLU(), nn.Linear(64, 64))
opt = torch.optim.Adam(list(enc.parameters()) + list(dec.parameters()), lr=.002)

def terms(x):
    mu, logvar = enc(x).chunk(2, dim=-1)
    z = mu + (.5*logvar).exp() * torch.randn_like(mu)
    recon = F.binary_cross_entropy_with_logits(dec(z), x, reduction="none").sum(1)
    kl = .5*(mu.square() + logvar.exp() - logvar - 1).sum(1)
    return recon, kl

for step in range(400):
    x = X[torch.randint(len(X), (64,))]
    recon, kl = terms(x)
    loss = (recon + kl).mean()
    opt.zero_grad()
    loss.backward()
    opt.step()
enc.eval(); dec.eval()
with torch.no_grad():
    recon, kl = terms(Xt)
    samples = dec(torch.randn(4, 8)).sigmoid().reshape(4, 8, 8)
    print("test negative ELBO estimate:", round((recon + kl).mean().item(), 3))
    print("reconstruction / KL:", round(recon.mean().item(), 3),
          round(kl.mean().item(), 3))
    print("sample probability maps:", tuple(samples.shape))
```

On our machine it prints a test negative ELBO of about 20.5 nats per image, split into roughly 17.0 nats of reconstruction and 3.5 nats of KL. Since the bound is a lower bound on log-likelihood, 20.5 nats is an **upper bound** on the test NLL. The output maps are Bernoulli probabilities; to obtain binary samples, draw each pixel with those probabilities. The test estimate is stochastic, so average several latent draws for a more stable number. No test-based checkpoint selection happens here.

Figure 3 shows what a VAE with only a **two-dimensional** latent space learns on the same data (trained longer by the figure script). Even though labels are never used, digits of the same class land near each other, and decoding a regular grid of $z$ values produces smooth transitions between digit shapes. This is the property an ordinary autoencoder lacks: every point near the prior's bulk decodes to something digit-like.

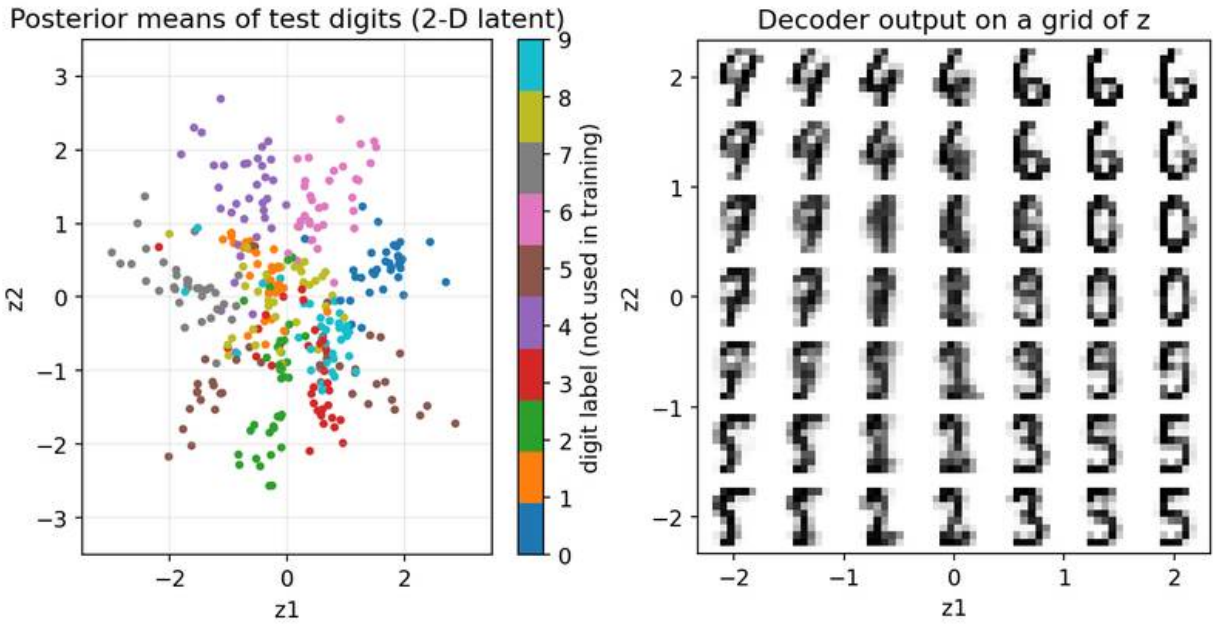



*Figure 3 — A VAE with a 2-D latent space on binarized 8 × 8 digits. Left: encoder means for test images (labels used only for coloring). Right: decoder outputs on a grid of latent values.*

### Posterior collapse, β-VAE and deeper VAEs

If the decoder is very powerful (for example, an autoregressive decoder that can model pixels on its own), the cheapest way to raise the ELBO may be to ignore $z$ entirely: set $q_\phi(z \mid x) = p(z)$ so the KL term is zero, and let the decoder do all the work. This is **posterior collapse**. Bowman et al. [7] observed it when training VAEs on text and proposed **KL annealing**: start with a small weight on the KL term and raise it to 1 during training, so the model learns to use the code before the penalty bites. Monitoring the KL per latent dimension is the simplest diagnostic — dimensions with KL near zero carry no information.

Higgins et al. [8] go the other way: the **β-VAE** multiplies the KL term by $\beta > 1$, pressuring the model toward a more factorized, "disentangled" code at the cost of reconstruction. Be precise about what this means: with $\beta \neq 1$ the objective is no longer the ELBO, so the value is not a bound on $\log p_\theta(x)$, and Kingma and Welling's tutorial review [9] discusses this and many other variants. Finally, single-layer VAEs with factorized Gaussians tend to produce blurry images, partly because the Gaussian decoder averages over uncertainty. **Hierarchical VAEs** stack several layers of latent variables; NVAE [10] showed that a carefully engineered deep hierarchy with stabilizing regularization can produce sharp samples on natural images up to 256 × 256 pixels.

*Online only: VAE — learning through a tractable lower bound — https://blog.shyandram.dev/en/learn/ml/generative-models/papers/vae–learning-through-a-tractable-lower-bound/*

## GANs: learning with an adversary

**Plain version.** A generator learns to fool a discriminator that separates generated samples from data. This competing objective can improve samples, but training can be unstable.

### The minimax game

Let $G$ map noise $z \sim p(z)$ to samples $G(z)$, whose distribution we call $p_g$. Let $D(x) \in (0,1)$ be the discriminator's estimated probability that $x$ is real. The original GAN objective of Goodfellow et al. [11] is

$$\min_G \max_D \; V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))].$$

The discriminator maximizes $V$: it is ordinary binary cross-entropy with real data labeled 1 and fakes labeled 0. The generator minimizes $V$: it wants $D(G(z))$ to be large.

### What the game minimizes

For a fixed generator, write $V$ as an integral over $x$: $V = \int [p_{\text{data}}(x)\log D(x) + p_g(x)\log(1 - D(x))]dx$. For each $x$ separately, the function $a\log D + b\log(1-D)$ with $a, b > 0$ is maximized at $D = a/(a+b)$ (set the derivative $a/D - b/(1-D)$ to zero). So the **optimal discriminator** is

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}.$$

It is the probability that a point came from the data rather than the generator, if both were equally likely a priori. Substituting $D^*$ back in and adding and subtracting $\log 4$ gives

$$V(D^*, G) = -\log 4 + 2\,\text{JSD}(p_{\text{data}} \,\|\, p_g),$$

where $\text{JSD}(p\|q) = \frac{1}{2}D_{\text{KL}}(p\|m) + \frac{1}{2}D_{\text{KL}}(q\|m)$ with $m = \frac{1}{2}(p+q)$ is the **Jensen–Shannon divergence**. So, with a perfect discriminator, the generator minimizes a symmetric divergence between data and model, and the global optimum is $p_g = p_{\text{data}}$, where $D^* = 1/2$ everywhere [11]. Notice that no likelihood was ever computed: the discriminator acts as a learned, adaptive test of "does this look like the data?"

### The non-saturating loss and how to implement it

Early in training the generator is poor, $D(G(z)) \approx 0$, and $\log(1 - D(G(z)))$ is flat, so the generator's gradient vanishes. In practice the generator instead minimizes $-\mathbb{E}\log D(G(z))$, the **non-saturating** loss proposed in the same paper [11], which has strong gradients precisely when the generator is losing. It has the same fixed point but different dynamics.

Updates alternate. When updating $D$, **detach** generated samples so no gradient reaches $G$; when updating $G$, let gradients flow through $D$ into $G$ but step only $G$'s optimizer. Using logits with

`binary_cross_entropy_with_logits` is numerically safer than applying a sigmoid and then a log.

### Code: a GAN on eight Gaussians

The classic test bed for mode coverage is a ring of eight tight Gaussian clusters in 2-D. It is small enough to train on a CPU in under a minute, and you can **see** whether every mode is covered.

```
import math
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(0)
torch.set_num_threads(1)
angles = torch.arange(8) * (2 * math.pi / 8)
centers = 2.0 * torch.stack([angles.cos(), angles.sin()], dim=1)

def sample_data(n):
    idx = torch.randint(8, (n,))
    return centers[idx] + 0.05 * torch.randn(n, 2)

def mlp(i, o):
    return nn.Sequential(nn.Linear(i, 128), nn.ReLU(),
                         nn.Linear(128, 128), nn.ReLU(), nn.Linear(128, o))

G, D = mlp(8, 2), mlp(2, 1)
opt_g = torch.optim.Adam(G.parameters(), lr=1e-3, betas=(.5, .999))
opt_d = torch.optim.Adam(D.parameters(), lr=1e-3, betas=(.5, .999))
for step in range(3000):
    real = sample_data(256)
    fake = G(torch.randn(256, 8))
    # Discriminator: logits, real -> 1, fake -> 0; detach so G is untouched.
    d_loss = (F.binary_cross_entropy_with_logits(D(real), torch.ones(256, 1))
              + F.binary_cross_entropy_with_logits(D(fake.detach()),
                                                   torch.zeros(256, 1)))
    opt_d.zero_grad(); d_loss.backward(); opt_d.step()
    # Generator: non-saturating loss -log D(G(z)).
    g_loss = F.binary_cross_entropy_with_logits(D(fake), torch.ones(256, 1))
    opt_g.zero_grad(); g_loss.backward(); opt_g.step()

with torch.no_grad():
    x = G(torch.randn(5000, 8))
    dist = torch.cdist(x, centers)
    nearest = dist.min(1)
    good = nearest.values < 0.15          # within 3 std of some mode
    counts = torch.bincount(nearest.indices[good], minlength=8)
print("fraction near a mode:", round(good.float().mean().item(), 3))
print("modes with >= 1% of samples:", int((counts >= 50).sum()))
print("samples per mode:", counts.tolist())
```

It prints:

```
fraction near a mode: 0.896
modes with >= 1% of samples: 7
samples per mode: [590, 14, 710, 276, 651, 731, 863, 646]
```

About 90% of samples land on a mode, but the mode at 45° receives only 14 of 5,000 samples and another is under-represented. That is **partial mode collapse** caught in the act. Figure 4 shows the generator's journey: an unstructured cloud early on, a ring that covers every mode but also the space between them, and finally sharper clusters with one mode nearly abandoned. Exact numbers depend on the seed and library version; the qualitative picture is typical.

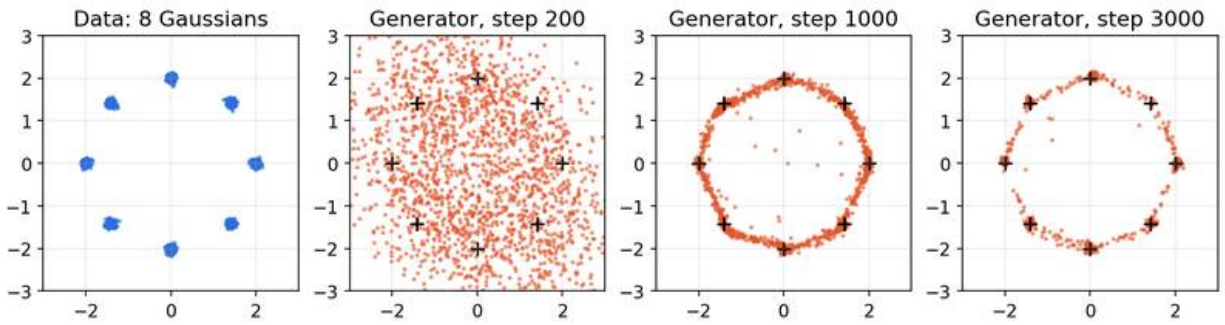



*Figure 4 — A small GAN learning a ring of eight Gaussians (crosses mark the true modes). By step 3000 samples are sharp but one mode is nearly dropped: sharpness and coverage are separate achievements.*

### Mode collapse and why it happens

**Mode collapse** occurs when the generator produces too few kinds of output. The generator is rewarded for samples the current discriminator accepts, not for diversity. If it finds one region the discriminator likes, it can pile samples there; the discriminator then learns to reject that region, the generator hops to another, and the pair can chase each other around instead of converging. Unlike maximum likelihood, nothing in the per-sample generator loss directly punishes **missing** a mode. Salimans et al. [12] proposed early remedies such as feature matching and minibatch discrimination, which let the discriminator look at statistics of a whole batch so that a lack of diversity becomes detectable.

A collection of sharp, convincing images can therefore still cover only a small part of the data distribution. Always measure coverage separately from quality — we return to this under evaluation.

### Wasserstein GANs

When $p_{\text{data}}$ and $p_g$ live on thin, non-overlapping sets (as images do in pixel space), the JSD is stuck at its maximum $\log 2$ and provides no useful gradient about **how far apart** they are. Arjovsky et al. [13] proposed the **Wasserstein-1 (earth mover's) distance** instead: the minimum cost of

moving probability mass from one distribution to the other. Its dual form, from the Kantorovich–Rubinstein theorem, is

$$W(p_{\text{data}}, p_g) = \sup_{\|f\|_L \le 1} \mathbb{E}_{x \sim p_{\text{data}}}[f(x)] - \mathbb{E}_{x \sim p_g}[f(x)],$$

where the supremum is over all 1-Lipschitz functions $f$ (functions whose output changes by at most $\|x - x'\|$ when the input changes from $x'$ to $x$). The network $f$ is now called a **critic**: it outputs an unbounded score, not a probability. The Wasserstein distance shrinks smoothly as the generator's samples move toward the data, even when the supports do not overlap.

The hard part is enforcing the Lipschitz constraint. The original WGAN clips every critic weight to a small interval, which is crude and can make the critic too simple. **WGAN-GP** [14] instead adds a gradient penalty

$$\lambda \, \mathbb{E}_{\hat{x}}[(\|\nabla_{\hat{x}} f(\hat{x})\|_2 - 1)^2],$$

where $\hat{x}$ is drawn uniformly on straight lines between pairs of real and generated samples and $\lambda$ is a weight (the paper uses 10). Two other widely used stabilizers act on the discriminator directly: **spectral normalization** [15] divides each weight matrix by its largest singular value, bounding the Lipschitz constant layer by layer, and the **R1 penalty** analyzed by Mescheder et al. [16] penalizes $\frac{\gamma}{2}\|\nabla_x D(x)\|^2$ on real data, which they show gives local convergence where unregularized GAN training can oscillate.

### Architectures and where GANs are used now

StyleGAN [17] illustrates how architecture matters as much as the loss: it maps noise to an intermediate latent space and injects that "style" at every resolution, giving controllable, high-quality face synthesis. GANs sample with a single generator pass, which makes them fast at inference. Adversarial losses also appear as **components** of other systems — for example, inside image tokenizers trained to reconstruct sharp detail. That does not make every model using such a tokenizer a GAN generator; the distribution may be modeled by something else entirely.

*Online only: GAN — a distribution game and its idealized guarantee — https://blog.shyandram.dev/en/learn/ml/generative-models/papers/gan–a-distribution-game-and-its-idealized-guarantee/*

## Normalizing flows: invertible transformations

**Plain version.** A normalizing flow transforms a simple density through invertible maps. Tracking each map's volume change makes the modeled density computable.

### Change of variables

Let $z \sim p_0(z)$ be a simple base distribution and $x = f_\theta(z)$ an invertible, differentiable map with the same dimension. Then

$$\log p_\theta(x) = \log p_0(f_\theta^{-1}(x)) - \log \left| \det \frac{\partial f_\theta}{\partial z} \right|_{z=f_\theta^{-1}(x)}.$$

$\partial f_\theta / \partial z$ is the Jacobian matrix of the map, and its determinant measures how much the map scales a small volume around $z$. Probability mass is conserved, so if a region is stretched to twice its volume, its density must halve. In one dimension, $x = 2z$ doubles every interval and adds $-\log 2$ to the log density. A composition $f = f_K \circ \cdots \circ f_1$ simply adds up the log-determinants of the steps.

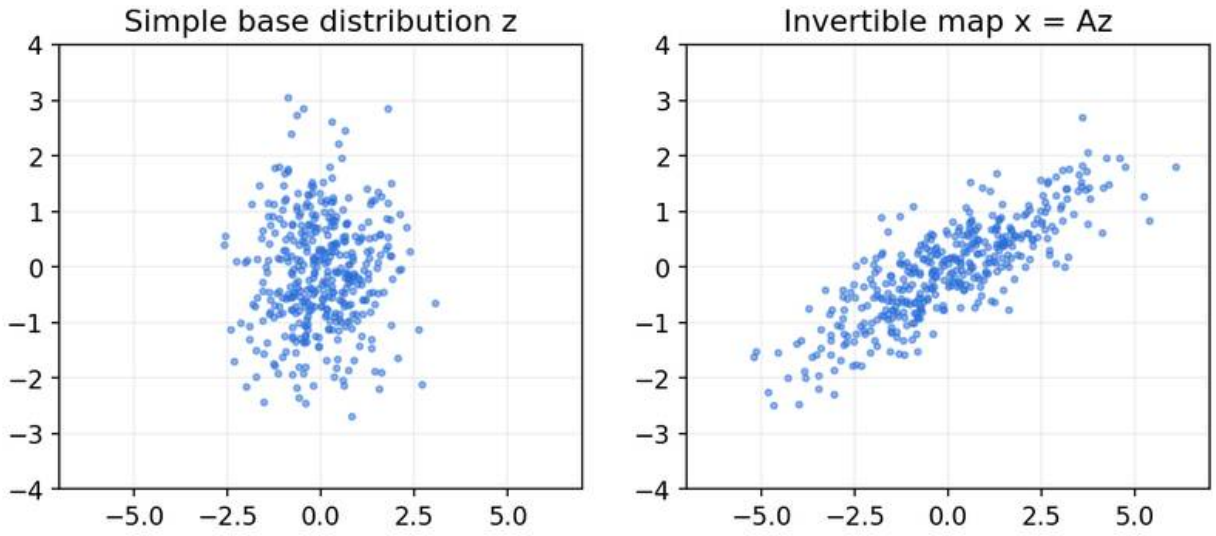



*Figure 5 — Change of variables preserves total probability while changing density. This affine map has an analytically tractable determinant.*

### Coupling layers

The formula is exact, but a general $D \times D$ determinant costs $O(D^3)$ — hopeless for images. Flow layers are designed so that the Jacobian is **triangular**, whose determinant is just the product of the diagonal. The affine coupling layer of RealNVP [18] splits $x$ into two halves $(x_1, x_2)$ and computes

$$y_1 = x_1, \qquad y_2 = x_2 \odot \exp\big(s(x_1)\big) + t(x_1),$$

where $s$ and $t$ are arbitrary neural networks. The Jacobian is block-triangular with the identity on the first block and $\operatorname{diag}(\exp s(x_1))$ on the second, so $\log|\det J| = \sum_j s_j(x_1)$. The inverse is just as cheap: $x_2 = (y_2 - t(y_1)) \odot \exp(-s(y_1))$. Crucially, $s$ and $t$ never need to be inverted, so they can be as complex as you like. Stacking layers with alternating splits lets every coordinate be transformed. Glow [19] added learned invertible $1 \times 1$ convolutions as a generalized channel permutation between coupling layers.

### Code: verify a coupling layer's log-determinant

The snippet below builds one affine coupling layer and checks its cheap log-determinant against the full Jacobian computed by autograd.

```
import torch
from torch import nn

torch.manual_seed(0)

class AffineCoupling(nn.Module):
    """RealNVP-style layer: keep x1, scale-and-shift x2 using x1."""
    def __init__(self, dim, hidden=32):
        super().__init__()
        self.d = dim // 2
        self.net = nn.Sequential(nn.Linear(self.d, hidden), nn.Tanh(),
                                 nn.Linear(hidden, 2 * (dim - self.d)))

    def forward(self, x):
        x1, x2 = x[..., :self.d], x[..., self.d:]
        s, t = self.net(x1).chunk(2, dim=-1)
        s = torch.tanh(s)                       # keep scales moderate
        y2 = x2 * s.exp() + t
        return torch.cat([x1, y2], -1), s.sum(-1)   # output, log|det J|

    def inverse(self, y):
        y1, y2 = y[..., :self.d], y[..., self.d:]
        s, t = self.net(y1).chunk(2, dim=-1)
        s = torch.tanh(s)
        return torch.cat([y1, (y2 - t) * (-s).exp()], -1)

layer = AffineCoupling(4)
x = torch.randn(4)
y, logdet = layer(x)
J = torch.autograd.functional.jacobian(lambda v: layer(v)[0], x)
print("log|det J| from layer:   ", round(logdet.item(), 5))
print("log|det J| from autograd:", round(torch.linalg.slogdet(J)[1].item(), 5))
print("upper-right block is zero:", bool(torch.allclose(J[:2, 2:], torch.zeros(2, 2))))
print("inverse error:", (layer.inverse(y) - x).abs().max().item() < 1e-6)

# Log density of y under a standard-normal base after one layer:
base = torch.distributions.Normal(0., 1.)
z = layer.inverse(y)
log_px = base.log_prob(z).sum() - layer(z)[1]
print("log p(y):", round(log_px.item(), 4))
```

Both log-determinants print 0.14867, the upper-right Jacobian block is exactly zero (the triangular structure), and the inverse recovers the input. The last line is the change-of-variables formula in action. Training a flow simply maximizes this `log p` over data with a stack of such layers.

### Strengths, limits and continuous flows

Flows give exact likelihoods, exact inversion (useful for editing and for mapping data to a well-behaved latent space) and one-pass sampling. The price is architectural: every layer must be invertible with a cheap determinant, and the latent must have the **same dimension** as the data, which wastes capacity on representing noise in all pixel directions. Rezende and Mohamed [20] introduced flows as a way to make the approximate posterior of a VAE more flexible, which shows that the same tool serves inference as well as generation.

Continuous normalizing flows replace a discrete stack by an ordinary differential equation $\mathrm{d}x/\mathrm{d}t = v_\theta(x, t)$. Neural ODEs [21] showed that the log-density then evolves through the **divergence** (trace of the Jacobian) of $v_\theta$ rather than a determinant, which removes the architectural restriction but makes likelihood evaluation require an ODE solve. **Flow matching** [22], covered in ML 08, trains such a vector field by simple regression instead of maximum likelihood. The word "flow" therefore refers to related transformations, but the training objective need not be likelihood.

## Energy-based models

**Plain version.** Instead of describing how to generate an image, give every possible image a score — an "energy" — where low energy means "looks real." Probability is high wherever energy is low.

An energy-based model (EBM) defines

$$p_\theta(x) = \frac{\exp(-E_\theta(x))}{Z_\theta}, \qquad Z_\theta = \int \exp(-E_\theta(x))\, dx,$$

where $E_\theta$ is any network that outputs a scalar and $Z_\theta$ is the **partition function** that makes the density integrate to one. EBMs are extremely flexible — any architecture works — but $Z_\theta$ is intractable in high dimensions, so neither the likelihood nor exact samples are directly available. Goodfellow, Bengio and Courville [23] cover the classical tools (the partition-function chapter and approximate inference), and Song and Kingma [24] survey modern training routes: maximum likelihood with MCMC samples (for example, Langevin dynamics), **score matching**, which fits $\nabla_x \log p_\theta(x) = -\nabla_x E_\theta(x)$ and thereby never needs $Z_\theta$, and noise-contrastive methods. Score matching is the bridge to diffusion models.

## Autoregressive models, briefly

**Plain version.** Generate an image one piece at a time, each piece conditioned on everything generated so far, like writing a sentence word by word.

The chain rule of probability is exact for any ordering of the $D$ elements of $x$:

$$p_\theta(x) = \prod_{i=1}^{D} p_\theta(x_i \mid x_1, \ldots, x_{i-1}).$$

Each factor is an ordinary classification or regression problem, so training is simple, stable maximum likelihood with an exact likelihood. PixelRNN and PixelCNN [25] modeled raw pixels this way with a 256-way softmax per subpixel. The drawback is sequential sampling: $D$ network evaluations for $D$ elements. Modern systems instead predict compressed tokens, groups of tokens or whole scales at once. ML 09 develops this family in detail.

## Diffusion models, briefly

**Plain version.** Teach a network to remove a little noise from a noisy image. Then start from pure noise and apply the denoiser many times, slowly sculpting an image.

A diffusion model defines a fixed forward process that gradually adds Gaussian noise to data until nothing but noise remains, and learns a reverse process that undoes it step by step. Ho et al. [26] showed that training reduces, up to weighting, to predicting the noise that was added — a simple regression — and that the objective is a reweighted variational bound, which is why the taxonomy in Figure 1 lists diffusion next to the VAE. The connection to score matching (see the energy-based models section) gives a second, equivalent view. The survey by Yang et al. [27] maps the many variants. Diffusion and flow matching now dominate high-quality image generation and are the subject of ML 08.

## The family map

| Family | Basic training signal | Sampling mechanism | Likelihood | Main qualification |
|---|---|---|---|---|
| VAE | Variational bound (ELBO) | Sample latent, then decoder (one pass) | Lower bound | Bound depends on approximate inference; samples can be blurry |
| GAN | Adversarial discrimination | Generator pass (one pass) | None | Unstable game; mode collapse |
| Normalizing flow | Change-of-variables likelihood | Invertible transform (one pass) | Exact | Invertibility and same-dimension constraints |
| Autoregressive | Conditional likelihoods | Sequential conditional sampling | Exact | Generation dependencies can be costly |
| Diffusion | Denoising/score objectives | Reverse noise process (many steps) | Bound or ODE-based | Quality depends on model and sampler |
| Flow matching | Velocity regression | Integrate learned vector field | Needs extra computation | Integration cost |
| Energy-based | Energy learning (MCMC, score matching) | Often iterative sampling | Unnormalized | Normalization and mixing can be hard |

Bond-Taylor et al. [3] summarize these trade-offs as a triangle that is hard to satisfy all at once: fast sampling, high sample quality, and good mode coverage. GANs are fast and sharp but can

miss modes; likelihood models cover modes but were historically slower or blurrier; diffusion models reach high quality and coverage at the cost of many sampling steps. This table is a teaching synthesis, not an exclusive taxonomy. Real models combine families: a VAE compresses an image, a transformer models its latent distribution, and a diffusion objective trains that transformer.

## Evaluating generative models

**Plain version.** A generator can be good in different ways: it can assign high probability to real data, make individually convincing samples, cover all the variety in the data, or follow its conditioning. No single number captures all of these, so use several measures and know what each misses.

Theis et al. [5] made this point sharply: average log-likelihood, sample quality and older proxies such as Parzen-window estimates can be **largely independent** of one another in high dimensions. A model can have excellent likelihood and poor samples, or the reverse. Evaluate against the purpose you have in mind.

### Likelihood

For likelihood-based models, held-out NLL (or bits per dimension) is principled and comparable when dequantization and preprocessing match. It cannot be computed for GANs. And a high likelihood need not mean reliable out-of-distribution detection: Nalisnick et al. [28] found that flows, VAEs and PixelCNNs trained on CIFAR-10 can assign **higher** likelihood to the unrelated SVHN digit images, because likelihood is dominated by low-level statistics such as smoothness rather than semantic content.

### Inception Score

Salimans et al. [12] proposed the **Inception Score (IS)**. Feed each generated image $x$ to a pre-trained ImageNet classifier (Inception-v3) to get a label distribution $p(y \mid x)$, and let $p(y) = \mathbb{E}_x[p(y \mid x)]$ be the average over samples. Then

$$\mathrm{IS} = \exp\left(\mathbb{E}_{x \sim p_g} D_{\mathrm{KL}}(p(y \mid x) \,\|\, p(y))\right).$$

IS is high when each image is confidently classified ($p(y \mid x)$ peaked) and the set covers many classes ($p(y)$ spread out). It never looks at real data, so it cannot tell whether samples resemble **your** dataset, and it is meaningful only for ImageNet-like content.

### Fréchet Inception Distance

The **Fréchet Inception Distance (FID)** of Heusel et al. [29] compares real and generated images in a feature space. Extract Inception-v3 features for both sets, fit a Gaussian to each — means $\mu_r$, $\mu_g$ and covariances $\Sigma_r, \Sigma_g$ — and compute the Fréchet distance between the Gaussians:

$$\mathrm{FID} = \|\mu_r - \mu_g\|_2^2 + \mathrm{Tr}\left(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}\right).$$

Lower is better. FID uses real data and is sensitive to both quality and diversity, which made it the standard metric. But it is not an image-by-image truth measure: it depends on the feature extractor, image resizing, and especially the **number of samples** (it is biased for finite sets, so compare FIDs only with equal sample counts), and it folds quality and coverage into one number.

### Precision and recall

To separate the two, Sajjadi et al. [30] defined **precision and recall for distributions**: precision asks what fraction of generated samples fall where real data lives (quality), recall asks what fraction of the real distribution the generator covers (diversity). Kynkäänniemi et al. [31] gave a simple, robust estimator that we implement below. Approximate each set's manifold by a union of balls around its points, with each ball's radius equal to the distance to the point's $k$-th nearest neighbor in the same set. Then

- **precision** = fraction of generated samples inside some real sample's ball;
- **recall** = fraction of real samples inside some generated sample's ball.

On images, the points are deep features rather than pixels. On our 2-D ring we can use coordinates directly:

```
import numpy as np

rng = np.random.default_rng(0)
angles = np.arange(8) * 2 * np.pi / 8
centers = 2 * np.stack([np.cos(angles), np.sin(angles)], 1)

def ring(n, modes):
    idx = rng.choice(modes, n)
    return centers[idx] + .05 * rng.standard_normal((n, 2))

def knn_radius(a, k):
    d = np.linalg.norm(a[:, None] - a[None], axis=-1)
    return np.sort(d, axis=1)[:, k]          # column 0 is the point itself

def coverage(query, ref, k=3):
    """Fraction of query points inside some reference point's k-NN ball."""
    r = knn_radius(ref, k)
    d = np.linalg.norm(query[:, None] - ref[None], axis=-1)
    return np.mean((d <= r[None]).any(axis=1))

real = ring(1000, range(8))
fakes = {
    "all 8 modes": ring(1000, range(8)),
    "only 2 modes": ring(1000, [0, 4]),
    "8 modes, blurry": ring(1000, range(8)) + .4 * rng.standard_normal((1000, 2)),
}
for name, fake in fakes.items():
    precision = coverage(fake, real)   # are fake samples on the real manifold?
```

```
    recall = coverage(real, fake)      # is the real manifold covered by fakes?
    print(f"{name:16s} precision={precision:.2f} recall={recall:.2f}")
```

```
all 8 modes      precision=0.97 recall=0.96
only 2 modes     precision=0.97 recall=0.26
8 modes, blurry  precision=0.07 recall=0.99
```

The mode-collapsed generator keeps high precision — each of its samples is perfect — but its recall drops to about 2/8. The blurry generator is the mirror image: it covers every mode but most samples are off the data. A single scalar such as FID would rank both as "worse" without saying **why**. Figure 6 shows the three cases.

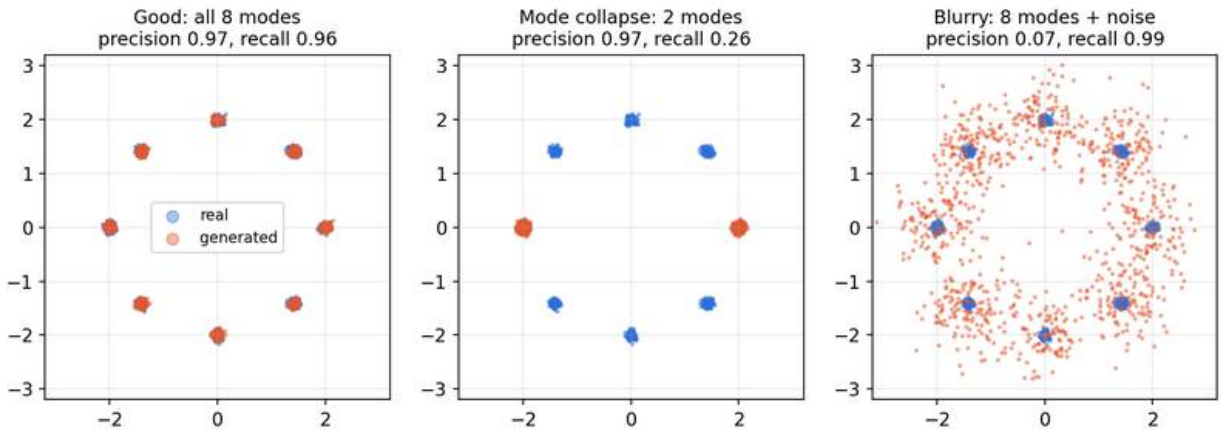



*Figure 6 — k-nearest-neighbor precision and recall on a toy ring: mode collapse lowers recall, blur lowers precision.*

### Known limitations of current metrics

Evaluation is itself an active research area. Stein et al. [32] ran large human studies and found that no existing automatic metric strongly correlates with human judgments of realism; in particular, FID computed on Inception features under-rates the perceptual realism of diffusion models. They recommend richer self-supervised feature spaces (DINOv2 ViT-L/14) for metrics like FID and precision/recall. Jayasumana et al. [33] argue that FID's normality assumption and Inception features are poor fits for modern text-to-image models and propose **CMMD**, a maximum mean discrepancy computed on CLIP embeddings, which needs no Gaussian assumption. Practical advice: report several complementary metrics with their exact protocol (feature extractor, sample count, resolution, reference set), show uncurated samples, and run a human study when the claim is about perceived quality.

For conditional tasks, add a measure of **alignment** (does the image match the text or the low-resolution input?). For restoration, a beautiful generated texture may be unsupported by the

observation; the distinction between plausible samples and measured evidence is central in DLCV 09.

## Modern view

**Surveys worth reading.** Four reviews cover this chapter's ground from different angles.

- **Bond-Taylor et al. [3]** (IEEE TPAMI 2022) is the best single comparative review of VAEs, GANs, normalizing flows, energy-based and autoregressive models. It organizes each family by its core technique and its known pathologies, and compares them along training speed, sampling speed, sample quality, diversity and likelihood, concluding that each family pays for its strengths with a weakness somewhere else, and that hybrids are the natural way forward.
- **Papamakarios et al. [4]** (JMLR 2021) is a unified treatment of normalizing flows: their expressive power, the design space of coupling, autoregressive, residual and continuous flows, and their uses for density estimation, inference and simulation.
- **Kingma and Welling [9]** (Foundations and Trends in ML 2019) is the authoritative tutorial-review of VAEs by their inventors: the derivation, the reparameterization estimator, tighter bounds, more flexible posteriors (including flows), hierarchical models and the β-weighted variants.
- **Yang et al. [27]** (ACM Computing Surveys 2023) surveys diffusion models — sampling acceleration, likelihood improvements, structured data, and the links to VAEs, GANs, flows and EBMs — and is the bridge to the next chapter.

The textbooks of Bishop and Bishop [1] and Prince [2] give coherent derivations of all the families, with Prince's notebooks providing runnable companions.

**Where the field is (2023–2026).** Diffusion and flow-matching models, usually operating in a learned latent space, now produce most state-of-the-art image and video generation. That makes the **autoencoder** a load-bearing component again: VQ-VAE [34] showed how to learn discrete latent codes that a separate prior can model, and latent diffusion [35] showed that running the generative model inside a compressed autoencoder latent space cuts cost dramatically while keeping detail. A tokenizer must balance compression, reconstruction and how easy its latent space is to model; evaluate its reconstruction first, because no downstream prior can restore detail the encoder discarded.

GANs have not disappeared. GigaGAN [36] scaled a GAN to text-to-image synthesis and emphasized what GANs still do best: single-pass sampling fast enough for interactive use, high-resolution upsampling, and a smooth, editable latent space. Huang et al. [37] revisited the training objective itself, deriving a regularized relativistic GAN loss with local convergence guarantees and showing that a minimalist modern baseline (R3GAN) without the traditional bag of tricks can surpass StyleGAN2 on standard benchmarks. Adversarial losses also live on inside other systems, for example to sharpen tokenizer reconstructions.

On evaluation, the 2023–2024 critiques of FID [32], [33] have moved the field toward richer feature extractors, separate precision/recall-style measures and human studies, but there is still no accepted single metric.

**Open problems.** (1) Reliable evaluation that agrees with humans and separates quality, diversity and memorization. (2) Detecting when a model reproduces training images rather than generalizing. (3) Calibrated likelihoods that are actually useful for out-of-distribution detection [28]. (4) The speed–quality–coverage trade-off: getting diffusion-level quality with GAN-level speed without losing modes. (5) Learning latent spaces that are simultaneously compressive, faithful and easy to model.

## Key takeaways

- A generator needs a defined distribution or sampling procedure; reconstruction alone does not supply one.
- Maximum likelihood fits observed data probabilities; high likelihood and convincing samples are different goals.
- The ELBO lower-bounds log likelihood, with a gap determined by approximate-posterior mismatch.
- Adversarial objectives depend on discriminator behavior; training improvements do not guarantee convergence or prevent mode collapse.
- Invertible flows track density through Jacobian determinants; tractable transformations constrain their design.
- Report complementary metrics with their feature extractors, sample counts and conditioning protocols.

## Exercises

1. **KL sanity check.** Substitute $\mu = 0$ and $\sigma^2 = 1$ into the Gaussian KL. Then compute the KL for $\mu = 1, \sigma^2 = 1$ in one dimension.

   **Hint**

   For $\mu = 0, \sigma^2 = 1$ every term is zero, because posterior and prior are the same distribution. For $\mu = 1$, $\sigma^2 = 1$ you get $\frac{1}{2}(1 + 1 - 0 - 1) = 0.5$ nats: shifting the mean by one standard deviation costs half a nat.

2. **Loss scaling.** Average the VAE's reconstruction across pixels rather than summing. What happens if KL is unchanged?

   **Hint**

   Reconstruction becomes 64 times smaller relative to KL. This is equivalent to a β-VAE with $\beta = 64$ on the summed loss: it effectively changes regularization (often toward posterior collapse), rather than merely changing reported units.

3. **Optimal discriminator.** At a point $x$ where $p_{\text{data}}(x) = 0.3$ and $p_g(x) = 0.1$, what is $D^*(x)$? What is $D^*$ everywhere when $p_g = p_{\text{data}}$, and what is the value of the game then?

**Hint**

$D^*(x) = 0.3/(0.3 + 0.1) = 0.75$. When the distributions match, $D^* = 1/2$ everywhere and $V = \log \frac{1}{2} + \log \frac{1}{2} = -\log 4$, consistent with JSD $= 0$.

4. **Affine flow.** For $x = Az + b$ with $z \sim \mathcal{N}(0, I)$, derive the density correction and explain why singular $A$ invalidates it. Check your answer against the known Gaussian $\mathcal{N}(b, AA^\top)$.

**Hint**

Subtract $\log|\det A|$. Since $\det(AA^\top)^{1/2} = |\det A|$, this matches the normalizing constant of $\mathcal{N}(b, AA^\top)$. A singular map is not invertible and collapses volume, so the same-dimensional change-of-variables formula does not apply.

5. **Spot collapse.** Propose different checks for VAE posterior collapse and GAN mode collapse. Then modify the GAN code to use WGAN-GP (critic without sigmoid, gradient penalty with $\lambda = 10$, several critic steps per generator step) and compare the per-mode counts.

**Hint**

For a VAE, inspect the KL per latent dimension and whether decoded outputs change when you vary $z$. For a GAN, count samples per mode (as in our code), compute recall, and look for repeated samples at fixed conditions. They are different failures. For WGAN-GP, sample $\hat{x} = \alpha x_{\text{real}} + (1 - \alpha)x_{\text{fake}}$ with $\alpha \sim U(0, 1)$, call `torch.autograd.grad` with `create_graph=True`, and lower the learning rate if training oscillates. Compare across several seeds before concluding anything.

6. **Metric disagreement.** Using the precision/recall code, construct a fake set that has roughly the same precision and recall as the good generator but places half its samples in only four modes. What does this tell you about the limits of the metric?

**Hint**

Unequal mode weights can still give high recall, because recall only asks whether each real point is near **some** fake point, not whether mode frequencies match. Precision/recall measure support overlap, not density; pair them with a distribution-level measure such as FID or a per-mode histogram.

## References


1. C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts*, Springer, 2024. Book
2. S. J. D. Prince, *Understanding Deep Learning*, MIT Press, 2023. Book and notebooks
3. S. Bond-Taylor, A. Leach, Y. Long and C. G. Willcocks, "Deep Generative Modelling: A Comparative Review of VAEs, GANs, Normalizing Flows, Energy-Based and Autoregressive Models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022. Paper
4. G. Papamakarios et al., "Normalizing Flows for Probabilistic Modeling and Inference," *JMLR*, 2021. Review

5. L. Theis, A. van den Oord and M. Bethge, “A note on the evaluation of generative models,” ICLR, 2016. Paper
6. D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” ICLR, 2014. Paper
7. S. R. Bowman et al., “Generating Sentences from a Continuous Space,” CoNLL, 2016. Paper
8. I. Higgins et al., “β-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework,” ICLR, 2017. Paper
9. D. P. Kingma and M. Welling, “An Introduction to Variational Autoencoders,” *Foundations and Trends in Machine Learning*, 2019. Review
10. A. Vahdat and J. Kautz, “NVAE: A Deep Hierarchical Variational Autoencoder,” NeurIPS, 2020. Paper
11. I. Goodfellow et al., “Generative Adversarial Nets,” NeurIPS, 2014. Paper
12. T. Salimans et al., “Improved Techniques for Training GANs,” NeurIPS, 2016. Paper
13. M. Arjovsky, S. Chintala and L. Bottou, “Wasserstein GAN,” ICML, 2017. Paper
14. I. Gulrajani et al., “Improved Training of Wasserstein GANs,” NeurIPS, 2017. Paper
15. T. Miyato et al., “Spectral Normalization for Generative Adversarial Networks,” ICLR, 2018. Paper
16. L. Mescheder, A. Geiger and S. Nowozin, “Which Training Methods for GANs do actually Converge?” ICML, 2018. Paper
17. T. Karras, S. Laine and T. Aila, “A Style-Based Generator Architecture for Generative Adversarial Networks,” CVPR, 2019. Paper
18. L. Dinh, J. Sohl-Dickstein and S. Bengio, “Density estimation using Real NVP,” ICLR, 2017. Paper
19. D. P. Kingma and P. Dhariwal, “Glow: Generative Flow with Invertible 1x1 Convolutions,” NeurIPS, 2018. Paper
20. D. J. Rezende and S. Mohamed, “Variational Inference with Normalizing Flows,” ICML, 2015. Paper
21. R. T. Q. Chen, Y. Rubanova, J. Bettencourt and D. Duvenaud, “Neural Ordinary Differential Equations,” NeurIPS, 2018. Paper
22. Y. Lipman et al., “Flow Matching for Generative Modeling,” ICLR, 2023. Paper
23. I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016, ch. 18–20. Book
24. Y. Song and D. P. Kingma, “How to Train Your Energy-Based Models,” arXiv, 2021. Paper
25. A. van den Oord, N. Kalchbrenner and K. Kavukcuoglu, “Pixel Recurrent Neural Networks,” ICML, 2016. Paper
26. J. Ho, A. Jain and P. Abbeel, “Denoising Diffusion Probabilistic Models,” NeurIPS, 2020. Paper
27. L. Yang et al., “Diffusion Models: A Comprehensive Survey of Methods and Applications,” *ACM Computing Surveys*, 2023. Survey
28. E. Nalisnick et al., “Do Deep Generative Models Know What They Don’t Know?” ICLR, 2019. Paper
29. M. Heusel et al., “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium,” NeurIPS, 2017. Paper
30. M. S. M. Sajjadi, O. Bachem, M. Lucic, O. Bousquet and S. Gelly, “Assessing Generative Models via Precision and Recall,” NeurIPS, 2018. Paper
31. T. Kynkäänniemi, T. Karras, S. Laine, J. Lehtinen and T. Aila, “Improved Precision and Recall Metric for Assessing Generative Models,” NeurIPS, 2019. Paper

32. G. Stein et al., “Exposing flaws of generative model evaluation metrics and their unfair treatment of diffusion models,” NeurIPS, 2023. Paper
33. S. Jayasumana et al., “Rethinking FID: Towards a Better Evaluation Metric for Image Generation,” CVPR, 2024. Paper
34. A. van den Oord, O. Vinyals and K. Kavukcuoglu, “Neural Discrete Representation Learning,” NeurIPS, 2017. Paper
35. R. Rombach et al., “High-Resolution Image Synthesis with Latent Diffusion Models,” CVPR, 2022. Paper
36. M. Kang et al., “Scaling up GANs for Text-to-Image Synthesis,” CVPR, 2023. Paper
37. Y. Huang, A. Gokaslan, V. Kuleshov and J. Tompkin, “The GAN is dead; long live the GAN! A Modern GAN Baseline,” NeurIPS, 2024. Paper

# ML 08 • Diffusion Models and Flow Matching

Derive noise, clean-image and velocity prediction, compare pixel and latent generation with JiT and representation autoencoders, and understand pretrained image/video diffusion and flow pipelines.

Online edition: https://blog.shyandram.dev/en/learn/ml/diffusion-flow-matching/

## What you'll learn

- Build a forward noise process and a reverse prediction task.
- Relate variational training to noise, clean-image and velocity targets.
- Separate model predictions from numerical sampling methods.
- Compare conditioning, generative representations and flow matching.
- Evaluate complete image/video pipelines and few-step methods.

## The big picture

A diffusion model learns to undo controlled corruption. During training we take a real example, add a known amount of Gaussian noise, and ask a network to tell us what noise we added. Because we added the noise ourselves, the correct answer is always available: no labels, no adversary, just regression. During generation we start from pure noise and apply the learned "denoise a little" step many times until an example appears.

> **Explain like I'm five**
>
> Imagine slowly covering a drawing with pencil scribbles. You keep copies of the drawing at many scribble levels: a few lines, lots of lines, so many that the drawing is gone. A model practices one small job: look at a scribbled page, and guess which scribbles were added. After lots of practice you hand it a page that is *only* scribbles. It erases a little, looks again, erases a little more, and a brand-new drawing slowly appears — one that was never in the pile.

Why does this matter? Since 2021, diffusion-style models have produced many of the state-of-the-art results in image, video and audio synthesis, and the same machinery is used for inpainting, super-resolution, image editing, 3-D generation and scientific data. Diffusion also replaced two pain points of earlier families covered in ML 07: GANs need a fragile two-player game, and VAEs tend to produce blurry samples. Diffusion training is a stable regression, and its samples are sharp because generation is spread over many small, easy steps.

The price is sampling cost: a classic sampler calls the network hundreds of times per image. Much of the research since 2021 — DDIM, fast ODE solvers, distillation, consistency models, flow matching, rectified flow — is about keeping the quality while cutting the number of network calls.

The chapter follows the organization of the major surveys. Yang et al. [1] group the field into efficient sampling, improved likelihood and data with special structure; Croitoru et al. [27] group it by formulation (DDPM, noise-conditioned score networks, SDEs); Lipman et al.'s flow-matching guide [2] and the monograph by Lai et al. [29] present diffusion and flows as one family of learned transport. Prince's textbook [14] is a gentle companion for the derivations. Throughout, we use **one explicit time convention per section** and say when it changes — mixing conventions is the most common source of sign errors in this area.

## Forward diffusion has a closed form

**Plain version.** The forward process gradually weakens the image signal and adds Gaussian noise. A suitable terminal schedule makes the distribution approximately standard Gaussian.

The idea goes back to Sohl-Dickstein et al. [15], who proposed destroying structure with a slow forward diffusion and learning to reverse it; DDPM [3] made it practical. Let $x_0 \in \mathbb{R}^d$ be a data example (an image flattened into $d$ numbers, scaled to roughly $[-1, 1]$). Pick a **noise schedule** $\beta_1, \dots, \beta_T$ with $0 < \beta_t < 1$ and write $\alpha_t = 1 - \beta_t$. The forward (noising) process is a Markov chain

$$q(x_t \mid x_{t-1}) = \mathcal{N}\big(\sqrt{\alpha_t}\, x_{t-1},\ \beta_t I\big), \qquad t = 1, \dots, T,$$

where $\mathcal{N}(\mu, \Sigma)$ is a Gaussian with mean $\mu$ and covariance $\Sigma$, and $I$ is the $d \times d$ identity. The factor $\sqrt{\alpha_t}$ shrinks the signal slightly so that the total variance stays bounded instead of growing; this is why this family is called **variance preserving (VP)**.

### Deriving the one-shot formula

Write one step as $x_t = \sqrt{\alpha_t} x_{t-1} + \sqrt{1-\alpha_t}\,\epsilon_t$ with $\epsilon_t \sim \mathcal{N}(0, I)$. Substituting the previous step,

$$\begin{aligned} x_t &= \sqrt{\alpha_t}\left(\sqrt{\alpha_{t-1}} x_{t-2} + \sqrt{1-\alpha_{t-1}}\,\epsilon_{t-1}\right) + \sqrt{1-\alpha_t}\,\epsilon_t \\ &= \sqrt{\alpha_t \alpha_{t-1}}\, x_{t-2} + \underbrace{\sqrt{\alpha_t(1-\alpha_{t-1})}\,\epsilon_{t-1} + \sqrt{1-\alpha_t}\,\epsilon_t}_{\text{sum of independent Gaussians}}. \end{aligned}$$

The two noise terms are independent zero-mean Gaussians, so their sum is Gaussian with variance $\alpha_t(1-\alpha_{t-1}) + (1-\alpha_t) = 1 - \alpha_t\alpha_{t-1}$. Repeating the argument down to $x_0$ gives, with $\bar\alpha_t = \prod_{s=1}^{t} \alpha_s$,

$$q(x_t \mid x_0) = \mathcal{N}\big(\sqrt{\bar\alpha_t}\, x_0,\ (1-\bar\alpha_t) I\big) \quad \Longleftrightarrow \quad x_t = \sqrt{\bar\alpha_t}\, x_0 + \sqrt{1-\bar\alpha_t}\,\epsilon, \qquad \epsilon \sim \mathcal{N}(0, I).$$

Here $\bar\alpha_t$ is the fraction of signal variance that survives to step $t$, and $\epsilon$ is a single standard Gaussian vector. This formula is what makes training cheap: we can jump straight to any noise level without simulating the earlier steps.

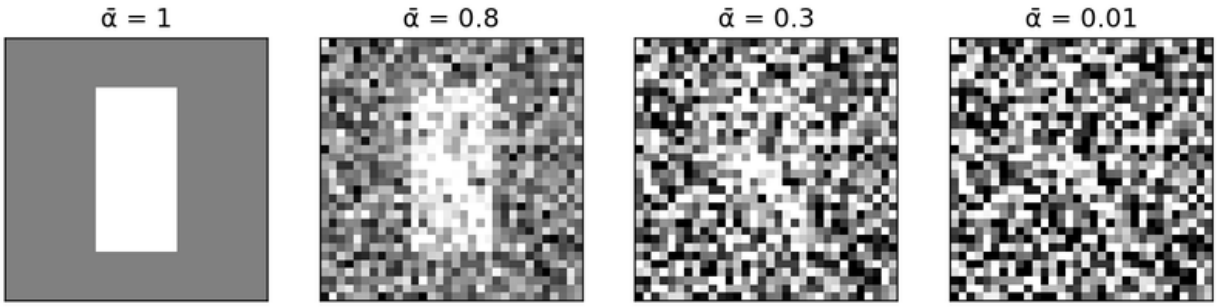



*Figure 1 — The closed-form forward process at several signal levels. A shared noise realization makes the effect visible; training draws fresh noise for every example and every step.*

### Noise schedules and the signal-to-noise ratio

The **signal-to-noise ratio** at step $t$ is

$$\mathrm{SNR}(t) = \frac{\bar{\alpha}_t}{1 - \bar{\alpha}_t},$$

the ratio of signal variance to noise variance (for unit-variance data). It decreases monotonically from very large at $t = 1$ to nearly zero at $t = T$. Two requirements constrain the schedule: $\bar{\alpha}_T$ must be close to 0, so that $x_T$ is practically a standard Gaussian (the sampler starts from $\mathcal{N}(0, I)$); and the steps in between should spend the training budget on noise levels where the network has something useful to learn.

DDPM [3] used $T = 1000$ and a **linear schedule** $\beta_t$ from $10^{-4}$ to 0.02. Nichol and Dhariwal [19] observed that this destroys information quickly: for a large part of the chain the image is already nearly pure noise. Their **cosine schedule** defines $\bar{\alpha}_t = f(t)/f(0)$ with $f(t) = \cos^2\!\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right)$ and a small offset $s = 0.008$, and clips each $\beta_t$ at 0.999. Figure 2 compares the two. Karras et al. [20] later argued that the schedule, the network's input/output scaling and the distribution of training noise levels are separate design choices that are best tuned independently.

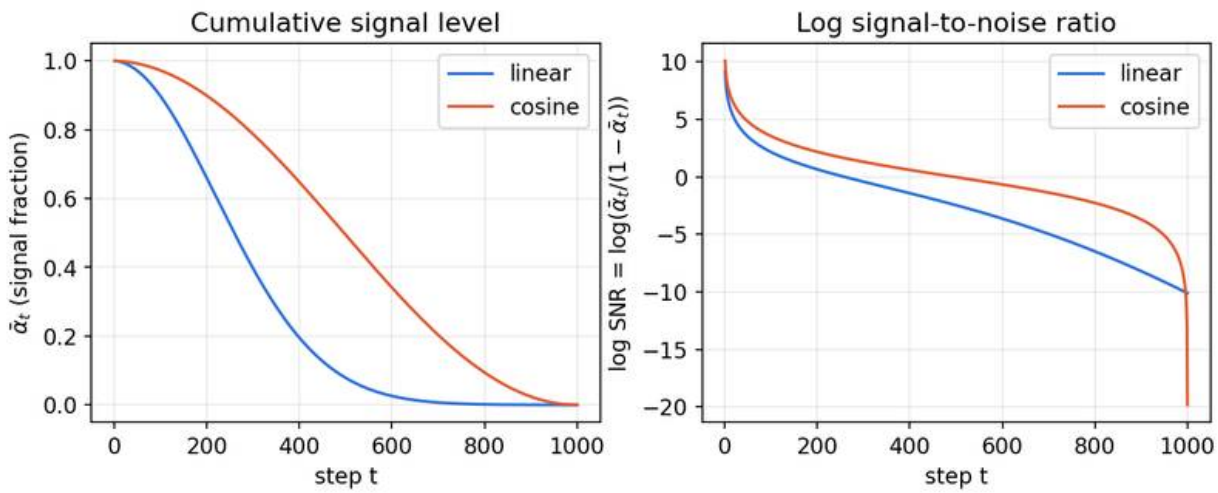



*Figure 2 — Linear versus cosine schedule for $T = 1000$. The cosine schedule keeps more signal for longer and spreads the log-SNR more evenly over the steps.*

## The reverse process and the variational bound

**Plain version.** To generate, we need the reverse step: "given a noisy example, what did the slightly less noisy example look like?" We cannot compute this directly for real data, but we *can* compute it if we also know the clean example — and that turns out to give a simple training loss.

### The model and the bound

The generative model is a reverse Markov chain that starts at $p(x_T) = \mathcal{N}(0, I)$ and uses learned Gaussian transitions

$$p_\theta(x_{t-1} \mid x_t) = \mathcal{N}\big(\mu_\theta(x_t, t),\ \sigma_t^2 I\big),$$

where $\mu_\theta$ is a neural network (with parameters $\theta$) and $\sigma_t^2$ is a variance we will choose. This is a latent-variable model with latents $x_{1:T}$, exactly like a deep VAE whose encoder $q$ is fixed rather than learned (see ML 07). The same Jensen-inequality argument gives an evidence lower bound. Following [3], [15], the negative bound can be rearranged into a sum of KL divergences:

$$-\log p_\theta(x_0) \;\le\; \mathbb{E}_q\Big[\underbrace{D_{\mathrm{KL}}(q(x_T \mid x_0)\,\|\,p(x_T))}_{L_T} + \sum_{t=2}^{T} \underbrace{D_{\mathrm{KL}}(q(x_{t-1} \mid x_t, x_0)\,\|\,p_\theta(x_{t-1} \mid x_t))}_{L_{t-1}} \underbrace{-\log p_\theta(x_0 \mid x_1)}_{L_0}\Big].$$

$D_{\mathrm{KL}}$ is the Kullback–Leibler divergence. $L_T$ has no learnable parameters (it is small when $\bar{\alpha}_T \approx 0$); $L_0$ is a decoder term for the last step; and each $L_{t-1}$ compares the model's reverse step with the **forward posterior** $q(x_{t-1} \mid x_t, x_0)$ — the reverse step that would be exact if the clean example $x_0$ were known.

### The forward posterior is Gaussian

By Bayes' rule, $q(x_{t-1} \mid x_t, x_0) \propto q(x_t \mid x_{t-1})\, q(x_{t-1} \mid x_0)$. Both factors are Gaussian in $x_{t-1}$, so their product is Gaussian. Collecting the quadratic and linear terms in $x_{t-1}$ (completing the square) gives

$$q(x_{t-1} \mid x_t, x_0) = \mathcal{N}\big(\tilde{\mu}_t(x_t, x_0),\ \tilde{\beta}_t I\big),$$

$$\tilde{\mu}_t = \frac{\sqrt{\bar{\alpha}_{t-1}}\,\beta_t}{1-\bar{\alpha}_t}\, x_0 + \frac{\sqrt{\alpha_t}\,(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\, x_t, \qquad \tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\,\beta_t.$$

The posterior mean is a weighted average of the clean example and the current noisy one; the posterior variance $\tilde{\beta}_t$ is slightly smaller than $\beta_t$. You do not have to trust the algebra: the snippet below samples the forward chain and fits a regression of $x_{t-1}$ on $x_t$.

```
import numpy as np

rng = np.random.default_rng(0)
beta = np.linspace(1e-4, 0.02, 1000)
alpha = 1 - beta
abar = np.cumprod(alpha)
t, x0, n = 300, 1.3, 400_000           # check q(x_{t-1} | x_t, x_0) at step t

# Sample the forward chain: x_{t-1} from q(x_{t-1}|x_0), then one more step.
x_prev = np.sqrt(abar[t-1])*x0 + np.sqrt(1-abar[t-1])*rng.normal(size=n)
x_t = np.sqrt(alpha[t])*x_prev + np.sqrt(beta[t])*rng.normal(size=n)

# Empirical regression of x_{t-1} on x_t (the posterior mean is linear in x_t).
slope, intercept = np.polyfit(x_t, x_prev, 1)
resid_var = np.var(x_prev - (slope*x_t + intercept))

# Closed-form posterior from the derivation.
c_x0 = np.sqrt(abar[t-1])*beta[t] / (1-abar[t])
c_xt = np.sqrt(alpha[t])*(1-abar[t-1]) / (1-abar[t])
var = beta[t]*(1-abar[t-1]) / (1-abar[t])
print(f"slope     {slope:.4f}  vs  {c_xt:.4f}")
print(f"intercept {intercept:.4f}  vs  {c_x0*x0:.4f}")
print(f"variance  {resid_var:.6f}  vs  {var:.6f}")
```

The output matches to three or four decimals (slope 0.9934 vs 0.9930, variance 0.006050 vs 0.006052); the small gaps are Monte Carlo error.

### From the bound to the simplified loss

Now use the forward formula backwards: $x_0 = (x_t - \sqrt{1-\bar{\alpha}_t}\,\epsilon)/\sqrt{\bar{\alpha}_t}$. Substituting into $\tilde{\mu}_t$ and simplifying with $\bar{\alpha}_t = \alpha_t \bar{\alpha}_{t-1}$,

$$\tilde{\mu}_t = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\,\epsilon\right).$$

At sampling time the network sees $x_t$ but not $\epsilon$. So DDPM parameterizes the model mean in exactly the same form, with the unknown noise replaced by a prediction $\epsilon_\theta(x_t, t)$:

$$\mu_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}} \left( x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right).$$

The KL between two Gaussians with the same fixed covariance $\sigma_t^2 I$ is a scaled squared distance between their means, so

$$L_{t-1} = \frac{1}{2\sigma_t^2} \|\tilde{\mu}_t - \mu_\theta\|^2 + C = \frac{\beta_t^2}{2\sigma_t^2 \, \alpha_t \, (1-\bar{\alpha}_t)} \, \|\epsilon - \epsilon_\theta(x_t, t)\|^2 + C,$$

where $C$ does not depend on $\theta$. Every term of the bound is therefore a **weighted noise-prediction error**. Ho et al. [3] found that dropping the weight, and sampling $t$ uniformly, trains better:

$$\mathcal{L}_{\text{simple}} = \mathbb{E}_{x_0,\, t\sim \mathcal{U}\{1,\dots,T\},\, \epsilon\sim\mathcal{N}(0,I)} \left\| \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\, \epsilon,\ t) \right\|^2.$$

Read it as an algorithm: pick a training example, pick a random step, pick random noise, build $x_t$ in one line, and regress the noise. The time $t$ is fed to the network (usually through a sinusoidal embedding) because the same pixel values mean different things at different noise levels.

Two points are easy to get wrong. First, $\mathcal{L}_{\text{simple}}$ is *not* the likelihood bound: removing the weights relatively down-weights the low-noise steps, which helps sample quality but hurts the reported log-likelihood. Improved DDPM [19] recovered good likelihoods by learning the variances $\sigma_t^2$ (as an interpolation between $\beta_t$ and $\tilde{\beta}_t$) with a hybrid loss. Second, the same network also yields a **clean estimate**,

$$\hat{x}_0(x_t, t) = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\, \epsilon_\theta(x_t, t)}{\sqrt{\bar{\alpha}_t}},$$

which every modern sampler uses internally. When $\bar{\alpha}_t$ is tiny, the division amplifies prediction errors, so $\hat{x}_0$ at very high noise is a blurry average, not a sample.

*Online only: DDPM — noise prediction and the quality–likelihood tradeoff — https://blog.shyandram.dev/en/learn/ml/diffusion-flow-matching/papers/ddpm–noise-prediction-and-the-qualitylikelihood-tradeoff/*

## Noise, clean data, velocity and the score

**Plain version.** Predicting the noise, predicting the clean image and predicting "which direction makes this example more typical" are three ways to express the same knowledge. They differ in scaling, and scaling changes how the training loss weights different noise levels.

### Three parameterizations

Because $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$ is a fixed linear relation, a network can output any one of

- the noise $\epsilon$ (DDPM [3]),
- the clean example $x_0$,
- the **velocity** $v = \sqrt{\bar{\alpha}_t}\,\epsilon - \sqrt{1-\bar{\alpha}_t}\,x_0$, introduced by Salimans and Ho [22] for distillation, which stays well scaled at both ends of the schedule.

Given $x_t$ and $t$, each converts to the others exactly. However, an unweighted MSE on $\epsilon$, on $x_0$ or on $v$ weights the noise levels differently, so the choice affects training. Karras et al. [20] made this explicit with input/output "preconditioning" coefficients that keep the network's inputs and targets at unit variance for every noise level. Note that this diffusion "velocity" is a reparameterization of signal and noise; it is related to, but not identical with, the transport velocity of flow matching later in this chapter.

### Exact conversion does not make finite networks equivalent

For the variance-preserving path, write $a = \sqrt{\bar{\alpha}_t}$ and $b = \sqrt{1-\bar{\alpha}_t}$. Thus $a^2+b^2 = 1$ and $x_t = ax_0+b\epsilon$. The diffusion velocity $v_{\mathrm{VP}} = a\epsilon - bx_0$ completes a rotation:

$$x_0 = ax_t - bv_{\mathrm{VP}}, \qquad \epsilon = bx_t + av_{\mathrm{VP}},$$
$$\hat{x}_0^{(\epsilon)} = (x_t - b\hat{\epsilon})/a, \quad \hat{\epsilon}^{(x)} = (x_t - a\hat{x}_0)/b.$$

Here $a, b$ are known schedule coefficients; $v_{\mathrm{VP}}$ is the target used in [22]; hats denote estimates; and superscripts identify the network's direct output. The last two conversions require nonzero divisors. Their instability near an endpoint is a numerical and statistical issue, even though the algebra is exact in the interior.

A noise error $\delta_\epsilon$ becomes a clean-image error $-(b/a)\delta_\epsilon$. Therefore, for corresponding predictions at a fixed noise level,

$$\|\hat{x}_0 - x_0\|^2 = \frac{b^2}{a^2}\|\hat{\epsilon} - \epsilon\|^2 = \frac{1}{\mathrm{SNR}(t)}\|\hat{\epsilon} - \epsilon\|^2, \qquad \|\hat{x}_0^{(v)} - x_0\|^2 = b^2\|\hat{v}_{\mathrm{VP}} - v_{\mathrm{VP}}\|^2.$$

$\mathrm{SNR}(t) = a^2/b^2$ is the signal-to-noise ratio; the second identity follows from the rotation above. Equal unweighted losses consequently emphasize different times. Changing the target without recording time sampling and loss weighting confounds an experiment [20], [22].

There is also a capacity issue. If a network directly outputs a clean image, its final representation can discard nuisance noise. A network directly outputting noise may need to preserve more coordinates of the noisy input, or use a skip connection that supplies those coordinates. Converting one network's output afterward does not make its architecture identical to a network trained to output another target. This distinction becomes especially relevant for large pixel patches and high-dimensional latents; the JiT discussion below supplies recent evidence [42]. It does not imply that noise prediction generally fails: the earlier DDPM examples and dense image denoisers are counterexamples.

```
import numpy as np

rng = np.random.default_rng(7)
```

```
x0 = rng.normal(size=32)
eps = rng.normal(size=32)
a, b = np.sqrt(.2), np.sqrt(.8)
xt = a*x0 + b*eps
v_vp = a*eps - b*x0
assert np.allclose(a*xt - b*v_vp, x0)
assert np.allclose(b*xt + a*v_vp, eps)

# A deliberately imperfect noise estimate: compare equivalent losses.
eps_hat = eps + .03*rng.normal(size=32)
x_hat = (xt - b*eps_hat)/a
noise_mse = np.mean((eps_hat - eps)**2)
clean_mse = np.mean((x_hat - x0)**2)
assert np.allclose(clean_mse/noise_mse, b*b/(a*a))
print("Clean/noise MSE ratio:", round(clean_mse/noise_mse, 3))

# A straight flow uses a different path and a different velocity.
t = .35
zt = (1-t)*eps + t*x0
u = x0 - eps
x_from_flow = zt + (1-t)*u
u_from_clean = (x0 - zt)/(1-t)
assert np.allclose(x_from_flow, x0)
assert np.allclose(u_from_clean, u)
print("VP velocity equals straight-flow target:", np.allclose(v_vp, u))
```

The example prints a loss ratio of 4.0 and `False` for the two velocity targets. Both use the letter $v$ in parts of the literature; a checkpoint's scheduler must identify the actual path, normalization and time direction.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

### What is a score, and why does denoising estimate it?

The **score** of a density $p$ is $\nabla_x \log p(x)$: a vector field that points toward higher probability. If you know the score, you can sample with **Langevin dynamics**: repeatedly step a little along the score and add a little noise. Song and Ermon [17] trained "noise-conditional score networks" at many noise levels and sampled with annealed Langevin dynamics — a parallel route to diffusion that turned out to be the same idea.

The bridge is **denoising score matching** (Vincent [16]). For Gaussian corruption, the best possible noise predictor (in mean squared error) is a conditional expectation, $\epsilon^*(x_t, t) = \mathbb{E}[\epsilon \mid x_t]$, and **Tweedie's formula** [40] links it to the score of the noisy marginal $p_t$:

$$\nabla_{x_t} \log p_t(x_t) = -\frac{\mathbb{E}[\epsilon \mid x_t]}{\sqrt{1-\bar{\alpha}_t}} \quad \Longrightarrow \quad s_\theta(x_t, t) \approx -\frac{\epsilon_\theta(x_t, t)}{\sqrt{1-\bar{\alpha}_t}},$$

with $s_\theta$ the learned score. The same formula in "clean" units reads $\mathbb{E}[x_0 \mid x_t] = (x_t + (1 - \bar{\alpha}_t)\nabla \log p_t(x_t))/\sqrt{\bar{\alpha}_t}$. Two warnings: this is the score of the *noisy marginal distribution* at level $t$, not the gradient for any single training image; and it holds for the optimal predictor — a trained network approximates it.

You can verify Tweedie's formula where everything is known: one-dimensional Gaussian data, for which the best noise predictor is linear in $x_t$.

```
import numpy as np

rng = np.random.default_rng(0)
s, abar, n = 0.7, 0.4, 500_000     # data x0 ~ N(0, s^2); one noise level
x0 = s*rng.normal(size=n)
eps = rng.normal(size=n)
xt = np.sqrt(abar)*x0 + np.sqrt(1-abar)*eps

# "Train" the best linear noise predictor eps_hat = w * x_t by least squares.
w = (xt @ eps) / (xt @ xt)
score_from_eps = -w / np.sqrt(1-abar)       # s(x_t) = -eps_hat / sqrt(1-abar)

# Exact: x_t ~ N(0, abar s^2 + 1 - abar), so grad log p_t(x) = -x / var.
exact = -1 / (abar*s**2 + 1 - abar)
print(f"score slope from noise predictor: {score_from_eps:.4f}")
print(f"exact score slope:                {exact:.4f}")
```

It prints $-1.2560$ versus $-1.2563$: a regression on noise recovers the score.

### The SDE view

Song et al. [4] took the step count to infinity. With continuous time $t \in [0, 1]$ and a rate $\beta(t)$, the VP forward process becomes the **stochastic differential equation (SDE)**

$$dx = -\tfrac{1}{2}\beta(t)\, x\, dt + \sqrt{\beta(t)}\, dw,$$

where $w$ is a standard Wiener process (Brownian motion). A classical result says that this process run backwards in time is again an SDE, and that the only unknown in it is the score:

$$dx = \Big[ -\tfrac{1}{2}\beta(t)\, x - \beta(t)\, \nabla_x \log p_t(x) \Big] dt + \sqrt{\beta(t)}\, d\bar{w},$$

with $dt$ negative and $\bar{w}$ a reverse-time Wiener process. Moreover, there is a deterministic **probability-flow ODE** whose solutions have the same marginal distributions $p_t$ at every time:

$$\frac{dx}{dt} = -\tfrac{1}{2}\beta(t)\Big[ x + \nabla_x \log p_t(x) \Big].$$

This unifies the field. DDPM's ancestral sampler relates to a discretization of the reverse SDE; NCSN [17] corresponds to a "variance exploding" SDE. The probability-flow ODE defines a continuous normalizing flow whose log-density follows an instantaneous change-of-variables identity [4]. The identity is exact; a computed likelihood still has ODE integration error and, when the

divergence is estimated stochastically, estimator error. With an approximate learned score, the ODE and reverse SDE need not have identical marginals. The ODE perspective also motivates deterministic sampling and general-purpose numerical solvers.

## Sampling is a reverse process

**Plain version.** A loss evaluated at one random noise level is not a generator. To generate, you must chain many reverse steps with a consistent schedule, starting from pure noise — and you can choose how many steps and how much fresh randomness to inject.

### DDPM ancestral sampling

DDPM samples by following the learned reverse chain:

$$x_{t-1} = \mu_\theta(x_t, t) + \sigma_t z, \qquad z \sim \mathcal{N}(0, I), \qquad \sigma_t^2 \in \{\beta_t,\ \tilde{\beta}_t\},$$

with no noise added at the final step. Ho et al. [3] report similar results for $\sigma_t^2 = \beta_t$ and $\sigma_t^2 = \tilde{\beta}_t$; learned variances [19] help most when using fewer steps.

### Code: DDPM on a two-dimensional distribution

This complete CPU example (about 10 seconds) uses two Gaussian clusters centered at $x = \pm 1.5$ with standard deviation 0.25. It contains every operation of an image model — schedule, random-time training, noise prediction, reverse mean — without an image-scale budget. With 100 steps and this schedule, almost no signal survives to $t = T$.

```
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(0)
torch.set_num_threads(1)
T = 100
beta = torch.linspace(.001, .15, T)
alpha = 1-beta
abar = alpha.cumprod(0)
previous = torch.cat([torch.ones(1), abar[:-1]])
post_var = beta * (1-previous) / (1-abar)
net = nn.Sequential(nn.Linear(3, 64), nn.SiLU(),
                    nn.Linear(64, 64), nn.SiLU(), nn.Linear(64, 2))
opt = torch.optim.Adam(net.parameters(), lr=.002)
def data(n):
    side = torch.randint(2, (n, 1))*2-1
    return torch.cat([side*1.5, torch.zeros(n, 1)], 1) + .25*torch.randn(n, 2)
for step in range(3000):
    x0 = data(128)
    t = torch.randint(T, (len(x0),))
```

```
    eps = torch.randn_like(x0)
    a = abar[t, None]
    xt = a.sqrt()*x0 + (1-a).sqrt()*eps
    predicted = net(torch.cat([xt, t[:, None].float()/(T-1)], 1))
    loss = F.mse_loss(predicted, eps)
    opt.zero_grad(); loss.backward(); opt.step()
net.eval()
with torch.no_grad():
    x = torch.randn(512, 2)
    for t in reversed(range(T)):
        time = torch.full((len(x), 1), t/(T-1))
        eps = net(torch.cat([x, time], 1))
        mean = (x-beta[t]/(1-abar[t]).sqrt()*eps)/alpha[t].sqrt()
        x = mean + post_var[t].sqrt()*torch.randn_like(x) if t else mean
    assert torch.isfinite(x).all()
    print("terminal signal:", round(abar[-1].item(), 6))
    left = x[x[:, 0] < 0]
    print("fraction in left cluster:", round((x[:, 0] < 0).float().mean().item(), 3))
    print("left cluster: mean x", round(left[:, 0].mean().item(), 2),
          "std", round(left[:, 0].std().item(), 2), "(data: -1.5 and 0.25)")
```

On our machine it prints a terminal signal of 0.000349, a left-cluster fraction of 0.441, and a left-cluster mean and standard deviation of $-1.47$ and 0.25. The cluster *shapes* are right; the 44/56 split shows the small network has not learned the mode weights exactly. This is why a single summary number is never enough: always plot samples, and check mode coverage, within-mode spread and outliers against fresh data.

### DDIM: same training, different sampler

Song, Meng and Ermon [5] noticed that $\mathcal{L}_{\text{simple}}$ only depends on the marginals $q(x_t \mid x_0)$, not on the Markov chain that produced them. Many non-Markovian forward processes share those marginals, and each gives a different reverse sampler — all usable with the *same* trained $\epsilon_\theta$. The **DDIM** update, for any decreasing subsequence of steps $\tau_1 > \tau_2 > \cdots$, is

$$x_{\tau_{i+1}} = \sqrt{\bar{\alpha}_{\tau_{i+1}}}\,\hat{x}_0 + \sqrt{1-\bar{\alpha}_{\tau_{i+1}}-\sigma^2}\,\epsilon_\theta(x_{\tau_i},\tau_i) + \sigma z, \qquad \sigma = \eta\sqrt{\frac{1-\bar{\alpha}_{\tau_{i+1}}}{1-\bar{\alpha}_{\tau_i}}\Big(1-\frac{\bar{\alpha}_{\tau_i}}{\bar{\alpha}_{\tau_{i+1}}}\Big)},$$

where $\hat{x}_0$ is the clean estimate at $x_{\tau_i}$ and $\eta \geq 0$ controls stochasticity. Read it as: "estimate the clean example, then re-noise it to the next, lower noise level using the predicted noise direction." With $\eta = 1$ and consecutive steps, this recovers DDPM's posterior variance; with $\eta = 0$ the sampler is **deterministic** and is a discretization of the probability-flow ODE. Determinism has practical consequences: the same starting noise always gives the same output, so you can interpolate in noise space or invert a real image to its noise (DDIM inversion), which is the basis of many editing methods.

The following standalone example repeats the same small training setup so it can run independently. When experimenting in one session, reuse the already trained `net`, `abar` and `T` instead:

```
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(0)
torch.set_num_threads(1)
T = 100
beta = torch.linspace(.001, .15, T)
alpha = 1-beta
abar = alpha.cumprod(0)
previous = torch.cat([torch.ones(1), abar[:-1]])
post_var = beta * (1-previous) / (1-abar)
net = nn.Sequential(nn.Linear(3, 64), nn.SiLU(),
                    nn.Linear(64, 64), nn.SiLU(), nn.Linear(64, 2))
opt = torch.optim.Adam(net.parameters(), lr=.002)
def data(n):
    side = torch.randint(2, (n, 1))*2-1
    return torch.cat([side*1.5, torch.zeros(n, 1)], 1) + .25*torch.randn(n, 2)
for step in range(3000):
    x0 = data(128)
    t = torch.randint(T, (len(x0),))
    eps = torch.randn_like(x0)
    a = abar[t, None]
    xt = a.sqrt()*x0 + (1-a).sqrt()*eps
    predicted = net(torch.cat([xt, t[:, None].float()/(T-1)], 1))
    loss = F.mse_loss(predicted, eps)
    opt.zero_grad(); loss.backward(); opt.step()
net.eval()

def ddim_sample(n, steps, eta=0.0, seed=1):
    g = torch.Generator().manual_seed(seed)
    taus = torch.linspace(T-1, 0, steps).round().long()
    x = torch.randn(n, 2, generator=g)
    with torch.no_grad():
        for i, t in enumerate(taus):
            time = torch.full((n, 1), t.item()/(T-1))
            eps = net(torch.cat([x, time], 1))
            a_t = abar[t]
            a_prev = abar[taus[i+1]] if i+1 < steps else torch.tensor(1.)
            x0_hat = (x - (1-a_t).sqrt()*eps) / a_t.sqrt()
            sigma = eta*((1-a_prev)/(1-a_t)*(1-a_t/a_prev)).sqrt()
            x = (a_prev.sqrt()*x0_hat + (1-a_prev-sigma**2).sqrt()*eps
                 + sigma*torch.randn(n, 2, generator=g))
    return x

for steps in [5, 20, 100]:
    s = ddim_sample(2000, steps)
    left = s[s[:, 0] < 0]
    print(f"{steps:3d} steps: left fraction {(s[:, 0] < 0).float().mean():.3f}, "
          f"left-cluster mean x {left[:, 0].mean():.2f}, std {left[:, 0].std():.2f}")
```

The output shows the typical pattern. With 5 steps the left cluster is centered at $-1.11$ with standard deviation 0.39 — the samples are pulled toward the middle and smeared, because large steps

follow a curved trajectory with straight lines. With 20 steps it is −1.37 and 0.30; with 100 steps −1.42 and 0.24, close to the data. No retraining was needed to change the step count.

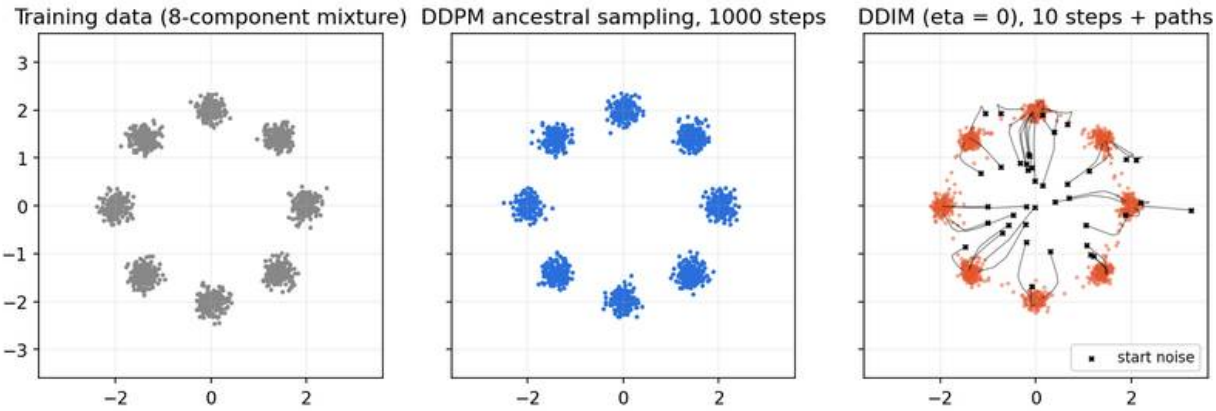



*Figure 3 — Samplers driven by the exact denoiser of an 8-component mixture, so only the sampling rule differs. DDPM with 1000 steps reproduces the data; deterministic DDIM with 10 steps lands near the modes but with slightly wider, smeared clusters. Black lines are DDIM trajectories from starting noise (×).*

### Faster solvers

Once sampling is "solve an ODE," numerical analysis applies. Karras et al. (EDM) [20] used a second-order Heun solver with a carefully chosen spacing of noise levels and showed that sampler, schedule, preconditioning and training noise distribution can be tuned separately. DPM-Solver [21] exploits the semi-linear structure of the diffusion ODE — the linear part can be integrated exactly — to reach good quality in around 10 network evaluations. Stochastic samplers add a little fresh noise at each step; that costs more steps but can correct accumulated errors. Whatever the solver, report the **number of function evaluations** (**NFE**): the number of network calls, which, rather than "steps," is the true sampling cost.

## Steering generation: conditioning and guidance

**Plain version.** Conditioning specifies the requested output; guidance adjusts how strongly sampling favors it. More guidance need not mean a better image.

### Conditioning

A conditional model learns $\epsilon_\theta(x_t, t, c)$ for a condition $c$. Class labels enter through embeddings added to the time embedding; text enters through **cross-attention** from image features to text-encoder tokens [7]; images (for super-resolution, inpainting, editing) are concatenated to the input channels or fed through extra encoders.

### Classifier guidance

Bayes' rule for the noisy marginals gives $\nabla \log p_t(x_t \mid y) = \nabla \log p_t(x_t) + \nabla \log p_t(y \mid x_t)$. Dhariwal and Nichol [18] trained a separate classifier $p_\phi(y \mid x_t)$ on *noisy* images and added its gradient, scaled by $w$, to an unconditional model. In noise units,

$$\hat{\epsilon} = \epsilon_\theta(x_t, t) - w\sqrt{1-\bar{\alpha}_t}\,\nabla_{x_t} \log p_\phi(y \mid x_t).$$

A scale $w > 1$ samples roughly from $p_t(x_t)\,p_t(y \mid x_t)^w$, a sharpened distribution that trades diversity for fidelity. Together with architecture improvements, this was the work that showed diffusion models beating GANs on ImageNet [35] sample quality [18]. Its drawback is the extra classifier, which must be trained on noisy inputs for every noise level.

### Classifier-free guidance (CFG)

Ho and Salimans [6] removed the classifier. During training, the condition is replaced by a "null" token for a random fraction of examples, so one network learns both $\epsilon_\theta(x_t, t, c)$ and $\epsilon_\theta(x_t, t, \varnothing)$. Bayes' rule says the classifier gradient equals the difference of the conditional and unconditional scores, so the implicit-classifier version of the formula above is

$$\epsilon_{\text{guided}} = \epsilon_\theta(x_t, t, \varnothing) + w(\epsilon_\theta(x_t, t, c) - \epsilon_\theta(x_t, t, \varnothing)).$$

In this convention $w = 0$ is unconditional, $w = 1$ is the ordinary conditional model, and $w > 1$ extrapolates past it. (Some papers write the same thing as $(1 + w')\epsilon_c - w'\epsilon_u$, so their "scale" is $w - 1$ in our notation; always check the convention before comparing numbers.)

The snippet below uses the exact noise predictor of a four-blob mixture, so any odd behaviour is due to guidance itself, not to a badly trained network. Class 1 is the two blobs with $x > 0$.

```
import numpy as np

# Data: four Gaussian blobs at the corners of a square; class c is the sign of x.
MU = np.array([[-1.5, -1.5], [-1.5, 1.5], [1.5, -1.5], [1.5, 1.5]])
SIG = 0.2
CLASS = np.array([0, 0, 1, 1])

def eps_exact(x, abar, comps):
    """Exact E[eps | x_t] for a Gaussian mixture restricted to `comps`."""
    m = np.sqrt(abar)*MU[comps]
    var = abar*SIG**2 + 1 - abar
    logw = -((x[:, None] - m[None])**2).sum(-1) / (2*var)
    w = np.exp(logw - logw.max(1, keepdims=True))
    w /= w.sum(1, keepdims=True)
    score = -(w[..., None]*(x[:, None] - m[None])).sum(1) / var
    return -np.sqrt(1-abar)*score

abar = np.cumprod(1 - np.linspace(1e-4, 0.02, 1000))
taus = np.linspace(999, 0, 50).round().astype(int)
everything, class1 = np.arange(4), np.where(CLASS == 1)[0]
print("data: mean distance to its mode", round(SIG*np.sqrt(np.pi/2), 3))
```

```
for w in [0, 1, 3, 8]:
    x = np.random.default_rng(0).normal(size=(4000, 2))
    for i, t in enumerate(taus):                    # deterministic DDIM
        e_u, e_c = eps_exact(x, abar[t], everything), eps_exact(x, abar[t], class1)
        e = e_u + w*(e_c - e_u)                     # classifier-free guidance
        x0 = (x - np.sqrt(1-abar[t])*e) / np.sqrt(abar[t])
        a_prev = abar[taus[i+1]] if i+1 < len(taus) else 1.0
        x = np.sqrt(a_prev)*x0 + np.sqrt(1-a_prev)*e
    dist = np.linalg.norm(x[:, None] - MU[None], axis=-1).min(1)
    print(f"w={w}: class-1 fraction {(x[:, 0] > 0).mean():.3f}, "
          f"mean distance to nearest mode {dist.mean():.3f}")
```

The data's mean distance to its mode is 0.251. With $w = 0$ about half the samples (0.490) are class 1; with $w = 1$ all of them are, at distance 0.225 (50 DDIM steps slightly under-disperse). At $w = 3$ the distance is 0.259 and at $w = 8$ it grows to 0.446: strong guidance does not just "make it more class 1" — it pushes samples *past* the data, a toy version of the over-saturated colours and distorted shapes seen with very high guidance in image models. Figure 4 shows the same effect on the 8-mode mixture.

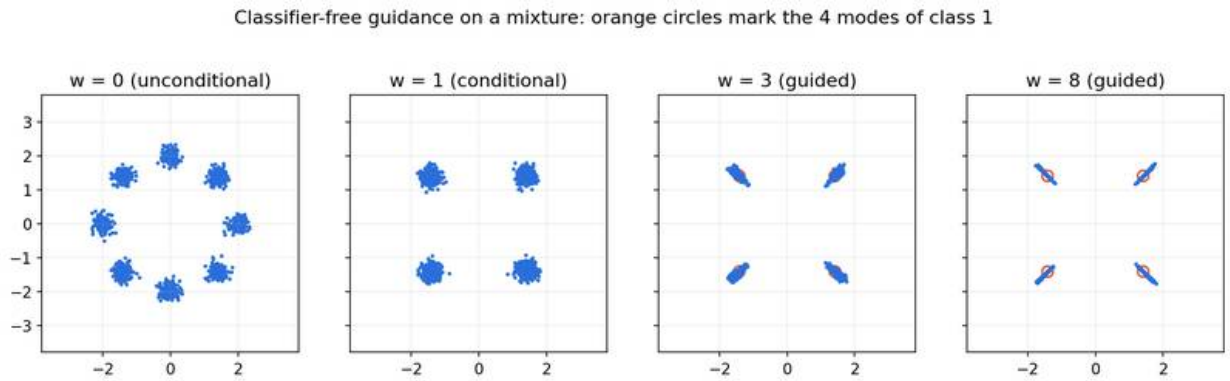



*Figure 4 — Exact classifier-free guidance on an 8-mode mixture whose class 1 has 4 modes (orange circles). $w = 1$ already restricts samples to class 1; larger $w$ shrinks clusters along some directions and stretches them along others — the samples no longer follow any data distribution.*

Guidance has a compute cost too: each step needs a conditional *and* an unconditional prediction, so a 30-step guided sampler costs 60 network evaluations unless the two are fused. And $w$ is a sampling knob, not a calibrated measure of how well the image matches the prompt.

## Latent diffusion and diffusion transformers

**Plain version.** Pixels are expensive. Compress the image into a smaller grid of features with an autoencoder, run diffusion in that compressed space, then decode. Separately, the denoising network itself can be a transformer instead of a convolutional U-Net [37].

### Latent diffusion

Rombach et al. [7] train an autoencoder first: an encoder $z = \mathcal{E}(x)$ maps an $H \times W \times 3$ image to an $h \times w \times c$ latent with a spatial downsampling factor $f = H/h$, and a decoder $\mathcal{D}$ maps back. The autoencoder is trained with reconstruction, perceptual and adversarial losses plus a light regularizer (a small KL term toward a Gaussian, or vector quantization) so that the latent space stays well behaved. Diffusion is then trained on $z$ exactly as before, and generation ends with $\hat{x} = \mathcal{D}(z_0)$. With $f = 8$, a $512 \times 512$ image becomes a $64 \times 64$ latent: 64 times fewer spatial positions for every network call.

The trade-off is a reconstruction bottleneck: information discarded by the encoder cannot be faithfully recovered from that code, although the decoder may synthesize plausible detail. The latent's scale must also match what the diffusion model was trained on; implementations rescale latents to roughly unit variance. Text conditioning in [7] uses cross-attention layers in the U-Net, the design behind the Stable Diffusion family.

### Diffusion transformers

DiT (Peebles and Xie [8]) replaces the U-Net with a Vision Transformer [41] (see ML 05) over latent patches. A latent of size $h \times w$ is cut into $p \times p$ patches, giving $hw/p^2$ tokens; smaller $p$ means more tokens, more compute and better quality. The timestep and class label condition every block through **adaptive layer norm**: they predict the scale and shift of each LayerNorm, and in the "adaLN-Zero" variant also a gating factor initialized to zero, so each residual block starts as the identity. The paper's main finding is a scaling trend: across model sizes and patch sizes, more transformer compute (GFLOPs) gave consistently lower FID [8].

It helps to keep three choices separate. The **architecture** (U-Net or transformer) is the function approximator. The **objective** (noise, $x_0$, $v$, or flow velocity) is what it regresses. The **sampler** (DDPM, DDIM, ODE solver) is how predictions become an image. Stable Diffusion 3 [9] is an example of a new combination: a latent transformer with separate weights for the image and text token streams that attend jointly ("MM-DiT"), trained with a rectified-flow objective and a timestep distribution biased toward intermediate noise levels.

For video, the latent and the attention must cover time as well as space, so compression and attention patterns become the main engineering problem. Melnik et al.'s survey [31] reviews architectures, temporal-consistency mechanisms and evaluation for video diffusion. Per-frame quality is not enough there: object permanence, motion plausibility and long-duration consistency need their own evaluation.

## Pixel space, latent space and representation autoencoders

**Plain version.** The generator needs a place to do its repeated updates. That place can be the image itself, a compact reconstruction code, or features learned by a visual representation model. A latent stage buys a particular computational and statistical trade-off; it is not a prerequisite for generative modeling.

### Why pixels become latents and then pixels again

Training images arrive as pixels, while the output must be viewable as pixels. Between these endpoints, an encoder can remove redundancy and produce a grid that is cheaper to update repeatedly [7]. Train the encoder and decoder on reconstruction first, then fit a generative distribution to encoded training images. These are two different tasks: reconstruction preserves a given example; generation learns which encoded examples are plausible.

The training and sampling routes are

$$\text{training: } x \xrightarrow{\mathcal{E}} z_{\text{data}} \xrightarrow{\text{corrupt}} z_t \xrightarrow{\text{prediction loss}} \theta,$$

$$\text{generation: } z_{\text{noise}} \xrightarrow{\text{learned sampler}} \hat{z} \xrightarrow{\mathcal{D}} \hat{x},$$

$$\text{reconstruction check: } x \xrightarrow{\mathcal{E}} \mathcal{E}(x) \xrightarrow{\mathcal{D}} \tilde{x}.$$

Here $x$ is a training image; $\mathcal{E}$ and $\mathcal{D}$ are the encoder and decoder; $z_{\text{data}}$ is an encoded image; $z_t$ is its corrupted state; $\theta$ parameterizes the generative predictor; $\hat{z}$ is a generated code; $\hat{x}$ is the generated image; and $\tilde{x}$ is a reconstruction. Pure text-to-image sampling does **not** encode an input image first. Editing and image-to-image generation can encode an actual input image and begin from a partially corrupted code.

The main saving occurs because the denoiser or flow predictor runs repeatedly, while encoding training data and decoding a sample occur much less often. A spatial factor $f$ reduces grid positions by $f^2$, but the number of latent channels also matters. For an image with three channels and a latent with $c$ channels, the scalar-count reduction is $3f^2/c$. It is incorrect to describe the grid reduction as the reduction in all floating-point values or total runtime.

For a transformer, token count and token width must also be distinguished. Full attention stores a matrix with a number of entries proportional to $N^2$, where $N$ is the token count; feature projections depend on token width as well [41]. A patch size can reduce $N$ in either pixels or latents. None of these counts alone predicts wall-clock speed because kernels, memory traffic, the decoder and conditioning contribute.

```
# Synthetic shape bookkeeping, not a benchmark or a specific checkpoint.
H = W = 512
channels, spatial_factor, patch = 4, 8, 2
pixel_values = H*W*3
h = H//spatial_factor
w = W//spatial_factor
latent_values = h*w*channels
pixel_tokens = (H//patch)*(W//patch)
latent_tokens = (h//patch)*(w//patch)
print("Spatial positions reduction:", (H*W)//(h*w))
print("Scalar values reduction:", pixel_values//latent_values)
print("Dense attention entries reduction:", pixel_tokens**2//latent_tokens**2)
assert pixel_values//latent_values == 48
assert pixel_tokens//latent_tokens == 64
assert pixel_tokens**2//latent_tokens**2 == 4096
```

This constructed example yields 64 times fewer spatial positions, 48 times fewer scalar values and 4096 times fewer attention entries **at the same numerical patch size in each coordinate space**. Changing patch sizes changes the comparison. A pixel transformer with much larger patches can have the same token count as a latent transformer.

The cost is information loss. If two images have the same code, a deterministic decoder cannot recover their different exact details. A decoder may invent plausible texture, but that does not restore information about the original image. The reconstruction check therefore deserves its own evaluation before generative training. Inspect fine lines, lettering and temporal changes, rather than relying only on average image error.

The following deliberately simple encoder averages blocks and the decoder repeats those averages. It demonstrates a bottleneck without pretending to reproduce a learned VAE or RAE.

```
import numpy as np

y, x = np.mgrid[:64, :64]
checker = ((x+y) % 2).astype(float)
smooth = x/63
image = .6*smooth + .4*checker
f = 4
latent = image.reshape(64//f, f, 64//f, f).mean(axis=(1, 3))
reconstruction = np.repeat(np.repeat(latent, f, axis=0), f, axis=1)
print("Synthetic reconstruction MSE:", round(np.mean((image-reconstruction)**2), 6))
assert np.mean((image-reconstruction)**2) > .04
# Recovering the exact latent does not recover the missing checkerboard.
assert np.allclose(reconstruction[0, :f], reconstruction[0, 0])
```

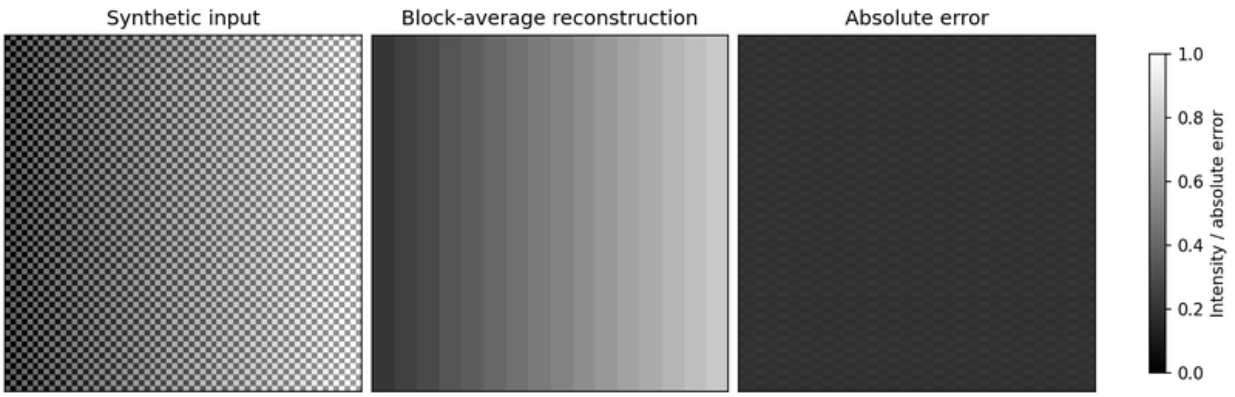



*Figure 5 — A latent bottleneck can remove detail before the generative model is trained. This block-average toy is an explanation of information loss, not an evaluation of a learned autoencoder.*

For editing, repeatedly encoding and decoding a result can accumulate distortion. Keep intermediate edits in the compatible latent space when the pipeline permits it, and inspect the eventual pixel output. A good unconditional generator may still be a poor reconstruction system. The

claim that an autoencoder sets a ceiling concerns faithfulness to encoded information; it does not mean that a generative decoder can never synthesize new fine texture.

### Reconstruction codes versus semantic representation codes

A VAE encoder is trained with a reconstruction objective and a distributional regularizer; its code is designed to support decoding and sampling [7]. A pretrained recognition encoder instead learns useful visual features. These goals can align, but recognition invariance may discard details needed for faithful reconstruction. The decoder and the retained feature structure determine whether the representation is a useful generative space.

Zheng, Ma, Tong and Xie's **Representation Autoencoders** (**RAEs**) [43] pair a frozen pretrained representation encoder with a trained decoder. Their ICLR 2026 study uses semantic visual features as the space for generative modeling. It examines high-dimensional token features, dimension-dependent noise scheduling, decoder robustness and a shallow wide diffusion head. This changes the representation being generated, rather than simply adding a feature-alignment loss to a conventional latent predictor.

Do not equate "latent" with "smaller in every dimension." A spatially sparse feature grid can have many channels per token. An RAE is also not automatically a variational model: a frozen feature extractor plus a decoder does not by itself define a Gaussian posterior, a KL regularizer or a likelihood bound. Keep reconstruction, representation quality and generative quality as separate measurements.

Tong et al. [44] extend this approach to freeform text-to-image generation in 2026. Their study compares RAE and VAE spaces at larger scales and examines decoder data composition. It finds that dimension-aware noise scheduling remains important while some small-scale architectural and decoder modifications offer little benefit at the tested larger scales. This is a reason to re-test recipes when scale and data change, rather than treating an ImageNet result as a universal design rule.

For a new application, inspect whether the chosen encoder preserves the content that matters: fine text for documents, small defects for inspection, or frame-to-frame differences for video. Evaluate the decoder on held-out real features and on generated features. A decoder trained only on exact training codes can be sensitive to small off-distribution errors in a sampled code. Normalize features using the training recipe, and record the encoder version with the generative checkpoint.

### JiT: direct clean-image prediction on pixels

Li and He's **Just image Transformers** (**JiT**) [42], published at CVPR 2026 after a 2025 preprint, revisit large-patch transformers operating directly on pixels. The network predicts the clean image; sampling derives a flow velocity from that prediction. Their controlled experiments support a manifold-based explanation: a bottleneck can retain structured clean-image information while making direct prediction of high-dimensional noise or velocity difficult. The reported setup uses no pretrained latent tokenizer or auxiliary representation loss. This is evidence for a particular

architecture and training regime, not a proof that pixel-space models always outperform latent generators.

The crucial distinction is **direct network output versus the quantity used in the loss or solver**. For the straight path defined below, use $y_t = (1-t)\epsilon + tx$, with noise at $t = 0$ and clean data at $t = 1$. A network that directly predicts $\hat{x}_\theta(y_t, t)$ can supply

$$\hat{u}_\theta(y_t, t) = \frac{\hat{x}_\theta(y_t, t) - y_t}{1-t}, \qquad \|\hat{u}_\theta - (x-\epsilon)\|^2 = \frac{\|\hat{x}_\theta - x\|^2}{(1-t)^2}.$$

Here $x$ is the clean image; $\epsilon$ is the noise draw; $y_t$ is their interpolation; $\hat{x}_\theta$ is the direct prediction; and $\hat{u}_\theta$ is the corresponding transport velocity. The identity holds for $t < 1$ before numerical clipping. An $x$-output network can thus train with a velocity-space loss. Calling it "clean prediction" does not mean its loss is an unweighted clean-image MSE. JiT's released implementation makes this distinction explicit [42].

Near $t = 1$, division by $1 - t$ needs an endpoint convention and numerical safeguards. For the interior algebra, a fixed clean-image error is amplified increasingly as the endpoint approaches. During sampling, use the checkpoint's time distribution, noise scaling and solver recipe. Changing the direct output label alone does not reproduce the method.

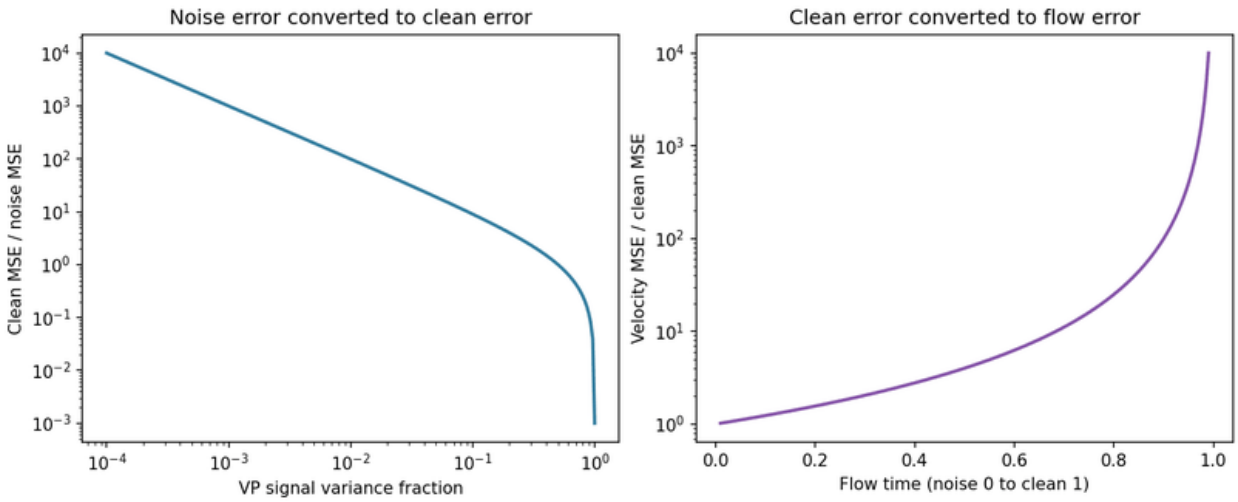



*Figure 6 — Equivalent predictions use different units. The left curve shows noise-error amplification at low signal; the right shows velocity-error amplification near the clean endpoint. The curves are algebraic identities, not model performance measurements.*

A useful comparison fixes training data, compute, resolution, token count and the sampler, then changes the representation or target one at a time. Compare a pixel transformer, a compact VAE-latent transformer and a representation-latent transformer on reconstruction needs, generated quality and total cost. Record encoder/decoder cost and any representation pretraining. A model that uses a pretrained visual encoder has inherited training even when its generative stage starts from scratch.

*Online only: JiT — clean-image prediction in pixel space — https://blog.shyandram.dev/en/learn/ml/diffusion-flow-matching/papers/jit–clean-image-prediction-in-pixel-space/*

## Flow matching: learn a transport velocity

**Plain version.** For a straight conditional path, pair noise with data and regress its velocity. Generation follows the learned velocity field, whose trajectories need not be straight.

**Time convention change.** In this section time runs from **noise at** $t = 0$ **to data at** $t = 1$, the opposite of the diffusion sections above (where large $t$ meant more noise).

### The conditional flow-matching objective

A continuous normalizing flow moves points with an ODE $dx_t/dt = v(x_t, t)$. If the velocity field $v$ is right, it transports the noise distribution $p_0 = \mathcal{N}(0, I)$ to the data distribution $p_1$. Lipman et al. [10] (and, concurrently, Liu et al. [11] and Albergo and Vanden-Eijnden's stochastic interpolants [34]) showed how to train $v$ without ever simulating the ODE.

Pair noise $z \sim \mathcal{N}(0, I)$ with a data example $x \sim p_1$ and define the straight **conditional path**

$$x_t = (1 - t)\, z + t\, x, \qquad \frac{dx_t}{dt} = u_t(x_t \mid z, x) = x - z.$$

Every pair moves at constant speed in a straight line. The marginal velocity we actually want is the average of these conditional velocities over all pairs that pass through the point:

$$u_t(y) = \mathbb{E}[\, x - z \mid x_t = y \,].$$

We cannot compute this average, but we do not need to. Compare the "ideal" loss with the one we can compute:

$$\mathcal{L}_{\mathrm{FM}} = \mathbb{E}_{t,x_t} \| v_\theta(x_t, t) - u_t(x_t) \|^2, \qquad \mathcal{L}_{\mathrm{CFM}} = \mathbb{E}_{t,z,x} \| v_\theta(x_t, t) - (x - z) \|^2,$$

with $t \sim \mathcal{U}[0, 1]$. Expand both squares. The $\|v_\theta\|^2$ terms are identical. The cross terms are equal because, by the tower rule,

$\mathbb{E}[v_\theta(x_t, t)^\top (x - z)] = \mathbb{E}[v_\theta(x_t, t)^\top \, \mathbb{E}[x - z \mid x_t]] = \mathbb{E}[v_\theta^\top u_t(x_t)]$. The remaining terms do not involve $\theta$. Therefore

$$\nabla_\theta \mathcal{L}_{\mathrm{CFM}} = \nabla_\theta \mathcal{L}_{\mathrm{FM}},$$

so regressing the per-pair velocity $x - z$ trains the marginal velocity field. This is the same trick as in diffusion: a target that is noisy for any single example has the right conditional mean. Generation solves $dx/dt = v_\theta(x, t)$ from $t = 0$ to $t = 1$ with any ODE solver.

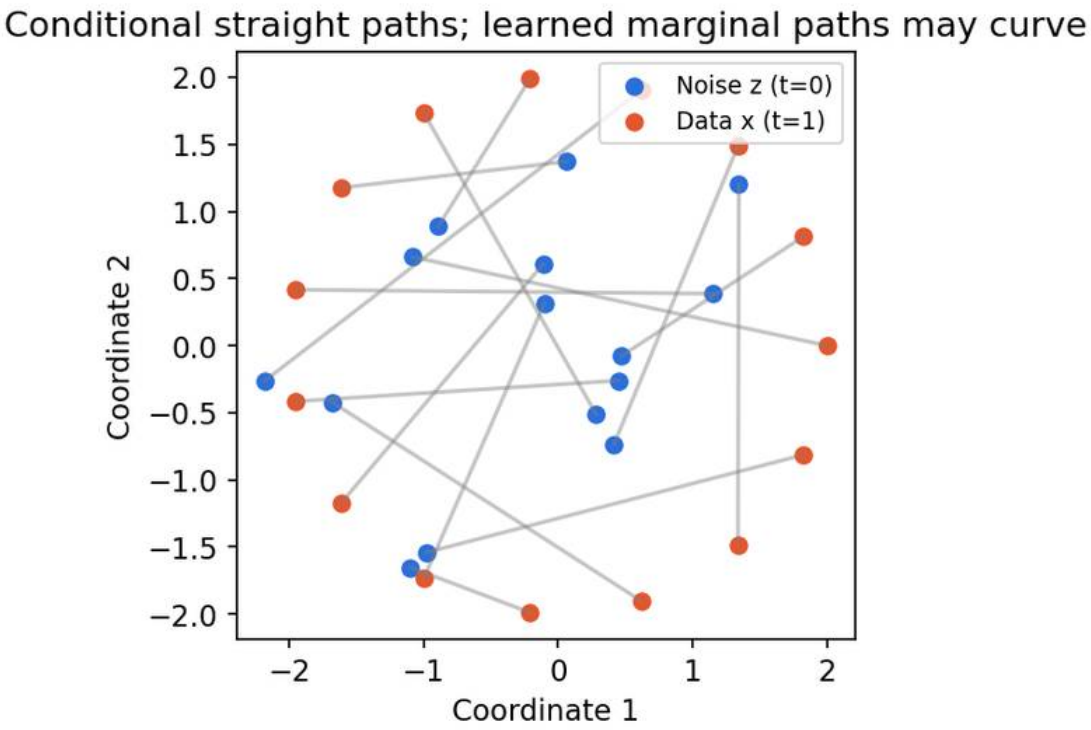



*Figure 7 — Conditional training paths are straight. Paths of different pairs cross, so the marginal velocity at a crossing point is an average, and the learned trajectories can curve.*

### How flow matching relates to diffusion

The linear path is a Gaussian path: given $x$, $x_t \sim \mathcal{N}(t\,x,\ (1-t)^2 I)$. Writing $\hat{z} = \mathbb{E}[z \mid x_t]$ (a noise prediction) and using $x = (x_t - (1-t)z)/t$,

$$u_t(x_t) = \frac{x_t - \hat{z}}{t}, \qquad \nabla \log p_t(x_t) = -\frac{\hat{z}}{1-t}, \qquad \text{so} \qquad u_t(x_t) = \frac{x_t + (1-t)\,\nabla \log p_t(x_t)}{t}.$$

So a flow-matching velocity, a noise predictor and a score are again interchangeable for this path; what differs is the path (straight lines versus the VP curve), the weighting of noise levels in the loss, and the time convention. Diffusion's probability-flow ODE is itself a flow along a particular Gaussian path. Lipman et al.'s guide [2] and the stochastic-interpolant framework [34] treat both as special cases of one construction. The practical advice is the same as for parameterizations: the targets are related by derivation, not interchangeable by renaming.

### Code: flow matching on two clusters

This standalone example learns the same two-cluster distribution as the DDPM snippet with linear conditional paths, then samples with Euler steps of size $1/N$.

```
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(0)
torch.set_num_threads(1)
net = nn.Sequential(nn.Linear(3, 64), nn.SiLU(),
                    nn.Linear(64, 64), nn.SiLU(), nn.Linear(64, 2))
opt = torch.optim.Adam(net.parameters(), lr=.002)
for step in range(3000):
    z = torch.randn(128, 2)
    side = torch.randint(2, (128, 1))*2-1
    x = torch.cat([1.5*side, torch.zeros(128, 1)], 1) + .25*torch.randn(128, 2)
    t = torch.rand(128, 1)
    xt = (1-t)*z+t*x
    loss = F.mse_loss(net(torch.cat([xt, t], 1)), x-z)
    opt.zero_grad(); loss.backward(); opt.step()
net.eval()
def euler_sample(n, steps, seed=1):
    x = torch.randn(n, 2, generator=torch.Generator().manual_seed(seed))
    with torch.no_grad():
        for k in range(steps):
            t = torch.full((n, 1), k/steps)
            x = x + net(torch.cat([x, t], 1))/steps
    return x

for steps in [1, 4, 50]:
    s = euler_sample(2000, steps)
    assert torch.isfinite(s).all()
    left = s[s[:, 0] < 0]
    print(f"{steps:2d} Euler steps: left fraction {(s[:, 0] < 0).float().mean():.3f}, "
          f"left-cluster mean x {left[:, 0].mean():.2f}, std {left[:, 0].std():.2f}")
```

It prints, for 1, 4 and 50 steps, left-cluster means of $-0.18$, $-1.20$ and $-1.45$ with standard deviations 0.12, 0.42 and 0.29. One Euler step lands almost at the overall data mean: at $t = 0$ the marginal velocity is $\mathbb{E}[x] - z$, so a single full step sends *every* noise point to $\mathbb{E}[x]$. Straight conditional paths do not make the learned marginal flow straight. Figure 8 shows the same effect with the exact velocity of an 8-mode mixture.

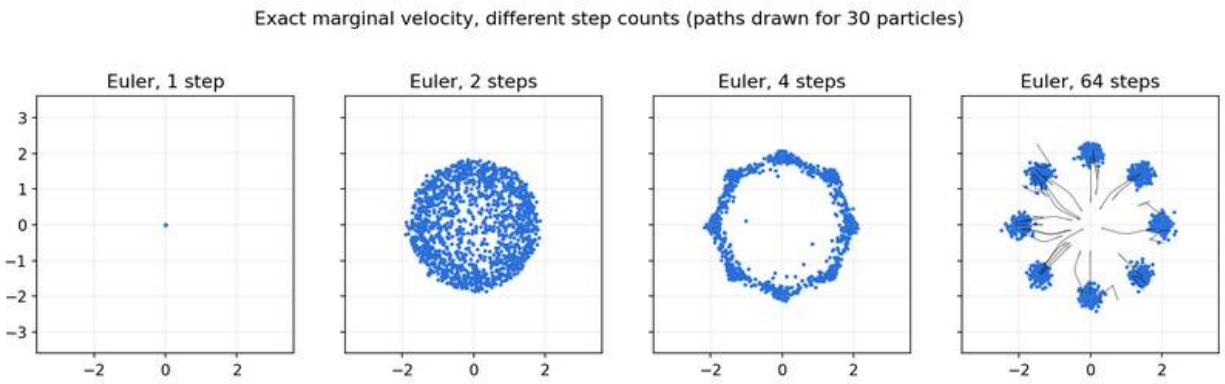


*Figure 8 — Even with the exact marginal velocity, Euler sampling needs many steps: one step collapses every sample to the data mean, and the 64-step trajectories (black) bend toward the modes late in the path.*

### Rectified flow and reflow

Liu, Gong and Liu [11] use the same straight interpolation ("rectified flow") and add **reflow**: after training, generate pairs $(z, \hat{x})$ by solving the learned ODE from each $z$, then retrain on these *coupled* pairs instead of independent ones. Because the new pairs are already connected by the flow, their straight lines cross less, the learned trajectories become straighter, and fewer Euler steps are needed; a straight enough flow can then be distilled into a one-step model. Stable Diffusion 3 [9] showed that rectified-flow training of a large latent transformer works at text-to-image scale.

*Online only: Flow Matching — regress a velocity, then integrate it — https://blog.shyandram.dev/en/learn/ml/diffusion-flow-matching/papers/flow-matching–regress-a-velocity-then-integrate-it/*

## After flow matching: intervals, image outputs and one-step pixels

**Plain version.** There are two separate questions after learning a flow: can one prediction cover a whole interval, and what should the network directly output? MeanFlow changes the interval being modeled; JiT changes the output representation. Pixel MeanFlow brings these directions together. Read them in that order instead of treating every new name as another sampler.

### First align the clocks

The preceding flow-matching example moves from noise at zero to data at one. MeanFlow [13], improved MeanFlow [49] and pixel MeanFlow [50] use the reverse convention:

$$z_t = (1-t)x + t\epsilon, \qquad t = 0:\ \text{data}, \quad t = 1:\ \text{noise}.$$

Here $x$ is a clean training example, $\epsilon$ is prior noise, and $z_t$ is their interpolated training state. Generation runs from one to zero. To compare with JiT's time $s$, use $s = 1 - t$ and reverse the velocity sign. This is a coordinate change, not a different generative principle. The conditional straight interpolation used to form training pairs still does not imply that the learned marginal ODE trajectories are straight.

### MeanFlow: replace a tangent with an interval average

An instantaneous velocity tells you which way to move **now**. MeanFlow [13] learns the average over an interval, so a prediction can describe its entire displacement:

$$u(z_t, r, t) = \frac{1}{t-r}\int_r^t v(z_s, s)\, ds, \qquad z_r = z_t - (t-r)u(z_t, r, t).$$

Here $v$ is the marginal instantaneous velocity, $z_s$ lies on its ODE trajectory, $r < t$ are interval endpoints, and $u$ is the average velocity along that trajectory. The zero-length boundary uses the

limit $u = v$. This integral defines the desired quantity; training does not obtain a teacher trajectory by evaluating it for every example.

Instead, differentiating the definition yields $u = v - (t - r)du/dt$. The total derivative follows the moving state and holds $r$ fixed; it is not merely a partial derivative with respect to the time input. The original recipe uses a Jacobian-vector product and a stopped-gradient target constructed from the current network. One-call generation then uses the full interval $(r, t) = (0, 1)$, provided the learned average is accurate enough.

The following exact curve illustrates the distinction. It is an algebra check, not a trained MeanFlow model:

```
r, t = 0.0, 1.0
start = (t, t*t)                    # curve z(t) = (t, t^2)
instant = (1.0, 2*t)                # tangent at the starting time
average = (1.0, t+r)                # exact interval-average derivative
one_euler = tuple(z-(t-r)*v for z, v in zip(start, instant))
one_average = tuple(z-(t-r)*u for z, u in zip(start, average))
assert one_average == (r, r*r)
print("Instantaneous Euler jump:", one_euler)
print("Average-velocity jump:", one_average)
```

The outputs are `(0.0, -1.0)` and `(0.0, 0.0)`. A correct local tangent can miss the endpoint after one large step, whereas the correct interval average reaches it. Learning that average is the difficult part.

*Online only: MeanFlow — average velocity over an interval — https://blog.shyandram.dev/en/learn/ml/diffusion-flow-matching/papers/meanflow–average-velocity-over-an-interval/*

### Improved MeanFlow: make prediction inputs and guidance explicit

iMF [49] rewrites training as a loss on instantaneous velocity while the network still outputs average velocity. The important change is inside the derivative calculation: the original recipe feeds the sampled conditional velocity into the JVP; iMF uses a network prediction of marginal velocity. This keeps the clean training label out of the compound predictor's inputs. It can reuse the zero-length boundary or use an auxiliary velocity head during training.

A separate improvement makes guidance scale and its active interval explicit conditions. A single trained model can then vary those settings at inference while retaining one-call generation. That is different from adding two conditional/unconditional network calls afterward. Its in-context condition tokens also change model size, so the final system comparison is not a pure loss-function ablation.

*Online only: Improved MeanFlow — prediction inputs and flexible guidance — https://blog.shyandram.dev/en/learn/ml/diffusion-flow-matching/papers/improved-meanflow–prediction-inputs-and-flexible-guidance/*

### JiT: image output is a different design axis

The earlier JiT discussion separates direct clean-image output from velocity-space loss [42]. In the time convention used here, the derived instantaneous velocity is $(z_t - \hat{x}_\theta)/t$, where $\hat{x}_\theta$ is the predicted clean image. This conversion needs care near zero. JiT still follows a multi-step trajectory; an image-shaped output does not automatically provide an interval-average velocity or a one-step sample.

The controlled comparison is especially useful: hold loss space fixed, then change direct output and patch dimension. It explains why algebraic conversions between targets do not make finite-capacity networks equally easy to train. Do not compare only the strongest headline models with different training recipes.

*Online only: JiT — clean-image prediction in pixel space — https://blog.shyandram.dev/en/learn/ml/diffusion-flow-matching/papers/jit–clean-image-prediction-in-pixel-space/*

### Pixel MeanFlow: connect image output to the interval field

Pixel MeanFlow [50] defines a two-time image-like field $x^\star(z_t, r, t) = z_t - t\,u(z_t, r, t)$. Here $x^\star$ is induced by the average-velocity field, not necessarily the clean training image paired with $z_t$. The network predicts this field as $x_\theta$; converting back gives $u_\theta = (z_t - x_\theta)/t$. This conversion uses $t > 0$ with endpoint safeguards.

The derived $u_\theta$ enters iMF's instantaneous-velocity loss: **image output → average velocity → instantaneous-velocity loss**. For the full interval, the update simplifies to the network's image output, enabling one-call pixel generation. At intermediate intervals, the image-like manifold interpretation is a hypothesis supported by the paper's visualizations and ablations, not an exact manifold guarantee.

No latent tokenizer is needed at generation, but the stronger recipe uses pretrained feature networks for perceptual training losses. Record that training dependency when comparing it with JiT's recipe or a latent generator. "No decoder," "one NFE" and "no pretrained component anywhere" are three different claims.

*Online only: Pixel MeanFlow — one-step pixels through a denoised field — https://blog.shyandram.dev/en/learn/ml/diffusion-flow-matching/papers/pixel-meanflow–one-step-pixels-through-a-denoised-field/*

### Keep neighboring approaches separate

Reflow [11] changes the coupling of training pairs to straighten trajectories. Consistency models [12] learn agreement of endpoint predictions along a trajectory; teacher-based distillation and teacher-free training are separate settings. MeanFlow instead parameterizes an interval average, while JiT addresses the network output space. The few-step overview compares these neighboring families.

Before choosing a method, write down its representation, direct output, loss target, time direction and sampling calls. Then compare matched resolution and conditioning, distinguish generator

training from representation pretraining, and include decoder cost where relevant. A lower NFE is useful evidence about calls, not a complete latency or quality evaluation.

## Pretrained image and video generators as complete pipelines

**Plain version.** A pretrained generator is a bundle of compatible components. Loading the denoiser weights alone does not establish the right representation, text conditioning, prediction target or sampling schedule. The image/video boundary also changes what a reconstruction must preserve.

### Read the components before running the checkpoint

A practical pipeline combines a text encoder, an optional image/video encoder, a time-conditioned predictor, a sampler and an optional decoder. A separate refinement model may follow the first generator. Each component expects particular tensor shapes and scales. A scheduler declaring `v_prediction` may mean VP velocity; a flow scheduler may expect a derivative along a straight interpolation. Check the training path rather than guessing from a configuration string.

The following verified examples illustrate architectural differences. They are fixed research baselines, rather than a ranking of the latest product releases. The references include their papers or technical reports and official release documentation.

| Baseline | Generation representation and predictor | What to inspect in the pretrained pipeline |
|---|---|---|
| SDXL [45] | Reconstruction latents with a convolutional denoiser | Text conditioning, latent scaling, optional base/refiner composition and compatible sampling configuration |
| Stable Diffusion 3 [9] | Latent rectified flow with a joint text–image transformer | Flow time convention, text streams, time sampling and autoencoder compatibility |
| FLUX.1 [46] | Latent rectified-flow transformer | Exact release variant, guidance behavior, text conditioning, autoencoder and release terms |
| JiT [42] | Class-conditioned clean-image prediction directly on pixels | Image normalization, patch/ resolution configuration, derived velocity and sampling recipe; this is not a freeform text-to-image checkpoint |
| HunyuanVideo [47] | Compressed video latents with a flow-matching transformer | Video autoencoder, image/video joint training, temporal dimensions and text-conditioning components |
| Wan2.1 [48] | Video diffusion transformer with a spatiotemporal autoencoder | Text-to-video versus image-to-video variant, causal video compression, temporal length and decode caching |

SDXL [45] documents a larger latent image-generation architecture and optional refinement stage. Stable Diffusion 3 [9] illustrates how the same broad latent pipeline can instead use rectified flow and joint text/image attention. FLUX.1's official model card [46] describes a released rectified-flow transformer; its `.1-dev` variant is guidance-distilled. These facts do not make the checkpoints interchangeable. Ordinary two-pass CFG and a distilled guidance input can have different meanings and costs.

For image-to-image generation, encoding an input and adding partial noise trades input preservation against the ability to change it. The actual noise level is determined by the schedule, not merely by a UI label such as "strength." To compare edits, keep the same input and record the initial time, seed, prompt, guidance and reconstruction baseline. A reconstruction artifact can otherwise be mistaken for an edit requested by the text.

### Video adds temporal compression and a consistency problem

Let a video have $F$ frames, height $H$ and width $W$. A hypothetical encoder compressing time by $f_t$ and each spatial axis by $f_s$ gives a rough latent grid of $(F/f_t) \times (H/f_s) \times (W/f_s)$ positions. Exact shapes depend on padding and special handling of the first frame. After patching, dense attention across all positions grows quadratically with the remaining token count; separately reducing time and space can therefore change cost substantially.

HunyuanVideo's technical report [47] combines a 3D video autoencoder with a flow-matching transformer and staged image/video training. Wan's report [48] explains a causal video VAE with special first-frame handling, spatial and temporal compression, and feature caching during chunked encoding/decoding. Causality in an autoencoder does not by itself imply that the entire video generator samples autoregressively; the generative attention and sampling procedure must be examined separately.

A frame can look excellent while a clip changes an object's identity, reverses motion unexpectedly or forgets an occluded object. Test reconstruction of held-out clips before generation, then evaluate appearance, motion, prompt correspondence and consistency across frames. Melnik et al. [31] provide the survey map for these video-specific issues. A per-image metric alone cannot diagnose temporal behavior.

A pretrained video generator also does not automatically become a reliable world model. Predicting plausible clips differs from predicting the consequences of a specified action. The world-model chapter develops this distinction through state, actions, rollouts and evaluation. Use the actual conditioning interface and evaluate the task the system claims to solve.

### A reproducible first experiment

Start by recording a pipeline manifest: checkpoint revision, encoder/decoder revisions, text encoder, representation shape and scaling, prediction target, time direction, solver, evaluation count, guidance, dtype and seed. Then perform a reconstruction check if an autoencoder is present and generate a fixed small prompt/seed set. Compare one change at a time. Include decoder latency and peak memory when reporting the cost of a latent method.

This chapter's CPU examples test the underlying identities and failure modes without downloading weights. Full-resolution pretrained generation belongs in a separately provisioned experiment using the official release instructions [42], [45]–[48]. Treat an actual measured run, a paper's published result and a toy calculation as three different kinds of evidence.

## Few-step and one-step generation

**Plain version.** If a sampler needs 50 network calls, train a second model that jumps along the trajectory in a few calls — or train one from scratch that is designed to jump.

The main families are:

- **Progressive distillation** [22]: a student learns to match two deterministic teacher steps with one step; repeating the procedure halves the step count each round.
- **Consistency models** [12]: learn a function $f_\theta(x_t, t)$ that maps *any* point on a probability-flow ODE trajectory to that trajectory's starting (clean) point. It must satisfy the boundary condition $f_\theta(x, \epsilon) = x$ at the smallest time $\epsilon$, enforced by writing $f_\theta(x,t) = c_{\text{skip}}(t)\, x + c_{\text{out}}(t)\, F_\theta(x,t)$ with $c_{\text{skip}}(\epsilon) = 1$ and $c_{\text{out}}(\epsilon) = 0$. Training asks outputs at adjacent points of a trajectory to agree. In *consistency distillation*, adjacent points come from one ODE-solver step of a pretrained diffusion model; in *consistency training*, from the same data example noised at two levels, with no teacher. Sampling takes one step, or a few steps of "jump to clean, re-noise, jump again." Improved training recipes [32] simplified continuous-time consistency models and scaled them to 1.5 billion parameters on ImageNet $512 \times 512$ with two-step sampling.
- **Adversarial distillation**: ADD [33] combines score distillation from a large pretrained diffusion teacher with an adversarial loss so that a student produces sharp images in one to four steps.
- **Average-velocity models**: MeanFlow [13] learns interval displacement through an average velocity; iMF [49] improves its regression and guidance, and pixel MeanFlow [50] connects it to image output. The post-flow walkthrough develops these differences and their separate pretraining dependencies.

A small step count is a learned capability, not a free consequence of a nicer equation. When comparing, count NFE including guidance evaluations, the decoder's cost in latent models, resolution, quality *and* diversity — and remember that a one-step network can be larger and slower per call than a multi-step one.

## Evaluating generative models

**Plain version.** "Does it look good?" hides at least three questions: are individual samples realistic (fidelity), does the set of samples cover the variety of the data (diversity), and does the sample match the condition (alignment)? No single number answers all three.

### Fréchet Inception Distance

**FID** (Heusel et al. [23]) embeds real and generated images with an Inception-v3 network [36], fits a Gaussian to each set of features, and computes the Fréchet (2-Wasserstein) distance between the two Gaussians:

$$\mathrm{FID} = \|\mu_r - \mu_g\|^2 + \mathrm{tr}\Big(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}\Big),$$

where $\mu_r, \Sigma_r$ and $\mu_g, \Sigma_g$ are the feature mean and covariance of the real and generated sets and tr is the matrix trace. Lower is better. The snippet applies the formula to two-dimensional "features."

```
import numpy as np
from scipy import linalg

def frechet_distance(a, b):
    """Frechet distance between Gaussians fitted to feature sets a and b."""
    mu1, mu2 = a.mean(0), b.mean(0)
    c1, c2 = np.cov(a, rowvar=False), np.cov(b, rowvar=False)
    covmean = linalg.sqrtm(c1 @ c2).real
    return float(((mu1 - mu2)**2).sum() + np.trace(c1 + c2 - 2*covmean))

rng = np.random.default_rng(0)
centers = np.array([[-1.5, 0.], [1.5, 0.]])
real = centers[rng.integers(2, size=5000)] + .25*rng.normal(size=(5000, 2))
good = centers[rng.integers(2, size=5000)] + .25*rng.normal(size=(5000, 2))
one_mode = centers[[0]] + .25*rng.normal(size=(5000, 2))      # mode collapse
blurry = .6*rng.normal(size=(5000, 2))                         # mass in the gap
print("good     :", round(frechet_distance(real, good), 3))
print("one mode :", round(frechet_distance(real, one_mode), 3))
print("blurry   :", round(frechet_distance(real, blurry), 3))
small = [frechet_distance(real[:50], good[i*50:(i+1)*50]) for i in range(20)]
print("good, 50 samples: mean", round(np.mean(small), 3), "std", round(np.std(small), 3))
```

It prints about 0.001 for a correct sampler, 3.9 for a sampler that only produces the left mode, and 0.976 for a "blurry" sampler that puts its mass between the modes. The last line shows a practical trap: with 50 samples, even a *perfect* sampler scores $0.077 \pm 0.073$, because FID is biased upward for small sample sizes. Compare FIDs only with the same number of samples, the same reference set and the same feature extractor and preprocessing.

### Beyond FID

- **Precision and recall** (Kynkäänniemi et al. [24]) separate fidelity from diversity: estimate the real and generated "manifolds" with $k$-nearest-neighbour balls in feature space; precision is the fraction of generated samples inside the real manifold, and recall the fraction of real samples inside the generated one. Our "one mode" sampler would have high precision and low recall.
- **Feature extractor matters.** Stein et al. [25] ran a large human study and found that no common metric correlated strongly with human judgments; in particular, the perceptual realism of diffusion models was not reflected in FID. They recommend features from DINOv2-ViT-L/14

[39] instead of Inception, and show that current metrics cannot separate memorization from underfitting or mode shrinkage.
- **CMMD** (Jayasumana et al. [26]) replaces Inception features with CLIP [38] embeddings and the Gaussian assumption with a maximum mean discrepancy using a Gaussian RBF kernel, giving an unbiased estimator that is more sample-efficient.
- **Likelihood.** A probability-flow ODE supports log-density evaluation through change of variables [4]; numerical tolerances and divergence estimation must be reported. A DDPM variational bound [19] is a different quantity. Converting a continuous density to image bpd also requires the stated dequantization and scaling convention. Neither score alone measures visual quality.
- **Conditional models** need alignment checks (does the image show what the prompt says?), usually by human rating or by vision–language models, and memorization checks against the training set.
- **Cost.** Report NFE, guidance evaluations, resolution and wall-clock time on stated hardware.

The evaluation chapter ML 11 returns to sample-size effects and honest comparisons.

## Modern view

**Surveys and reference texts.** The literature is large; these reviews are good maps.

- Yang et al. [1] (*ACM Computing Surveys*, 2023) organize diffusion research into three improvement directions — efficient sampling, improved likelihood estimation, and data with special structure (manifolds, discrete data, invariances) — and also cover combinations with VAEs, GANs and flows and applications across domains.
- Croitoru et al. [27] (*IEEE TPAMI*, 2023) focus on vision. They identify three generic formulations — DDPMs, noise-conditioned score networks and SDEs — and categorize vision applications such as generation, super-resolution, inpainting, editing, translation and segmentation.
- Cao et al. [28] (*IEEE TKDE*, 2024) take a methods angle: fundamental formulation, algorithmic enhancements (sampling acceleration, objective design, conditioning) and applications from images to biology and healthcare.
- Lipman et al. [2] (2024) is a guide-with-code to flow matching that covers Gaussian and general probability paths, the link to diffusion and extensions to discrete and Riemannian data.
- Lai, Song, Kim, Mitsufuji and Ermon [29] (2025) is a book-length monograph presenting the variational, score-based and flow-based views as three routes to the same time-dependent velocity field, followed by guidance, fast solvers and flow-map (few-step) models.
- Shen et al. [30] (*TMLR*, 2025) survey efficiency at the algorithm level (training, sampling, compression), system level (hardware and serving) and framework level.
- Melnik et al. [31] (*TMLR*, 2024) survey video diffusion: architectures, temporal-consistency mechanisms, datasets and evaluation.

**State of the art, 2023–2026.** Several trends are visible across these reviews and the primary papers in this chapter. (1) *Transformers in latent space*: DiT-style backbones [8] replaced U-Nets in many large systems, and joint text–image transformers trained with rectified-flow objectives [9] became

a standard text-to-image recipe. (2) *Flow matching as the default objective*: straight-path flow matching [10], [11] is now used alongside or instead of noise prediction, with diffusion understood as one choice of probability path [2], [29]. (3) *Fewer evaluations*: distillation [22], [33], consistency models [12], [32] and average-velocity models [13] bring sampling down to one to four network calls with quality approaching multi-step samplers. (4) *Better evaluation*: the community is moving from Inception-based FID toward richer feature spaces, human studies and alignment metrics [25], [26]. (5) *Beyond images*: video, 3-D, audio and scientific data reuse the same machinery, with video's temporal consistency and cost the main open challenges [31].

**Representation and prediction, 2025–2026.** JiT [42] tests direct clean-image prediction on pixels; RAE [43] tests generation in frozen semantic feature spaces; its text-to-image follow-up [44] examines larger-scale behavior. These studies separate where the model operates from what the network directly predicts. The pretrained examples above [45]–[48] show why component compatibility and temporal reconstruction matter in actual image/video pipelines. Compare these choices under matched compute and data, including representation pretraining.

**Open problems.** Sampling cost at high resolution and long video duration; principled guidance that does not trade away diversity or produce over-saturation; evaluation metrics that agree with people and detect memorization; understanding generalization (why models produce new samples rather than copies of training data); controllability and editing with guarantees; and data provenance, consent and misuse, which are social rather than purely technical questions. The next chapter, ML 09, covers the main alternative route: autoregressive generation over pixels and tokens, and its hybrids with diffusion.

## Key takeaways

- Known forward corruption supplies training targets at sampled noise levels.
- The simplified denoising objective changes the weighting of variational-bound terms; it is not itself the exact likelihood.
- Prediction targets convert under a specified path, but weighting, endpoint behavior and model capacity still affect training.
- A sampler needs a compatible schedule and prediction convention; report network evaluations and full latency.
- Guidance changes the sampled distribution; stronger conditioning can reduce diversity and introduce artifacts.
- Straight conditional training paths do not guarantee straight learned trajectories. Compare representations, solvers and few-step training under matched evaluation protocols.

## Exercises

1. **Recover the clean estimate.** Starting from $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$, derive $\hat{x}_0$ from a noise prediction. If $\epsilon_\theta$ has an error $\delta$, how large is the error in $\hat{x}_0$ at $\bar{\alpha}_t = 0.25$ and at $\bar{\alpha}_t = 10^{-4}$? Then derive the straight-flow velocity from a clean prediction and explain why JiT can output $x$ while using

a velocity loss.

**Hint**

$\hat{x}_0 = (x_t - \sqrt{1-\bar{\alpha}_t}\,\epsilon_\theta)/\sqrt{\bar{\alpha}_t}$, so the error is $\sqrt{1-\bar{\alpha}_t}\,\delta/\sqrt{\bar{\alpha}_t}$: about $1.7\,\delta$ at $\bar{\alpha}_t = 0.25$ and about $100\,\delta$ at $10^{-4}$. At high noise, $\hat{x}_0$ is an average over many plausible images, not one image.

2. **Posterior mean in two forms.** Substitute $x_0 = (x_t - \sqrt{1-\bar{\alpha}_t}\,\epsilon)/\sqrt{\bar{\alpha}_t}$ into $\tilde{\mu}_t$ and show that it equals $\frac{1}{\sqrt{\alpha_t}}(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon)$. Then change `t` in the posterior snippet to 50 and 900 and check that the formula still matches.

**Hint**

Use $\bar{\alpha}_t = \alpha_t\bar{\alpha}_{t-1}$ and $\beta_t = 1 - \alpha_t$. The coefficient of $x_t$ becomes $\frac{\beta_t + \alpha_t(1-\bar{\alpha}_{t-1})}{\sqrt{\alpha_t}(1-\bar{\alpha}_t)} = \frac{1}{\sqrt{\alpha_t}}$ because the numerator equals $1 - \bar{\alpha}_t$.

3. **Count compute.** A latent text-to-image model samples with 30 DDIM steps and classifier-free guidance using separate conditional and unconditional passes, then runs the decoder once. How many denoiser evaluations are needed? How does the count change for a 4-step distilled model that has guidance "baked in"? For the synthetic shape example, change the latent channels to 16 and the pixel patch size to 16. Recompute scalar counts and attention entries. Which savings remain, and which depend on patching?

**Hint**

60 denoiser evaluations plus one decoder pass; batching both CFG branches reduces wall-clock time but not the work. A guidance-distilled 4-step model needs 4 evaluations plus the decoder. Compare quality and diversity, not just the counts.

4. **Time convention.** Diffusion sections use "large $t$ = noisy"; the flow-matching section uses "$t = 0$ noise, $t = 1$ data." Reparameterize the flow with $s = 1 - t$. What is $dx/ds$, and what goes wrong if you only relabel the time input?

**Hint**

$dx/ds = -v(x, 1-s)$. If you feed $s$ to a network trained on $t$ without flipping the sign and the time input, you integrate in the wrong direction and move samples away from the data.

5. **Guidance sweep.** In the CFG snippet, add $w \in \{0.5, 1.5, 5\}$ and also record the fraction of class-1 samples that land closer to the wrong-class modes. Plot mean distance to the nearest mode against $w$. Where does guidance stop helping?

**Hint**

Already at $w = 1$ the exact conditional model puts every sample in class 1, so larger $w$ cannot improve class accuracy here — it only distorts the clusters. With a learned, imperfect conditional model, moderate $w > 1$ can help adherence, which is why it is used in practice.

6. **Sampler ablation.** Hold the trained flow-matching network fixed. Compare Euler with 1, 2, 5, 20 and 100 steps against a second-order Heun step (predict with Euler, then average the velocities at both ends). Use the same starting noise and measure cluster means and spreads. Does the 2-step Heun sampler beat 4-step Euler at the same number of network calls?

**Hint**

Heun uses two network calls per step. Plot samples, not only numbers. One Euler step of a model trained for instantaneous velocity does not test a separately trained one-step model such as a consistency or MeanFlow model.

## References


1. L. Yang, Z. Zhang, Y. Song, S. Hong, R. Xu, Y. Zhao, W. Zhang, B. Cui and M.-H. Yang, "Diffusion Models: A Comprehensive Survey of Methods and Applications," *ACM Computing Surveys*, 2023. arXiv
2. Y. Lipman, M. Havasi, P. Holderrieth, N. Shaul, M. Le, B. Karrer, R. T. Q. Chen, D. Lopez-Paz, H. Ben-Hamu and I. Gat, "Flow Matching Guide and Code," arXiv:2412.06264, 2024. arXiv
3. J. Ho, A. Jain and P. Abbeel, "Denoising Diffusion Probabilistic Models," *NeurIPS*, 2020. arXiv
4. Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon and B. Poole, "Score-Based Generative Modeling through Stochastic Differential Equations," *ICLR*, 2021. arXiv
5. J. Song, C. Meng and S. Ermon, "Denoising Diffusion Implicit Models," *ICLR*, 2021. arXiv
6. J. Ho and T. Salimans, "Classifier-Free Diffusion Guidance," NeurIPS 2021 Workshop on Deep Generative Models and Downstream Applications; arXiv:2207.12598, 2022. arXiv
7. R. Rombach, A. Blattmann, D. Lorenz, P. Esser and B. Ommer, "High-Resolution Image Synthesis with Latent Diffusion Models," *CVPR*, 2022. arXiv
8. W. Peebles and S. Xie, "Scalable Diffusion Models with Transformers," *ICCV*, 2023. arXiv
9. P. Esser, S. Kulal, A. Blattmann et al., "Scaling Rectified Flow Transformers for High-Resolution Image Synthesis," *ICML*, PMLR 235, 2024. PMLR
10. Y. Lipman, R. T. Q. Chen, H. Ben-Hamu, M. Nickel and M. Le, "Flow Matching for Generative Modeling," *ICLR*, 2023. arXiv
11. X. Liu, C. Gong and Q. Liu, "Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow," *ICLR*, 2023. arXiv
12. Y. Song, P. Dhariwal, M. Chen and I. Sutskever, "Consistency Models," *ICML*, 2023. arXiv
13. Z. Geng, M. Deng, X. Bai, J. Z. Kolter and K. He, "Mean Flows for One-step Generative Modeling," *NeurIPS*, 2025. arXiv
14. S. J. D. Prince, *Understanding Deep Learning*, MIT Press, 2023. book site
15. J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan and S. Ganguli, "Deep Unsupervised Learning using Nonequilibrium Thermodynamics," *ICML*, PMLR 37, 2015. PMLR
16. P. Vincent, "A Connection Between Score Matching and Denoising Autoencoders," *Neural Computation*, vol. 23, no. 7, pp. 1661–1674, 2011. doi
17. Y. Song and S. Ermon, "Generative Modeling by Estimating Gradients of the Data Distribution," *NeurIPS*, 2019. arXiv

18. P. Dhariwal and A. Nichol, “Diffusion Models Beat GANs on Image Synthesis,” *NeurIPS*, 2021. NeurIPS
19. A. Q. Nichol and P. Dhariwal, “Improved Denoising Diffusion Probabilistic Models,” *ICML*, PMLR 139, 2021. PMLR
20. T. Karras, M. Aittala, T. Aila and S. Laine, “Elucidating the Design Space of Diffusion-Based Generative Models,” *NeurIPS*, 2022. arXiv
21. C. Lu, Y. Zhou, F. Bao, J. Chen, C. Li and J. Zhu, “DPM-Solver: A Fast ODE Solver for Diffusion Probabilistic Model Sampling in Around 10 Steps,” *NeurIPS*, 2022. arXiv
22. T. Salimans and J. Ho, “Progressive Distillation for Fast Sampling of Diffusion Models,” *ICLR*, 2022. arXiv
23. M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler and S. Hochreiter, “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium,” *NeurIPS* (*NIPS*), 2017. arXiv
24. T. Kynkäänniemi, T. Karras, S. Laine, J. Lehtinen and T. Aila, “Improved Precision and Recall Metric for Assessing Generative Models,” *NeurIPS*, 2019. arXiv
25. G. Stein, J. C. Cresswell, R. Hosseinzadeh et al., “Exposing Flaws of Generative Model Evaluation Metrics and Their Unfair Treatment of Diffusion Models,” *NeurIPS*, 2023. arXiv
26. S. Jayasumana, S. Ramalingam, A. Veit, D. Glasner, A. Chakrabarti and S. Kumar, “Rethinking FID: Towards a Better Evaluation Metric for Image Generation,” *CVPR*, 2024. CVF
27. F.-A. Croitoru, V. Hondru, R. T. Ionescu and M. Shah, “Diffusion Models in Vision: A Survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023. doi
28. H. Cao, C. Tan, Z. Gao, Y. Xu, G. Chen, P.-A. Heng and S. Z. Li, “A Survey on Generative Diffusion Models,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 7, pp. 2814–2830, 2024. doi
29. C.-H. Lai, Y. Song, D. Kim, Y. Mitsufuji and S. Ermon, “The Principles of Diffusion Models,” monograph, arXiv:2510.21890, 2025. arXiv
30. H. Shen, J. Zhang, B. Xiong et al., “Efficient Diffusion Models: A Survey,” *Transactions on Machine Learning Research*, 2025. arXiv
31. A. Melnik, M. Ljubljanac, C. Lu, Q. Yan, W. Ren and H. Ritter, “Video Diffusion Models: A Survey,” *Transactions on Machine Learning Research*, 2024. arXiv
32. C. Lu and Y. Song, “Simplifying, Stabilizing and Scaling Continuous-Time Consistency Models,” *ICLR*, 2025. arXiv
33. A. Sauer, D. Lorenz, A. Blattmann and R. Rombach, “Adversarial Diffusion Distillation,” *ECCV*, 2024. arXiv
34. M. S. Albergo and E. Vanden-Eijnden, “Building Normalizing Flows with Stochastic Interpolants,” *ICLR*, 2023. arXiv
35. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database,” *CVPR*, 2009, pp. 248–255. doi
36. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,” *CVPR*, 2016, pp. 2818–2826. arXiv
37. O. Ronneberger, P. Fischer and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” *MICCAI*, LNCS 9351, pp. 234–241, 2015. arXiv
38. A. Radford, J. W. Kim, C. Hallacy et al., “Learning Transferable Visual Models From Natural Language Supervision,” *ICML*, 2021. arXiv
39. M. Oquab, T. Darcet, T. Moutakanni et al., “DINOv2: Learning Robust Visual Features without

Supervision," *Transactions on Machine Learning Research*, 2024. arXiv
40. B. Efron, "Tweedie's Formula and Selection Bias," *Journal of the American Statistical Association*, vol. 106, no. 496, pp. 1602–1614, 2011. doi
41. A. Dosovitskiy, L. Beyer, A. Kolesnikov et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *ICLR*, 2021. arXiv
42. T. Li and K. He, "Back to Basics: Let Denoising Generative Models Denoise," *CVPR*, 2026 (preprint 2025). paper, publication record, official JiT code.
43. B. Zheng, N. Ma, S. Tong and S. Xie, "Diffusion Transformers with Representation Autoencoders," *ICLR*, 2026 (preprint 2025). proceedings, manuscript.
44. S. Tong, B. Zheng, Z. Wang, B. Tang, N. Ma, E. Brown, J. Yang, R. Fergus, Y. LeCun and S. Xie, "Scaling Text-to-Image Diffusion Transformers with Representation Autoencoders," *CVPR Findings Track*, 2026. manuscript, author publication record.
45. D. Podell, Z. English, K. Lacey, A. Blattmann, T. Dockhorn, J. Müller, J. Penna and R. Rombach, "SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis," *ICLR*, 2024. proceedings, official model card.
46. Black Forest Labs, "FLUX.1 [dev]," official pretrained-model card, 2024. model card.
47. W. Kong, Q. Tian, Z. Zhang et al., "HunyuanVideo: A Systematic Framework For Large Video Generative Models," technical report, arXiv:2412.03603, 2024. report, official release.
48. Team Wan, A. Wang, B. Ai, B. Wen et al., "Wan: Open and Advanced Large-Scale Video Generative Models," technical report, arXiv:2503.20314, 2025. report, official Wan2.1 release.
49. Z. Geng, Y. Lu, Z. Wu, E. Shechtman, J. Z. Kolter and K. He, "Improved Mean Flows: On the Challenges of Fastforward Generative Models," *CVPR*, pp. 30467–30476, 2026. CVF
50. Y. Lu, S. Lu, Q. Sun, H. Zhao, Z. Jiang, X. Wang, T. Li, Z. Geng and K. He, "One-step Latent-free Image Generation with Pixel Mean Flows," *ICML*, PMLR 306, pp. 81265–81277, 2026. PMLR

# ML 09 • Autoregressive Vision: Pixels, Tokens and Scales

Factor an image distribution into conditionals: PixelCNN, VQ-VAE and VQGAN tokenizers, GPT-style token transformers, next-scale prediction, masked generative models, continuous tokens and how they compare with diffusion.

Online edition: https://blog.shyandram.dev/en/learn/ml/autoregressive-vision/

## What you'll learn

- Factor image distributions with the probability chain rule.
- Separate teacher-forced training from sequential sampling.
- Compare pixel prediction with learned visual tokenization.
- Distinguish next-token, next-scale, masked and continuous-token methods.
- Evaluate decoding controls, caches and comparisons with diffusion.

## The big picture

An **autoregressive** (**AR**) generator builds a sample piece by piece. At each step it looks at everything it has produced so far, predicts a probability distribution for the next piece, draws from it, and appends the result. The "piece" can be a single pixel value, an integer token from a learned codebook, a vector of continuous features, or an entire map at the next resolution.

> **Explain like I'm five**
>
> Imagine drawing a comic one panel at a time. Each new panel must fit the panels already drawn. When you practice, you copy real comics, so you always know what the earlier panels look like. When you invent a new comic, you have to live with your own earlier panels, even the slightly wobbly ones. An autoregressive image model works the same way: practice on real images, then create new ones one piece at a time.

Why does this matter? Autoregression is the same recipe that powers large language models: next-token prediction with a transformer. Bringing images into that recipe lets one architecture read and write text and pictures, reuse the engineering of language-model serving, and measure progress with an exact likelihood. The price is that images are not naturally sequences. Someone has to decide **what a token is** and **in which order tokens are produced**. Most of this chapter is about those two decisions.

Xiong et al.'s survey [1] organizes visual AR models into three families by their prediction unit: **pixel-based**, **token-based** and **scale-based**. We follow that map, then add masked (non-left-to-right) and continuous-token models, which blur the boundary with diffusion. The previous chapters on generative models and diffusion and flow matching provide the VAE and diffusion background used here; Prince [9] covers the general probabilistic material at textbook level.

## The chain rule needs no independence assumption

**Plain version.** Any joint distribution can be written as a product of "next element given the earlier ones" distributions. That is a fact of probability, not a modeling assumption.

For an ordered sequence $x = (x_1, \dots, x_N)$,

$$p_\theta(x) = \prod_{i=1}^{N} p_\theta(x_i \mid x_{<i}), \qquad \mathcal{L}_{\text{NLL}}(\theta) = -\sum_{i=1}^{N} \log p_\theta(x_i \mid x_{<i}).$$

Here $x_i$ is the $i$-th element in the chosen order, $x_{<i} = (x_1, \dots, x_{i-1})$ is everything before it, $\theta$ are the network parameters, and $\mathcal{L}_{\text{NLL}}$ is the negative log-likelihood of one training example (averaged over the dataset in practice).

The chain rule is exact for **any** order. Approximation enters only through the model: a finite network, a limited context window, or a restricted form for each conditional. For discrete values, each conditional is usually a softmax over the possible values; binary pixels can use a Bernoulli distribution; continuous features need a density or another conditional generator (we meet one in the MAR section).

An order is a computational choice, not a claim about physics. **Raster order** (left to right, top to bottom) is convenient but arbitrary for a two-dimensional image: the pixel below you is just as related as the pixel to your right. Different orders lead to different conditionals, and a finite model may find some orders easier to learn than others. Figure 1 shows three prediction units used in this chapter.

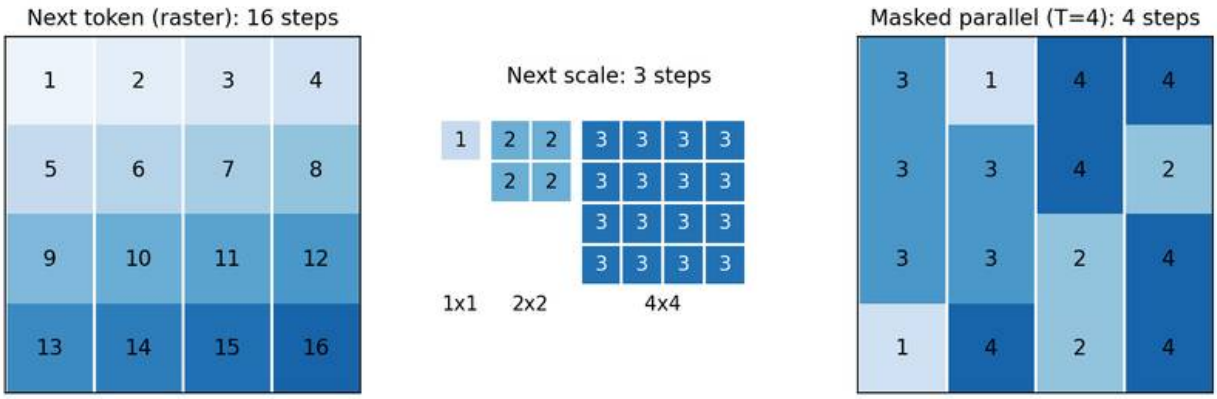



*Figure 1 — The same 16 final tokens produced by three generation schemes. Numbers give the sequential step that produces each token: raster autoregression needs 16 steps, next-scale prediction 3 (one per map), and masked parallel decoding with $T = 4$ needs 4. Diagram only; no model is trained.*

### Measuring an AR model: bits per dimension

Because the chain rule gives an exact $\log p_\theta(x)$, AR models are usually compared by **negative log-likelihood in bits per dimension (bpd)**:

$$\text{bpd}(x) = \frac{-\log_2 p_\theta(x)}{D},$$

where $D$ is the number of discrete values in the image (for example $D = 3 \cdot 32 \cdot 32$ for a $32 \times 32$ RGB image) and $\log_2$ converts nats to bits. A model that assigned every 8-bit value a uniform probability would score exactly 8 bpd; anything lower means the model has learned structure. Bpd is measured on held-out images, so it detects overfitting as well as underfitting.

Two cautions. First, bpd and visual quality can disagree: a model can spend most of its likelihood on imperceptible noise while still making odd global mistakes. Second, for a **token** model the likelihood is over codebook indices, not pixels. A lossy tokenizer throws information away, so a token-level likelihood is not an exact pixel-space likelihood and cannot be compared directly with pixel-model bpd.

## Teacher forcing, causal masks and exposure bias

**Plain version.** Teacher forcing supplies the true prefix during training; sampling uses generated prefixes. Input shifts and masks must hide each prediction's target.

With **teacher forcing**, every conditional in the training loss is computed using the ground-truth previous elements. Since the whole training image is available, all $N$ conditionals can be evaluated in a single forward pass, as long as position $i$ is prevented from reading positions $\geq i$. A causal transformer enforces this with an attention mask; a PixelCNN enforces it with masked convolution kernels.

Two input conventions are common, and the mask must match the convention:

- In a **shifted-input** decoder, feed `[BOS, x1, x2, ...]` and predict `[x1, x2, x3, ...]`. A lower-triangular mask that *includes* the diagonal is correct, because the input at position $i$ is the *previous* element.
- In a model that receives **unshifted** values, the dependency must be **strictly** earlier. Including the diagonal would let the network copy the value it is supposed to predict.

At inference, earlier sampled mistakes enter the context of later steps. This training/inference mismatch is often called **exposure bias**. It does not make maximum likelihood wrong (the factorization is still exact), but it matters in practice for imperfect models and long sequences: a small early error in the layout of an image can push later steps into contexts that never occurred in training.

## Pixel models: PixelRNN and PixelCNN

**Plain version.** Treat each pixel's intensity as a 256-way classification problem whose input is all pixels above and to the left.

PixelRNN and PixelCNN were introduced together by van den Oord, Kalchbrenner and Kavukcuoglu [2]. Both model each 8-bit value with a **256-way softmax** rather than a Gaussian,

so the distribution can be multimodal (a pixel can be "either black or white, but not grey"). Colour is handled by a fixed order inside each pixel, red then green then blue, so that green may depend on red and blue on both. PixelRNN uses two-dimensional LSTM layers (Row LSTM and Diagonal BiLSTM) whose dependency range is potentially unbounded; PixelCNN replaces them with stacked masked convolutions, which have a bounded receptive field but train much faster because all positions are computed in parallel.

### Masked convolutions

A masked convolution multiplies its kernel by a binary mask that zeros the weights at "future" positions. In [2], **mask A** is used only in the first layer: it excludes the centre pixel, so the first layer never sees the value it predicts (and, for colour, only sees channels already predicted). **Mask B** is used in all later layers: it allows the centre position, because after the first layer the centre feature already summarizes only earlier pixels. Formally, for a $k \times k$ kernel with centre $(c, c)$, $c = \lfloor k/2 \rfloor$,

$$M_{uv} = \begin{cases} 1 & u < c, \\ 1 & u = c \text{ and } v < c \text{ (A) or } v \le c \text{ (B)}, \\ 0 & \text{otherwise}, \end{cases}$$

where $u$ indexes kernel rows and $v$ kernel columns. The code below builds both masks, stacks five layers, and uses a gradient to find exactly which input pixels can influence the output at the centre of a $9 \times 9$ image.

```
import torch
from torch import nn
from torch.nn import functional as F

class MaskedConv2d(nn.Conv2d):
    """PixelCNN-style masked convolution for one channel (raster order)."""
    def __init__(self, cin, cout, k, mask_type):
        super().__init__(cin, cout, k, padding=k//2, bias=False)
        mask = torch.ones(k, k)
        c = k//2
        mask[c, c + (mask_type == "B"):] = 0   # centre (type A) and pixels to its right
        mask[c+1:] = 0                         # all rows below
        self.register_buffer("mask", mask)
    def forward(self, x):
        return F.conv2d(x, self.weight*self.mask, padding=self.padding)

torch.manual_seed(0)
layers = [MaskedConv2d(1, 8, 3, "A"), nn.Tanh()]
for _ in range(4):                             # 5 layers: reach of 5 pixels
    layers += [MaskedConv2d(8, 8, 3, "B"), nn.Tanh()]
net = nn.Sequential(*layers, MaskedConv2d(8, 1, 1, "B"))
print(net[0].mask.int())
x = torch.rand(1, 1, 9, 9, requires_grad=True)
net(x)[0, 0, 4, 4].backward()                  # logit for pixel (row 4, col 4)
dep = (x.grad[0, 0].abs() > 0).int()
print(dep)
```

```
later = torch.zeros(9, 9, dtype=torch.bool)
later[4, 4:] = True; later[5:] = True
assert dep[later].sum() == 0                  # never reads itself or later pixels
print("earlier pixels never seen (blind spot):", int(((dep == 0) & ~later).sum()))
```

The assertion passes: no pixel at or after the target influences its logit, so the model is a valid autoregressive factorization. But the printout reveals something else. Six pixels that come *earlier* in raster order, in a staircase above and to the right of the target, have zero gradient. This is the **blind spot** of stacked masked convolutions (Figure 2): each layer's receptive field grows up and to the left, but the region up and to the right is never reached. The model is still correct, just less informed than it could be.

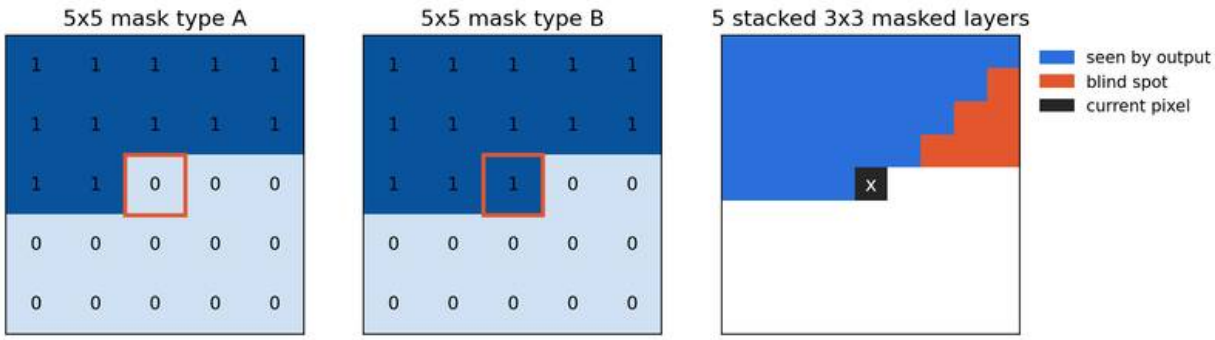



*Figure 2 — Left and centre: mask A excludes the centre weight; mask B keeps it. Right: input pixels that influence the prediction at the dark cell after five stacked 3×3 masked layers, measured with a gradient. Orange cells come earlier in raster order but are never seen: the blind spot.*

The Gated PixelCNN of van den Oord et al. [3] removed the blind spot by splitting the network into two stacks: a **vertical stack** that sees all rows above the current one without restriction and a **horizontal stack** that sees the current row up to the current pixel, combined after each layer. It also replaced ReLU with a gated unit $y = \tanh(W_f * x) \odot \sigma(W_g * x)$, where $*$ is convolution, $\sigma$ the logistic sigmoid and $\odot$ elementwise multiplication, and showed how to condition the model on class labels or embeddings. With these changes PixelCNN matched PixelRNN's likelihood on ImageNet [23] at much lower cost [3].

### Code: a minimal autoregressive model

Masked convolutions are one way to enforce causality; a transformer is not required either. The following complete CPU experiment generates 4×4 binary images whose rows all repeat the same four random bits. A single masked linear layer, whose strictly lower-triangular mask excludes the current pixel and all later ones, learns the conditionals directly.

```
import torch
from torch import nn
from torch.nn import functional as F
```

```
torch.manual_seed(0)
torch.set_num_threads(1)
class BinaryAR(nn.Module):
    def __init__(self, n=16):
        super().__init__()
        self.weight = nn.Parameter(torch.zeros(n, n))
        self.bias = nn.Parameter(torch.zeros(n))
        self.register_buffer("mask", torch.tril(torch.ones(n, n), diagonal=-1))
    def forward(self, x):
        return F.linear(x, self.weight*self.mask, self.bias)

model = BinaryAR()
opt = torch.optim.Adam(model.parameters(), lr=.04)
for step in range(600):
    first_row = torch.randint(2, (128, 4)).float()
    x = first_row.repeat(1, 4)
    loss = F.binary_cross_entropy_with_logits(model(x), x)
    opt.zero_grad(); loss.backward(); opt.step()

with torch.no_grad():
    a = torch.rand(2, 16)
    b = a.clone(); b[:, 8:] = torch.rand(2, 8)
    assert torch.allclose(model(a)[:, :9], model(b)[:, :9])
    samples = torch.zeros(8, 16)
    for i in range(16):
        p = model(samples)[:, i].sigmoid()
        samples[:, i] = torch.bernoulli(p)
    rows = samples.reshape(-1, 4, 4)
    print("one generated image:\n", rows[0].int())
    print("repeat agreement:", (rows[:, 1:] == rows[:, :1]).float().mean().item())
```

The assertion checks causality: changing positions 8 onward cannot alter the logits for positions 0 through 8, because each of those outputs reads only earlier inputs. Training then drives the agreement between later rows and the first row to 1.0, while the first row stays random (the model must not memorize one particular row).

The true distribution has 16 equally likely images, so its entropy is $\log_2 16 = 4$ bits per image, or 0.25 bits per pixel. Later rows contain no new uncertainty. A model that treated all 16 pixels as independent fair coins would need 16 bits per image. The gap between 16 and 4 bits is exactly the structure an autoregressive model captures and an independent-pixel model cannot.

### Why pixel models stopped scaling

A $256 \times 256$ RGB image has $3 \cdot 256 \cdot 256 = 196{,}608$ values, so pixel-level sampling needs about two hundred thousand sequential network evaluations. Most of the likelihood is also spent on fine, locally predictable detail, while the global layout that people notice is a small fraction of the bits. Image GPT [11] showed that a plain sequence transformer trained to predict pixels autoregressively on low-resolution images learns features that transfer well under linear probing, but

the pixel route stays expensive at higher resolutions. The next step was to learn a shorter, more meaningful sequence.

## Discrete visual tokens: VQ-VAE and VQGAN

**Plain version.** Train a separate network that compresses an image into a small grid of "words" from a learned dictionary, then do autoregression over the words instead of the pixels.

### VQ-VAE

VQ-VAE [4] has an encoder that produces a feature vector $z_e(x) \in \mathbb{R}^d$ at each position of a downsampled grid, a **codebook** $\{e_1, \dots, e_K\} \subset \mathbb{R}^d$ of $K$ learned vectors, and a decoder. Each feature is replaced by its nearest code:

$$k^* = \arg\min_k \|z_e(x) - e_k\|_2^2, \qquad z_q(x) = e_{k^*}.$$

The image is now represented by the integer grid of $k^*$ values. Training minimizes

$$\mathcal{L}_{\text{VQ}} = \underbrace{\|x - D(z_q)\|^2}_{\text{reconstruction}} + \underbrace{\|\text{sg}[z_e] - e_{k^*}\|^2}_{\text{codebook}} + \beta \underbrace{\|z_e - \text{sg}[e_{k^*}]\|^2}_{\text{commitment}},$$

where $D$ is the decoder, $\text{sg}[\cdot]$ is the **stop-gradient** operator (identity in the forward pass, zero gradient in the backward pass), and $\beta$ is a small weight on the commitment term (the snippet below uses 0.25). The reconstruction term is written as squared error here; any reconstruction likelihood can take its place.

The three terms do three jobs. The **codebook** term moves the chosen code toward the encoder output, like a k-means update. The **commitment** term pulls the encoder toward its code so that encoder outputs do not drift far from the codebook. The `argmin` has no useful gradient, so the reconstruction gradient reaches the encoder through the **straight-through estimator**: in the forward pass use $z_q$, in the backward pass copy the gradient from $z_q$ to $z_e$ as if quantization were the identity. In code this is the one-liner `z_q = z_e + (e - z_e).detach()`.

The toy below trains a 2-D "encoder", codebook and decoder on four clusters, with eight codes available:

```
import torch
from torch import nn
from torch.nn import functional as F

def quantize(z_e, codebook, beta=0.25):
    """z_e: (n, d) encoder outputs; codebook: (K, d). Returns z_q with straight-through
     ↪  gradient."""
    dist = torch.cdist(z_e, codebook)            # (n, K) Euclidean distances
    idx = dist.argmin(1)                         # nearest code index per vector
    e = codebook[idx]
    codebook_loss = F.mse_loss(e, z_e.detach())  # moves codes toward encoder outputs
```

```
    commit_loss = F.mse_loss(z_e, e.detach())    # keeps the encoder near its code
    z_q = z_e + (e - z_e).detach()               # forward: e; backward: identity
    return z_q, idx, codebook_loss + beta*commit_loss

torch.manual_seed(0)
torch.set_num_threads(1)
centres = torch.tensor([[2., 0.], [-2., 0.], [0., 2.], [0., -2.]])
encoder = nn.Linear(2, 2)                        # toy "encoder" and "decoder"
decoder = nn.Linear(2, 2)
codebook = nn.Parameter(torch.randn(8, 2)*.1)    # K = 8 codes, 4 are enough
opt = torch.optim.Adam([*encoder.parameters(), *decoder.parameters(), codebook], lr=.03)
for step in range(800):
    x = centres[torch.randint(4, (256,))] + .1*torch.randn(256, 2)
    z_q, idx, vq_loss = quantize(encoder(x), codebook)
    loss = F.mse_loss(decoder(z_q), x) + vq_loss
    opt.zero_grad(); loss.backward(); opt.step()

with torch.no_grad():
    x = centres.repeat_interleave(100, 0) + .1*torch.randn(400, 2)
    z_q, idx, _ = quantize(encoder(x), codebook)
    print("reconstruction MSE:", round(F.mse_loss(decoder(z_q), x).item(), 4))
    print("codes used:", idx.unique().numel(), "of", len(codebook))
    print("codes per cluster:", [idx[i*100:(i+1)*100].unique().tolist() for i in range(4)])
z = torch.randn(3, 2, requires_grad=True)
z_q, _, _ = quantize(z, codebook.detach())
z_q.sum().backward()
print("straight-through gradient:", z.grad[0].tolist())
```

It prints a reconstruction MSE of about 0.0096, uses **4 of the 8 codes** (one per cluster), and shows a straight-through gradient of `[1.0, 1.0]`. Two lessons hide in those numbers. The reconstruction error is almost exactly the within-cluster noise variance $0.1^2 = 0.01$: quantization has kept *which cluster* and discarded *where inside the cluster*. And half the codebook is **dead**: codes that are never selected receive no codebook gradient and never move (Figure 3). In real tokenizers, low codebook usage wastes capacity and is a standard failure to monitor.

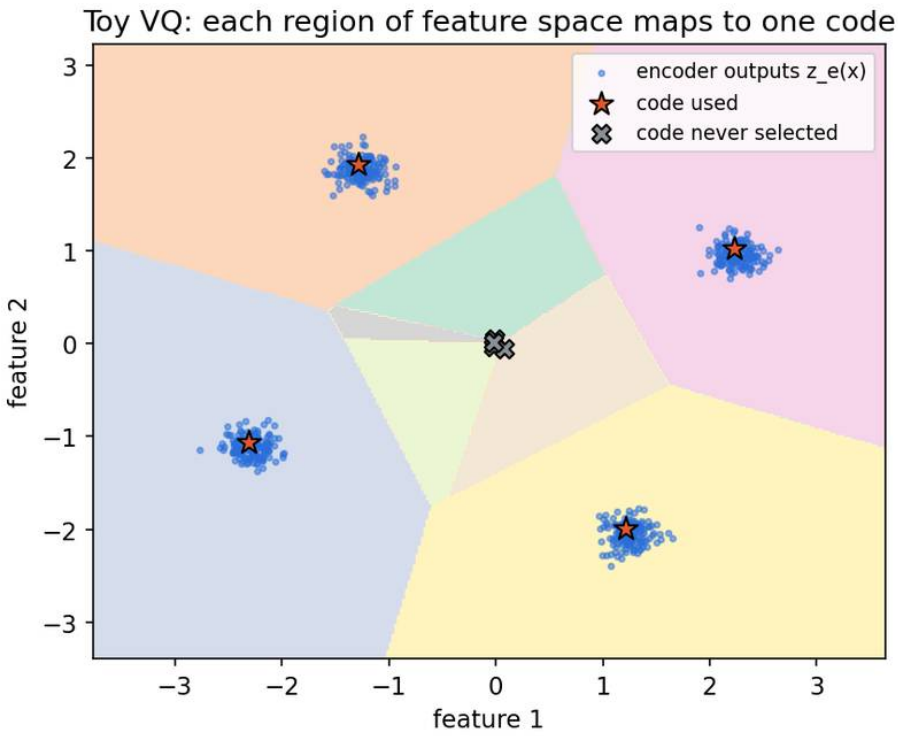



*Figure 3 — Toy vector quantization. Each coloured region is the set of features mapped to one code. Four codes (stars) are used, one per cluster; four (crosses) were never selected and stayed near their initialization.*

### Two stages: a tokenizer, then a prior

A VQ-VAE alone is an autoencoder; it does not by itself say which token grids are likely. Generation therefore has **two stages** (Figure 4). Stage 1 trains the tokenizer. Stage 2 freezes it, converts every training image into its index grid, and trains an autoregressive model $p_\psi(k_1, \ldots, k_M)$ over the $M$ indices. To sample, draw an index grid from the prior and decode it. In [4] the prior was a PixelCNN over codes; VQ-VAE-2 [10] used a hierarchy of two code maps (a coarse top level and a finer bottom level) with PixelCNN-style priors and produced diverse, high-fidelity ImageNet samples.

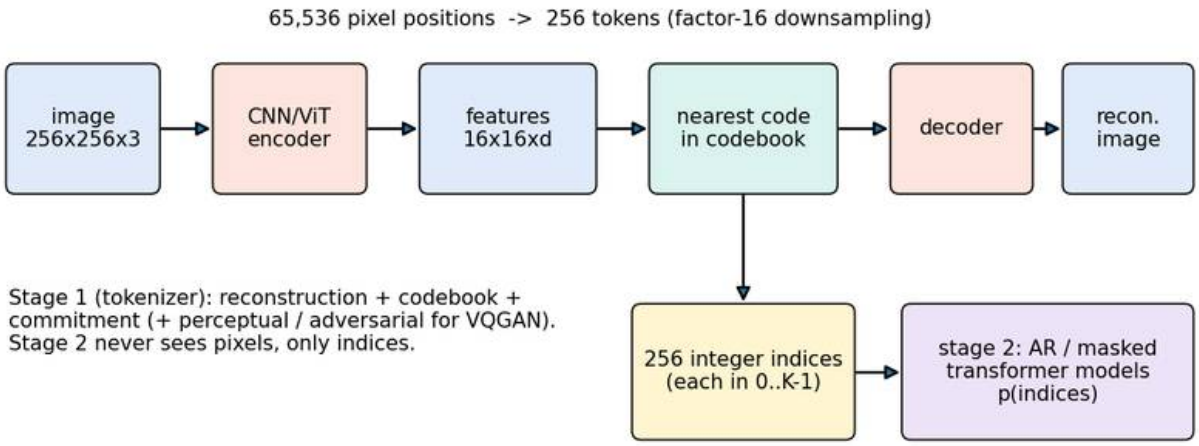



*Figure 4 — Two-stage generation. The tokenizer (top row) is trained for reconstruction; the prior (bottom right) only ever sees integer indices. With factor-16 downsampling, a 256×256 image becomes 256 tokens instead of 65,536 pixel positions.*

The arithmetic explains the appeal. If a tokenizer downsamples a $256 \times 256$ image by 16 in each direction, the prior sees $16 \times 16 = 256$ tokens rather than 65,536 pixel positions (or 196,608 colour values). Self-attention cost grows with the square of sequence length, so this is the difference between feasible and infeasible.

### VQGAN: perceptual tokens

Esser, Rombach and Ommer's VQGAN [5] kept the two-stage design but trained the tokenizer with a **perceptual loss** and a **patch-based adversarial loss** in addition to reconstruction, and modelled the token grid with a transformer. The adversarial and perceptual terms let the decoder produce sharp, realistic texture from very few tokens, because the tokens no longer need to encode every pixel exactly; they need to encode enough for the decoder to produce *a* plausible version. This is why VQGAN-style tokenizers became the default front end for token-based generation. Note the division of labour: the **tokenizer** uses adversarial training, but the **prior** is an ordinary autoregressive model trained by maximum likelihood.

### Codebook problems and alternatives

Real tokenizers face three recurring problems: dead codes (as in the toy), a gap between what the tokenizer reconstructs and what the prior can model, and loss of small details such as text and thin lines. Several designs respond to them:

- **ViT-VQGAN** [12] uses a Vision Transformer encoder/decoder, looks codes up in a low-dimensional **factorized** space and $\ell_2$**-normalizes** codes and features, which improved codebook usage and reconstruction.

- **RQ-VAE** [15] uses **residual quantization**: each position is represented by a short stack of codes, each one quantizing what the previous ones missed. This allows a coarser spatial grid (shorter sequence) at similar fidelity.
- **FSQ** [16] (finite scalar quantization) projects to a handful of dimensions and rounds each to a few fixed levels. The implicit codebook is the product of the levels; there is no learned codebook to collapse and no commitment loss.

Compression always creates a bottleneck. Measure codebook usage, reconstruction quality and the generator's quality **separately**: a generator cannot produce detail its tokenizer cannot represent.

*Online only: VQ-VAE — discrete codes and a separately learned prior — https://blog.shyandram.dev/en/learn/ml/autoregressive-vision/papers/vq-vae–discrete-codes-and-a-separately-learned-prior/*

## Autoregressive transformers over tokens

**Plain version.** A decoder-only Transformer can predict the next visual token as it predicts a text token. The visual tokenizer and decoder determine what those tokens represent.

### From pixels to tokens

Image GPT [11] applied a GPT-style transformer directly to sequences of low-resolution pixels. Its main finding was about **representation learning**: features from a model trained only to predict the next pixel transfer well when a linear classifier is trained on top. Working on tokens instead of pixels shortens sequences enough to generate at higher resolution. The VQGAN transformer [5] is the classic example: a GPT-like model over VQGAN indices in raster order, which let a transformer synthesize high-resolution images that would be far too long as pixel sequences.

### Text-to-image as translation: Parti

Parti [13] treats text-to-image generation like machine translation. A transformer **encoder** reads the text prompt; a transformer **decoder** autoregressively generates ViT-VQGAN image tokens [12] conditioned on it. The report scales this sequence-to-sequence model up to 20 billion parameters and introduces PartiPrompts, a benchmark of more than 1,600 English prompts spanning many categories of content.

### Language-model architecture, unchanged: LlamaGen

LlamaGen [14] asks whether a vanilla language-model architecture, with no image-specific inductive biases, can generate images well when scaled. It uses a Llama-style [24] decoder (RMSNorm pre-normalization, SwiGLU feed-forward layers, 2-D rotary position embeddings) over the tokens of a factor-16 VQ tokenizer with $K = 16{,}384$ codes of dimension 8. The class label or projected text embedding is supplied as a **prefix** token, and generation proceeds in raster order.

The paper releases class-conditional models from 111M to 3.1B parameters and reports that standard LLM serving software (vLLM) speeds up inference. Its broader message is architectural reuse: an image generator can share almost everything with a text model except the tokenizer.

### Classifier-free guidance for autoregressive models

Classifier-free guidance (CFG) [18] was introduced for diffusion models, but it transfers directly to AR logits. During training, the condition $c$ is randomly replaced by a learned **null** embedding so that one network learns both conditional and unconditional predictions. At each sampling step, compute both sets of logits and combine them:

$$\ell_g = \ell_u + s\,(\ell_c - \ell_u),$$

where $\ell_c$ are the next-token logits with the condition, $\ell_u$ the logits with the null condition, and $s$ the guidance scale; $s = 1$ recovers the ordinary conditional model. This is the form LlamaGen uses [14]. Larger $s$ usually improves prompt or class adherence and sample sharpness at the cost of diversity. It also doubles the per-step compute unless the two passes are batched, so step counts alone understate cost.

## Next-scale prediction (VAR)

**Plain version.** Instead of predicting the next token in a fixed reading order, predict the next, sharper version of the whole image, conditioned on all coarser versions.

Raster order forces a strange dependency: the top-left of an image is generated without any knowledge of the bottom-right. VAR (Visual AutoRegressive modeling) [6] argues that this one-directional order conflicts with the two-directional correlations of image tokens and with spatial locality, and replaces it with a **coarse-to-fine** order over token maps $r_1, \ldots, r_K$ of increasing resolution:

$$p(r_1, \ldots, r_K) = \prod_{k=1}^{K} p(r_k \mid r_1, \ldots, r_{k-1}),$$

where $r_k$ is an $h_k \times w_k$ grid of token indices, $r_1$ is $1 \times 1$ and $r_K$ has the full latent resolution. All tokens of $r_k$ are predicted **in parallel** given the coarser maps, so the number of sequential steps equals the number of scales, not the number of tokens.

The tokenizer is part of the method. VAR's multi-scale VQ encoder works **residually**: starting from the encoder feature map $f$, for $k = 1, \ldots, K$ it downsamples $f$ to $h_k \times w_k$, quantizes it to get $r_k$ (with one codebook shared across scales), upsamples the looked-up codes back to full resolution, passes them through a small convolution $\phi_k$, and subtracts the result from $f$ before moving to the next scale [6]. Each scale therefore encodes what the coarser scales have not explained. Simply resizing ordinary token maps would not reproduce this.

During training, a **block-wise causal** attention mask lets every token of $r_k$ attend to all tokens of $r_{\le k}$ but none of the finer scales; at inference a KV cache stores the coarser scales. VAR uses

CFG as well. On class-conditional ImageNet 256 × 256 the paper reports that VAR improves its AR baseline's FID from 18.65 to 1.73, with faster inference, and that its models follow power-law scaling trends similar to those of language models [6]. VAR received a NeurIPS 2024 Best Paper award. As always, those numbers apply to the paper's data, resolution and training protocol.

Parallel prediction within a scale is itself an assumption: tokens in $r_k$ are sampled independently given $r_{<k}$. The masked-model section shows what that assumption can cost on a toy problem.

## Masked generative models (MaskGIT and Muse)

**Plain version.** Start with a grid of blanks. At each round, guess every blank at once, keep only the guesses you are most confident about, and blank out the rest again.

MaskGIT [8] uses the same VQ tokens as an AR model, but a **bidirectional** transformer that sees every position. Training is a masked-token objective: choose a mask ratio, replace that fraction of tokens with a special `[MASK]` token, and minimize cross-entropy on the masked positions,

$$\mathcal{L}_{\text{mask}} = -\mathbb{E}\Big[\sum_{i \in \mathcal{M}} \log p_\theta(y_i \mid y_{\bar{\mathcal{M}}})\Big],$$

where $y$ is the token grid, $\mathcal{M}$ is the set of masked positions and $y_{\bar{\mathcal{M}}}$ the unmasked tokens. The mask ratio is drawn as $\gamma(r)$ with $r \sim \mathcal{U}(0,1)$ and a **mask schedule** $\gamma$ that decreases from 1 to 0.

Generation runs for $T$ iterations. At iteration $t$ the model predicts all masked tokens in parallel and samples them; each sample's probability serves as its **confidence**. The number of tokens that remain masked after the iteration is

$$n_t = \lceil \gamma(t/T)\, N \rceil,$$

with $N$ the total number of tokens, so the $n_t$ least confident new tokens are re-masked and everything else is fixed. MaskGIT compares several schedules and selects the **cosine** schedule $\gamma(r) = \cos(\pi r/2)$, which fixes few tokens early (when the model knows little) and many late; it generates 256 × 256 images in 8 iterations with a 16×16 token grid and a 1,024-entry codebook [8]. Muse [17] scales the same idea to text-to-image generation, conditioning the masked-token transformer on text embeddings from a pre-trained large language model.

MaskGIT is **not** ordinary left-to-right chain-rule autoregression: the order is chosen on the fly by confidence, and several tokens are sampled in the same step. Those simultaneous tokens are drawn **independently** given the current context. The toy below shows when that matters. It trains a small bidirectional network on the repeated-row $4 \times 4$ images from earlier and decodes with the cosine schedule.

```
import math
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(0)
torch.set_num_threads(1)
```

```
N, MASK = 16, 2                                      # 16 binary tokens; state 2 = [MASK]
gamma = lambda r: math.cos(math.pi*r/2)             # cosine schedule: fraction still masked
net = nn.Sequential(nn.Flatten(), nn.Linear(3*N, 128), nn.ReLU(), nn.Linear(128, N))

def data(n):                                         # 4x4 images whose rows repeat
    return torch.randint(2, (n, 4)).repeat(1, 4)

def logits(tokens):                                  # sees every position: bidirectional
    return net(F.one_hot(tokens, 3).float())

opt = torch.optim.Adam(net.parameters(), lr=3e-3)
for step in range(1500):
    x = data(256)
    ratio = torch.tensor([gamma(r) for r in torch.rand(256).tolist()])
    n_mask = (ratio*N).ceil().clamp(min=1)
    masked = torch.rand(256, N).argsort(1) < n_mask[:, None]   # random positions
    inp = torch.where(masked, MASK, x)
    loss = F.binary_cross_entropy_with_logits(logits(inp)[masked], x[masked].float())
    opt.zero_grad(); loss.backward(); opt.step()

@torch.no_grad()
def generate(n, T=4):
    tokens = torch.full((n, N), MASK)
    for t in range(1, T+1):
        p = logits(tokens).sigmoid()
        sample = torch.bernoulli(p).long()
        conf = torch.where(sample == 1, p, 1-p)              # prob. of the drawn value
        still = tokens == MASK
        conf = torch.where(still, conf, torch.inf)           # never re-mask fixed tokens
        proposal = torch.where(still, sample, tokens)
        n_keep_masked = math.floor(gamma(t/T)*N)             # 0 after the last step
        order = conf.argsort(1)                              # least confident first
        remask = torch.zeros_like(still)
        remask.scatter_(1, order[:, :n_keep_masked], True)
        tokens = torch.where(remask, MASK, proposal)
    return tokens

for T in (1, 2, 4, 8, 16):
    rows = generate(1000, T).reshape(-1, 4, 4)
    agree = (rows[:, 1:] == rows[:, :1]).float().mean().item()
    valid = (rows == rows[:, :1]).all(2).all(1).float().mean().item()
    print(f"T={T:2d}  repeat agreement={agree:.3f}  valid images={valid:.3f}")
```

(The code uses `floor` rather than the paper's ceiling so that the count reaches exactly zero at $t = T$ in this tiny example.) The output is:

```
T= 1  repeat agreement=0.511  valid images=0.000
T= 2  repeat agreement=0.812  valid images=0.100
T= 4  repeat agreement=0.908  valid images=0.360
T= 8  repeat agreement=0.998  valid images=0.982
T=16  repeat agreement=1.000  valid images=0.999
```

With $T = 1$ every token is sampled independently from the all-mask context, so rows almost never agree. With more iterations, the first few tokens are fixed early and later ones can condition on them; by $T = 8$ nearly every sample is valid. On natural images the dependence between tokens decoded in the same round is weaker, which is why a handful of iterations works, but the trade-off between **fewer steps** and **more independence assumptions** is real (Figure 5).

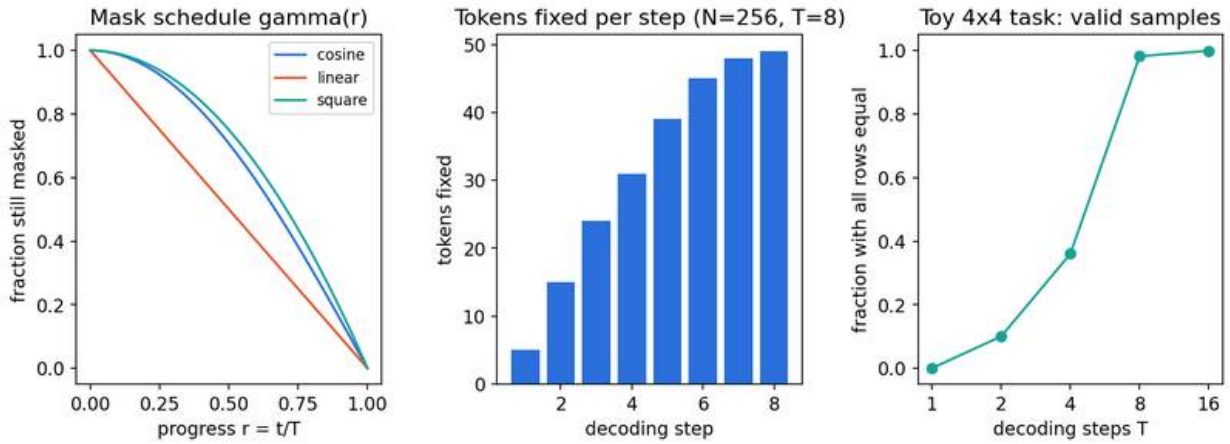



*Figure 5 — Left: candidate mask schedules $\gamma(r)$. Centre: with the cosine schedule, 256 tokens and $T = 8$, few tokens are fixed early and many late. Right: on the toy repeated-row task, the fraction of valid samples grows with the number of decoding iterations because fewer tokens are sampled independently in the same step.*

## Continuous tokens and diffusion conditionals (MAR)

**Plain version.** Autoregression defines the conditioning order; a small diffusion model can represent each continuous-token conditional distribution.

Vector quantization is lossy and awkward to train. Li et al. [7] observe that autoregression needs a way to model $p(x_i \mid x_{<i})$, not necessarily a categorical distribution. They let the AR (or masked AR) transformer output a conditioning vector $z_i$ for each continuous-valued token $x_i$, and train a small MLP denoiser $\epsilon_\theta$ with a **diffusion loss**:

$$\mathcal{L}(z_i, x_i) = \mathbb{E}_{\epsilon,t}\|\epsilon - \epsilon_\theta(x_{i,t} \mid t, z_i)\|^2, \qquad x_{i,t} = \sqrt{\bar{\alpha}_t}\, x_i + \sqrt{1-\bar{\alpha}_t}\, \epsilon,$$

where $\epsilon \sim \mathcal{N}(0, I)$, $t$ is a diffusion time step and $\bar{\alpha}_t$ the noise schedule from ML 08. The gradient also flows into the transformer through $z_i$. To sample a token, run the reverse diffusion process of the small denoiser conditioned on $z_i$; a temperature scales the injected noise. The paper's main variant, **MAR** (masked autoregressive), uses a random order and predicts a set of tokens per step, combining the outer structure of MaskGIT with an inner continuous sampler [7].

Consequently, "AR versus diffusion" is not an either/or choice. A system can decide the order and grouping of tokens autoregressively and sample each group by diffusion. Likewise, a language decoder can autoregressively generate *text about* an image without generating any pixels;

the unifying idea is factorization, not a particular output type.

## Sampling, temperature and caching

**Plain version.** The trained model gives a distribution at every step; the sampler decides how adventurous to be when drawing from it, and the cache avoids recomputing the past.

For logits $\ell_k$ over $K$ possible values, **temperature** $T$ rescales the distribution:

$$p_T(k) = \frac{\exp(\ell_k / T)}{\sum_{j=1}^{K} \exp(\ell_j / T)}.$$

Lower $T$ concentrates probability on the top values; larger $T$ spreads it (Figure 6). **Top-$k$** sampling keeps only the $k$ highest logits; **top-$p$** (nucleus) sampling keeps the smallest set of values whose probability mass reaches $p$. Both renormalize what remains.

```
import torch

def filtered_probs(logits, temperature=1.0, top_k=None, top_p=None):
    """Return the (renormalized) distribution actually sampled from."""
    logits = logits / temperature
    if top_k is not None:                                # keep the k largest logits
        kth = logits.topk(top_k).values[..., -1, None]
        logits = logits.masked_fill(logits < kth, -torch.inf)
    if top_p is not None:                                # smallest set with mass >= p
        sorted_logits, order = logits.sort(descending=True)
        cum = sorted_logits.softmax(-1).cumsum(-1)
        drop = cum - sorted_logits.softmax(-1) >= top_p  # mass before this token already >=
         ↪ p
        sorted_logits = sorted_logits.masked_fill(drop, -torch.inf)
        logits = torch.empty_like(logits).scatter(-1, order, sorted_logits)
    return logits.softmax(-1)

logits = torch.tensor([2.0, 1.0, 0.0, -1.0])
for name, kw in [("T=1", {}), ("T=0.5", {"temperature": .5}), ("T=2", {"temperature": 2.}),
                 ("top-k=2", {"top_k": 2}), ("top-p=0.8", {"top_p": .8})]:
    p = filtered_probs(logits, **kw)
    entropy = -(p[p > 0]*p[p > 0].log2()).sum().item()
    print(f"{name:10s}", [round(v, 3) for v in p.tolist()], f"entropy={entropy:.2f} bits")
```

The model's own distribution ($T = 1$) has 1.37 bits of entropy; $T = 0.5$ lowers it to 0.66 bits and $T = 2$ raises it to 1.80 bits. Top-$k = 2$ and top-$p = 0.8$ both keep the two most likely values here (the first value has mass 0.644, which is below 0.8, so the second is needed). Every one of these changes the distribution you sample from, so a sampler's visual quality should not be confused with the model's likelihood.

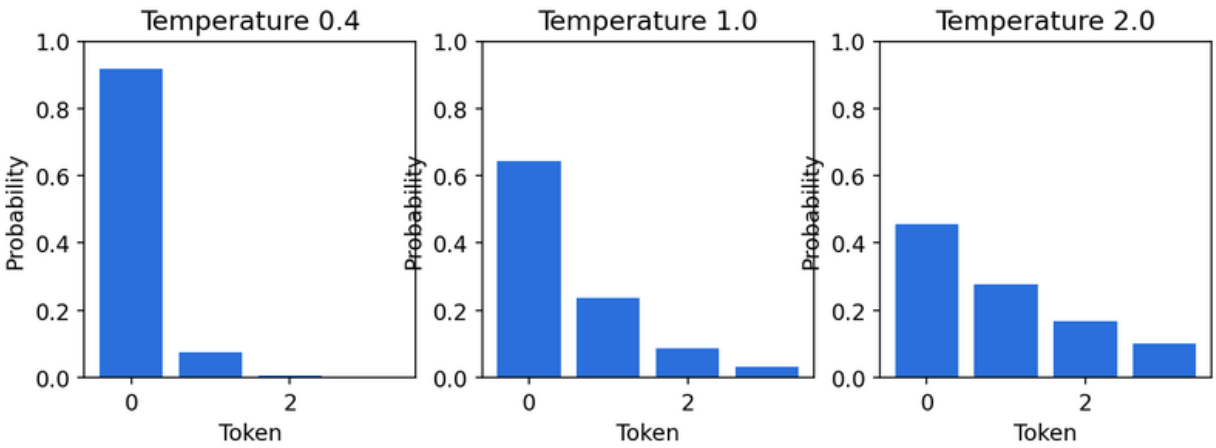



*Figure 6 — Temperature changes diversity at each conditional. It cannot recover information that a tokenizer has already lost.*

### The KV cache

In a causal transformer, the keys and values of earlier tokens do not change when a new token is appended. A **KV cache** stores them, so each new step computes the query, key and value of the newest token only and attends to the cache:

```
import torch
from torch.nn import functional as F

torch.manual_seed(0)
d, n = 8, 6
Wq, Wk, Wv = (torch.randn(d, d) for _ in range(3))
x = torch.randn(n, d)                                  # embeddings of n tokens

# Full causal attention over the whole sequence (what training computes).
q, k, v = x@Wq, x@Wk, x@Wv
mask = torch.ones(n, n).tril().bool()
scores = (q@k.T/d**.5).masked_fill(~mask, -torch.inf)
full = scores.softmax(-1)@v

# Incremental decoding: each step computes q, k, v for the newest token only.
K_cache, V_cache, outputs = [], [], []
for i in range(n):
    K_cache.append(x[i]@Wk); V_cache.append(x[i]@Wv)
    K, V = torch.stack(K_cache), torch.stack(V_cache)
    att = F.softmax((x[i]@Wq)@K.T/d**.5, -1)        # attends to tokens 0..i
    outputs.append(att@V)
assert torch.allclose(full, torch.stack(outputs), atol=1e-5)
print("cached decoding matches full causal attention")
print("dot products per step:", [i+1 for i in range(n)], "total:", n*(n+1)//2)
```

The assertion confirms that cached decoding gives exactly the training-time causal outputs (for a single layer here; real models cache every layer and head). The cache removes recomputation but not the sequential dependency, and it grows linearly with sequence length. Step $i$ still attends to $i$ cached keys, so total attention work over a sequence of length $n$ is $n(n+1)/2$, quadratic in $n$. This is why shorter sequences (better tokenizers) and fewer sequential steps (next-scale or masked decoding) matter so much.

For video, tokens can be ordered over space and time or grouped into chunks. Caching, compression and temporal ordering then dominate cost, and object consistency and long-duration drift need their own evaluation.

## Autoregressive versus diffusion

**Plain version.** Both families turn one hard generation problem into many easier ones. AR splits the *image* into pieces; diffusion splits the *noise level* into steps.

| Aspect | Autoregressive (token / scale) | Masked generative | Diffusion / flow (ML 08) |
|---|---|---|---|
| What is generated per step | One token (or one scale) | A confidence-chosen set of tokens | A refinement of the whole sample |
| Sequential steps | Tokens or scales | Fixed, typically few ($T$) | Solver steps (NFE) |
| Typical representation | Discrete codes (or continuous with a diffusion head [7]) | Discrete codes | Continuous pixels or latents |
| Training objective | Exact NLL (cross-entropy) | Masked cross-entropy | Denoising / velocity regression (ELBO-related) |
| Likelihood | Exact over tokens | Not a single exact chain-rule likelihood | Bound or ODE-based estimate |
| Natural strengths | Shares architecture and serving with LLMs; mixes with text | Few steps; easy inpainting and editing | Strong image quality; flexible guidance |
| Typical weaknesses | Long sequential decoding; tokenizer bottleneck; error accumulation | Independence within a step | Many network evaluations unless distilled |

Three points keep this table honest. First, the families mix: MAR [7] puts a diffusion model inside an AR loop, and latent diffusion also depends on a learned tokenizer-like autoencoder. Second, claims that one family "beats" the other (as in LlamaGen's title [14] or VAR's comparison with DiT [6]) hold for a stated benchmark, tokenizer, model size and guidance setting, not universally. Third, cost should be compared in network evaluations, guidance passes, sequence length and actual latency, not in "steps" alone.

## Modern view

The field has moved from raw pixels to learned representations, from fixed raster order to flexible prediction units, and from separate image generators to models that read and write both text and

images. Several surveys map this territory:

- **Xiong et al. [1]** (TMLR) is the most direct survey of visual autoregression. Its taxonomy by representation (pixel-, token- and scale-based models) is the one used in this chapter, and it covers applications in image, video, 3D and multimodal generation as well as embodied and medical settings.
- **Bond-Taylor et al. [19]** (TPAMI 2022) compare VAEs, GANs, normalizing flows, energy-based and autoregressive models side by side, including the trade-offs in run time, sample diversity and architectural constraints. It places AR models in the wider generative-model landscape: exact likelihood and stable training, paid for with slow sequential sampling.
- **Li et al. [20]** (TPAMI 2026) survey discrete tokenization for multimodal LLMs, organizing vector-quantization variants and their use in classical, single-modality LLM and multimodal LLM systems, and discussing codebook collapse and unstable gradient estimation.
- **Jia et al. [22]** survey discrete tokenizers across generation, comprehension, recommendation and retrieval, dissecting tokenizers into their sub-modules and mechanisms before organizing methods by application, and listing limitations and future directions.
- **Zhao et al. [21]** survey unified multimodal understanding-and-generation models, grouping them into diffusion-based, autoregressive-based and hybrid AR–diffusion designs, which is where most 2025–2026 activity sits.

**State of the art (2023–2026).** Three lines stand out in the verified literature. (i) *Scaling plain AR transformers over better tokenizers*: LlamaGen [14] shows that a language-model architecture with almost no image-specific design is competitive on class-conditional ImageNet when the tokenizer and training are good. (ii) *Changing the prediction unit*: VAR's next-scale prediction [6] and masked decoding in MaskGIT and Muse [8], [17] reduce the number of sequential steps; MAR [7] removes vector quantization by using a diffusion head per token. (iii) *Unified multimodal models*: the same next-token machinery is used to generate text and image tokens in one model, with tokenizers designed to serve both understanding and generation [20], [21]. Our interpretation is that choosing the **representation and prediction unit** now matters at least as much as choosing the backbone.

**Open problems.** The surveys repeatedly point to the following:

- *Tokenizer quality.* Reconstruction caps generation quality; small text, faces and fine geometry are the usual casualties. Better codebook usage and semantic structure without losing detail remain active topics [20], [22].
- *Discrete versus continuous tokens.* Discrete tokens fit LLM machinery; continuous tokens reconstruct better but need a continuous conditional model such as a diffusion head [7].
- *Speed.* Sequential decoding is latency-bound. Next-scale, masked and parallel decoding trade steps for independence assumptions.
- *Evaluation.* Token likelihood, FID-type metrics, prompt adherence and human preference measure different things; a lossy tokenizer's token likelihood is not a pixel likelihood.
- *Video and long contexts.* Token counts explode over time, and temporal consistency, object permanence and drift need dedicated evaluation [1].

## Key takeaways

- Autoregression is a distribution factorization; it does not prescribe a Transformer architecture.
- Teacher forcing supplies true prefixes; causal masks and shifted inputs must prevent target leakage.
- Tokenization shortens sequences while constraining the decoder's representational capacity.
- Token, scale and masked prediction use different factorizations or update rules; compare their dependency assumptions.
- Temperature and truncation change sampling probabilities; caching reuses computation without changing the intended conditional distribution.
- Compare autoregressive and diffusion pipelines using matched data, representations, guidance and cost.

## Exercises

1. **Leak the answer.** Change the `BinaryAR` mask to include the diagonal (`diagonal=0`). Explain why the training loss can fall almost to zero while sampling breaks.

   **Hint**

   Each output can now read its own target and simply copy it, so the training loss collapses. At sampling time the current position is still 0 (not yet drawn) when its logit is computed, so the learned "copy" rule produces nonsense. The causality assertion in the snippet would also fail.

2. **Bits per dimension.** For the repeated-row $4 \times 4$ binary data, compute the bits per pixel of (a) the true distribution, (b) a model that treats every pixel as an independent fair coin, and (c) a model that learns the first row perfectly but treats later rows as independent fair coins.

   **Hint**

   (a) 4 bits / 16 pixels = 0.25. (b) 16 / 16 = 1.0. (c) 4 + 12 = 16 bits per image, also 1.0 bits per pixel: learning the first row's marginal does not help, because its bits are already uniform. All gains come from modelling the dependence of later rows on the first.

3. **Tokenizer budget.** A $512 \times 512$ image is tokenized with factor-8 or factor-16 downsampling. Compute the sequence length in each case and the ratio of full self-attention costs. What might you lose with factor 16?

   **Hint**

   $64 \times 64 = 4096$ versus $32 \times 32 = 1024$ tokens. Attention cost scales with length squared, so the ratio is 16. With factor 16 each token must describe a $16 \times 16$ patch, so small text and thin structures are more likely to be lost unless the codebook or decoder compensates.

4. **Blind spot.** Modify the masked-convolution snippet to use 7 layers and an $11 \times 11$ input with the target at the centre. How many earlier pixels are never seen? Then sketch how a vertical stack plus a horizontal stack removes them.

**Hint**

The blind spot is the staircase above and to the right of the target that the up-left-growing receptive field never reaches; counting zero-gradient cells that are earlier in raster order gives 10. A vertical stack sees full rows above (no left/right restriction) and is fed into the horizontal stack, which handles the current row only, so every earlier pixel becomes reachable.

5. **Masked decoding steps.** Using $n_t = \lceil \gamma(t/T)N \rceil$ with the cosine schedule, $N = 256$ and $T = 8$, compute how many tokens are fixed at each step. Why does the schedule fix few tokens first?

**Hint**

Remaining masked counts are 252, 237, 213, 182, 143, 98, 50 and 0, so 4, 15, 24, 31, 39, 45, 48 and 50 tokens are fixed per step (the figure uses floor, which shifts counts by at most one). In exact arithmetic $\gamma(1) = 0$; in floating point $\cos(\pi/2)$ is a tiny positive number, so a literal ceiling leaves one token masked, another reason the snippet uses floor. Early on the model has little context, so its parallel guesses are the least reliable and the most likely to conflict; fixing only a few lets later steps condition on them.

6. **Classify the methods.** For PixelCNN, VQGAN transformer, Parti, LlamaGen, VAR, MaskGIT and MAR, give the prediction unit, the representation (pixels, discrete tokens, continuous tokens, multiscale maps), the conditional distribution type, and the number of sequential steps for a $16 \times 16$ token grid.

**Hint**

PixelCNN: pixels in raster order, 256-way softmax, one step per value. VQGAN transformer, Parti, LlamaGen: discrete tokens in raster order, softmax over the codebook, 256 steps. VAR: multiscale discrete maps, softmax per token with parallel tokens inside a scale, one step per scale. MaskGIT: discrete tokens, bidirectional masked prediction, $T$ steps (8 in the paper). MAR: continuous tokens in random-order groups, a per-token diffusion head, as many steps as groups (each with its own inner diffusion steps).

## References

1. J. Xiong, G. Liu, L. Huang et al., "Autoregressive Models in Vision: A Survey," *Transactions on Machine Learning Research* (*TMLR*), 2025 (arXiv:2411.05902, 2024). arXiv
2. A. van den Oord, N. Kalchbrenner and K. Kavukcuoglu, "Pixel Recurrent Neural Networks," in *Proc. ICML*, PMLR 48, 2016. PMLR
3. A. van den Oord, N. Kalchbrenner, O. Vinyals, L. Espeholt, A. Graves and K. Kavukcuoglu, "Conditional Image Generation with PixelCNN Decoders," in *NeurIPS*, 2016. NeurIPS

4. A. van den Oord, O. Vinyals and K. Kavukcuoglu, “Neural Discrete Representation Learning,” in *NeurIPS*, 2017. NeurIPS
5. P. Esser, R. Rombach and B. Ommer, “Taming Transformers for High-Resolution Image Synthesis,” in *Proc. IEEE/CVF CVPR*, 2021. CVF
6. K. Tian, Y. Jiang, Z. Yuan, B. Peng and L. Wang, “Visual Autoregressive Modeling: Scalable Image Generation via Next-Scale Prediction,” in *NeurIPS*, 2024 (Best Paper Award). arXiv
7. T. Li, Y. Tian, H. Li, M. Deng and K. He, “Autoregressive Image Generation without Vector Quantization,” in *NeurIPS*, 2024. arXiv
8. H. Chang, H. Zhang, L. Jiang, C. Liu and W. T. Freeman, “MaskGIT: Masked Generative Image Transformer,” in *Proc. IEEE/CVF CVPR*, 2022. CVF
9. S. J. D. Prince, *Understanding Deep Learning*, MIT Press, 2023. Book
10. A. Razavi, A. van den Oord and O. Vinyals, “Generating Diverse High-Fidelity Images with VQ-VAE-2,” in *NeurIPS*, 2019. NeurIPS
11. M. Chen, A. Radford, R. Child, J. Wu, H. Jun, D. Luan and I. Sutskever, “Generative Pretraining From Pixels,” in *Proc. ICML*, PMLR 119, 2020. PMLR
12. J. Yu, X. Li, J. Y. Koh et al., “Vector-quantized Image Modeling with Improved VQGAN,” in *ICLR*, 2022. arXiv
13. J. Yu, Y. Xu, J. Y. Koh et al., “Scaling Autoregressive Models for Content-Rich Text-to-Image Generation,” *Transactions on Machine Learning Research*, 2022. arXiv
14. P. Sun, Y. Jiang, S. Chen, S. Zhang, B. Peng, P. Luo and Z. Yuan, “Autoregressive Model Beats Diffusion: Llama for Scalable Image Generation,” arXiv:2406.06525, 2024. arXiv
15. D. Lee, C. Kim, S. Kim, M. Cho and W.-S. Han, “Autoregressive Image Generation using Residual Quantization,” in *Proc. IEEE/CVF CVPR*, 2022. CVF
16. F. Mentzer, D. Minnen, E. Agustsson and M. Tschannen, “Finite Scalar Quantization: VQ-VAE Made Simple,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2024. arXiv
17. H. Chang, H. Zhang, J. Barber et al., “Muse: Text-To-Image Generation via Masked Generative Transformers,” in *Proc. ICML*, PMLR 202, 2023. PMLR
18. J. Ho and T. Salimans, “Classifier-Free Diffusion Guidance,” arXiv:2207.12598, 2022 (short version at the NeurIPS 2021 Workshop on Deep Generative Models and Downstream Applications). arXiv
19. S. Bond-Taylor, A. Leach, Y. Long and C. G. Willcocks, “Deep Generative Modelling: A Comparative Review of VAEs, GANs, Normalizing Flows, Energy-Based and Autoregressive Models,” *IEEE TPAMI*, vol. 44, no. 11, 2022. doi
20. J. Li, Y. Fu, J. Liu, L. Cao, W. Ji, M. Yang, I. King and M.-H. Yang, “Discrete Tokenization for Multimodal LLMs: A Comprehensive Survey,” *IEEE TPAMI*, 2026 (arXiv:2507.22920). doi
21. S. Zhao, X. Zhang, J. Guo et al., “Unified Multimodal Understanding and Generation Models: Advances, Challenges, and Opportunities,” arXiv:2505.02567, 2025; revised 2026. arXiv
22. J. Jia, J. Gao, B. Xue et al., “From Principles to Applications: A Comprehensive Survey of Discrete Tokenizers in Generation, Comprehension, Recommendation, and Information Retrieval,” arXiv:2502.12448, 2025. arXiv
23. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database,” *CVPR*, 2009, pp. 248–255. doi
24. H. Touvron, T. Lavril, G. Izacard et al., “LLaMA: Open and Efficient Foundation Language Models,” arXiv:2302.13971, 2023. arXiv

# ML 10 • Foundation Models and Vision–Language Learning

What makes a model a foundation model: scaling laws, pre-training objectives, CLIP and SigLIP, DINOv2 features, promptable segmentation, multimodal language models, adaptation with linear probes and LoRA, and how to evaluate the risks.

Online edition: https://blog.shyandram.dev/en/learn/ml/foundation-models/

## What you'll learn

- Define foundation models through pretraining and reuse.
- Interpret scaling laws within their measured regimes.
- Compare supervision sources and reusable vision capabilities.
- Trace image–text alignment, visual prompts and multimodal connectors.
- Choose adaptation methods and test task performance, robustness and limitations.

## The big picture

For most of the history of computer vision, each task had its own model: one network for classifying flowers, another for detecting cars, another for segmenting tumors, each trained from scratch or from an ImageNet [43] classifier. A **foundation model** is trained once, on broad data and at large scale, and then **adapted** to many downstream tasks [1]. The word describes how the model is trained and reused, not a new layer type. A DINOv2 [15] or DINOv3 [16] image encoder, a CLIP-style image–text model [3], a promptable segmenter such as SAM [5], a multimodal language model and a broadly pretrained image generator can all play this role.

> **Explain like I'm five**
>
> Instead of teaching a new student from the beginning for every job, start with someone who has already looked at millions of pictures and read millions of captions. They already know what edges, wheels, faces and dogs look like. For a new job you only need to give them a few examples, a short instruction, or a small new tool — not a whole new education.

Why does this matter? Three reasons. First, **labels are expensive** and pretraining moves most of the learning to data that needs no hand labels (captions, the images themselves). Second, **one backbone serves many tasks**, which concentrates engineering effort. Third, it changes the **interface**: instead of retraining, you can often specify a task with a text prompt, a click or a few examples. Bommasani et al. call the resulting concentration of methods *homogenization*, and warn that it also concentrates failure: a flaw in one shared backbone is inherited by every system built on it [1].

This chapter builds on ML 06 ·Representation Learning (what makes a feature reusable), ML 05 ·Vision Transformers (the usual backbone) and the generative chapters ML 08 and ML 09. The textbook treatment of transfer learning and representation in Bishop and Bishop [14] is a good companion. We follow the architectural map of the vision foundation-model survey by Awais et al. [39] and the multimodal survey by Caffagni et al. [2], with representative reports verified for this October 2026 edition. Examples illustrate families; they do not rank commercial products or claim that one checkpoint is best everywhere.

## What makes a model a "foundation" model

**Plain version.** A foundation model is a large model pretrained on broad data with a generic objective, so that its internal representations or its interface can be reused for tasks it was never specifically trained on.

Bommasani et al. [1] coined the term in 2021 and emphasized two properties:

- **Emergence.** Capabilities arise from scale and the training objective rather than being designed explicitly. For example, CLIP was trained only to match images with captions, yet it can classify images into categories named in text [3]; DINO was trained without any labels, yet its attention maps outline objects [24].
- **Homogenization.** Many applications converge on a small number of pretrained backbones and recipes.

Be careful with "emergence". In this chapter it means only *a capability that was not a training target*. It does not mean an unpredictable or mysterious jump; whether a capability appears suddenly often depends on the metric used to measure it.

Formally, let $f_\theta$ be a model pretrained on an upstream dataset $\mathcal{D}_{\text{up}}$ with objective $\mathcal{L}_{\text{pre}}$:

$$\theta^\star = \arg\min_\theta \; \mathbb{E}_{x \sim \mathcal{D}_{\text{up}}}[\mathcal{L}_{\text{pre}}(f_\theta; x)].$$

A downstream task $k$ supplies a (usually small) dataset $\mathcal{D}_k$ and a task loss $\ell_k$. Adaptation chooses a small set of new or changed parameters $\phi_k$ and solves

$$\phi_k^\star = \arg\min_{\phi_k} \sum_{(x,y)\in\mathcal{D}_k} \ell_k(h_{\phi_k}(f_{\theta^\star}, x), y),$$

where $h_{\phi_k}$ is whatever we put on top of or inside the pretrained model: a linear head, an adapter, a LoRA update, a prompt, or nothing at all (pure prompting, $\phi_k = \emptyset$). The whole chapter is about the two lines above: what $\mathcal{L}_{\text{pre}}$ and $\mathcal{D}_{\text{up}}$ should be, and how small $\phi_k$ can be.

A foundation model does **not** have to produce text. A frozen image encoder whose features feed a hundred different classifiers is a foundation model. Conversely, a large model trained for one narrow task is not one, however big it is.

## Scaling laws

**Plain version.** Scaling laws fit how loss changes with model size, data and compute. They help plan runs within comparable regimes, but do not guarantee extrapolated gains.

Kaplan et al. [17] trained families of Transformer language models and reported that the loss scales as a power law in model size, dataset size and compute, with some trends spanning more than seven orders of magnitude, while width and depth mattered little within the tested range. A power law in model size $N$ reads

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N},$$

where $L$ is held-out cross-entropy loss, $N$ the number of non-embedding parameters, $N_c$ a fitted constant with the same units as $N$, and $\alpha_N > 0$ the fitted exponent. Taking logs gives $\log L = \alpha_N \log N_c - \alpha_N \log N$: a straight line with slope $-\alpha_N$.

Real curves bend because some loss can never be removed (the data itself is noisy or ambiguous). Hoffmann et al. [18] therefore fitted a form with an irreducible term:

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}},$$

where $D$ is the number of training tokens, $E$ the irreducible loss, and $A, B, \alpha, \beta$ fitted constants. For dense Transformers a common approximation of training compute is $C \approx 6ND$ floating-point operations [17] (roughly two for the forward and four for the backward pass, per parameter per token). Fixing $C$ and minimizing $L$ over $N$ (with $D = C/6N$) gives a **compute-optimal** model size. Hoffmann et al. trained over 400 language models and concluded that, for compute-optimal training, model size and training tokens should be scaled equally — every doubling of model size should come with a doubling of tokens [18]. That differed from Kaplan et al.'s earlier recommendation of very large models trained on relatively modest data [17]; the disagreement is a useful reminder that fitted exponents depend on the experimental protocol (learning-rate schedules, which runs are included, what is counted as compute).

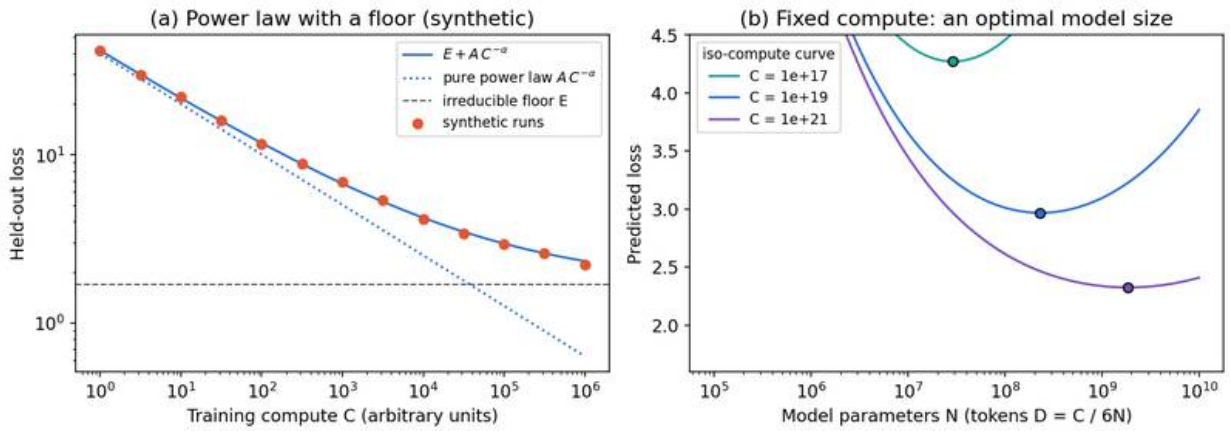



*Figure 1 — (a) A power law with an irreducible floor E: a pure power law (dotted) keeps falling and over-*

*promises at large scale. (b) With compute fixed, too small a model under-fits and too large a model sees too few tokens, so each budget has an optimal size. All constants are synthetic.*

Vision has its own scaling studies. Zhai et al. [19] characterized the relationship between error rate, data and compute for Vision Transformers [21] trained on large labeled datasets, scaling to two-billion-parameter models. Cherti et al. [20] trained CLIP-style models on the public LAION dataset [34] and found power-law scaling for zero-shot classification, retrieval, linear probing and fine-tuning — but also that OpenAI's CLIP and OpenCLIP, with the same architectures and similar recipes, scale differently because their **training distributions** differ. Data is not a scalar.

### Code: fit a scaling law and extrapolate

The snippet fits $L(C) = E + A\,C^{-\alpha}$ to the cheaper of 13 synthetic training runs and then predicts the expensive ones. Everything is synthetic; the point is the procedure.

```
import numpy as np
from scipy.optimize import curve_fit

rng = np.random.default_rng(0)
# Synthetic "runs": loss = E + A * C^(-alpha), with 2% multiplicative noise.
E_true, A_true, alpha_true = 1.7, 40.0, 0.3
compute = np.logspace(0, 6, 13)            # arbitrary compute units
loss = (E_true + A_true * compute**-alpha_true) * np.exp(.02 * rng.normal(size=13))

def law(c, E, A, alpha):
    return E + A * c**-alpha

# Fit only the cheaper runs, then extrapolate to the larger ones.
small = compute <= 1e4
(E, A, alpha), _ = curve_fit(law, compute[small], loss[small], p0=(1, 10, .5))
print(f"fit: E={E:.2f}  A={A:.1f}  alpha={alpha:.3f}")
for c, l in zip(compute[~small], loss[~small]):
    print(f"C={c:9.0f}  observed {l:.3f}  predicted {law(c, E, A, alpha):.3f}")

# Naive straight line in log-log space ignores the floor E.
slope = np.polyfit(np.log(compute[small]), np.log(loss[small]), 1)[0]
print(f"pure power-law slope without floor: {slope:.3f}")
```

Output: the fit recovers $E \approx 1.73$, $A \approx 40.0$, $\alpha \approx 0.300$; the extrapolated losses are within about 7% of the noisy observations; and a straight log–log line without the floor gives a slope of about $-0.25$, a wrong exponent. Two lessons transfer to real practice: always include the floor, and treat extrapolation over many orders of magnitude as a forecast with error bars.

### What scaling laws do not tell you

- They predict **average loss** on the pretraining distribution, not accuracy on your task. Downstream metrics may improve smoothly, saturate, or move in steps.

- They hold **for a fixed recipe and data distribution** [20]. Better data filtering or a different objective changes the constants.
- They say nothing about **inference cost**. A smaller model trained on more data can be cheaper to deploy at the same loss, which is why many deployed models are trained beyond the compute-optimal point.
- Data is finite. If the high-quality data runs out, the $B/D^{\beta}$ term stops shrinking however much compute you add.

## Pre-training objectives: where does the target come from?

**Plain version.** Every pretraining method invents a prediction task. The difference is where the "answer" comes from: a human label, a caption, a hidden part of the same image, another network, or the next token.

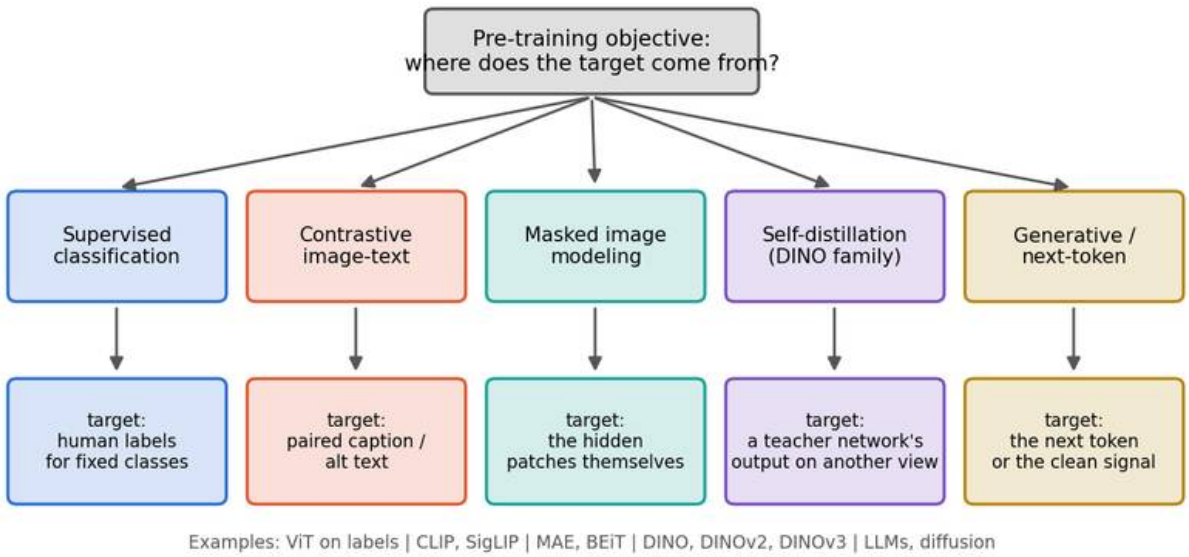



*Figure 2 — Five families of pre-training objectives, organized by the source of the training target.*

| Objective | Target comes from | Representative models | Typical strength |
|---|---|---|---|
| Supervised classification | Human labels | ViT on large labeled sets [19], [21] | Strong global semantics for the label space |
| Contrastive image–text | Paired captions / alt text | CLIP [3], ALIGN [25], SigLIP [4] | Text-defined recognition, retrieval |
| Masked image modeling | Hidden patches of the same image | BEiT [23], MAE [22] | Strong after fine-tuning; local detail |
| Self-distillation | A teacher network's output on another view | DINO [24], DINOv2 [15], DINOv3 [16] | Frozen features, dense correspondence |
| Generative / next-token | Next token or clean signal | LLMs, diffusion (ML 07), AR (ML 08) | Generation; language interface |

### Supervised pre-training

Train a classifier with cross-entropy on a large labeled dataset, then drop the final layer. This was the ImageNet recipe for a decade and still works at scale [19], [21]. Its weakness is that the features are shaped by a fixed label set: distinctions that the labels do not require may be discarded.

### Contrastive image–text learning

Captions come for free with web images, so CLIP [3] and ALIGN [25] train an image encoder and a text encoder together so that matching pairs score higher than mismatched ones within a batch. ALIGN showed that a large, noisy alt-text corpus with minimal filtering can still give strong representations [25]. The loss is detailed in the next section.

### Masked image modeling

Hide most of an image and predict what is missing. BEiT [23] masks patches and predicts discrete visual tokens for them, borrowing the masked-language-modeling idea from BERT [44]. MAE [22] predicts the raw pixels directly and observes that masking a high proportion, e.g. 75% of patches, creates a nontrivial task; its encoder sees only the visible patches and a lightweight decoder reconstructs the rest, which makes training efficient. With $\mathcal{M}$ the set of masked patch indices, $x_p$ the (normalized) pixels of patch $p$ and $\hat{x}_p$ the reconstruction,

$$\mathcal{L}_{\text{MAE}} = \frac{1}{|\mathcal{M}|} \sum_{p \in \mathcal{M}} \|\hat{x}_p - x_p\|_2^2.$$

The loss is computed only on masked patches: predicting the visible ones would be trivial copying. Masked-modeling features are often excellent after fine-tuning but weaker as frozen features for linear probing than contrastive or self-distilled ones.

### Self-distillation: the DINO family

DINO [24] — described by its authors as self-distillation with no labels — trains a **student** network $g_s$ to match a **teacher** network $g_t$ on different augmented views (crops) of the same image. Both output $K$-dimensional scores; probabilities are softmaxes with temperatures $\tau_s$ and $\tau_t$:

$$P_s(x') = \text{softmax}\left(\frac{g_s(x')}{\tau_s}\right), \qquad P_t(x) = \text{softmax}\left(\frac{g_t(x) - c}{\tau_t}\right), \qquad \mathcal{L} = -\sum_{k=1}^{K} P_t(x)_k \log P_s(x')_k.$$

Here $x$ and $x'$ are two views, $c$ is a running mean of teacher outputs (**centering**), and a small $\tau_t$ makes the teacher's distribution sharp (**sharpening**). Centering stops one dimension from dominating; sharpening stops the output from becoming uniform; together they avoid **collapse**, the trivial solution where every image maps to the same output. The teacher receives no gradients. Its weights are an exponential moving average (the momentum encoder) of the student's:

$$\theta_t \leftarrow \lambda\,\theta_t + (1-\lambda)\,\theta_s, \qquad 0 < \lambda < 1.$$

DINO also uses multi-crop training (several small local crops for the student, global crops for the teacher), which forces local-to-global correspondence. Its authors report that the resulting ViT features contain explicit information about object segmentation that does not emerge as clearly with supervised ViTs or convnets, and that they work well as k-nearest-neighbor classifiers [24].

DINOv2 [15] scales this recipe and, crucially, curates the data: an automatic pipeline builds a dedicated, diverse, curated image dataset instead of using uncurated web images, a 1-billion-parameter ViT is trained, and it is distilled into smaller models. The authors report that these frozen features surpass OpenCLIP on most of their image-level and pixel-level benchmarks [15]. DINOv3 [16] scales data and model further and introduces **Gram anchoring**, a technique aimed at a known problem: dense (per-patch) feature maps degrading during long training schedules. It also adds post-hoc strategies for flexibility in resolution, model size and alignment with text.

**Why dense features matter.** A classifier needs one vector per image. Segmentation, depth estimation, correspondence and tracking need a good vector **per patch**. Self-distilled encoders are popular frozen backbones for these dense tasks precisely because their patch features stay semantically meaningful without fine-tuning.

### Generative pre-training

Predicting the next token (language models, autoregressive image models in ML 09) or denoising (diffusion in ML 08) is also a pretraining objective. For language it produces the most flexible interface — instructions in natural language — which is why multimodal models attach vision to a pretrained language model rather than the other way round.

## Image and text in a shared space: CLIP and SigLIP

**Plain version.** Align paired image and text embeddings, then score images against candidate descriptions. Text-defined classification still depends on prompts and learned coverage.

CLIP [3] normalizes image embeddings $v_i$ and text embeddings $t_j$ to unit length and forms scores $s_{ij} = v_i^\top t_j / \tau$ with a learned temperature $\tau$. For a batch of $B$ paired examples,

$$\mathcal{L}_{\text{CLIP}} = \frac{1}{2}\left[\frac{1}{B}\sum_i -\log\frac{\exp(s_{ii})}{\sum_j \exp(s_{ij})} + \frac{1}{B}\sum_j -\log\frac{\exp(s_{jj})}{\sum_i \exp(s_{ij})}\right].$$

Each image should select its matching caption among the $B$ captions (first term, a row-wise softmax), and each caption its matching image (second term, column-wise). All other pairs in the batch act as negatives. Training uses paired text supervision, usually from existing captions or alt text; "zero-shot" downstream use does not mean the model was trained without supervision.

At inference, encode candidate descriptions such as "a photo of a dog" and compare an image with them. The text embeddings replace the class weights of a learned classifier:

$$\hat{y} = \arg\max_k v(x)^\top t(\text{prompt}_k).$$

The candidate set and wording affect the result; averaging the embeddings of several templates ("a photo of a $k$", "a close-up of a $k$") is a common way to reduce sensitivity. Zero-shot means no task-specific parameter training under the stated protocol; it does not guarantee that the classes or even the test images were absent from pretraining (see *Risks and evaluation*).

### Code: paired loss and retrieval mechanics

These **synthetic embeddings** make the computation transparent. They are not outputs from a pretrained model, and their matches do not demonstrate real-world zero-shot accuracy.

```
import torch
from torch.nn import functional as F

images = F.normalize(torch.tensor([[1., .1, 0.], [.1, 1., 0.],
                                   [0., .1, 1.]]), dim=-1)
texts = F.normalize(torch.eye(3), dim=-1)
scores = images @ texts.T / .1
labels = torch.arange(3)
loss = (F.cross_entropy(scores, labels) +
        F.cross_entropy(scores.T, labels))/2
print("matching text:", scores.argmax(1).tolist())
print("paired loss:", round(loss.item(), 4))
ambiguous = F.normalize(torch.tensor([[.5, .5, .2]]), dim=-1)
print("cosines:", (ambiguous @ texts.T).tolist())
```

It prints `matching text: [0, 1, 2]`, a paired loss of about `0.0002`, and cosines of about `0.68, 0.68, 0.27` for the ambiguous image: it is equally close to two descriptions, and no softmax can tell you which is right.

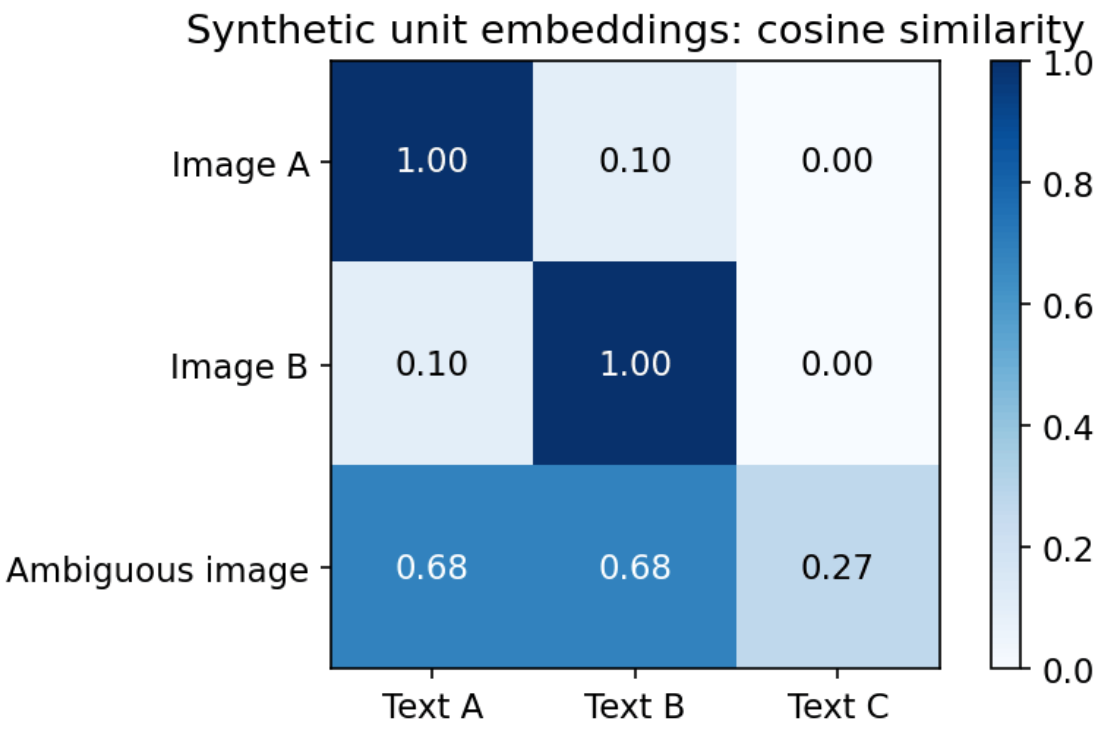



*Figure 3 — Cosine similarity ranks candidates. It is not a calibrated probability that a concept is present.*

### SigLIP: a sigmoid instead of a softmax

SigLIP [4] replaces the batch-normalized softmax with an independent binary decision for every image–text pair. With labels $y_{ij} = +1$ for matching pairs and $-1$ otherwise, a learned scale $a$ and bias $b$,

$$\mathcal{L}_{\text{SigLIP}} = -\frac{1}{B}\sum_{i=1}^{B}\sum_{j=1}^{B}\log\sigma(y_{ij}(a\,v_i^{\top}t_j + b)),$$

where $\sigma(u) = 1/(1+e^{-u})$, so each term equals $\log(1+\exp(-y_{ij}(a\,v_i^{\top}t_j+b)))$. Because no term needs a sum over the whole batch, the loss does not require a global normalization across devices, which simplifies large-batch training. Since there are $B$ positives and $B^2 - B$ negatives, the bias $b$ is initialized so that the initial predictions are strongly negative.

Shared embeddings support retrieval and open-vocabulary recognition, but fine-grained counting, negation ("a dog **without** a collar"), spatial relations and rare domain terms are known weak points. Evaluate those requirements directly rather than inferring them from a strong overall classification score.

*Online only: CLIP — from paired captions to a zero-shot classifier — https://blog.shyandram.dev/en/learn/ml/foundation-models/papers/clip–from-paired-captions-to-a-zero-shot-classifier/*

## Promptable segmentation: the Segment Anything family

**Plain version.** A prompt tells the model which region or concept you want, and the output is a mask.

SAM [5] combines a heavy image encoder (run once per image), a light prompt encoder and a light mask decoder, so many prompts can be answered quickly on the same image. Spatial prompts — points, boxes or rough masks — select objects without a fixed class vocabulary. Because a single click is ambiguous (a click on a shirt could mean the shirt or the whole person), SAM predicts several candidate masks with confidence scores. It was trained on SA-1B, over 1 billion masks on 11 million licensed and privacy-respecting images, collected with a model-in-the-loop data engine [5].

SAM 2 [6] extends promptable segmentation to video with a streaming memory: the model stores features of past frames and of the prompted object, and propagates the mask through time; corrective clicks on later frames update the memory.

SAM 3 [7], released in 2025 with a report revised in March 2026, introduces **promptable concept segmentation**: given a short noun phrase (e.g. "yellow school bus"), image exemplars, or both, it returns masks and identities for **all** matching instances. Its image-level detector and memory-based video tracker share one backbone, and a **presence head** decouples recognizing whether the concept occurs at all from localizing its instances [7].

These interfaces supply different information: a spatial point says *where*, a noun phrase says *what*, an exemplar says *what it looks like*. Report prompt creation cost and whether prompts come from people, ground truth or an automatic detector. A "zero-shot segmentation" score obtained with ground-truth boxes as prompts hides substantial annotation information.

## Multimodal large language models

**Plain version.** Turn visual features into "words" a language model can read, then teach it to follow visual instructions.

A common design has

$$\text{image/video} \xrightarrow{\text{vision encoder}} h \xrightarrow{\text{connector}} z \xrightarrow{\text{language decoder with text}} \text{answer}.$$

Here $h$ is a grid of visual features (for a ViT, one vector per patch), $z$ is a sequence of vectors in the language model's token-embedding space, and the decoder treats $z$ like the embeddings of ordinary words.

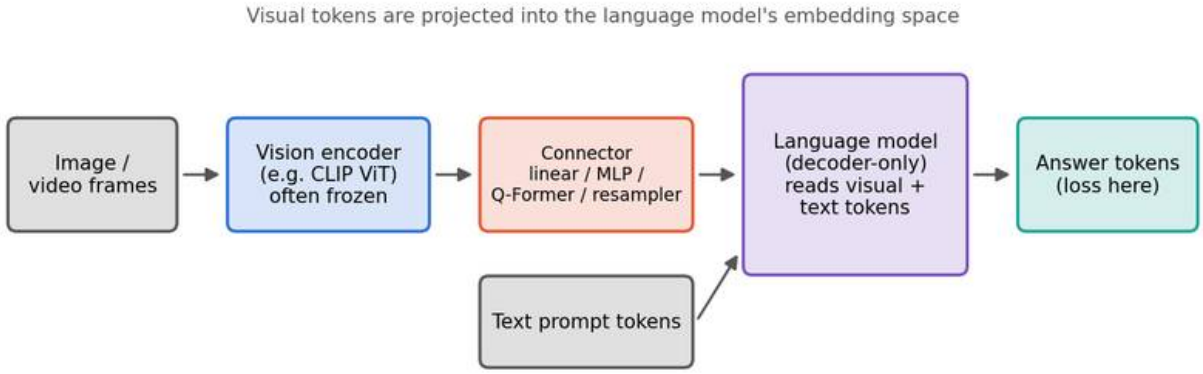



*Figure 4 — The typical multimodal LLM pipeline. Visual features are converted into tokens in the language model's embedding space.*

### Connectors

- **Linear or MLP projection.** LLaVA [8] uses a single projection from CLIP features to the language model's embedding space; LLaVA-1.5 [28] reports that a simple MLP connector (with a CLIP ViT-L at 336-pixel resolution and academic VQA data) is surprisingly powerful and data-efficient. Every patch becomes one visual token.
- **Learned queries.** BLIP-2 [26] inserts a lightweight Querying Transformer (Q-Former) between a frozen image encoder and a frozen language model. A fixed set of learned query vectors cross-attends to the image features, so the language model always sees a small, fixed number of visual tokens. BLIP-2 trains the Q-Former in two stages: representation learning against the frozen image encoder, then generative learning against the frozen language model.
- **Resampling and cross-attention.** Flamingo [27] bridges frozen vision and language models and handles arbitrarily interleaved sequences of images and text, which enabled few-shot learning from examples written directly in the prompt.

The trade-off is token count versus detail: one token per patch keeps fine detail but makes long sequences (expensive for many images or video); a few learned queries are cheap but can lose small text.

### Visual instruction tuning

LLaVA [8] used a language-only GPT-4 [45] to generate multimodal instruction-following data from image captions and boxes, then trained in two stages: first align the connector (feature alignment), then fine-tune connector and language model on the instructions. The loss is ordinary next-token cross-entropy, applied to the **answer** tokens only:

$$\mathcal{L} = -\sum_{i \in \mathcal{A}} \log p_\theta(w_i \mid z,\ w_{<i}),$$

where $w_i$ are text tokens, $\mathcal{A}$ is the set of positions belonging to the assistant's answer, and $z$ are the visual tokens. Instruction and prompt tokens are visible as context but masked out of the loss.

Qwen3-VL's 2025 report [9] illustrates the current direction: a family of dense and mixture-of-experts models with natively interleaved image, video and text contexts of up to 256K tokens. It is a representative report, not evidence that every task benefits from longer contexts or more generated reasoning.

### What the diagram hides

Reading a document may require preserving small text through tiling or dynamic resolution. Video questions need timestamps and temporal order. These details change token counts and preprocessing; a generic "vision encoder + LLM" diagram is only the starting point. A fluent answer can also contain objects that are not in the image (**object hallucination**) or wrong counts. Distinguish object grounding, OCR, spatial reasoning, factual answering and stylistic instruction following — they need different evidence and metrics (see *Risks and evaluation*). For a survey of the whole design space, see Yin et al. [40] and Caffagni et al. [2].

## Adapting a foundation model

**Plain version.** Use the least expensive adaptation that solves the target task, and check it on data that looks like deployment.

| Strategy | Trainable components | Useful starting situation |
|---|---|---|
| Prompting / in-context examples | Nothing | The model's interface already fits the task |
| Frozen features + linear head (linear probe) | Small supervised readout | Few labels, good feature transfer |
| Prompt tuning / visual prompt tuning | A few learned input tokens | Many tasks share one frozen backbone |
| Adapters / LoRA | Small inserted or low-rank modules | Domain adaptation under memory limits |
| Partial fine-tuning | Selected blocks plus head | Some adaptation without full optimizer cost |
| Full fine-tuning (optionally LP-FT) | All model parameters | Enough data and compute; transfer alone insufficient |

### Linear probe

Freeze the encoder $f$, compute features once, and fit a linear classifier $\hat{y} = \mathrm{softmax}(Wf(x) + b)$. It is cheap, hard to overfit, and is also the standard **measurement** of representation quality: if a linear readout works, the information is linearly accessible in the features.

### Code: why pretrained features help most with few labels

This toy experiment simulates the whole story. Observations are 300-dimensional and mix 8 **semantic** factors (which determine the class) with 8 large **nuisance** factors (think lighting and pose) plus noise. "Pretraining" sees 50 upstream classes with plenty of labels and learns an 8-dimensional projection (linear discriminant analysis, standing in for a deep encoder). Downstream, there are four **new** classes and only a few labels per class.

```
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.linear_model import LogisticRegression

rng = np.random.default_rng(0)
D, k_sem, k_nui = 300, 8, 8
sem_map = rng.normal(size=(k_sem, D))   # semantic factors -> pixels
nui_map = rng.normal(size=(k_nui, D))   # nuisance factors (light, pose...)

def sample(centers, n):
    y = rng.integers(len(centers), size=n)
    z = centers[y] + 0.7 * rng.normal(size=(n, k_sem))
    u = 4.0 * rng.normal(size=(n, k_nui))          # large nuisance variation
    x = z @ sem_map + u @ nui_map + rng.normal(size=(n, D))
    return x, y

# 1) "Pre-training": many upstream classes with plenty of labels.
up_centers = rng.normal(size=(50, k_sem))
x_up, y_up = sample(up_centers, 10_000)
encoder = LinearDiscriminantAnalysis(n_components=k_sem).fit(x_up, y_up)

# 2) Downstream: four NEW classes, few labels, linear probe only.
new_centers = rng.normal(size=(4, k_sem))
x_te, y_te = sample(new_centers, 2_000)
f_te = encoder.transform(x_te)          # frozen features, computed once
for shots in [1, 5, 20, 100]:
    raw, feat = [], []
    for repeat in range(10):
        x_tr, y_tr = sample(new_centers, 4 * shots)
        raw.append(LogisticRegression(max_iter=3000)
                   .fit(x_tr, y_tr).score(x_te, y_te))
        feat.append(LogisticRegression(max_iter=3000)
                    .fit(encoder.transform(x_tr), y_tr).score(f_te, y_te))
    print(f"{shots:>3} labels/class | raw pixels {np.mean(raw):.3f}"
          f" | frozen features {np.mean(feat):.3f}")
```

On our machine it prints accuracies of roughly 0.29 vs 0.71 with one label per class, 0.62 vs 0.95 with five, 0.96 vs 0.99 with twenty and 0.98 vs 0.99 with a hundred (raw inputs vs frozen features; chance is 0.25). The pretrained projection already discards the nuisance directions, so a handful of labels suffice; with enough labels, the raw-input probe learns to ignore the nuisance itself and the gap closes. The upstream classes are different from the downstream ones — the features transfer because both share the same semantic factors. If the downstream task depended on a

factor that pretraining threw away (here, the nuisance), frozen features would fail no matter how many labels you add. That is the failure mode to test for in practice.

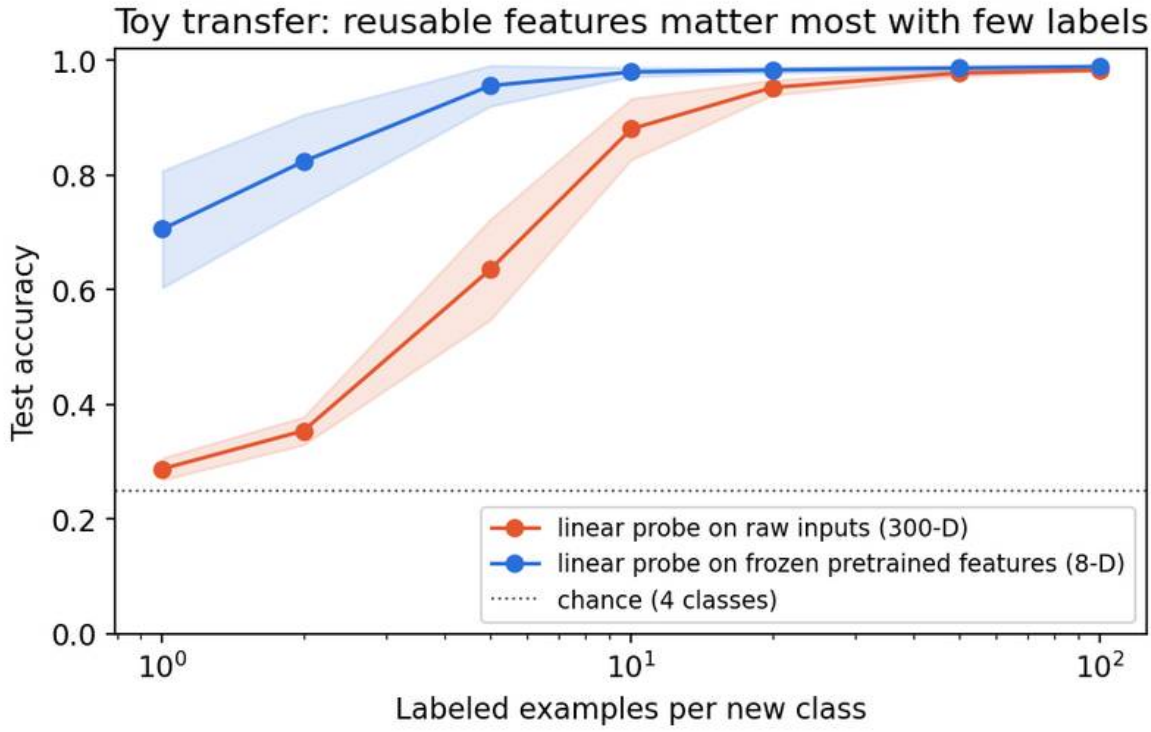



*Figure 5 — The toy transfer experiment over more label budgets (mean ± one standard deviation over 10 draws). Reusable features matter most when labels are scarce.*

### Fine-tuning, LP-FT and robust fine-tuning

Full fine-tuning updates every weight and usually gives the best in-distribution accuracy when data suffices. But Kumar et al. [32] showed that it can distort good pretrained features: across their benchmarks, fine-tuning gave on average higher in-distribution but lower out-of-distribution accuracy than linear probing. Their explanation is that a randomly initialized head sends large, noisy gradients into the backbone early in training. **LP-FT** — first fit a linear probe, then fine-tune everything starting from that head — reduced this distortion and outperformed both alternatives in their experiments [32]. **WiSE-FT** [33] takes another route: linearly interpolate the weights of the zero-shot and fine-tuned models, $\theta_\gamma = (1-\gamma)\theta_{\text{zero-shot}} + \gamma\,\theta_{\text{ft}}$, which improved robustness under distribution shift for CLIP-style models while keeping strong target accuracy. Both are cheap to try and worth knowing before declaring that fine-tuning "hurt robustness".

### Parameter-efficient fine-tuning (PEFT)

**Adapters** [29] insert small bottleneck MLPs inside each Transformer block and train only them, adding a few parameters per task while the original network stays fixed. **Prompt tuning** learns a few extra input tokens; **Visual Prompt Tuning (VPT)** [30] does this for ViTs, prepending learnable tokens to the patch sequence of a frozen backbone. Xu et al. [42] review the PEFT landscape with a taxonomy of additive, partial, reparameterized, hybrid and unified fine-tuning, plus experiments on parameter and memory efficiency.

**LoRA** [10] reparameterizes an update to a frozen weight $W_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ as

$$W = W_0 + \frac{\alpha}{r} BA, \qquad A \in \mathbb{R}^{r \times d_{\text{in}}}, \quad B \in \mathbb{R}^{d_{\text{out}} \times r},$$

with rank $r \ll \min(d_{\text{in}}, d_{\text{out}})$ and a scaling constant $\alpha$. Freeze $W_0$ and learn $A, B$. Trainable weights drop from $d_{\text{out}} d_{\text{in}}$ to $r(d_{\text{in}} + d_{\text{out}})$. Initialize $B = 0$ so the initial update is zero, and $A$ randomly so learning can start. After training, $BA$ can be merged into $W_0$, so inference costs nothing extra. QLoRA [31] stores the frozen base weights in 4-bit precision and backpropagates through them into LoRA adapters, cutting memory enough to fine-tune very large models on a single GPU.

### Code: a low-rank update with a frozen base

This standalone example trains one adapter step on synthetic input. It shows the parameterization and initial behavior; it does not reproduce a full pretrained-model fine-tuning pipeline.

```
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(0)
class LoRALinear(nn.Module):
    def __init__(self, din, dout, rank=4, alpha=4):
        super().__init__()
        self.base = nn.Linear(din, dout, bias=False)
        self.base.weight.requires_grad_(False)
        self.A = nn.Parameter(torch.randn(rank, din) * .02)
        self.B = nn.Parameter(torch.zeros(dout, rank))
        self.scale = alpha/rank
    def forward(self, x):
        return self.base(x) + self.scale * F.linear(F.linear(x, self.A), self.B)

layer = LoRALinear(32, 16)
x = torch.randn(8, 32)
assert torch.allclose(layer(x), layer.base(x))
opt = torch.optim.Adam([layer.A, layer.B], lr=.01)
loss = F.mse_loss(layer(x), torch.randn(8, 16))
opt.zero_grad(); loss.backward(); opt.step()
assert layer.base.weight.grad is None
print("trainable:", sum(p.numel() for p in layer.parameters() if p.requires_grad))
```

It prints `trainable: 192`, which is $r(d_{\text{in}} + d_{\text{out}}) = 4 \times (32 + 16)$, against 512 for the full matrix. The first assertion checks that the zero-initialized $B$ leaves the model unchanged at the start; the second, that the frozen base receives no gradient.

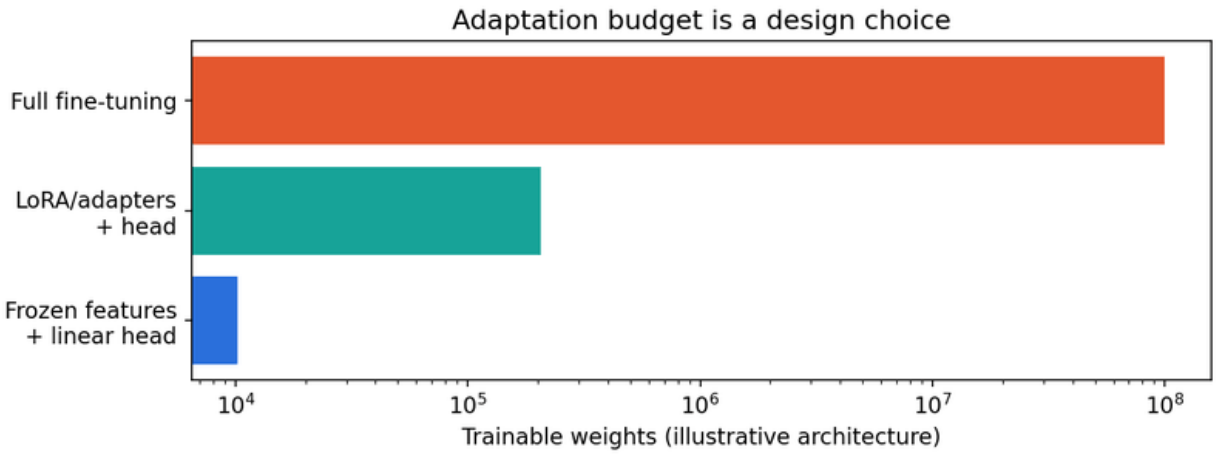



*Figure 6 — Analytical trainable-weight budgets for an illustrative architecture (feature size 1024, 10 classes, rank 8 on 12 matrices, 100M parameters in total). Activation memory, frozen weight storage and optimizer state must be counted separately.*

LoRA is a parameterization, not a guarantee against overfitting or forgetting. Adapters still require domain validation. Frozen BatchNorm statistics, image normalization, output-label mapping and checkpoint-specific preprocessing also need deliberate treatment.

### Prompting and in-context learning

The cheapest adaptation changes no weights: write a better text prompt for CLIP, give SAM a box, or put a few solved examples into a multimodal model's context, as Flamingo did [27]. The vision–language survey by Zhang et al. [41] compares prompting, feature adaptation and fine-tuning across vision tasks. Prompting is fast to iterate but brittle: small wording changes can move results, so fix and report the prompt protocol.

## Video, geometry and world models

**Plain version.** Beyond recognizing a frame, a model may predict change over time, infer three-dimensional structure, or support action planning.

V-JEPA 2 [11] studies self-supervised video models that predict in feature space rather than pixel space, with an action-conditioned extension used for planning. VGGT [12] predicts camera parameters, depth, point maps and tracks from one or many views in a single feed-forward pass. These illustrate two routes beyond classification: temporal state modeling and geometric inference.

A generated video that looks convincing need not obey physical constraints. A scene reconstruction can be useful without generating photorealistic pixels. Measure multi-view consistency, geometry, action-conditioned outcomes and temporal behavior according to the claimed capability. DLCV 13 ·World Models develops partial observability, learned dynamics and planning in more detail.

## Risks and evaluation

**Plain version.** Broad pretraining can carry data biases, unsupported predictions and test-set overlap into evaluation. Check these separately from aggregate benchmark performance.

### Data provenance, bias and privacy

Web-scale image–text datasets are too large to inspect by hand. LAION-5B [34] made such a dataset public (5.85 billion CLIP-filtered pairs) and thereby enabled open replication such as OpenCLIP [20]; audits of its predecessor LAION-400M by Birhane et al. [35] documented explicit and stereotyping content in image–text pairs. Downstream systems inherit these associations. Licenses and consent also matter: training data, model weights and generated outputs may each carry different terms, and personal data (faces, names, medical images) raises privacy obligations that do not disappear because data was public. SAM's SA-1B is an example of a dataset whose description emphasizes licensed and privacy-respecting images [5]; record such properties for every model you deploy.

### Hallucination

Multimodal LLMs can mention objects that are absent from the image. POPE [37] measures object hallucination with a polling approach: it asks yes/no questions about the presence of specific objects (including frequent and co-occurring distractors), which makes the measurement less dependent on the model's generation style than scoring free-form captions. A low POPE error does not cover counting, attributes or relations; test those separately.

### Contamination and "zero-shot"

If benchmark images (or near-duplicates) are in the pretraining set, a "zero-shot" score partly measures memory. Mayilvahanan et al. [36] asked whether CLIP's generalization mainly stems from high train–test similarity; after pruning LAION to remove images too similar to common out-of-distribution benchmarks and retraining, performance dropped on some benchmarks but remained strong overall. The lesson is two-sided: similarity explains part of the reported performance, not all of it — so measure it rather than assume either way. Practical checks: deduplicate the test set against available training data, report results on data collected after the model's data cutoff, and state the full zero-shot protocol (class names, templates, candidate set).

### Benchmarks

Aggregate multimodal benchmarks such as MMMU [38] — 11.5K college-level questions from exams, quizzes and textbooks across six disciplines, with charts, diagrams, maps, tables and other image types — test knowledge and reasoning, but a single number mixes perception, OCR, domain knowledge and answer formatting. For a deployment, build a small, task-specific evaluation set with known failure cases, report variance over seeds or prompt wordings, and compare against a simple baseline (a linear probe on frozen features is often surprisingly strong). ML 11 · Evaluation and Deployment covers calibration, splits and deployment in depth.

## Modern view

**Surveys worth reading.**

- **Bommasani et al. [1]** — the report that named foundation models. It covers capabilities (language, vision, robotics, reasoning), applications (law, health, education), technology (modeling, training, adaptation, evaluation, data) and societal impact. Its key takeaways — emergence, homogenization, and inherited failure — still frame the field.
- **Awais et al. [39]** — a computer-vision survey organizing foundation models by architecture design, training objective (contrastive versus generative), datasets, fine-tuning mechanisms and prompting, and listing open challenges in evaluation, real-world applicability, adversarial robustness and interpretability.
- **Zhang et al. [41]** — a TPAMI survey of vision–language models for vision tasks: network architectures, pre-training objectives, downstream tasks, datasets, transfer learning and knowledge distillation, with benchmarking of reviewed methods.
- **Yin et al. [40]** and **Caffagni et al. [2]** — surveys of multimodal large language models: architecture (encoder, connector, LLM), training strategy and data (pre-training, instruction tuning, alignment), evaluation, and hallucination.
- **Han et al. [13]** — a survey broadening the map to multimodal *generative* models, organized by generative category and technique.
- **Xu et al. [42]** — a critical review of parameter-efficient fine-tuning that sorts methods into additive, partial, reparameterized, hybrid and unified families and compares representative ones experimentally.

**State of the art, 2023–2026.** Four trends stand out. (1) **Better frozen features**: DINOv2 [15] and DINOv3 [16] made self-supervised encoders strong enough that many dense tasks start from frozen patch features. (2) **Concept-level prompting**: segmentation moved from "click an object" (SAM [5]) to video memory (SAM 2 [6]) to "segment every instance of this phrase" (SAM 3 [7]). (3) **Long, interleaved multimodal contexts**: models such as Qwen3-VL [9] read many images, video and text together. (4) **Beyond recognition**: video prediction for planning (V-JEPA 2 [11]) and feed-forward geometry (VGGT [12]) treat foundation models as world and scene estimators.

**Open problems.**

- Grounded reliability: reducing hallucination and making models say "not visible" when ap-

propriate [37].
- Honest evaluation under contamination and benchmark saturation [36].
- Data: curation as a scaling axis [15], [20], provenance, licensing and consent [34], [35].
- Efficient adaptation and inference: token budgets for high-resolution images and video; PEFT that preserves robustness [32], [33].
- Dense, 3D and temporal understanding that matches the semantic strength of image-level models.

For any new "foundation" model, record its input/output interface, training objective, data sources, supported adaptation, checkpoint license, preprocessing and task-level evaluation. This avoids treating model size or an open-ended chat interface as proof of general competence.

## Key takeaways

- Foundation models combine broad pretraining with reuse; the term does not specify one architecture or require text generation.
- Scaling laws are empirical fits tied to data, objectives and training recipes; extrapolation requires validation.
- Pretraining objectives shape capabilities but do not uniquely determine downstream performance.
- Image–text similarity supports retrieval and text-defined classes; similarity is not calibrated confidence.
- Visual prompts and connectors control information access; report prompt construction and image-resolution constraints.
- Start with inexpensive adaptation baselines, then test stronger methods for gains, cost, shift sensitivity, hallucination and contamination.

## Exercises

1. **Candidate dependence.** Add a fourth description to the synthetic CLIP example. Can the softmax "confidence" for an image change without changing the image?

   **Hint**

   Yes: the denominator of the softmax changes. Similarity to a fixed description is unchanged, but the normalized probability depends on the candidate set. Try adding a text vector close to the ambiguous image's embedding.

2. **Compute-optimal allocation.** Using the parametric law $L(N, D) = E + A/N^{\alpha} + B/D^{\beta}$ and $C = 6ND$, show that the optimal model size grows as $N^{\star} \propto C^{\beta/(\alpha+\beta)}$. What does $\alpha = \beta$ imply?

   **Hint**

Substitute $D = C/(6N)$, differentiate with respect to $N$ and set to zero: $\alpha A N^{-\alpha-1} = \beta B (6N/C)^{\beta}/N$. Solving gives $N^{\alpha+\beta} \propto C^{\beta}$. With $\alpha = \beta$, $N^{\star} \propto C^{1/2}$ and therefore $D^{\star} \propto C^{1/2}$ too: scale parameters and tokens equally, as Hoffmann et al. concluded.

3. **Break the transfer.** In the linear-probe experiment, make the downstream class depend on the *nuisance* factors instead of the semantic ones. What happens to the frozen-feature probe as labels increase, and what does that say about choosing a foundation model?

**Hint**

Draw class centers for `u` instead of `z`. The frozen 8-D features were trained to discard the nuisance directions, so they stay near chance however many labels you add, while the raw-input probe improves. A backbone is only useful if its pretraining kept the information your task needs — check with a linear probe before investing in adaptation.

4. **LoRA budget.** A ViT-B block has four $768 \times 768$ attention projections and two MLP matrices of sizes $3072 \times 768$ and $768 \times 3072$. Count trainable weights for full fine-tuning of these six matrices and for rank-8 LoRA on all of them.

**Hint**

Full: $4{\cdot}768^2 + 2{\cdot}3072{\cdot}768 = 2{,}359{,}296 + 4{,}718{,}592 = 7{,}077{,}888$. LoRA: $8{\cdot}[4{\cdot}(768{+}768) + 2{\cdot}(3072{+}768)] = 8 \cdot (6144 + 7680) = 110{,}592$, about 1.6% of the full count.

5. **Prompt fairness.** Design a SAM evaluation that separates mask quality from prompt quality.

**Hint**

Fix the prompt protocol and count human interactions. Evaluate automatic proposals separately from masks produced with ideal ground-truth prompts, and report both. For SAM 3, compare noun-phrase prompts with exemplar prompts on the same images.

6. **Contamination audit.** You are given a new image–text model that reports high zero-shot accuracy on a public benchmark. List three checks you would run before trusting the number.

**Hint**

(a) Near-duplicate search of test images against any available training data (e.g. by embedding similarity); (b) evaluation on a freshly collected set from after the data cutoff; (c) sensitivity to the prompt templates and the candidate class list; optionally compare with a linear probe and a simple baseline.

## References


1. R. Bommasani et al., "On the Opportunities and Risks of Foundation Models," Stanford CRFM report, arXiv:2108.07258, 2021; revised 2022. arXiv
2. D. Caffagni et al., "The Revolution of Multimodal Large Language Models: A Survey," *Findings of ACL*, 2024. arXiv
3. A. Radford et al., "Learning Transferable Visual Models From Natural Language Supervision," *ICML*, 2021. arXiv
4. X. Zhai, B. Mustafa, A. Kolesnikov and L. Beyer, "Sigmoid Loss for Language Image Pre-Training," *ICCV*, 2023. arXiv
5. A. Kirillov et al., "Segment Anything," *ICCV*, 2023. arXiv
6. N. Ravi et al., "SAM 2: Segment Anything in Images and Videos," *ICLR*, 2025. arXiv
7. N. Carion et al., "SAM 3: Segment Anything with Concepts," arXiv:2511.16719, 2025; revised March 2026. arXiv
8. H. Liu, C. Li, Q. Wu and Y. J. Lee, "Visual Instruction Tuning," *NeurIPS*, 2023. arXiv
9. S. Bai et al., "Qwen3-VL Technical Report," arXiv:2511.21631, 2025. arXiv
10. E. J. Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models," *ICLR*, 2022. arXiv
11. M. Assran et al., "V-JEPA 2: Self-Supervised Video Models Enable Understanding, Prediction and Planning," arXiv:2506.09985, 2025. arXiv
12. J. Wang et al., "VGGT: Visual Geometry Grounded Transformer," *CVPR*, 2025. arXiv
13. L. Han et al., "A Survey of Generative Categories and Techniques in Multimodal Generative Models," arXiv:2506.10016, 2025. arXiv
14. C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts*, Springer, 2024. Book
15. M. Oquab et al., "DINOv2: Learning Robust Visual Features without Supervision," *Transactions on Machine Learning Research*, 2024. arXiv
16. O. Siméoni et al., "DINOv3," arXiv:2508.10104, 2025. arXiv
17. J. Kaplan et al., "Scaling Laws for Neural Language Models," arXiv:2001.08361, 2020. arXiv
18. J. Hoffmann et al., "Training Compute-Optimal Large Language Models," arXiv:2203.15556, 2022; published as "An empirical analysis of compute-optimal large language model training," *NeurIPS*, 2022. arXiv
19. X. Zhai, A. Kolesnikov, N. Houlsby and L. Beyer, "Scaling Vision Transformers," *CVPR*, 2022. arXiv
20. M. Cherti et al., "Reproducible Scaling Laws for Contrastive Language-Image Learning," *CVPR*, 2023. arXiv
21. A. Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *ICLR*, 2021. arXiv
22. K. He, X. Chen, S. Xie, Y. Li, P. Dollár and R. Girshick, "Masked Autoencoders Are Scalable Vision Learners," *CVPR*, 2022. arXiv
23. H. Bao, L. Dong, S. Piao and F. Wei, "BEiT: BERT Pre-Training of Image Transformers," *ICLR*, 2022. arXiv
24. M. Caron et al., "Emerging Properties in Self-Supervised Vision Transformers," *ICCV*, 2021. arXiv
25. C. Jia et al., "Scaling Up Visual and Vision-Language Representation Learning With Noisy Text Supervision," *ICML*, 2021. arXiv

26. J. Li, D. Li, S. Savarese and S. Hoi, “BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models,” *ICML*, 2023. PMLR
27. J.-B. Alayrac et al., “Flamingo: a Visual Language Model for Few-Shot Learning,” *NeurIPS*, 2022. arXiv
28. H. Liu, C. Li, Y. Li and Y. J. Lee, “Improved Baselines with Visual Instruction Tuning,” *CVPR*, 2024. arXiv
29. N. Houlsby et al., “Parameter-Efficient Transfer Learning for NLP,” *ICML*, 2019. PMLR
30. M. Jia et al., “Visual Prompt Tuning,” *ECCV*, 2022. arXiv
31. T. Dettmers, A. Pagnoni, A. Holtzman and L. Zettlemoyer, “QLoRA: Efficient Finetuning of Quantized LLMs,” *NeurIPS*, 2023. arXiv
32. A. Kumar, A. Raghunathan, R. Jones, T. Ma and P. Liang, “Fine-Tuning can Distort Pretrained Features and Underperform Out-of-Distribution,” *ICLR*, 2022. arXiv
33. M. Wortsman et al., “Robust Fine-Tuning of Zero-Shot Models,” *CVPR*, 2022. arXiv
34. C. Schuhmann et al., “LAION-5B: An Open Large-Scale Dataset for Training Next Generation Image-Text Models,” *NeurIPS Datasets and Benchmarks Track*, 2022. arXiv
35. A. Birhane, V. U. Prabhu and E. Kahembwe, “Multimodal Datasets: Misogyny, Pornography, and Malignant Stereotypes,” arXiv:2110.01963, 2021. arXiv
36. P. Mayilvahanan, T. Wiedemer, E. Rusak, M. Bethge and W. Brendel, “Does CLIP’s Generalization Performance Mainly Stem from High Train-Test Similarity?,” *ICLR*, 2024. arXiv
37. Y. Li et al., “Evaluating Object Hallucination in Large Vision-Language Models,” *EMNLP*, 2023. arXiv
38. X. Yue et al., “MMMU: A Massive Multi-discipline Multimodal Understanding and Reasoning Benchmark for Expert AGI,” *CVPR*, 2024. arXiv
39. M. Awais et al., “Foundational Models Defining a New Era in Vision: A Survey and Outlook,” arXiv:2307.13721, 2023. arXiv
40. S. Yin et al., “A Survey on Multimodal Large Language Models,” *National Science Review*, 2024. arXiv
41. J. Zhang, J. Huang, S. Jin and S. Lu, “Vision-Language Models for Vision Tasks: A Survey,” *IEEE TPAMI*, vol. 46, no. 8, 2024. arXiv
42. L. Xu et al., “Parameter-Efficient Fine-Tuning Methods for Pretrained Language Models: A Critical Review and Assessment,” arXiv:2312.12148, 2023. arXiv
43. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database,” *CVPR*, 2009, pp. 248–255. doi
44. J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *NAACL-HLT*, 2019, pp. 4171–4186. doi
45. OpenAI, “GPT-4 Technical Report,” arXiv:2303.08774, 2023. arXiv

# ML 11 • Evaluation, Efficiency and Deployment

Split data without leakage, cross-validate, attach confidence intervals, check calibration and robustness under shift, measure FLOPs, memory and latency honestly, then export, compress, monitor and reproduce a model.

Online edition: https://blog.shyandram.dev/en/learn/ml/evaluation-deployment/

## What you'll learn

- Design baselines and data splits that support the deployment claim.
- Choose cross-validation and uncertainty estimates for the sampling structure.
- Distinguish accuracy, calibration and robustness under shift.
- Measure operations, memory and latency on the intended system.
- Validate exported models, monitor deployment and document reproducible runs.

## The big picture

Training a model is the middle of a project, not the end. Before anyone relies on it, you need to answer four questions. Does it work on the cases it will really see? How sure are we of that number? Does it know when it is likely to be wrong? Does it fit the device, latency and budget it has to run on? After it ships, a fifth question appears: is it still working?

> **Explain like I'm five**
>
> A student who memorized last year's exam gets full marks on last year's exam. That tells you nothing. A fair test uses new questions (a clean split). One test can be lucky, so you want to know how much the score would wobble on another day (a confidence interval). A good student also says "I'm not sure" on the questions they might get wrong (calibration). If the real exam is in a noisy hall, you check that too (robustness). If it is a timed exam, being right slowly is not enough (latency). And once the student starts a job, someone still checks their work now and then (monitoring).

This chapter closes the ML track. It gathers the practical methodology from Goodfellow et al. [1, ch. 11] and the statistical-learning view of James et al. [2], and adds the engineering steps that turn a trained network into a dependable service. The task-specific metrics of vision (top-1, mAP, PSNR, FID and so on) are covered in DLCV 02 and DLCV 03; here we focus on what all tasks share.

## Start with a meaningful baseline

**Plain version.** Before spending more compute, find out how well a simple method does on exactly the same split. The baseline tells you what the complicated model actually buys.

Good baselines for image tasks include: logistic regression on raw pixels or hand-designed features; a nearest-neighbor or SVM classifier; and a small linear head on frozen features from a pretrained model (see ML 10). Logistic regression learns a linear boundary; a support vector machine (SVM) maximizes a regularized margin; nearest neighbors compares examples directly; tree ensembles partition the feature space. The classical pattern-classification chapter, IP 13, develops these methods.

For reference, the binary linear SVM minimizes

$$\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\max(0,\, 1 - y_i(w^\top x_i + b)), \qquad y_i \in \{-1, +1\},$$

where $w$ and $b$ are the weights and bias, $x_i$ is the $i$-th feature vector, and $C > 0$ trades margin width against training errors. The term $\max(0, 1 - y_i f(x_i))$ is the **hinge loss**: it is zero for points on the correct side with margin at least 1 and grows linearly otherwise.

### Code: baselines with preprocessing inside the pipeline

This example uses only bundled data. Model choice uses a validation split; the chosen model is evaluated on the test split once. Scaling and PCA are fitted inside each training pipeline, never on the full dataset.

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, balanced_accuracy_score

X, y = load_digits(return_X_y=True)
Xa, Xt, ya, yt = train_test_split(X, y, test_size=.2, stratify=y, random_state=0)
Xtr, Xv, ytr, yv = train_test_split(Xa, ya, test_size=.25, stratify=ya, random_state=1)
models = {      # listed from simplest to most complex
    "linear": make_pipeline(StandardScaler(),
                            LogisticRegression(max_iter=2000)),
    "PCA + RBF SVM": make_pipeline(StandardScaler(), PCA(n_components=30),
                                   SVC(C=10, gamma="scale")),
}
scores = {}
for name, model in models.items():
    model.fit(Xtr, ytr)
    scores[name] = accuracy_score(yv, model.predict(Xv))
best = max(scores, key=scores.get)   # tie -> first listed, i.e. the simpler model
```

```
pred = models[best].predict(Xt)
print("validation:", scores)
print("selected:", best)
print("test accuracy:", accuracy_score(yt, pred))
print("test balanced accuracy:", balanced_accuracy_score(yt, pred))
```

On this split both candidates score exactly 0.9639 on validation (347 of 360 images). The rule "on a tie, prefer the simpler model" was fixed in advance, so the linear model is selected, and it scores 0.9639 accuracy and 0.9636 balanced accuracy on test. Two lessons hide in this tiny run. First, a tie-break rule is part of the protocol and should be written down before you look. Second, with 360 validation images a single image is worth 0.28 percentage points, so many apparent "wins" in small experiments are noise; the section on confidence intervals makes this precise.

## Data splits and leakage

**Plain version.** Make test data new in the way deployment will be new. Shared patients, scenes or time periods can inflate results when the claim concerns unseen groups or future data.

### Choose the split unit from the claim

Images from one patient, one video, one production batch or one camera are correlated. Randomly splitting adjacent video frames puts almost the same picture in train and test. Ask what "new" means in deployment and split by that unit: by patient if the model will see new patients, by video if it will see new videos, by site if it will be installed in new hospitals, by time if it must predict the future.




*Figure 1 — Choose the split unit according to the generalization claim. Group separation prevents near-duplicate context from crossing the split.*

Stratification keeps class proportions similar across splits; it does not replace grouping. Deduplicate before splitting, and keep exact or near duplicates together. When the deployment population drifts over time, a time-based split (train on the past, test on the most recent period) is the honest choice.

### The many faces of leakage

**Leakage** is any path by which information that would not be available at prediction time reaches the model or the model-selection process. Kapoor and Narayanan found leakage errors affecting 329 papers across 17 scientific fields that use ML, and propose a taxonomy of leakage types, among them missing train/test separation, preprocessing on the full dataset, duplicates, temporal leakage and illegitimate features [4]. The scikit-learn documentation lists the most common coding mistakes [3]. A practical checklist:

| Leak | Example | Fix |
|---|---|---|
| Preprocessing on all data | Fit `StandardScaler`, PCA or a vocabulary on train + test | Put preprocessing inside a `Pipeline` fitted on the training fold only |
| Correlated units | Frames of one video, slices of one scan, crops of one photo | Split by group |
| Duplicates | The same image scraped twice | Hash or embed and deduplicate before splitting |
| Temporal | Using next week's data to predict this week | Time-ordered splits |
| Target proxy | A feature recorded *after* the outcome (e.g. "treatment given") | Audit each feature's timestamp |
| Test reuse | Tuning hyperparameters while watching the test score | Validation for decisions, test once |

The last row is subtle. Every time you look at the test score and change something, the test set becomes a little more like a validation set. Public benchmarks suffer from this at community scale.

## Cross-validation

**Plain version.** Instead of one train/validation split, rotate: split the data into $K$ parts, train $K$ times, each time holding out a different part, and average. You use every example for evaluation once and you see how much the score varies.

### K-fold and its variants

Partition the indices into $K$ disjoint folds $F_1, \dots, F_K$. For each $k$, train $\hat{f}^{(-k)}$ on all folds except $F_k$ and compute its error on $F_k$. The cross-validation estimate is

$$\widehat{\mathrm{Err}}_{\mathrm{CV}} = \frac{1}{K} \sum_{k=1}^{K} \frac{1}{|F_k|} \sum_{i \in F_k} \ell(y_i, \hat{f}^{(-k)}(x_i)),$$

where $\ell$ is the per-example loss (for example 0–1 error) and $|F_k|$ is the fold size. Common choices are $K = 5$ or $K = 10$ [2]. The variants differ only in how folds are formed:

- **Stratified** $K$**-fold** keeps class proportions equal across folds; useful when classes are imbalanced.
- **Group** $K$**-fold** assigns each group (patient, video, site) to exactly one fold, so no group appears in both training and held-out data.
- **Time-series split** always trains on earlier data and validates on later data.
- **Leave-one-group-out** is group $K$-fold with one group per fold; useful with few groups.

The fold scores are not independent (training sets overlap), so the standard deviation across folds understates the true uncertainty. Treat it as a rough indication of stability, not as a formal confidence interval.

### Code: what random K-fold does to grouped data

We simulate 40 patients with 25 images each. Every patient's images share a "fingerprint" (patient-specific appearance), and the label is fixed per patient. The real class signal is weak.

```
import numpy as np
from sklearn.model_selection import KFold, GroupKFold, cross_val_score
from sklearn.ensemble import RandomForestClassifier

rng = np.random.default_rng(0)
n_groups, per_group = 40, 25                    # e.g. 40 patients, 25 images each
groups = np.repeat(np.arange(n_groups), per_group)
y = np.repeat(rng.integers(0, 2, n_groups), per_group)   # label fixed per patient
identity = rng.normal(size=(n_groups, 20))[groups]       # patient "fingerprint"
signal = 0.3 * y[:, None] * np.ones((1, 20))             # weak real class signal
X = identity + signal + 0.3 * rng.normal(size=(len(y), 20))

model = RandomForestClassifier(n_estimators=200, random_state=0)
random_cv = cross_val_score(model, X, y, cv=KFold(5, shuffle=True, random_state=0))
group_cv = cross_val_score(model, X, y, cv=GroupKFold(5), groups=groups)
print(f"random K-fold accuracy: {random_cv.mean():.3f} +/- {random_cv.std():.3f}")
print(f"group  K-fold accuracy: {group_cv.mean():.3f} +/- {group_cv.std():.3f}")
```

Output:

```
random K-fold accuracy: 0.999 +/- 0.002
group  K-fold accuracy: 0.657 +/- 0.158
```

The random split reports near-perfect accuracy because the forest recognizes each patient's fingerprint and copies that patient's label. Group $K$-fold, which matches the claim "works on new patients", reports 0.657 with large variation across folds. Neither number is wrong as arithmetic; only one answers the question.

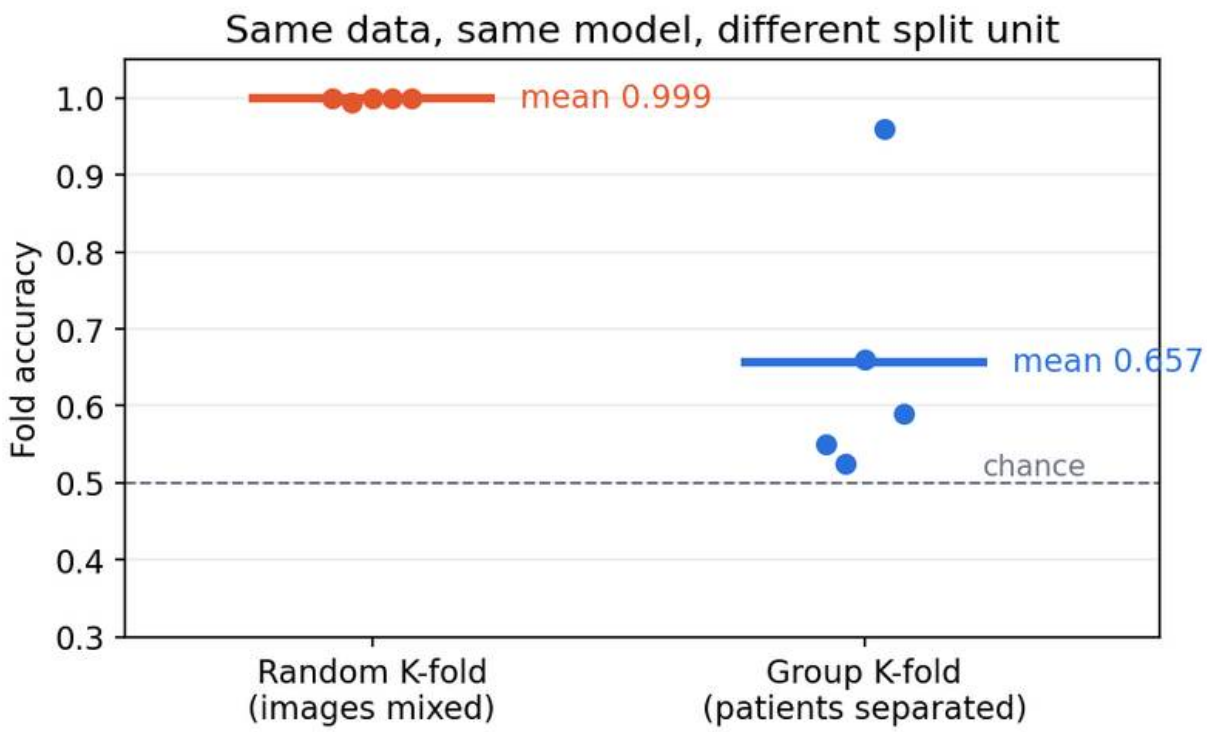



*Figure 2 — Output of the code above, one dot per fold. The split unit, not the model, causes the 34-point gap.*

### Nested cross-validation

If you use cross-validation both to **choose** hyperparameters and to **report** performance, the reported number is optimistically biased: you picked the configuration that happened to look best on those folds. Varma and Simon demonstrated this with "null" data in which the two classes were statistically identical: after tuning an SVM by cross-validation, the CV error estimate fell below 30% on 38% of the simulated datasets, while the tuned classifiers were no better than chance on independent test data [5]; Cawley and Talbot showed that over-fitting the model-selection criterion can be as large as the differences between learning algorithms [6].

**Nested CV** fixes this with two loops. The outer loop holds out a fold for evaluation. Inside each outer training set, an inner CV selects hyperparameters. The outer fold is touched only to score the configuration the inner loop chose. In scikit-learn this is simply a `GridSearchCV` placed inside `cross_val_score`. The result estimates the performance of the whole *procedure* (including tuning), which is what you will run when you retrain.

For deep networks a full nested CV is often too expensive. The usual compromise is a single fixed train/validation/test split, a stated tuning budget for every method compared, and a test set that is opened once.

## Metrics: a recap

**Plain version.** Decide what an error costs before choosing a number. One scalar rarely tells the whole story.

| Task | Useful measurements | Common trap |
|---|---|---|
| Classification | Per-class precision/recall, macro-F1, balanced accuracy, AUROC/AUPRC | Accuracy hides rare failures; PR curves depend on prevalence |
| Detection/segmentation | AP, IoU/Dice, boundary errors | Comparing different IoU thresholds or protocols |
| Retrieval | Recall@$k$, mAP under a defined relevance set | Gallery duplicates inflate matches |
| Restoration | PSNR/SSIM, perceptual metrics, consistency with the observation | Plausible texture may not be true detail |
| Generation | FID/KID, coverage, conditional alignment, human study | Feature realism does not establish prompt fidelity |
| Video/geometry | Temporal consistency, task accuracy, geometric error | Good individual frames can hide inconsistent structure |

The definitions and pitfalls of classification metrics (confusion matrix, top-$k$, ROC/PR curves) are worked through in DLCV 02, and those of detection (IoU, AP, COCO mAP) in DLCV 03. For generative models, FID compares Gaussian fits (means and covariances) of feature distributions; KID instead uses a kernel maximum-mean-discrepancy estimate that is unbiased, which makes it better behaved with small sample counts [7]. Neither is a universal measure of human preference. Always state the feature extractor, reference set, sample count, resolution and sampling settings.

With 1% positives, an always-negative classifier gets 99% accuracy and zero recall, so accuracy alone hides the failure. Precision also depends on prevalence. If only the class proportions change while the class-conditional input distributions and decision threshold stay fixed, recall and false-positive rate stay fixed, but precision changes [17]. This is the label-shift assumption; in a new hospital, camera or population, class-conditional appearances can also change, so all three rates must be remeasured rather than presumed stable.

## Statistical significance and confidence intervals

**Plain version.** A test score is computed from a finite sample, so it would come out a little different on another sample. Report a range, not just a point, and compare two models on the *same* examples.

### Standard error of an accuracy

If the $n$ test examples are independent and the model's true accuracy is $p$, the number of correct predictions is binomial, and the observed accuracy $\hat{p}$ has standard error

$$\mathrm{SE}(\hat{p}) = \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}, \qquad 95\%\ \text{interval} \approx \hat{p} \pm 1.96\,\mathrm{SE}.$$

Here $\hat{p}$ is the fraction correct and 1.96 is the 97.5th percentile of the standard normal distribution. For $\hat{p} = 0.972$ and $n = 540$, SE $\approx 0.0071$, so the interval is about $\pm 1.4$ percentage points. This normal approximation is poor when $\hat{p}$ is very close to 0 or 1 or $n$ is small; Wilson or exact binomial intervals are better there. If test examples are grouped (several images per patient), the effective sample size is closer to the number of groups, and the interval must be computed by resampling groups.

### The bootstrap

The **bootstrap** estimates the sampling distribution of any statistic by resampling the observed data with replacement [9]. Draw $B$ resamples of size $n$, recompute the statistic on each, and take the 2.5th and 97.5th percentiles as an approximate 95% interval (the *percentile* interval). It works for metrics with no simple formula, such as macro-F1, AUROC or mAP.

To compare two models, use a **paired** bootstrap: resample example indices once, and evaluate both models on the same resample. Because the models usually fail on many of the same hard examples, pairing removes that shared variance and gives a much tighter interval for the difference than two separate intervals would.

### McNemar's test

For two classifiers evaluated on the same test set, only the examples where they disagree carry information about which is better. Let $n_{01}$ count examples model A gets right and B gets wrong, and $n_{10}$ the reverse. Under the null hypothesis that both models have the same error rate, each discordant example is equally likely to fall either way, so $n_{10} \sim \mathrm{Binomial}(n_{01} + n_{10}, \frac{1}{2})$. The exact McNemar test computes a two-sided binomial p-value from that. Dietterich compared five tests for comparing learning algorithms and found that, for algorithms that can be run only once (typical for expensive models), McNemar's test was the only one with acceptable type I error; when ten runs are affordable he recommended the 5×2 cross-validated t test [8].

### Code: paired bootstrap and McNemar

```
import numpy as np
from scipy.stats import binomtest
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

X, y = load_digits(return_X_y=True)
```

```
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3, stratify=y, random_state=0)
a = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000)).fit(Xtr, ytr)
b = make_pipeline(StandardScaler(), SVC(C=10, gamma="scale")).fit(Xtr, ytr)
ca, cb = a.predict(Xte) == yte, b.predict(Xte) == yte    # per-example correctness

# 1) Percentile bootstrap: resample test examples with replacement.
rng = np.random.default_rng(0)
idx = rng.integers(0, len(yte), size=(10_000, len(yte)))
diff = cb[idx].mean(axis=1) - ca[idx].mean(axis=1)       # paired: same resample
lo, hi = np.percentile(diff, [2.5, 97.5])
print(f"accuracy A={ca.mean():.4f}  B={cb.mean():.4f}  n={len(yte)}")
print(f"B - A = {cb.mean() - ca.mean():+.4f}, 95% bootstrap CI [{lo:+.4f}, {hi:+.4f}]")

# 2) Exact McNemar test: only the discordant examples carry information.
n01 = int(np.sum(ca & ~cb))      # A right, B wrong
n10 = int(np.sum(~ca & cb))      # A wrong, B right
p = binomtest(n10, n01 + n10, 0.5).pvalue
print(f"discordant pairs: A-only={n01}, B-only={n10}, exact McNemar p={p:.4f}")
```

Output:

```
accuracy A=0.9722  B=0.9852  n=540
B - A = +0.0130, 95% bootstrap CI [-0.0019, +0.0278]
discordant pairs: A-only=5, B-only=12, exact McNemar p=0.1435
```

The SVM is 1.3 points better, but the interval includes zero and McNemar's p-value is 0.14. Only 17 of 540 images separate the two models, and 12 versus 5 is not unusual under a fair coin. With this test set, the honest conclusion is "B is probably at least as good; the evidence that it is better is weak."

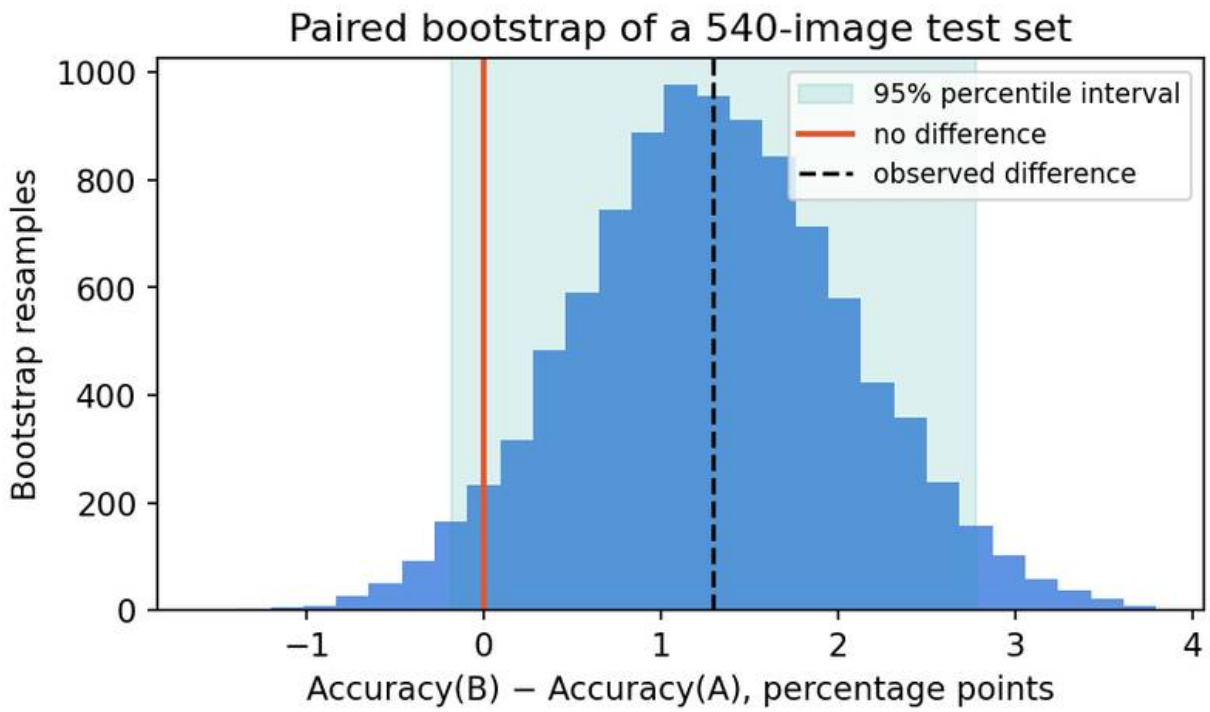



*Figure 3 — The paired bootstrap distribution of Accuracy$(B)$ − Accuracy$(A)$ from the code above. The 95% interval crosses zero.*

### Other sources of variance

The test sample is not the only source of randomness. For a neural network, the random seed changes initialization, data order and augmentation; the data split itself is random; and some GPU kernels are nondeterministic. Bouthillier et al. modeled the whole benchmarking process and found that variance from data sampling, parameter initialization and hyperparameter choice markedly affects conclusions; they recommend randomizing as many of these sources as possible across repeated runs [10]. Practical rules:

- Run each configuration with several seeds (3–5 is common) and report mean and spread.
- Compare methods with paired tests on the same examples and splits.
- If you compare many models or many datasets, some will "win" by chance; adjust for multiple comparisons or confirm on a fresh set.
- Report effect sizes with intervals. A statistically significant difference can be practically irrelevant, and a large practical difference can be statistically unproven on a small set.

## Calibration: knowing when you might be wrong

**Plain version.** A model's confidence should match how often it is right. Among all predictions made with 80% confidence, about 80% should be correct. Accuracy and calibration are separate properties.

### Measuring probability quality

Let $p_{ik}$ be the predicted probability of class $k$ for example $i$, and $y_{ik} \in \{0, 1\}$ the one-hot label. Two **proper scoring rules** (scores minimized in expectation by the true probabilities) are:

$$\mathrm{NLL} = -\frac{1}{n}\sum_{i=1}^{n} \log p_{i,y_i}, \qquad \mathrm{Brier} = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} (p_{ik} - y_{ik})^2,$$

where $y_i$ is the true class of example $i$ and $K$ is the number of classes. Some libraries divide the Brier score by $K$; say which convention you use.

The **expected calibration error** (ECE) groups predictions into $M$ bins $B_1, \dots, B_M$ by their top-class confidence and compares average confidence with accuracy in each bin:

$$\mathrm{ECE} = \sum_{m=1}^{M} \frac{|B_m|}{n} \left|\mathrm{acc}(B_m) - \mathrm{conf}(B_m)\right|,$$

where $\mathrm{acc}(B_m)$ is the fraction correct in bin $m$ and $\mathrm{conf}(B_m)$ the mean confidence there. A **reliability diagram** plots acc against conf per bin; a perfectly calibrated model lies on the diagonal. ECE is not a proper scoring rule and depends on the bin count and the bin scheme, so report it alongside NLL or Brier and a reliability diagram.

### Temperature scaling

Guo et al. observed that modern deep networks, which are much larger than those of earlier decades, are often overconfident, and that a single scalar fix works surprisingly well [11]. **Temperature scaling** divides the logits $\ell$ by $T > 0$ before the softmax:

$$\hat{p}_k = \frac{\exp(\ell_k / T)}{\sum_j \exp(\ell_j / T)}, \qquad T^\star = \arg\min_{T>0} \mathrm{NLL}_{\mathrm{cal}}(T).$$

$T > 1$ softens overconfident predictions; $T < 1$ sharpens underconfident ones. Because dividing all logits by the same positive number never changes which is largest, the predicted class (and so accuracy) is unchanged. $T$ must be fitted on a held-out calibration set, never on the test set. Later work by Minderer et al. found that the most recent image classifiers, notably non-convolutional ones such as Vision Transformers, are among the best calibrated, and that older trends such as calibration decaying under distribution shift are less pronounced for them [12].

### Code: temperature scaling and what shift does to it

```
import numpy as np
from scipy.optimize import minimize_scalar
from scipy.special import log_softmax, softmax
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

X, y = load_digits(return_X_y=True)
X = X / 16.0
Xa, Xte, ya, yte = train_test_split(X, y, test_size=0.4, stratify=y, random_state=0)
Xtr, Xcal, ytr, ycal = train_test_split(Xa, ya, test_size=0.3, stratify=ya, random_state=0)
net = MLPClassifier(hidden_layer_sizes=(128,), alpha=1e-6, max_iter=300,
                    random_state=0).fit(Xtr, ytr)

def logits(X):                        # log-probabilities = logits up to a constant
    return np.log(np.clip(net.predict_proba(X), 1e-12, 1.0))

def ece(prob, y, bins=10):
    conf, pred = prob.max(1), prob.argmax(1)
    edges = np.linspace(0, 1, bins + 1)
    total = 0.0
    for lo, hi in zip(edges[:-1], edges[1:]):
        m = (conf > lo) & (conf <= hi)
        if m.any():
            total += m.mean() * abs((pred[m] == y[m]).mean() - conf[m].mean())
    return total

def nll(T, z, y):
    return -log_softmax(z / T, axis=1)[np.arange(len(y)), y].mean()

T = minimize_scalar(nll, bounds=(0.05, 20), method="bounded",
                    args=(logits(Xcal), ycal)).x
print(f"fitted temperature T = {T:.2f}")
noise = np.random.default_rng(0).normal(size=Xte.shape)
for sigma in [0.0, 0.3, 0.6]:         # Gaussian pixel noise = a simple covariate shift
    Xs = np.clip(Xte + sigma * noise, 0, 1)
    z = logits(Xs)
    p_raw, p_ts = softmax(z, axis=1), softmax(z / T, axis=1)
    print(f"noise {sigma:.1f}: acc={(p_raw.argmax(1) == yte).mean():.3f}  "
          f"ECE raw={ece(p_raw, yte):.3f}  ECE scaled={ece(p_ts, yte):.3f}  "
          f"NLL raw={nll(1, z, yte):.3f}  NLL scaled={nll(T, z, yte):.3f}")
```

Output (scikit-learn may print a convergence warning, which is harmless here):

```
fitted temperature T = 1.19
noise 0.0: acc=0.972  ECE raw=0.007  ECE scaled=0.015  NLL raw=0.108  NLL scaled=0.110
noise 0.3: acc=0.794  ECE raw=0.088  ECE scaled=0.063  NLL raw=0.693  NLL scaled=0.639
noise 0.6: acc=0.435  ECE raw=0.380  ECE scaled=0.347  NLL raw=2.998  NLL scaled=2.598
```

Three observations. On clean data this small network is already well calibrated, and fitting $T$ on

324 calibration images slightly *worsens* ECE: a fix fitted on a small set can add noise. Under pixel noise the model becomes badly overconfident: at $\sigma = 0.6$ it is right 43.5% of the time while its confidence suggests far more. Temperature scaling fitted on clean data helps only a little under shift. This matches the large study by Ovadia et al., who found that the calibration of many uncertainty methods deteriorates as shift increases [16].

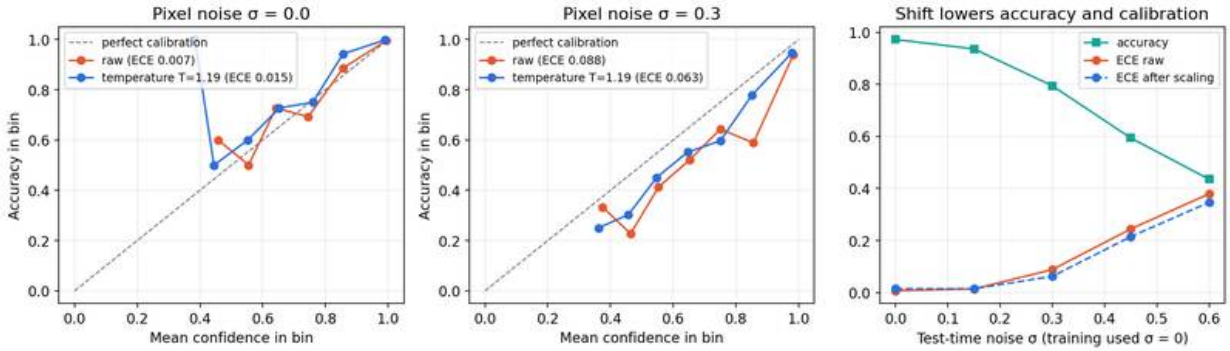



*Figure 4 — Reliability diagrams and a shift sweep for the network above (same noise pattern, scaled by $\sigma$). Points below the diagonal mean overconfidence.*

*Online only: Temperature Scaling — confidence without changing the class — https://blog.shyandram.dev/en/learn/ml/evaluation-deployment/papers/temperature-scaling–confidence-without-changing-the-class/*

## Robustness and distribution shift

**Plain version.** New cameras, lighting or populations can change performance and calibration. Test relevant shifts rather than assuming deployment matches training.

### Three kinds of shift

Write the joint distribution as $p(x, y) = p(y \mid x)\, p(x) = p(x \mid y)\, p(y)$, with $x$ the input and $y$ the label. Shifts are named by what changes between training ($p_{\text{tr}}$) and deployment ($p_{\text{te}}$):

- **Covariate shift:** $p_{\text{te}}(x) \neq p_{\text{tr}}(x)$ while $p(y \mid x)$ is unchanged. New camera, new lighting, new scanner.
- **Label (prior) shift:** $p_{\text{te}}(y) \neq p_{\text{tr}}(y)$ while $p(x \mid y)$ is unchanged. A disease becomes more common during an outbreak.
- **Concept shift:** $p(y \mid x)$ itself changes. The definition of "spam" evolves.

Under pure covariate shift, the expected test loss can be rewritten as a weighted training loss, $\mathbb{E}_{\text{te}}[\ell] = \mathbb{E}_{\text{tr}}[w(x)\, \ell]$ with importance weight $w(x) = p_{\text{te}}(x)/p_{\text{tr}}(x)$. These weights are hard to estimate for images, but the identity explains why shift matters most where test inputs fall in regions with little training data. Under label shift, Lipton et al. showed that a fixed black-box

classifier's confusion matrix on held-out training data, combined with its prediction frequencies on unlabeled test data, is enough to estimate the new class priors and re-weight the model [17].

### Benchmarks that probe shift

- **ImageNet-C** applies 15 corruption types (noise, blur, weather, digital) at several severities to ImageNet validation images, and **ImageNet-P** adds perturbation sequences; together they measure robustness to common corruptions rather than adversarial attacks [13]. Our pixel-noise sweep is a miniature version.
- **ImageNetV2** re-created the ImageNet test set by following the original collection process as closely as possible. Accuracy dropped by 11–14% across many models, yet gains on the original test set translated into even larger gains on the new one, so the drop reflects slightly harder images rather than over-fitting to the reused test set [14]. Even a careful re-collection is a shift.
- **WILDS** gathers ten datasets with naturally occurring shifts, including tumor tissue from different hospitals, camera traps at new locations and satellite images from new countries, and reports a large gap between in-distribution and out-of-distribution performance that existing mitigation methods did not close [15].

The practical lesson: always add at least one *shifted* evaluation set built from the change you expect in deployment (new site, new device, later time), and report in-distribution and shifted results side by side.

## Efficiency: FLOPs, memory and latency

**Plain version.** "Fast" and "small" each have several meanings. A model with fewer operations is not automatically quicker, and a model with fewer parameters does not automatically use less memory. Measure on the target device.

### Counting operations

One **multiply–accumulate** (MAC) is a multiplication plus an addition, so 1 MAC $\approx$ 2 floating-point operations (FLOPs). For a fully connected layer with $d_{\text{in}}$ inputs and $d_{\text{out}}$ outputs, and a convolution with $C_{\text{in}}$ input channels, $C_{\text{out}}$ output channels, a $k \times k$ kernel and an $H' \times W'$ output:

$$\text{FLOPs}_{\text{linear}} \approx 2\, d_{\text{in}} d_{\text{out}}, \qquad \text{FLOPs}_{\text{conv}} \approx 2\, C_{\text{in}} k^2 C_{\text{out}} H' W'$$

per example (bias and activation costs are small). Libraries are inconsistent about whether they report MACs or FLOPs, so state which.

### Why FLOPs do not equal speed

Dehghani et al. showed that common cost indicators (parameter count, FLOPs, throughput, latency) can disagree with each other and argued that reporting only one of them gives a misleading

picture of efficiency [18]. The main reasons:

- **Memory traffic.** A layer whose weights must be read from memory for very little arithmetic is limited by memory bandwidth, not compute. A linear layer applied to a single example reads $4d_{\text{in}}d_{\text{out}}$ bytes of fp32 weights to perform $2d_{\text{in}}d_{\text{out}}$ FLOPs: 0.5 FLOPs per byte, far too little to keep a modern accelerator busy. Batching raises this ratio (called *arithmetic intensity*).
- **Parallelism.** Depthwise convolutions and many small layers have few FLOPs but use the hardware poorly.
- **Kernel support.** An operation without an optimized kernel on the target runtime may be slow regardless of its FLOP count.

### Memory accounting

At inference, memory holds weights (parameters × bytes per value), activations of the layers currently alive, workspace for kernels, and for autoregressive Transformers a **KV cache** that grows with context length and batch size. During training add gradients and optimizer states: Adam keeps two extra values per parameter, so an fp32 model with Adam needs roughly 4× its weight memory before counting activations.

### A latency protocol

Wall-clock latency depends on hardware, runtime, batch size, input resolution, numeric precision and thread settings, and the first calls are slower because of one-off initialization. A sound protocol:

1. Fix and report device, runtime version, precision, batch size, input size and thread count.
2. Run warm-up iterations and discard them.
3. Time many runs and report percentiles (p50 and p90 or p99), not just the mean: users experience the tail.
4. On GPUs, synchronize the device before stopping the timer, because kernels are launched asynchronously.
5. Measure end to end (pre-processing, transfers, decoding) as well as the network alone.

A **Pareto frontier** contains the candidates for which no other candidate is at least as good on every axis (accuracy, latency, memory) and strictly better on one. Presenting the frontier makes trade-offs visible without inventing a single "best model".

## Deployment: export, compression and runtimes

**Plain version.** Export, compression and runtime choices can change predictions and performance. Re-evaluate the artifact that will actually run on the target device.

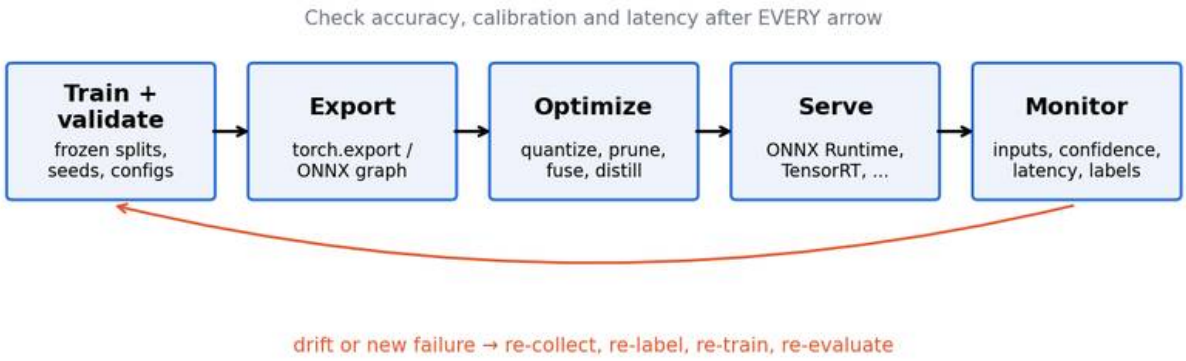



*Figure 5 — A deployment loop. Accuracy, calibration and latency should be re-checked after every step, because export, compression and runtimes all change numerics.*

### Export and inference runtimes

Training code is written in a flexible, Python-driven framework. Deployment prefers a static description of the computation:

- `torch.export` traces a PyTorch module ahead of time into an `ExportedProgram`, a graph of tensor operations that can be serialized and executed with different inputs without the original Python class [19].
- **ONNX** (Open Neural Network Exchange) is an open standard file format for ML models, so a model trained in one framework can be run by tools from others [20].
- **ONNX Runtime** is a cross-platform inference engine for ONNX models. It partitions the graph into subgraphs and hands each to a hardware-specific *execution provider* (for example CUDA, TensorRT, OpenVINO or CoreML) [21].
- **NVIDIA TensorRT** is an SDK that compiles models (from frameworks or ONNX) into optimized engines for NVIDIA GPUs, using layer fusion and reduced precisions such as FP16, BF16, FP8 and INT8 [22].

These runtimes gain speed by fusing operations, choosing fast kernels for the exact shapes and lowering precision. The same model can therefore give slightly different outputs on different runtimes. Compare exported and original outputs on a validation batch and re-measure accuracy after conversion.

### Quantization

**Quantization** stores and computes with fewer bits. In the common **affine (asymmetric) uniform** scheme, a real value $x$ is mapped to an integer $q$ and back:

$$q = \mathrm{clip}\Big(\mathrm{round}\big(\tfrac{x}{s}\big) + z,\ 0,\ 2^b - 1\Big), \qquad \hat{x} = s\,(q - z),$$

where $b$ is the bit width, $s > 0$ the **scale** (the step between representable values), $z$ the integer **zero point** chosen so that the real value 0 is represented exactly, and $\hat{x}$ the dequantized approximation. For a range $[x_{\min}, x_{\max}]$ that includes 0, $s = (x_{\max} - x_{\min})/(2^b - 1)$ and $z = \text{round}(-x_{\min}/s)$. The *symmetric* variant fixes $z = 0$ and uses a signed range. The integer-arithmetic formulation of Jacob et al. showed how to run whole networks with 8-bit integer matrix multiplication and how to train for it [23].

Error comes from two sources: **rounding** inside the range (at most $s/2$ per value) and **clipping** outside it. Both depend on the **granularity** of the scale: one scale per tensor, per output channel, or per small group of weights. Nagel et al. give a practical guide covering these choices, *post-training quantization* (PTQ: calibrate ranges on a small representative dataset after training) and *quantization-aware training* (QAT: simulate rounding during training so the weights adapt) [24]. Gholami et al. survey the wider design space [25].

```
import numpy as np

def quantize(x, bits=8, axis=None):
    """Asymmetric (affine) uniform quantization, per tensor or per channel."""
    qmin, qmax = 0, 2**bits - 1
    lo = x.min(axis=axis, keepdims=True) if axis is not None else x.min()
    hi = x.max(axis=axis, keepdims=True) if axis is not None else x.max()
    lo, hi = np.minimum(lo, 0), np.maximum(hi, 0)      # keep 0 exactly representable
    s = (hi - lo) / (qmax - qmin)                       # scale
    z = np.round(qmin - lo / s)                         # zero point (integer)
    q = np.clip(np.round(x / s) + z, qmin, qmax)        # stored integers
    return s * (q - z)                                  # dequantized values

rng = np.random.default_rng(0)
# 64 output channels whose weight magnitudes differ by up to 100x
W = rng.normal(size=(64, 256)) * np.logspace(-2, 0, 64)[:, None]
for bits in [8, 4]:
    e_t = np.abs(quantize(W, bits) - W).mean()
    e_c = np.abs(quantize(W, bits, axis=1) - W).mean()
    print(f"{bits}-bit mean |error|: per-tensor {e_t:.5f}  per-channel {e_c:.5f}")
print("zero maps back to:", quantize(np.array([-1.0, 0.0, 3.0]))[1])
```

Output:

```
8-bit mean |error|: per-tensor 0.00632  per-channel 0.00124
4-bit mean |error|: per-tensor 0.06718  per-channel 0.02112
zero maps back to: 0.0
```

When channels have very different ranges, one shared scale wastes most of the grid on the largest channel and rounds small channels coarsely. Per-channel scales cut the error about fivefold at 8 bits here. In large language models, a few outlier activation channels cause the same problem at a larger scale, which is what methods such as GPTQ (layer-wise error-compensating weight rounding) and AWQ (protecting salient weight channels identified from activations) address for 3–4-bit weight-only quantization [42], [43].

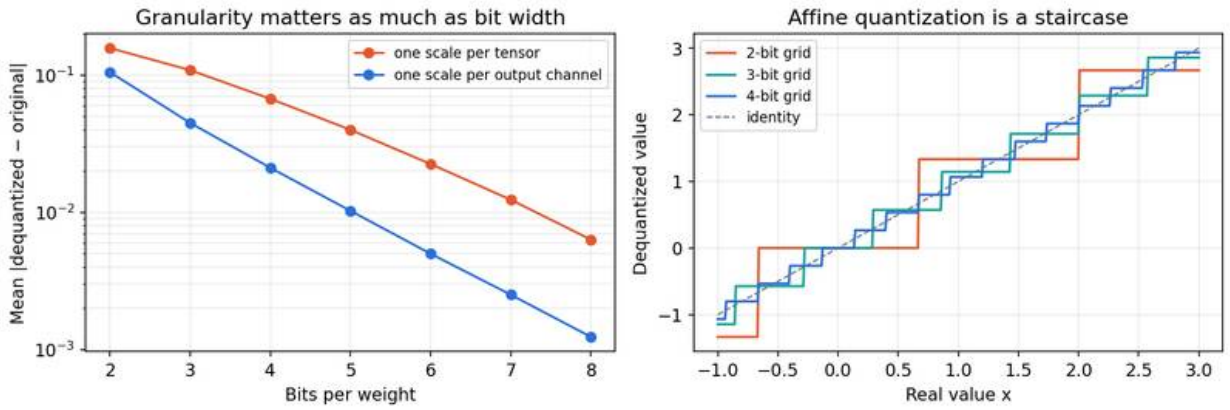



*Figure 6 — Quantization error falls roughly by half for each extra bit, and the choice of scale granularity shifts the whole curve.*

## Pruning

**Pruning** removes weights, channels, attention heads or blocks that contribute little. *Unstructured* pruning zeros individual weights; it can remove a large fraction of parameters, as in Deep Compression, which combined pruning, weight sharing via quantization, and Huffman coding to shrink networks substantially [26]. But scattered zeros save time only if the runtime has sparse kernels for that pattern. *Structured* pruning removes whole channels or heads, producing smaller dense tensors that any runtime can exploit. Blalock et al. reviewed 81 pruning papers and found that inconsistent baselines and metrics made it hard to tell which methods actually work, and proposed a standardized benchmark [27]. Hoefler et al. survey sparsity in training and inference in depth [28].

## Knowledge distillation

**Distillation** trains a small *student* to imitate a large *teacher*. Hinton et al. proposed matching the teacher's softened output distribution [29]. With temperature $T$, logits $\ell^t$ (teacher) and $\ell^s$ (student), softened probabilities $p^{(T)} = \mathrm{softmax}(\ell/T)$, and weight $\lambda \in [0,1]$:

$$\mathcal{L} = (1-\lambda)\,\mathcal{L}_{\mathrm{CE}}(y,\, p^{s,(1)}) + \lambda\, T^2\, D_{\mathrm{KL}}(p^{t,(T)} \,\|\, p^{s,(T)}).$$

The softened teacher distribution carries "dark knowledge" about which wrong classes are similar. The gradients of the soft term scale as $1/T^2$, so multiplying by $T^2$ keeps the two terms balanced when $T$ changes [29]. Gou et al. organize later work by *what* is transferred (responses, intermediate features, relations between examples) and *how* (offline, online, self-distillation) [30]. A student inherits the teacher's mistakes and biases as well as its skill.

### Code: weight-only int8, latency and export

This sketch stores the weights of a small MLP as int8 with one scale per output channel and measures size and latency on the CPU. It uses only core PyTorch.

```
import io, time
import numpy as np
import torch
from torch import nn

torch.manual_seed(0)
model = nn.Sequential(nn.Linear(512, 2048), nn.ReLU(),
                      nn.Linear(2048, 2048), nn.ReLU(),
                      nn.Linear(2048, 10)).eval()
x = torch.randn(32, 512)

class Int8WeightLinear(nn.Module):
    """Weight-only int8 storage, symmetric per output channel."""
    def __init__(self, lin):
        super().__init__()
        w = lin.weight.detach()
        scale = w.abs().amax(dim=1, keepdim=True) / 127
        self.register_buffer("q", torch.round(w / scale).to(torch.int8))
        self.register_buffer("scale", scale)
        self.register_buffer("bias", lin.bias.detach().clone())
    def forward(self, x):                       # dequantize, then fp32 matmul
        return nn.functional.linear(x, self.q.float() * self.scale, self.bias)

int8 = nn.Sequential(*[Int8WeightLinear(m) if isinstance(m, nn.Linear) else m
                       for m in model]).eval()

def size_mb(m):
    buf = io.BytesIO()
    torch.save(m.state_dict(), buf)
    return buf.getbuffer().nbytes / 1e6

def latency_ms(m, x, warmup=10, runs=100):
    with torch.inference_mode():
        for _ in range(warmup):                 # first calls pay one-off costs
            m(x)
        times = []
        for _ in range(runs):
            t0 = time.perf_counter()
            m(x)
            times.append((time.perf_counter() - t0) * 1e3)
    return np.percentile(times, [50, 90]).round(2)

with torch.inference_mode():
    gap = (model(x) - int8(x)).abs().max().item()
    agree = (model(x).argmax(1) == int8(x).argmax(1)).float().mean().item()
print(f"fp32 file: {size_mb(model):.1f} MB  p50/p90 latency {latency_ms(model, x)} ms")
print(f"int8 file: {size_mb(int8):.1f} MB  p50/p90 latency {latency_ms(int8, x)} ms")
print(f"max |logit difference| = {gap:.4f}, top-1 agreement = {agree:.3f}")
```

```
# Ahead-of-time graph capture: a portable program, independent of this Python class.
exported = torch.export.export(model, (x,))
ops = [n.target for n in exported.graph.nodes if n.op == "call_function"]
print(type(exported).__name__, "->", [str(o).split(".")[1] for o in ops])
```

One CPU run (four threads) printed:

```
fp32 file: 21.1 MB  p50/p90 latency [2.74 4.49] ms
int8 file: 5.3 MB  p50/p90 latency [ 9.23 11.06] ms
max |logit difference| = 0.0018, top-1 agreement = 1.000
ExportedProgram -> ['linear', 'relu', 'linear', 'relu', 'linear']
```

Your latencies will differ; the pattern is the point. The file is four times smaller and the outputs barely change. Yet the int8 version is *slower*, because it converts the weights back to fp32 on every call and then runs the same fp32 matrix multiplication. Lower precision becomes faster only when the runtime has integer (or low-precision) kernels for that operation, or when the workload is limited by memory bandwidth so that reading fewer bytes dominates. This is exactly the gap between "compressed" and "accelerated" that runtimes such as TensorRT, ONNX Runtime and PyTorch's own `torchao` library [46] are built to close. The FLOP count of this MLP is $2 \times (512 \cdot 2048 + 2048 \cdot 2048 + 2048 \cdot 10) \approx 10.5$ MFLOPs per example, identical for both versions.

## Monitoring after deployment

**Plain version.** A model can get worse without any code changing, because the data changes. Watch the inputs and outputs, collect fresh labels when possible, and decide in advance what triggers an investigation.

Sculley et al. describe how ML systems accumulate hidden "technical debt": entangled features, undeclared consumers of predictions, feedback loops where the model's outputs influence its future training data, and dependence on external data that changes without notice [33]. Paleyes et al. collected published deployment case studies and found problems at every stage of the workflow [39]. Useful signals, from cheapest to most informative:

- **System health:** latency percentiles, error rates, memory, throughput.
- **Input statistics:** image brightness, resolution, class of device, missing fields; compared with a reference window from training time.
- **Output statistics:** distribution of predicted classes and of confidence. These need no labels.
- **Delayed labels:** a small, regularly labelled sample of live traffic gives the only direct estimate of accuracy.

To compare a live window with the reference, a two-sample test is a simple starting point. The **Kolmogorov–Smirnov statistic** for one-dimensional samples with empirical cumulative distribution functions $F_{\text{ref}}$ and $F_{\text{live}}$ is

$$D = \sup_t |F_{\text{ref}}(t) - F_{\text{live}}(t)|,$$

the largest vertical gap between the two curves. Rabanser et al. compared shift detectors and found that a two-sample test applied after reducing the inputs with a pre-trained classifier (for example to its softmax outputs) performed best across the shifts they studied [32].

```
import numpy as np
from scipy.stats import ks_2samp
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

X, y = load_digits(return_X_y=True)
X = X / 16.0
Xtr, Xlive, ytr, ylive = train_test_split(X, y, test_size=0.5, random_state=0)
Xtr, Xref, ytr = Xtr[:600], Xtr[600:], ytr[:600]   # reference window from training time
clf = LogisticRegression(max_iter=2000).fit(Xtr, ytr)
ref_conf = clf.predict_proba(Xref).max(1)     # label-free signal: top-class confidence

rng = np.random.default_rng(0)
for name, noise in [("week 1 (clean)", 0.0), ("week 2 (new camera)", 0.25)]:
    batch = rng.choice(len(Xlive), 200, replace=False)
    Xb = np.clip(Xlive[batch] + noise * rng.normal(size=Xlive[batch].shape), 0, 1)
    conf = clf.predict_proba(Xb).max(1)
    stat, p = ks_2samp(ref_conf, conf)
    acc = (clf.predict(Xb) == ylive[batch]).mean()   # usually NOT available live
    print(f"{name}: mean confidence {conf.mean():.3f}, KS={stat:.3f}, p={p:.2g}, "
          f"(hidden) accuracy {acc:.3f}")
```

Output:

```
week 1 (clean): mean confidence 0.846, KS=0.089, p=0.28, (hidden) accuracy 0.950
week 2 (new camera): mean confidence 0.757, KS=0.277, p=1.3e-08, (hidden) accuracy 0.885
```

The confidence distribution flags week 2 without a single label, and the hidden accuracy confirms that something real happened. A detector says *that* the data changed, not *whether* accuracy fell: harmless shifts also trigger it, and with large windows even tiny changes become "significant". Use it to trigger investigation and labelling, not automatic retraining.

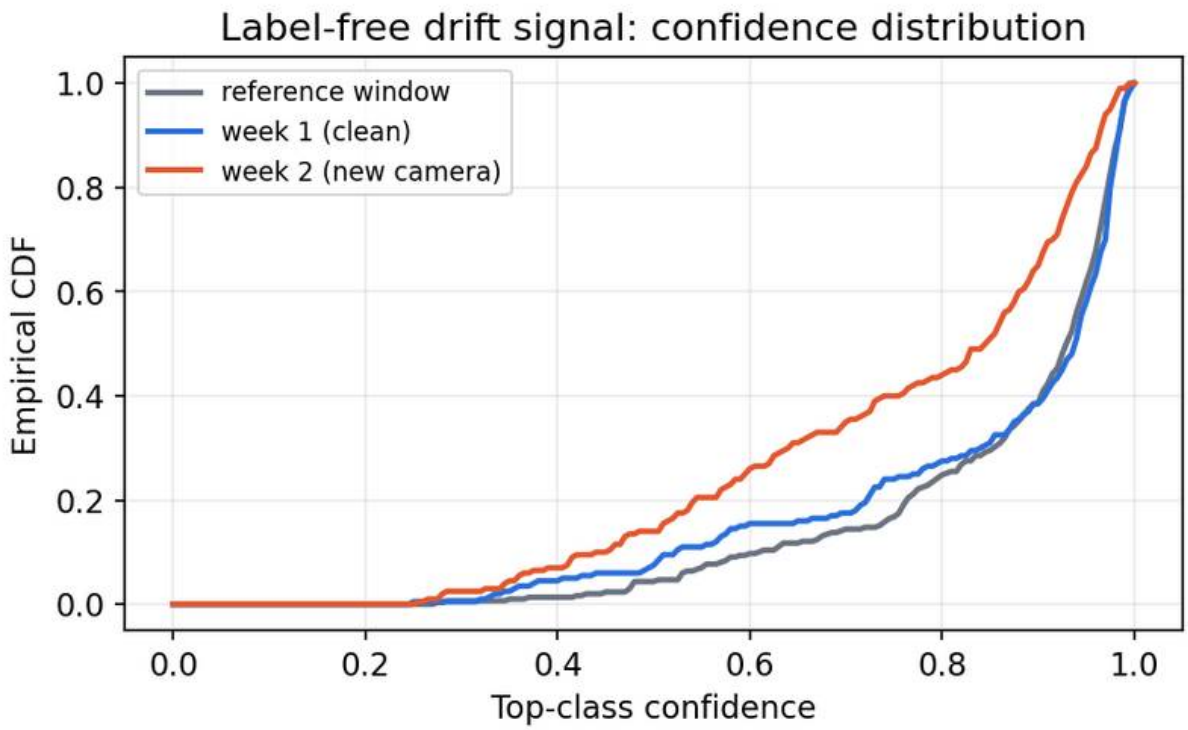



*Figure 7 — The Kolmogorov–Smirnov statistic is the largest vertical gap between the reference curve and a live window's curve.*

## Reproducibility and documentation

**Plain version.** Someone else (including you in six months) should be able to rerun your experiment and get the same numbers, and a reader should be able to tell what the model is for and where it should not be used.

### Making runs repeatable

Fix and record random seeds for every library, but know their limits: the PyTorch documentation states that completely reproducible results are not guaranteed across releases, commits or platforms, and recommends seeding, `torch.use_deterministic_algorithms(True)` and disabling cuDNN benchmarking when bit-exact repetition matters [34]. Beyond seeds, record:

- the dataset version, split IDs (the actual lists of example IDs) and deduplication rule;
- preprocessing and augmentation code, pretrained checkpoint and its source;
- the full configuration (optimizer, schedule, stopping rule), software environment and hardware;
- every validation decision made, and the single final test evaluation.

The NeurIPS 2019 reproducibility program, reported by Pineau et al., introduced a code-submission policy, a community reproducibility challenge and a checklist covering exactly these items [37].

### Documenting models and data

**Model cards** are short documents that accompany a trained model and report its intended use, the evaluation data, performance broken down by relevant groups and conditions, and known limitations [35]. **Datasheets for datasets** ask dataset creators to document motivation, composition, collection process, preprocessing, recommended uses and maintenance [36]. Both make the evaluation decisions in this chapter visible to the people who will rely on them.

### A capstone experiment

Choose one task with a modest, legally usable dataset. Define the deployment population and the split unit. Compare three candidates: a classical baseline, frozen pretrained features with a small head, and one fine-tuned network. For each, report the test metric with a bootstrap interval, a paired comparison against the best baseline, a reliability diagram, results on one shifted set, and latency percentiles on the target device. Write a one-page model card. Open the final test set once; if it reveals a failure and you revise the model, treat the old result as development evidence and obtain a fresh evaluation for the revised claim.

## Modern view

The engineering of evaluation and deployment is now a research field in its own right. Several surveys and reviews map it.

**Leakage and reproducibility.** Kapoor and Narayanan's review of ML-based science documents leakage in 329 papers across 17 fields and proposes "model info sheets" that make authors argue why their train/test separation is valid [4]. Combined with the variance study of Bouthillier et al. [10] and the NeurIPS reproducibility program [37], the consensus is that most over-optimistic results come from protocol errors, not from exotic statistics.

**Uncertainty and calibration.** Gawlikowski et al. survey uncertainty estimation in deep networks: they separate reducible model (epistemic) uncertainty from irreducible data (aleatoric) uncertainty, organize methods into deterministic networks, Bayesian networks, ensembles and test-time augmentation, and review calibration methods and their practical limits [38]. Ovadia et al. add the empirical warning that post-hoc calibration falls short under shift while methods that marginalize over models, such as ensembles, hold up better [16].

**Deployment in practice.** Paleyes et al. survey published case studies, map the challenges onto the stages of the deployment workflow and show that practitioners hit problems at every stage [39], extending the technical-debt view of Sculley et al. [33].

**Compression.** Cheng et al. give the classic overview of pruning, low-rank factorization, compact convolutions and distillation for CNNs [31]; Gholami et al. [25] and Nagel et al. [24] focus on quantization; Hoefler et al. on sparsity [28]; Gou et al. on distillation [30]. For large language models, Zhu et al. survey quantization, pruning, distillation and low-rank methods specific to models with billions of parameters [41]. A visible 2023–2026 trend is that weight-only quantization to 4 bits (GPTQ [42], AWQ [43]) and low-precision formats such as FP8 have moved from papers into production toolkits: TensorRT lists FP8, INT8, FP4 and INT4 among its supported precisions [22], and PyTorch's quantization tooling is consolidating in the `torchao` library [46].

**Holistic evaluation.** For foundation models, HELM evaluates many models on many scenarios and on several metrics at once (accuracy, calibration, robustness, fairness, bias, toxicity and efficiency) and argues that a single leaderboard number hides these trade-offs [40]. The same multi-metric habit applies to vision models.

**Scaling.** Scaling laws fit empirical relationships among model size, data, compute and loss. Hoffmann et al. found that, for a fixed compute budget, language models should scale parameters and training tokens roughly equally [44]; Zhai et al. studied how ViT accuracy scales with model size, data and compute [45]. Fitted constants from one domain should not be copied into another; the transferable lesson is to report the compute and data behind every result.

**Open problems.** Evaluation sets leaking into web-scale pretraining data (contamination) is hard to rule out; calibration under unknown shift remains unsolved; latency claims rarely transfer between hardware generations; and evaluation of generative and interactive systems (see DLCV 13) still lacks agreed protocols.

## Key takeaways

- Split by what will be new at deployment and fit preprocessing within each training fold.
- Use grouped or temporal folds for dependent data; separate model selection from reported evaluation.
- Compare models on paired examples and estimate uncertainty using the actual independent sampling unit.
- Calibration differs from accuracy; temperature scaling can help on matched data but does not guarantee calibration after shift.
- Operation counts, parameter counts, memory and latency measure different costs; benchmark the complete target pipeline.
- Re-evaluate after export or compression, monitor changing data, and record versions, protocols and intended use.

## Exercises

1. **Leakage audit.** A colleague fits PCA on all 1,797 digits, then splits into train and test and reports test accuracy. Explain the leak, and why it exists even though PCA never sees labels. How would you fix the code?

**Hint**

The principal directions are computed using the test images, so the representation is tuned to the test distribution. Put `PCA` inside a `Pipeline` so it is fitted only on each training fold. The bias is often small for PCA, but for supervised preprocessing (feature selection using labels) it can be large.

2. **Rare positives.** A dataset has 1% positive images. What accuracy, positive recall and balanced accuracy does an always-negative classifier achieve? Which metric would you report to a doctor, and why?

**Hint**

Accuracy 99%, recall 0, balanced accuracy 0.5 (mean of recall 0 on positives and 1 on negatives). Report recall (sensitivity) and specificity or precision at a clinically chosen operating point, plus their intervals, because the cost of a missed positive is what matters.

3. **How big must the test set be?** You expect accuracy around 90% and want a 95% confidence interval of about $\pm 1$ percentage point. Using the normal approximation, roughly how many independent test examples do you need? What changes if the images come from 50 patients?

**Hint**

Solve $1.96\sqrt{0.9 \cdot 0.1/n} = 0.01$, giving $n \approx 3{,}460$. With 50 patients the examples are not independent; the effective sample size is closer to 50 than to the number of images, so the interval will be much wider. Use a bootstrap that resamples patients.

4. **Temperature by hand.** A three-class model outputs logits $(4, 1, 0)$. Compute the top-class probability with $T = 1$ and $T = 2$. Does the predicted class change? When would a fitted $T$ be less than 1?

**Hint**

$T = 1$: $e^4/(e^4 + e^1 + e^0) \approx 0.936$. $T = 2$: $e^2/(e^2 + e^{0.5} + 1) \approx 0.736$. The argmax is unchanged. $T < 1$ is fitted when the model is underconfident, for example after heavy regularization or label smoothing.

5. **Compression claim.** A pruned model has half the parameters of the original but identical latency on your GPU. Give three possible explanations and one experiment to tell them apart.

**Hint**

Unstructured zeros are still multiplied by dense kernels; the remaining cost is dominated by activations, memory traffic or unchanged layers; or the batch is so small that kernel-launch overhead dominates. Profile per-layer time, and compare with a structured-pruned model whose tensors are actually smaller.

6. **Design a monitor.** Your defect detector runs on a factory line; labels arrive two weeks late from manual inspection. Design a monitoring plan: what you log, which statistics you compare with which reference, what threshold triggers an alert, and what happens after an alert.

**Hint**

Log latency, image statistics (brightness, blur), predicted-defect rate and confidence histograms daily; compare with a reference window from validation using KS or a population-stability measure; alert on sustained change rather than single days. After an alert, label a sample immediately, compute accuracy with an interval, and only then decide whether to recalibrate, fine-tune or roll back.

## References


1. I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016, ch. 11 “Practical Methodology”. book
2. G. James, D. Witten, T. Hastie, R. Tibshirani and J. Taylor, *An Introduction to Statistical Learning: with Applications in Python*, Springer, 2023. book
3. scikit-learn developers, “Common pitfalls and recommended practices,” scikit-learn documentation. docs
4. S. Kapoor and A. Narayanan, “Leakage and the reproducibility crisis in machine-learning-based science,” *Patterns*, vol. 4, no. 9, 2023. doi
5. S. Varma and R. Simon, “Bias in error estimation when using cross-validation for model selection,” *BMC Bioinformatics*, vol. 7, art. 91, 2006. doi
6. G. C. Cawley and N. L. C. Talbot, “On over-fitting in model selection and subsequent selection bias in performance evaluation,” *Journal of Machine Learning Research*, vol. 11, pp. 2079–2107, 2010. paper
7. M. Bińkowski, D. J. Sutherland, M. Arbel and A. Gretton, “Demystifying MMD GANs,” ICLR, 2018. arXiv
8. T. G. Dietterich, “Approximate statistical tests for comparing supervised classification learning algorithms,” *Neural Computation*, vol. 10, no. 7, 1998. doi
9. B. Efron, “Bootstrap methods: another look at the jackknife,” *The Annals of Statistics*, vol. 7, no. 1, 1979. doi
10. X. Bouthillier et al., “Accounting for variance in machine learning benchmarks,” MLSys, 2021. arXiv
11. C. Guo, G. Pleiss, Y. Sun and K. Q. Weinberger, “On calibration of modern neural networks,” ICML, 2017. arXiv
12. M. Minderer et al., “Revisiting the calibration of modern neural networks,” NeurIPS, 2021. arXiv
13. D. Hendrycks and T. Dietterich, “Benchmarking neural network robustness to common corruptions and perturbations,” ICLR, 2019. arXiv
14. B. Recht, R. Roelofs, L. Schmidt and V. Shankar, “Do ImageNet classifiers generalize to ImageNet?,” ICML, 2019. paper
15. P. W. Koh et al., “WILDS: A benchmark of in-the-wild distribution shifts,” ICML, 2021. paper
16. Y. Ovadia et al., “Can you trust your model’s uncertainty? Evaluating predictive uncertainty under dataset shift,” NeurIPS, 2019. arXiv
17. Z. C. Lipton, Y.-X. Wang and A. Smola, “Detecting and correcting for label shift with black box predictors,” ICML, 2018. arXiv

18. M. Dehghani, A. Arnab, L. Beyer, A. Vaswani and Y. Tay, “The efficiency misnomer,” ICLR, 2022. arXiv
19. PyTorch, “torch.export,” PyTorch documentation. docs
20. ONNX, “Open Neural Network Exchange: the open standard for machine learning interoperability.” site
21. ONNX Runtime, “Welcome to ONNX Runtime,” documentation. docs
22. NVIDIA, “NVIDIA TensorRT documentation.” docs
23. B. Jacob et al., “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” CVPR, 2018. doi
24. M. Nagel et al., “A white paper on neural network quantization,” arXiv:2106.08295, 2021. arXiv
25. A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney and K. Keutzer, “A survey of quantization methods for efficient neural network inference,” book chapter in *Low-Power Computer Vision*; arXiv:2103.13630, 2021. arXiv
26. S. Han, H. Mao and W. J. Dally, “Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding,” ICLR, 2016. arXiv
27. D. Blalock, J. J. Gonzalez Ortiz, J. Frankle and J. Guttag, “What is the state of neural network pruning?,” MLSys, 2020. arXiv
28. T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden and A. Peste, “Sparsity in deep learning: pruning and growth for efficient inference and training in neural networks,” *Journal of Machine Learning Research*, vol. 22, 2021. arXiv
29. G. Hinton, O. Vinyals and J. Dean, “Distilling the knowledge in a neural network,” NIPS Deep Learning Workshop, 2014; arXiv:1503.02531. arXiv
30. J. Gou, B. Yu, S. J. Maybank and D. Tao, “Knowledge distillation: a survey,” *International Journal of Computer Vision*, vol. 129, 2021. doi
31. Y. Cheng, D. Wang, P. Zhou and T. Zhang, “Model compression and acceleration for deep neural networks: the principles, progress, and challenges,” *IEEE Signal Processing Magazine*, vol. 35, no. 1, 2018. doi arXiv
32. S. Rabanser, S. Günnemann and Z. C. Lipton, “Failing loudly: an empirical study of methods for detecting dataset shift,” NeurIPS, 2019. arXiv
33. D. Sculley et al., “Hidden technical debt in machine learning systems,” NeurIPS, 2015. paper
34. PyTorch, “Reproducibility,” PyTorch documentation. docs
35. M. Mitchell et al., “Model cards for model reporting,” ACM FAT*, 2019. arXiv
36. T. Gebru et al., “Datasheets for datasets,” *Communications of the ACM*, vol. 64, no. 12, 2021. doi
37. J. Pineau et al., “Improving reproducibility in machine learning research (a report from the NeurIPS 2019 reproducibility program),” *Journal of Machine Learning Research*, vol. 22, 2021. paper
38. J. Gawlikowski et al., “A survey of uncertainty in deep neural networks,” *Artificial Intelligence Review*, vol. 56, 2023. doi
39. A. Paleyes, R.-G. Urma and N. D. Lawrence, “Challenges in deploying machine learning: a survey of case studies,” *ACM Computing Surveys*, vol. 55, no. 6, 2022. doi
40. P. Liang et al., “Holistic evaluation of language models,” *Transactions on Machine Learning Research*, 2023. arXiv
41. X. Zhu, J. Li, Y. Liu, C. Ma and W. Wang, “A survey on model compression for large language models,” *Transactions of the Association for Computational Linguistics*, vol. 12, 2024. doi

42. E. Frantar, S. Ashkboos, T. Hoefler and D. Alistarh, “GPTQ: accurate post-training quantization for generative pre-trained transformers,” ICLR, 2023. arXiv
43. J. Lin et al., “AWQ: activation-aware weight quantization for LLM compression and acceleration,” MLSys, 2024. arXiv
44. J. Hoffmann et al., “Training compute-optimal large language models,” arXiv:2203.15556, 2022; NeurIPS 2022 version titled “An empirical analysis of compute-optimal large language model training.” arXiv
45. X. Zhai, A. Kolesnikov, N. Houlsby and L. Beyer, “Scaling vision transformers,” CVPR, 2022. doi
46. PyTorch, “torchao documentation.” docs

# Part II

# Deep Learning for Computer Vision

# DLCV 01 • Deep Learning for Image Processing and Computer Vision

How deep learning reorganizes image processing, image analysis and computer vision: from the classical pattern-recognition pipeline to CNNs, Vision Transformers, self-supervised backbones and promptable foundation models.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/deep-learning-for-image-processing/

## What you'll learn

- Distinguish image-to-image tasks from image-to-semantics tasks.
- Explain end-to-end learning and pretrained backbone–head pipelines.
- Relate convolution, stride, padding and pooling to resolution and receptive field.
- Compare CNN, encoder–decoder and Transformer design choices.
- Choose a starting point for recognition, restoration, generation or vision–language tasks.

## The big picture

This tutorial is the entry point of the deep-learning track. It does not go deep into any one task. Instead it gives you the map: what kinds of image problems exist, which network designs solve them, and why those designs look the way they do. The later tutorials then zoom in on image classification, object detection, image restoration, low-light enhancement and object tracking.

> **Explain like I'm five**
>
> Think of a photo lab with two kinds of workers. The **repair crew** takes a scratched, blurry or dark photo and hands back a cleaner photo: picture in, picture out. The **detective** looks at a photo and tells you *what* is in it: "a cat, sitting on a sofa, on the left." Old-style computer vision gave both workers a fixed toolbox designed by experts. Deep learning lets them build their own tools by looking at millions of examples. Today there is even a "super-worker" who learned from billions of pictures and can be told what to do with a few words.

Why it matters: almost every modern camera pipeline, medical-imaging product, driver-assistance system and photo app is built from the pieces in this tutorial. Knowing the map lets you pick a sensible starting point for a new problem, read papers critically, and understand which classical ideas from the Digital Image Processing series survive inside the networks (many do).

> **About this tutorial**

This tutorial is expanded from the author's earlier post, written in Traditional Chinese, "深度學習的數位影像處理介紹" (An introduction to digital image processing with deep learning) [1]. It keeps the post's structure and viewpoints and adds the details, math and code.

**Related chapters.** DLCV 01A turns the ideas here into a practical training workflow: data pipelines, learning-rate schedules, regularization, debugging and fair experiments. DLCV 01B collects the building blocks (residual and attention blocks, normalization, upsampling, skips) and shows how classification and restoration networks compose them differently.

## Three levels of image tasks

**Plain version.** Image work can improve pixels, extract structures or properties, or infer scene meaning. Classify the task by its input and output.

**Precise version.** Following Gonzalez and Woods [2], we can sort image work into three levels by what goes in and what comes out.

| Level | Input → output | Typical operations | Classical examples |
|---|---|---|---|
| **Image processing** (narrow sense; low level) | image → image | primitive, pixel- or neighborhood-wise steps | histogram equalization, filtering, Fourier transform (IP 03, IP 04) |
| **Image analysis** (mid level) | image → attributes or highlighted regions | segmentation, feature extraction, description, using prior knowledge | skin-color regions in a face image, vessel regions in an X-ray (IP 10, IP 12) |
| **Computer vision** (high level; understanding) | image → meaning | recognition, interpretation, decision | classification, "is there a pedestrian?" (IP 13) |

The boundaries are soft. Real systems mix all three. A vessel-segmentation tool, for instance, first denoises (low level), then extracts a vessel map (mid level), and may then flag a stenosis (high level). The levels describe *kinds of computation*, not separate departments.

### How deep learning redraws the map

In deep-learning papers you will almost never see the word "mid-level". The field splits tasks into just two families, according to the **shape of the output** [1]:

- **Low-level vision**: the output is an image of (roughly) the same size as the input, and the target is a pixel-accurate signal. Denoising, deblurring, super-resolution, dehazing, low-light enhancement, image-to-image translation.
- **High-level vision**: the output is a *semantic* description. A label, boxes with labels, or a label per pixel.

The old "image analysis" level is absorbed into one of the two. Segmentation produces a per-pixel map, so it *looks* like an image-to-image task. But its values are class labels that require recogni-

tion, so it is grouped with high-level vision. Edge or vessel enhancement that outputs a cleaner intensity image is grouped with low-level vision.

The reason for the split is practical. The two families differ in the loss (pixel regression versus classification), the architecture (full-resolution encoder–decoders versus downsampling backbones), the training data (pairs of degraded and clean images versus human labels) and the metrics (fidelity versus accuracy). Figure 1 shows the reorganization.

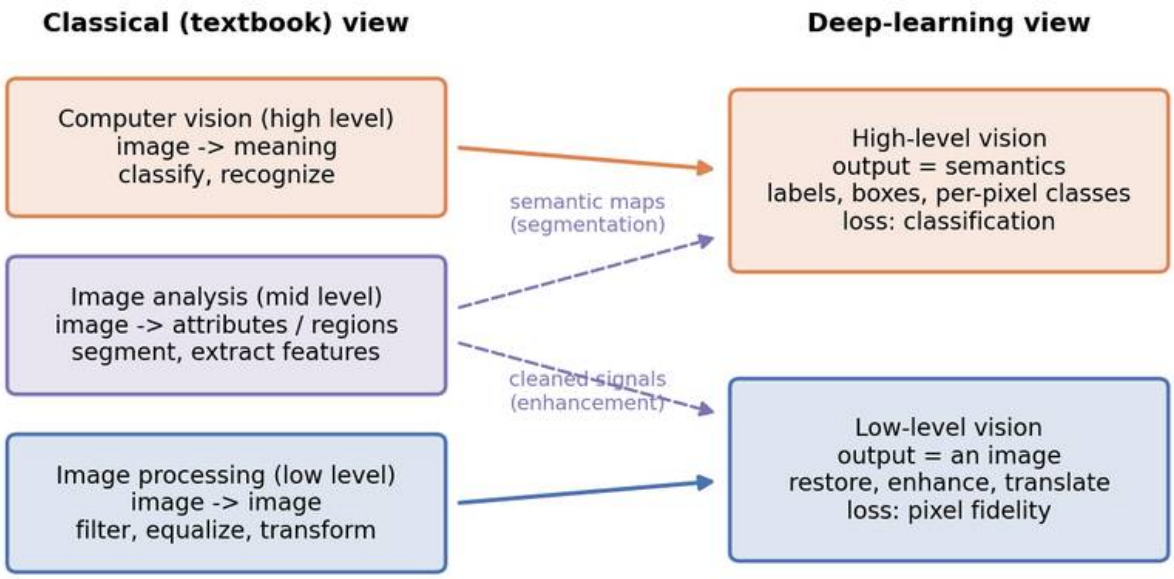



*Figure 1 — The classical three levels (left) and the two families used in deep learning (right). Mid-level image analysis is split between them according to whether the output is a cleaned-up signal or a semantic map.*

## From the pattern-recognition pipeline to end-to-end learning

**Plain version.** Classical recognition separates hand-designed features from classification. End-to-end learning trains both together.

**Precise version.** The classical pipeline [3] has four stages:

1. **Sensor.** Capture the pattern (camera, scanner, microscope).
2. **Feature generation and selection.** Compute a vector $\mathbf{x} = \phi(I) \in \mathbb{R}^d$ from image $I$, using a hand-designed map $\phi$ (moments, texture statistics, edge histograms, keypoint descriptors), then keep the most informative entries.
3. **Classifier design.** Learn a decision rule $\hat{y} = g(\mathbf{x}; \theta)$ with parameters $\theta$, for example a Bayes classifier, a nearest-neighbor rule or a support-vector machine.

4. **System evaluation.** Measure the error rate on held-out data and go back to step 2 or 3 if it is too high.

Only $\theta$ is learned. The feature map $\phi$ is fixed by an engineer, so the whole system is only as good as $\phi$. The chapter on feature extraction (IP 12) shows how much ingenuity went into those maps.

**End-to-end learning** makes the feature map learnable too. A deep network computes

$$\hat{y} = g(\phi(I; \mathbf{w}); \theta),$$

where $\phi(\cdot\,; \mathbf{w})$ is a stack of layers with weights $\mathbf{w}$, $g(\cdot\,; \theta)$ is a small head, and both $\mathbf{w}$ and $\theta$ are fitted jointly by minimizing a loss $\mathcal{L}(\hat{y}, y)$ over labeled pairs $(I, y)$ with gradient descent and back-propagation. The input is raw pixels; nothing is hand-designed except the architecture and the loss.

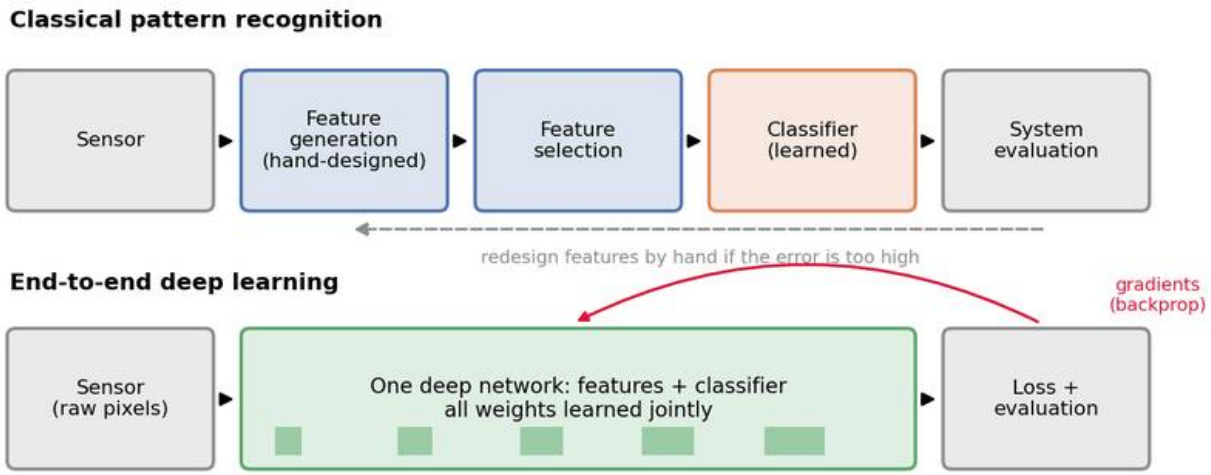



*Figure 2 — Classical pattern recognition (top) versus end-to-end learning (bottom). In the deep network the functional split into 'features' and 'classifier' still exists, but there is no clean boundary between them.*

The functional split still exists inside a network: early layers behave like feature extractors and the last layers like a classifier. But in practice you usually cannot point at a layer and say "here the features end" [1]. Two consequences follow.

- **Data replaces design.** End-to-end systems need many labeled examples; the hand-designed $\phi$ encoded prior knowledge that the network must now learn from data.
- **The split came back at a larger scale.** Since about 2021 the standard recipe is a **pretrained backbone** (trained once, often without labels, on a huge dataset) plus a **light task head** trained on your small dataset. That is "feature extractor + classifier" again, except the feature extractor is learned rather than designed. We return to this in the section on self-supervised backbones.

## Convolutional neural networks

**Plain version.** A convolutional layer slides a small window over the image and computes a weighted sum at every position, exactly like the spatial filters of IP 03. The difference is that a CNN *learns* the weights of its filters, uses many filters at once, and stacks dozens of such layers with nonlinearities in between.

### A convolutional layer is a bank of learned filters

**Precise version.** Let the input feature map be $X \in \mathbb{R}^{C_{\text{in}} \times H \times W}$ (channels, height, width). A convolutional layer with $C_{\text{out}}$ filters of size $k \times k$ computes

$$Y_o(x,y) = b_o + \sum_{c=1}^{C_{\text{in}}} \sum_{i=-r}^{r} \sum_{j=-r}^{r} W_{o,c}(i,j)\, X_c(x+i,\, y+j), \qquad o = 1, \dots, C_{\text{out}},$$

where $r = (k-1)/2$ for odd $k$, $W_{o,c}(i,j)$ is the weight of filter $o$ on input channel $c$ at offset $(i, j)$, and $b_o$ is a bias. Strictly this is a **correlation** (no kernel flip), as explained in the convolution tutorial; since the weights are learned, the flip makes no difference. An elementwise nonlinearity such as $\text{ReLU}(z) = \max(0, z)$ follows. Without it, a stack of convolutions would collapse into one big linear filter [1].

Two properties make this layer the right tool for images:

- **Local connectivity.** Each output depends only on a $k \times k$ neighborhood, because nearby pixels are strongly related and distant ones much less so.
- **Weight sharing.** The same filter is used at every position. This makes the layer **translation-equivariant** (shift the input, and the output shifts the same way) and slashes the number of parameters.

The saving is huge. A fully connected layer from a $224 \times 224 \times 3$ image to 64 units needs $224 \cdot 224 \cdot 3 \cdot 64 + 64 \approx 9.6$ million parameters. A $3 \times 3$ convolution from 3 to 64 channels needs $3 \cdot 3 \cdot 3 \cdot 64 + 64 = 1{,}792$, and it produces 64 *full-resolution* feature maps instead of 64 numbers.

The code below sets the two filters of a PyTorch convolution by hand to Sobel kernels, which turns the layer into an edge detector. Training would instead set these numbers by gradient descent.

```
import torch, torch.nn as nn, torch.nn.functional as F
from skimage import data, img_as_float

x = torch.tensor(img_as_float(data.camera()), dtype=torch.float32)[None, None]  # (1, 1, 512,
 ↪  512)
conv = nn.Conv2d(1, 2, kernel_size=3, padding=1, bias=False)
sobel_x = torch.tensor([[-1., 0., 1.], [-2., 0., 2.], [-1., 0., 1.]])
with torch.no_grad():
    conv.weight[0, 0] = sobel_x        # channel 0: responds to vertical edges
    conv.weight[1, 0] = sobel_x.T      # channel 1: responds to horizontal edges
    y = F.relu(conv(x))                # (1, 2, 512, 512): two feature maps
```

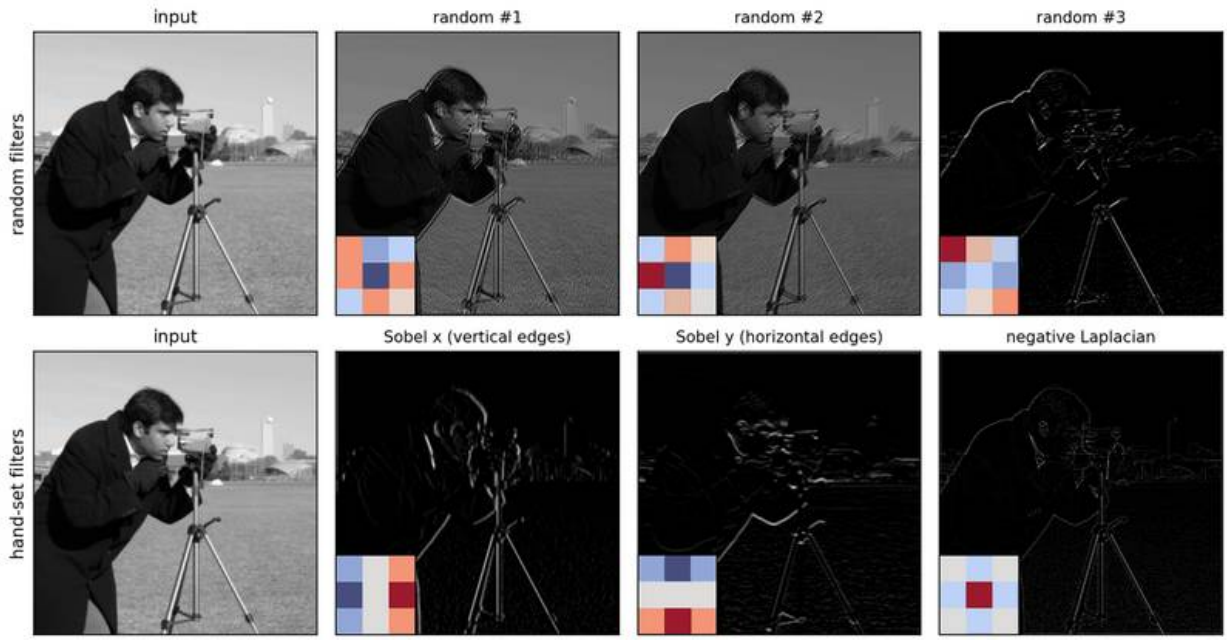



*Figure 3 — Feature maps (after ReLU) of 3×3 convolutions on the camera image. Top: random initial weights, which give noisy mixtures of smoothing and edges. Bottom: hand-set Sobel and Laplacian weights. Training moves the random filters toward useful ones.*

What do trained first-layer filters look like? We can train a tiny CNN on the 8×8 handwritten digits that ship with scikit-learn [36] in a few seconds on a CPU.

```
import torch, torch.nn as nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

torch.manual_seed(0)
X, y = load_digits(return_X_y=True)                    # 1,797 images of 8x8 pixels
X = torch.tensor(X / 16.0, dtype=torch.float32).reshape(-1, 1, 8, 8)
Xtr, Xte, ytr, yte = train_test_split(X, torch.tensor(y), test_size=0.25, random_state=0)

net = nn.Sequential(
    nn.Conv2d(1, 8, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),     # 8x8 -> 4x4
    nn.Conv2d(8, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),    # 4x4 -> 2x2
    nn.Flatten(), nn.Linear(16 * 2 * 2, 10),
)
opt = torch.optim.Adam(net.parameters(), lr=1e-2)
for epoch in range(30):
    for i in range(0, len(Xtr), 64):
        loss = nn.functional.cross_entropy(net(Xtr[i:i + 64]), ytr[i:i + 64])
        opt.zero_grad(); loss.backward(); opt.step()
print((net(Xte).argmax(1) == yte).float().mean())       # about 0.97
filters = net[0].weight.detach()[:, 0]                   # (8, 3, 3) learned kernels
```

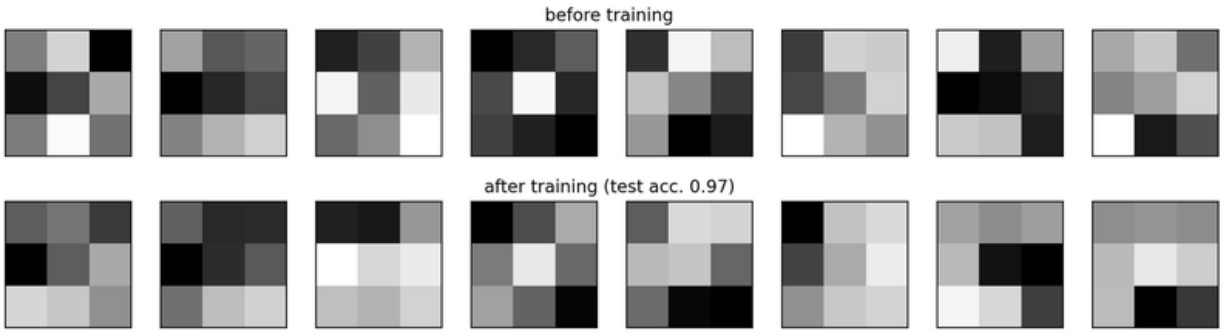



*Figure 4 — The eight first-layer 3×3 filters of the tiny digit CNN, before (top) and after (bottom) training. Several drift toward oriented light-to-dark transitions, the derivative-like patterns a hand designer would choose; nobody told the network to do so. (Tiny 8×8 inputs and 3×3 kernels keep the effect modest; large networks trained on natural images show it much more clearly.)*

### Stride, padding and output size

**Plain version.** Padding adds a border so the window can sit on edge pixels. Stride makes the window jump more than one pixel at a time, which shrinks the output.

**Precise version.** For an input of width $W$, kernel size $k$, padding $p$ (pixels added on each side) and stride $s$, the output width is

$$W_{\text{out}} = \left\lfloor \frac{W + 2p - k}{s} \right\rfloor + 1.$$

With $k = 3$, $p = 1$, $s = 1$ the size is preserved ("same" padding); with $s = 2$ it is halved. Zero padding is the default; reflect or replicate padding, the same boundary choices discussed in IP 03, are preferred in low-level networks because zero borders create dark artifacts at image edges.

### Pooling

**Pooling** summarizes each small window by one number, usually the maximum (max pooling) or the mean (average pooling), typically with a $2 \times 2$ window and stride 2. It halves resolution, makes the representation tolerant to small shifts, and cheaply enlarges the receptive field. Many modern networks replace it with strided convolutions, which learn how to downsample. At the very end of a classification network, **global average pooling** turns a $C \times H \times W$ map into a $C$-vector for the classifier head.

### Receptive field

**Plain version.** The receptive field of a neuron is the patch of the input image that can influence it. One $3 \times 3$ layer sees $3 \times 3$ pixels. Stack two, and each output sees $5 \times 5$. Downsampling makes it grow much faster.

**Precise version.** Number the layers $\ell = 1, \dots, L$ with kernel size $k_\ell$ and stride $s_\ell$. Let $j_\ell$ be the **jump**, the distance in input pixels between two neighboring units of layer $\ell$, and $r_\ell$ the receptive-field size, with $r_0 = 1$, $j_0 = 1$ for the input. Then

$$r_\ell = r_{\ell-1} + (k_\ell - 1)\, j_{\ell-1}, \qquad j_\ell = j_{\ell-1}\, s_\ell.$$

Each layer adds $k_\ell - 1$ units of its input, and each input unit is worth $j_{\ell-1}$ pixels. With stride 1 everywhere, $L$ layers of $3 \times 3$ give $r_L = 2L + 1$: linear growth. Every stride-2 step doubles the jump, so later layers grow the field twice as fast, giving roughly exponential growth in depth.

```
def receptive_field(layers):
    r, j = 1, 1                       # receptive field and jump, measured in input pixels
    for k, s in layers:               # (kernel size, stride) of each layer
        r, j = r + (k - 1) * j, j * s
    return r

print(receptive_field([(3, 1)] * 3))                                                  # 7
print(receptive_field([(3, 1), (3, 1), (2, 2), (3, 1), (3, 1), (2, 2), (3, 1)])) # 24
```

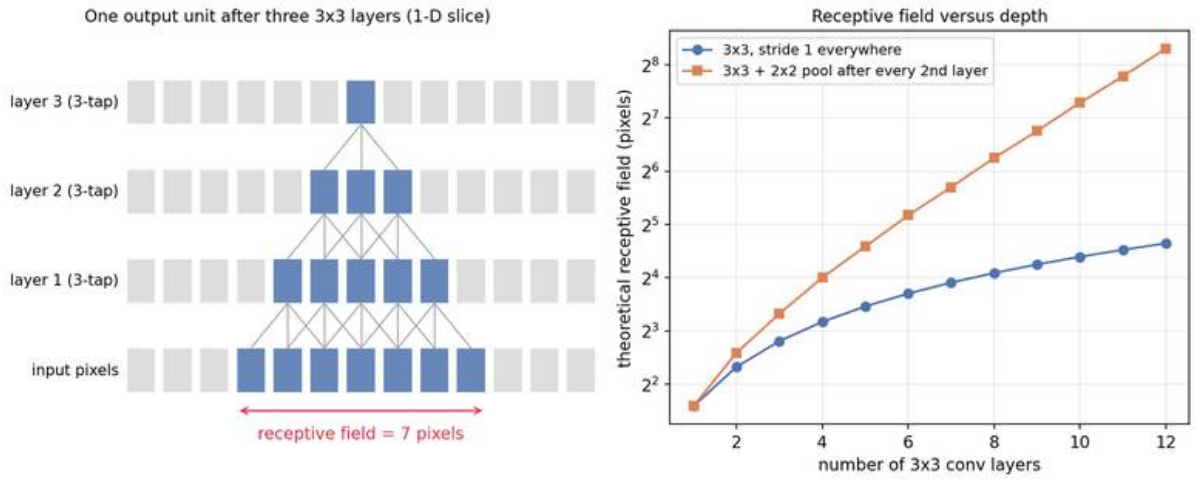



*Figure 5 — Left: three stacked 3-tap layers give one output a 7-pixel receptive field. Right: theoretical receptive field versus depth, for stride 1 everywhere and for 2× downsampling after every second layer.*

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

Two refinements matter in practice. First, the formula gives the **theoretical** receptive field. Luo et al. [12] showed that the **effective** receptive field, measured by how much each input pixel actually affects the output gradient, is roughly Gaussian and occupies only a fraction of the theoretical one. Pixels near the center dominate. Second, the receptive field decides what a network *can* use: a denoiser whose receptive field is smaller than the noise correlation length cannot remove that noise, and a classifier whose last layer sees only part of an object cannot recognize it from context.

### Feature hierarchies

Because the receptive field grows with depth while resolution shrinks, a CNN builds a **feature hierarchy**: early layers respond to edges and color blobs (Figure 4), middle layers to textures and parts, late layers to object-level patterns over large regions. A typical backbone is organized in **stages**, each ending with a 2× downsampling and doubling the channel count, so features go from "high-resolution, few channels, local" to "low-resolution, many channels, semantic". High-level tasks read the late stages. Dense prediction and low-level tasks need the early, high-resolution stages too, which is why encoder–decoders exist (below).

## Landmark CNN architectures

**Plain version.** Between 2012 and 2016 a series of networks, each deeper than the last, won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [6]. Each one introduced a design idea that is still in use.

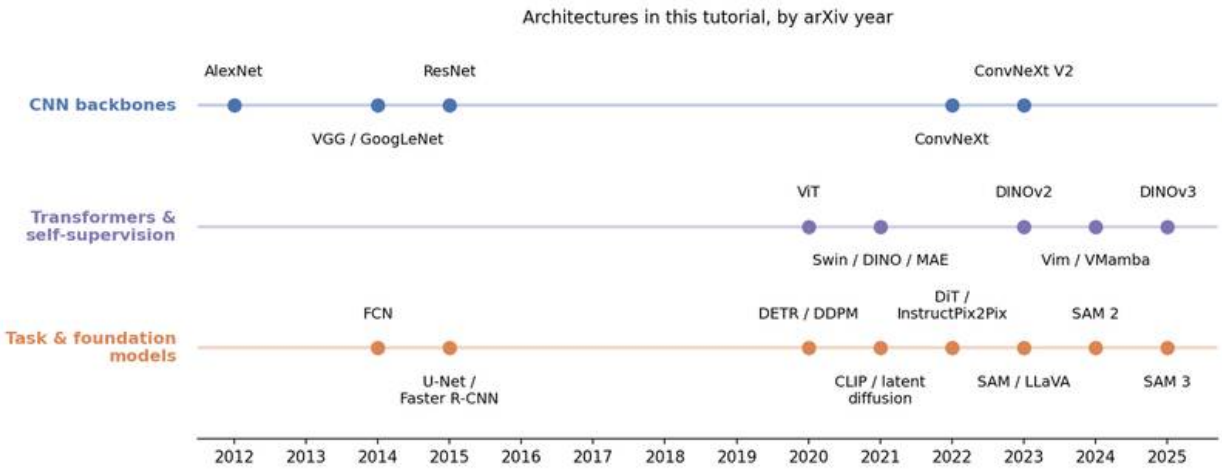



*Figure 6 — A timeline of the architectures discussed in this tutorial, by arXiv year. LeNet-5 (1998) and the earlier convolutional networks of the late 1980s sit to the left of this axis.*

### LeNet-5 and where CNNs really began

The post's original heading said "CNN (1998~)". That year is when LeCun, Bottou, Bengio and Haffner published LeNet-5 [5], not when CNNs started: convolutional networks trained by back-propagation were already reading handwritten ZIP-code digits in 1989 [4].

LeNet-5 is a textbook pattern-recognition system in network form [1]. Layers C1 to C5 are the **feature extractor**: convolutions (C1, C3, C5) alternating with subsampling layers (S2, S4) that turn a 32×32 digit image into a 120-dimensional feature vector. F6 and the output layer are the **classifier**. Everything, features included, is trained by gradient descent on the classification loss.

This was the end-to-end idea, already in place, but it needed 15 more years of data and compute to dominate.

### AlexNet (2012)

AlexNet [7] has five convolutional layers and three fully connected layers (eight learned layers, about 60 million parameters). Its ingredients were not individually new, but the combination was: ReLU activations, which train much faster than saturating sigmoids; an efficient GPU implementation; heavy data augmentation; and dropout in the fully connected layers. It won ILSVRC 2012 by a wide margin, which is why 2012 is usually given as the start of the deep-learning era in vision.

### VGG (2014)

Simonyan and Zisserman [8] asked a single question: what happens if you use only $3 \times 3$ convolutions and make the network deeper? Their best configurations had 16 and 19 weight layers. The reason $3 \times 3$ is enough is the receptive-field formula above: two stacked $3 \times 3$ layers see $5 \times 5$, three see $7 \times 7$. For $C$ channels in and out, three $3 \times 3$ layers cost $3 \cdot 9C^2 = 27C^2$ weights versus $49C^2$ for one $7 \times 7$ layer, and they insert two extra nonlinearities. VGG's uniform design made it the default pretrained feature extractor for years.

### GoogLeNet / Inception (2014)

GoogLeNet [9], 22 layers deep, won ILSVRC 2014. Its **Inception module** runs $1 \times 1$, $3 \times 3$ and $5 \times 5$ convolutions and a pooling branch in parallel and concatenates their outputs, so each block sees several scales at once. To keep this affordable, cheap $1 \times 1$ convolutions first reduce the number of channels. A $1 \times 1$ convolution mixes channels at each pixel without looking at neighbors; it is a per-pixel fully connected layer, and it remains a basic building block everywhere.

### ResNet (2015) and why residual connections work

Deeper should be better, but in practice it was not: He et al. [10] observed that a plain 56-layer network had *higher training error* than a 20-layer one. That is not overfitting. It is an optimization failure: the deeper net could in principle copy the shallow one and set the extra layers to the identity, but gradient descent does not find that solution.

The fix is the **residual block** (Figure 7). Instead of asking a stack of layers to learn a mapping $H(\mathbf{x})$ directly, let it learn the residual $F(\mathbf{x}) = H(\mathbf{x}) - \mathbf{x}$ and add the input back:

$$\mathbf{y} = \sigma(\mathbf{x} + F(\mathbf{x}; W)),$$

where $\mathbf{x}$ is the block input, $F$ is typically conv–BN–ReLU–conv–BN with weights $W$ (BN is batch normalization), and $\sigma$ is ReLU. When channel counts or resolution change, the shortcut uses a $1 \times 1$ convolution instead of the identity.

*Online only: Batch normalization: standardize inside the network — https://blog.shyandram.dev/en/learn/deep-learning/deep-learning-for-image-processing/notes/batch-normalization-standardize-inside-the-network/*

Why it works:

- **Identity is easy.** If extra depth is not useful, the block only has to drive $F$ toward zero, which is far easier than making a stack of nonlinear layers imitate the identity. Initializing the last BN scale of $F$ at zero makes each block start as the identity.
- **Gradients have a highway.** Ignoring the final nonlinearity, $\partial \mathbf{y}/\partial \mathbf{x} = I + \partial F/\partial \mathbf{x}$. The identity term carries the gradient from the loss to early layers without passing through every weight matrix, so it does not vanish with depth.
- **An ensemble view.** Unrolling $L$ blocks gives a sum over many paths of different lengths, so a deep ResNet behaves partly like a collection of shallower networks.

```
class ResidualBlock(nn.Module):
    """Basic block: y = ReLU(x + F(x)), with F = conv-BN-ReLU-conv-BN."""
    def __init__(self, c):
        super().__init__()
        self.f = nn.Sequential(
            nn.Conv2d(c, c, 3, padding=1, bias=False), nn.BatchNorm2d(c), nn.ReLU(),
            nn.Conv2d(c, c, 3, padding=1, bias=False), nn.BatchNorm2d(c),
        )
    def forward(self, x):
        return torch.relu(x + self.f(x))

blk = ResidualBlock(16)
nn.init.zeros_(blk.f[4].weight)                 # zero the last BN scale, so F(x) = 0
x = torch.randn(2, 16, 32, 32)
print(torch.allclose(blk(x), torch.relu(x)))    # True: the block starts as the identity
 ↪  (then ReLU)
```

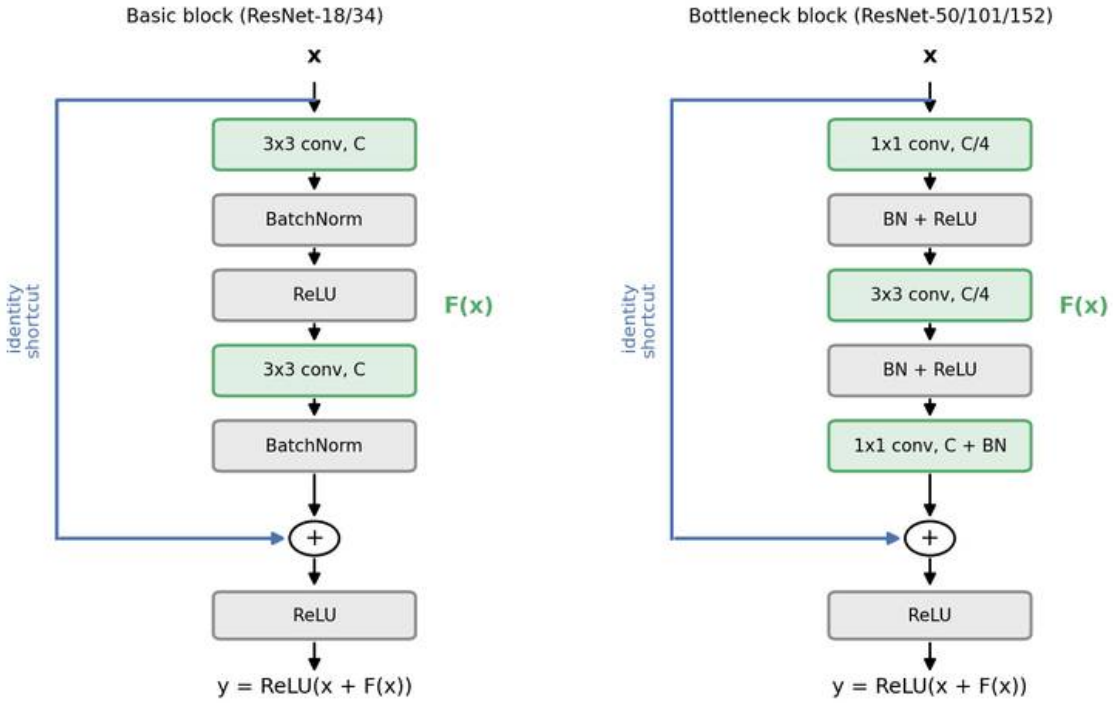



*Figure 7 — Left: the basic residual block. Right: the bottleneck block used in ResNet-50/101/152, where 1×1 convolutions shrink and restore the channel count around a cheap 3×3 convolution.*

On ImageNet the paper trained ResNets with **18, 34, 50, 101 and 152 layers**; the deeper three use the bottleneck block. The famous **1202-layer** network appears only in the paper's CIFAR-10 experiments [10], [11] (32×32 images), where it trained fine but overfit and tested worse than the 110-layer version. The ResNet family won ILSVRC 2015, and the residual connection is now in almost every deep network, Transformers included.

### Encoder–decoders for dense prediction

**Plain version.** A classification backbone throws away resolution to gain meaning. For segmentation or restoration we need an answer at every pixel, so we add a second half that brings the resolution back.

**Precise version.** Long et al.'s **fully convolutional network** (**FCN**) [13] replaced the fully connected layers of a classifier with convolutions, so the network outputs a coarse class-score map for an input of any size, then upsampled it with learned transposed convolutions and fused it with scores from earlier, finer layers. **U-Net** [14] made this symmetric: a contracting **encoder** (conv blocks and downsampling) and an expanding **decoder** (upsampling and conv blocks), with **skip connections** that concatenate each encoder stage's feature map to the decoder stage of the same resolution. The encoder supplies *what*; the skips supply *where*. U-Net was designed for biomedical images with few labels and strong augmentation, and the same shape now underlies medical segmentation, almost every restoration network covered in DLCV 09, and the denoiser inside

many diffusion models.

## The Vision Transformer

**Plain version.** The Transformer [15] was built for sentences: it treats the input as a sequence of tokens and lets every token look at every other token. The Vision Transformer (ViT) [16] cuts an image into small square patches, treats each patch as a "word", and feeds the sequence to an unmodified Transformer encoder.

### Patchify, embed, attend

**Precise version.** For an image $I \in \mathbb{R}^{C\times H\times W}$ and patch size $P$:

1. **Patchify.** Split $I$ into $N = HW/P^2$ non-overlapping patches and flatten each into a vector $\mathbf{p}_i \in \mathbb{R}^{CP^2}$. For a $224 \times 224$ RGB image and $P = 16$, $N = 196$ and each vector has 768 entries.
2. **Embed.** Map each patch linearly to a $D$-dimensional token, $\mathbf{z}_i = E\,\mathbf{p}_i$ with $E \in \mathbb{R}^{D\times CP^2}$. Prepend a learnable class token $\mathbf{z}_{\text{cls}}$ and add learnable **position embeddings** $\mathbf{e}_i$, because attention by itself has no notion of where a token came from: $Z^{(0)} = [\mathbf{z}_{\text{cls}}; \mathbf{z}_1; \dots; \mathbf{z}_N] + [\mathbf{e}_0; \dots; \mathbf{e}_N]$.
3. **Encode.** Apply $L$ Transformer encoder blocks, each a multi-head self-attention (MSA) sublayer and an MLP sublayer, both with layer normalization (LN) and residual connections:

$$Z' = Z^{(\ell-1)} + \mathrm{MSA}(\mathrm{LN}(Z^{(\ell-1)})),$$
$$Z^{(\ell)} = Z' + \mathrm{MLP}(\mathrm{LN}(Z')).$$

4. **Head.** Feed the final class token (or the average of all tokens) to an MLP head, which plays the role of the pattern-recognition classifier [1].

A single attention head computes, for token matrix $Z \in \mathbb{R}^{(N+1)\times D}$,

$$\mathrm{Attn}(Z) = \mathrm{softmax}\!\left(\frac{QK^\top}{\sqrt{d}}\right)V, \qquad Q = ZW_Q,\; K = ZW_K,\; V = ZW_V,$$

where $W_Q, W_K, W_V \in \mathbb{R}^{D\times d}$ are learned projections and $d$ is the head dimension. Row $i$ of the softmax matrix says how much token $i$ draws from every other token. Compared with a convolution, the "filter" is computed from the content and covers the whole image from the first layer on. The cost is quadratic in the number of tokens, $O(N^2 d)$, which is why patches rather than pixels are used.

```
img = torch.randn(1, 3, 224, 224)
P, D = 16, 768
patches = img.unfold(2, P, P).unfold(3, P, P)                    # (1, 3, 14, 14, 16, 16)
patches = patches.permute(0, 2, 3, 1, 4, 5).reshape(1, -1, 3 * P * P)
print(patches.shape)                                              # (1, 196, 768)
```

```
embed = nn.Conv2d(3, D, kernel_size=P, stride=P)                # patchify + linear embed
 ↪  in one op
tokens = embed(img).flatten(2).transpose(1, 2)                  # (1, 196, 768)
```

The last two lines show a useful identity: patch embedding *is* a convolution with kernel size and stride both equal to $P$.

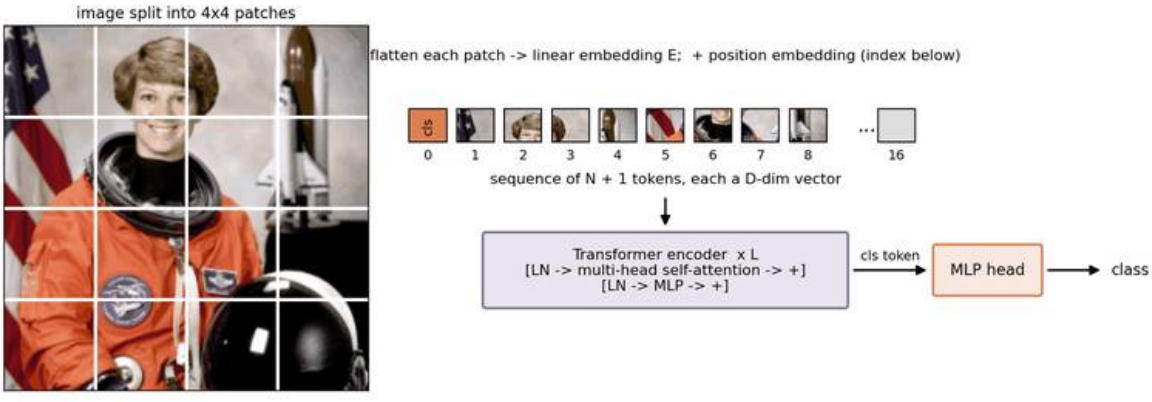



*Figure 8 — ViT in one picture: patchify, linearly embed, add position embeddings and a class token, run a Transformer encoder, classify from the class token. (A 4×4 grid is drawn for legibility; ViT-B/16 on 224×224 inputs uses 14×14 patches.)*

### Inductive bias and the need for scale

The original post said that ViT is "harder to converge". The more precise statement [1], [16] is about **inductive bias**, the assumptions built into an architecture before it sees data. A CNN assumes locality and weight sharing, so it already "knows" that nearby pixels matter together and that a pattern means the same thing anywhere in the image. ViT assumes almost nothing: apart from the patch split, all spatial structure, even the 2-D layout of the position embeddings, must be learned.

That is a trade-off, not a defect. With little data, the CNN's assumptions are a valuable head start, and ViT trained from scratch on ImageNet-1k alone underperforms comparable ResNets. With enough data, the assumptions become a limitation, and ViT pretrained on very large datasets matched or beat the best CNNs when fine-tuned [16]. The practical lesson is that ViTs are used **pretrained**, either on huge labeled sets or, increasingly, with self-supervision.

## Backbones after ViT

**Plain version.** After ViT, Swin added image-specific structure, ConvNeXt revisited CNN design, and DINO/MAE families developed reusable self-supervised features.

### Swin Transformer: locality returns

Swin [17] computes attention only inside non-overlapping local windows (for example $7 \times 7$ tokens), which makes the cost linear in image size. In alternate blocks the window grid is **shifted** by half a window, so information crosses window borders. Between stages, neighboring tokens are merged to halve resolution. The result is a hierarchical, multi-scale feature pyramid shaped like a CNN backbone, which is exactly what detection and segmentation heads expect. Swin put locality and hierarchy, the CNN's inductive biases, back into a Transformer.

### ConvNeXt: the CNN counterpoint

Liu et al. [18] started from a ResNet-50 and modernized it one step at a time, using the training recipe and design choices of Transformers: a "patchify" stem (a $4 \times 4$ convolution with stride 4), large $7 \times 7$ depthwise convolutions, an inverted-bottleneck MLP, fewer activation and normalization layers, and layer normalization. The resulting pure CNN, **ConvNeXt**, matched Swin in accuracy and scaling on classification, detection and segmentation. The lesson: much of ViT's advantage came from training recipes and macro design, not from attention itself. **ConvNeXt V2** [18] then added masked-autoencoder pretraining adapted to convolutions (the fully convolutional masked autoencoder, FCMAE) and a global response normalization layer. Today large foundation models mostly use ViT backbones, but convolutions remain common in low-level vision and on edge devices [1].

### Self-supervised pretraining: DINO versus MAE

Self-supervised learning creates a training signal from the images themselves, so no labels are needed. Two recipes dominate, and they are often confused.

**DINO: self-distillation** [19]. A student network $g_s$ and a teacher $g_t$ with the same architecture each output a probability vector over $K$ prototype dimensions. Several random crops of an image are made: two large "global" views and several small "local" views. The teacher sees only global views, the student sees all of them, and the student is trained to match the teacher's output:

$$\min_{\theta_s} \sum_{x \in \{x_1^g, x_2^g\}} \sum_{\substack{x' \in V \\ x' \neq x}} H(P_t(x),\, P_s(x')), \qquad H(a, b) = -a^\top \log b,$$

where $V$ is the set of all views, $P_t$ and $P_s$ are the softmax outputs of teacher and student, and $\theta_s$ are the student's weights. The teacher is not trained by gradients. Its weights are an exponential moving average of the student's, $\theta_t \leftarrow \lambda\theta_t + (1-\lambda)\theta_s$ with $\lambda$ close to 1, and its outputs are centered and sharpened so that training does not collapse to a constant output. Matching "small crop" to "whole image" forces the network to recognize objects from parts. A remarkable emergent property is that the attention maps of a DINO ViT segment the main objects without ever seeing a mask.

*Online only: Knowledge distillation: learning from a teacher's soft answers — https://blog.shyandram.*

*dev/en/learn/deep-learning/deep-learning-for-image-processing/notes/knowledge-distillation-learning-from-a-teachers-soft-answers/*

**MAE: masked prediction** [20]. The masked autoencoder hides a large random subset of patches (75 % works best), encodes only the visible ones with a ViT, and asks a light decoder to reconstruct the missing pixels:

$$\mathcal{L}_{\text{MAE}} = \frac{1}{|\mathcal{M}|} \sum_{i \in \mathcal{M}} \|\hat{\mathbf{p}}_i - \mathbf{p}_i\|_2^2,$$

where $\mathcal{M}$ is the set of masked patch indices, $\mathbf{p}_i$ the (normalized) pixels of patch $i$ and $\hat{\mathbf{p}}_i$ the reconstruction. This "hide and predict" game is the image analogue of masked-word prediction in language models; DINO, by contrast, is a teacher–student distillation method [1]. MAE is cheap because the encoder processes only a quarter of the tokens, and it fine-tunes very well. DINO-style features are better **frozen**, used as-is with a linear head.

### DINOv2 and DINOv3: frozen general-purpose features

**DINOv2** [21] scaled DINO-style self-distillation to a 1-billion-parameter ViT trained on a large, automatically **curated** image collection, then distilled it into smaller models. Its frozen features work well for both image-level tasks (classification, retrieval) and pixel-level tasks (segmentation, depth estimation) with only a light head. **DINOv3** [21] scaled data and model size further and introduced **Gram anchoring**, a regularizer that keeps dense patch features from degrading during very long training, so the same frozen backbone gives high-quality dense features.

This closes the loop with the pattern-recognition pipeline. "Feature extractor + classifier" has returned as **pretrained frozen backbone + light head**. The feature extractor is now learned once, at very large scale, without labels.

### State-space backbones

**Vision Mamba** (**Vim**) and **VMamba** [22] replace attention with selective state-space models, which scan the token sequence with cost linear in its length. Images are not sequences, so these models scan the patch grid along more than one path (Vim is bidirectional; VMamba's 2-D selective scan traverses several routes). They are attractive for high-resolution inputs but are still mostly a research direction rather than a default backbone.

## High-level vision tasks

**Plain version.** High-level tasks differ in how *where* the answer must be: one label for the whole image, a box per object, or a label for every pixel.

| Task | Output | Typical loss | Typical metric | Representative models |
|---|---|---|---|---|
| Classification | one label (or a probability vector) | cross-entropy | top-1 / top-5 accuracy | ResNet, ViT, ConvNeXt [10], [16], [18] |
| Object detection | a set of (box, class, score) | classification + box regression, or set matching | average precision (AP) over IoU thresholds | Faster R-CNN, DETR [23] |
| Semantic segmentation | a class per pixel | per-pixel cross-entropy, Dice (an overlap loss, below) | mean IoU | FCN, U-Net [13], [14] |
| Instance segmentation | a mask and class per object | detection + mask losses | mask AP | promptable: SAM family [24] |

### Classification

Assign one label to the whole image. The network is a backbone plus global pooling plus a linear layer, trained with cross-entropy $\mathcal{L} = -\log \hat{p}_y$, where $\hat{p}_y$ is the predicted probability of the true class $y$. Classification is also the **pretraining task** that produced most backbones before self-supervision. DLCV 02 covers it in depth.

### Object detection

Find every object, its class and its location as an axis-aligned box (a region of interest, ROI, given by its corner coordinates or center, width and height) [1]. Two design lineages illustrate the field [23]. **Faster R-CNN** is two-stage: a region proposal network suggests candidate boxes from shared CNN features, and a second head classifies and refines each one; duplicates are removed by non-maximum suppression. **DETR** treats detection as **set prediction**: a Transformer decoder emits a fixed set of predictions, and a bipartite matching between predictions and ground-truth objects defines the loss, which removes hand-designed anchors and non-maximum suppression. Box quality is measured by intersection over union,

$$\mathrm{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|},$$

where $A$ and $B$ are the predicted and true regions. The closely related Dice coefficient $2|A \cap B|/(|A| + |B|)$ is used, in a differentiable soft form, as a segmentation loss. Average precision summarizes the precision–recall curve at one or more IoU thresholds. Details are in DLCV 03.

### Semantic and instance segmentation

Here the region of interest is the object's exact set of pixels, not a box [1].

- **Semantic segmentation** labels each pixel with a class. Two overlapping people become one "person" region.

- **Instance segmentation** also separates objects of the same class. The two people become "person 1" and "person 2".

Semantic segmentation is the natural job of the encoder–decoders above [13], [14]. Instance segmentation adds a detection-like component that proposes one mask per object.

### Promptable segmentation: SAM, SAM 2, SAM 3

Segmentation now has **promptable foundation models** [24]. **SAM** (Segment Anything) takes an image and a prompt (points, a box, or a rough mask) and returns a mask for the indicated object. A heavy ViT image encoder runs once per image, and a light prompt encoder and mask decoder run per prompt, so interaction is fast. It was trained on SA-1B, a dataset of more than 1 billion masks on 11 million images, built with a model-in-the-loop "data engine". **SAM 2** extends this to video with a streaming memory that carries the object from frame to frame, trained on a new video dataset (SA-V). **SAM 3** adds **promptable concept segmentation**: given a short noun phrase such as "yellow school bus", image exemplars, or both, it finds, segments and (in video) tracks *all* matching instances, rather than the one object a click points at. With SAM 3 the line between detection, instance segmentation and tracking becomes thin; tracking is covered in DLCV 07.

## Low-level vision tasks

**Plain version.** Low-level tasks take an image and return an improved or transformed image of the same scene. The output is judged pixel by pixel, or by how natural it looks.

The general formulation is an inverse problem. A degraded observation $\mathbf{y}$ is modeled as

$$\mathbf{y} = \mathcal{D}(\mathbf{x}) + \mathbf{n},$$

where $\mathbf{x}$ is the clean image, $\mathcal{D}$ a degradation operator (blur, downsampling, haze, low exposure) and $\mathbf{n}$ noise, the same model as in IP 05. A network $f_\theta$ is trained on pairs to minimize, for example, $\|f_\theta(\mathbf{y}) - \mathbf{x}\|_1$. Because real paired data is scarce, pairs are usually **synthesized** by applying a modeled degradation to clean images, and the gap between synthetic and real degradations is the field's central difficulty.

- **Image restoration** recovers the clean image: denoising, deblurring, super-resolution (small to large), dehazing (hazy to clear), deraining. Architectures are full-resolution encoder–decoders (U-Net-like) or residual networks without downsampling, now often with window attention. See DLCV 09.
- **Image enhancement** makes content easier to see without a single "true" target: low-light enhancement (bringing out detail in dark images, roughly a smart brightening; turning night into day would be translation instead) and underwater enhancement. See DLCV 09C.
- **Image alignment** (registration) estimates the geometric transform or dense flow that maps one image onto another, a prerequisite for burst fusion, HDR and medical image comparison.
- **Image-to-image translation** maps an image from one domain or style to another: day to night, summer to winter, sketch to photo. Face-swapping "deepfakes" are a (misused) application of

the same idea. Since diffusion models, translation and editing are often done by a generative model conditioned on the input image, as discussed next.

## Generation and vision–language models

**Plain version.** Two newer families do not fit the old map well: models that **create** images from text, and models that **talk** about images. Both rely on large-scale pretraining and on connecting images with language.

### Diffusion models: from noise to image

A **denoising diffusion probabilistic model** (**DDPM**) [27] defines a forward process that gradually adds Gaussian noise to an image $\mathbf{x}_0$ over $T$ steps. In closed form,

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\epsilon}, \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, I),$$

where $t \in \{1, \dots, T\}$ is the step and $\bar{\alpha}_t \in (0, 1)$ is a fixed schedule that decreases from nearly 1 to nearly 0. A network $\boldsymbol{\epsilon}_\theta$ is trained to predict the noise that was added,

$$\mathcal{L} = \mathbb{E}_{\mathbf{x}_0, t, \boldsymbol{\epsilon}} \|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|_2^2,$$

and generation runs the process backwards, starting from pure noise and denoising step by step. In other words, a generator is a **learned denoiser applied many times**, which ties generation directly to the restoration problems above.

- **Latent diffusion** [28] runs the diffusion in the compressed latent space of a pretrained autoencoder rather than on pixels, which cuts the cost enough for high-resolution text-to-image synthesis; text conditioning enters through cross-attention. It is the basis of Stable Diffusion.
- **DiT** (Diffusion Transformer) [28] replaces the U-Net denoiser with a Transformer on latent patches and shows that quality improves steadily as the model's compute grows.
- **InstructPix2Pix** [29] edits a given image according to a written instruction ("make it winter"), so classic image-to-image translation becomes a special case of conditional generation.

### Vision–language models

Visual question answering (VQA), answering a free-form question about an image, once required a pipeline of image features, captioning and text question answering [1]. Two steps made it a single model.

**CLIP** [25] trains an image encoder $f$ and a text encoder $g$ on 400 million image–text pairs with a **contrastive** objective. In a batch of $B$ pairs, with normalized embeddings $\mathbf{u}_i = f(I_i)$ and $\mathbf{v}_j = g(T_j)$ and temperature $\tau$, the image-to-text loss is

$$\mathcal{L}_{I \to T} = -\frac{1}{B} \sum_{i=1}^{B} \log \frac{\exp(\mathbf{u}_i^\top \mathbf{v}_i / \tau)}{\sum_{j=1}^{B} \exp(\mathbf{u}_i^\top \mathbf{v}_j / \tau)},$$

and the total loss averages it with the symmetric text-to-image term. Matching pairs are pulled together and all other pairs in the batch pushed apart. Because the label space is now "any sentence", CLIP can classify zero-shot: embed the prompts "a photo of a cat", "a photo of a dog", ... and pick the one closest to the image.

**LLaVA** [26] connects a CLIP vision encoder to a large language model (LLM) through a projection layer that maps visual tokens into the LLM's word-embedding space, then fine-tunes on instruction-following conversations about images. Most current multimodal LLMs keep this **vision encoder + projector + LLM** pattern, so VQA, captioning and many recognition tasks are now handled by prompting one model.

## Modern view

The trend across this tutorial is a move from "one task, one model" to "one foundation model + prompt or light head". The surveys below are a good map of how the field got there.

**Surveys.** Six reviews map the ground covered here. Li et al. [30] lay out the CNN design space component by component. Khan et al. [31] organize vision Transformer work by task and stress that Transformers need large-scale pretraining or architectural priors to be data-efficient. Jing and Tian [32] and Balestriero et al. [33] cover self-supervised learning, from early pretext tasks to today's contrastive, self-distillation and masked-modeling families. Awais et al. [34] and Zhang et al. [35] survey vision foundation models and vision–language models, including prompt types, transfer methods and open challenges such as evaluation, bias and robustness.

*Online only: Guide to the Modern view surveys — https://blog.shyandram.dev/en/learn/deep-learning/deep-learning-for-image-processing/papers/guide-to-the-modern-view-surveys/*

**State of the art, 2024–2026.** Without quoting benchmark numbers, the landscape looks like this:

- **Backbones.** Large ViTs pretrained with self-supervision or image–text data are the default for high-level tasks, with frozen DINOv2/DINOv3 features [21] a strong baseline for dense tasks. Modern CNNs (ConvNeXt V2 [18]) remain competitive and are preferred where latency, memory or high resolution dominate. State-space backbones [22] are an active alternative.
- **Segmentation and tracking** have converged on promptable models: SAM 2 for video objects and SAM 3 for open-vocabulary concepts [24].
- **Generation** has moved to latent diffusion with Transformer denoisers [28], and editing to instruction-conditioned diffusion [29].
- **Understanding** is increasingly done by multimodal LLMs in the LLaVA pattern [26].
- **Low-level vision** still relies heavily on task-specific encoder–decoders trained on synthetic degradations, now increasingly with generative (diffusion) priors; see DLCV 04 and DLCV 05.

**Open problems.** (1) *Data and evaluation*: benchmarks saturate and may overlap with web-scale pretraining data, so it is hard to tell generalization from memorization [34]. (2) *Real-world degradations*: models trained on synthetic noise or blur still fail on real camera pipelines. (3) *Fidelity versus hallucination*: generative priors produce sharp but sometimes invented detail, which is un-

acceptable in medical or forensic use. (4) *Efficiency*: attention is quadratic in tokens, and foundation models are hard to run on devices. (5) *Dense features at scale*: DINOv3's Gram anchoring [21] exists because dense feature quality can degrade as training scales. (6) *Bias, robustness and interpretability* of models trained on uncurated web data [34].

## Key takeaways

- Define the required output first: an image, a label, a box, a mask or generated content.
- End-to-end learning trains features with the task; pretrained backbones let new heads reuse those features.
- Convolution shares local filters. Stride and pooling change resolution; stacked layers expand the receptive field.
- Residual paths support optimization; encoder–decoder skips help retain spatial detail.
- ViT relies heavily on training scale; Swin adds locality and hierarchy, while ConvNeXt modernizes CNN design.
- DINO learns through self-distillation; MAE learns through masked prediction. Evaluate transferred features on the target task.
- Prompts and language interfaces broaden model reuse, but each task still needs its own evaluation.

## Exercises

1. **Receptive field by hand.** A network has, in order: conv $7 \times 7$ stride 2; max-pool $3 \times 3$ stride 2; then four conv $3 \times 3$ stride 1. Compute the theoretical receptive field and the jump after each layer, and check your answer with the `receptive_field` function.

   **Hint**

   Start with $r = 1, j = 1$. After the first conv: $r = 7, j = 2$. After pooling: $r = 7 + 2 \cdot 2 = 11, j = 4$. Each following $3 \times 3$ layer adds $2 \cdot 4 = 8$, so $r = 19, 27, 35, 43$ with $j = 4$.

2. **Parameter budget.** For $C = 256$ input and output channels, compare the weights (ignore biases) of one $5 \times 5$ convolution, two stacked $3 \times 3$ convolutions, and a bottleneck ($1 \times 1$ to 64 channels, $3 \times 3$ at 64, $1 \times 1$ back to 256). Which have the same receptive field?

   **Hint**

   $25C^2 \approx 1.64$M; $2 \cdot 9C^2 \approx 1.18$M; bottleneck $256 \cdot 64 + 9 \cdot 64^2 + 64 \cdot 256 \approx 0.07$M. The first two both see $5 \times 5$; the bottleneck sees $3 \times 3$ but is far cheaper, which is why ResNet-50 and deeper use it.

3. **Why the identity is hard without a shortcut.** Inside a network, a block's input usually comes from a previous ReLU, so it is non-negative. Show that for such inputs a residual block with $F \equiv 0$ outputs exactly $\mathbf{y} = \mathbf{x}$. Then explain what a plain block (conv–BN–ReLU–conv–BN–ReLU) would have to learn to output $\mathbf{y} = \mathbf{x}$. What does this imply for very deep plain networks?

**Hint**

For $\mathbf{x} \geq 0$, $\mathrm{ReLU}(\mathbf{x} + 0) = \mathbf{x}$. A plain block must instead learn weights that exactly undo two convolutions and two normalizations, a precise and fragile solution that gradient descent rarely finds. The residual block only needs $F = 0$, which weight decay and zero-initialization already push it toward. Deep plain networks therefore struggle to "do nothing" in extra layers, which matches the degradation the ResNet authors observed.

4. **ViT token count and cost.** For a $512 \times 512$ input, compute the number of tokens for patch sizes 32, 16 and 8, and the relative cost of one global attention layer ($\propto N^2$). How does Swin's windowed attention with $7 \times 7$ windows change the scaling?

**Hint**

$N = (512/P)^2$: 256, 1024, 4096 tokens; costs scale $1 : 16 : 256$. Windowed attention costs $N \cdot 49$ per layer (each token attends to 49 others), so it grows linearly with $N$.

5. **Learned versus hand-set filters.** Modify the digit CNN so that its first layer is frozen to four hand-set kernels (two Sobels, a Laplacian and a box blur) plus four random kernels. Train only the rest. Compare test accuracy with the fully learned network and inspect what happens to the random kernels if you unfreeze them.

**Hint**

Set `conv.weight.requires_grad_(False)` after assigning the weights, and pass only the remaining parameters to the optimizer. On this easy dataset both versions reach similar accuracy; the point is that hand-set derivative filters are a reasonable first layer, and that unfrozen random kernels tend to drift toward oriented, derivative-like patterns (compare Figure 4).

6. **Choose a recipe.** You have 2,000 labeled microscopy images and need (a) a per-image quality label, (b) per-pixel cell masks, (c) denoised images. For each, propose a backbone, a head and a training strategy using this tutorial's map, and say which would benefit most from a frozen self-supervised backbone.

**Hint**

(a) Frozen DINOv2/v3 or fine-tuned ConvNeXt + linear head. (b) U-Net, possibly with a pretrained encoder, or SAM prompted by points and then fine-tuned. (c) A full-resolution restoration network trained on synthetic noisy/clean pairs that match the microscope's noise. Frozen semantic features help (a) and (b) most; (c) depends on pixel-accurate statistics more than on semantics.

## References

1. Shyandram, "深度學習的數位影像處理介紹 (An introduction to digital image processing with deep learning)," Medium post, 2024; updated 2026. link

2. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018. publisher page
3. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press, 2008. publisher page
4. Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989. doi:10.1162/neco.1989.1.4.541
5. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. doi:10.1109/5.726791
6. O. Russakovsky, J. Deng, H. Su, et al., "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015. doi:10.1007/s11263-015-0816-y ·arXiv:1409.0575
7. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *NeurIPS*, 2012. proceedings
8. K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *ICLR*, 2015. arXiv:1409.1556
9. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CVPR*, 2015. doi:10.1109/CVPR.2015.7298594 ·arXiv:1409.4842
10. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CVPR*, 2016. doi:10.1109/CVPR.2016.90 ·arXiv:1512.03385
11. A. Krizhevsky, "Learning multiple layers of features from tiny images," technical report, University of Toronto, 2009 (the CIFAR-10/100 datasets). dataset page
12. W. Luo, Y. Li, R. Urtasun, and R. Zemel, "Understanding the effective receptive field in deep convolutional neural networks," *NeurIPS*, 2016. arXiv:1701.04128
13. J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," *CVPR*, 2015. arXiv:1411.4038
14. O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," *MICCAI*, 2015. arXiv:1505.04597
15. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *NeurIPS*, 2017. ML anthology
16. A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," *ICLR*, 2021. arXiv:2010.11929
17. Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin Transformer: Hierarchical vision Transformer using shifted windows," *ICCV*, 2021. doi:10.1109/ICCV48922.2021.00986 ·arXiv:2103.14030
18. Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A ConvNet for the 2020s," *CVPR*, 2022, arXiv:2201.03545; and S. Woo, S. Debnath, R. Hu, X. Chen, Z. Liu, I. S. Kweon, and S. Xie, "ConvNeXt V2: Co-designing and scaling ConvNets with masked autoencoders," *CVPR*, 2023, arXiv:2301.00808.
19. M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin, "Emerging properties in self-supervised vision Transformers," *ICCV*, 2021. doi:10.1109/ICCV48922.2021.00951 ·arXiv:2104.14294
20. K. He, X. Chen, S. Xie, Y. Li, P. Dollár, and R. Girshick, "Masked autoencoders are scalable

vision learners," *CVPR*, 2022. doi:10.1109/CVPR52688.2022.01553 ·arXiv:2111.06377
21. M. Oquab, T. Darcet, T. Moutakanni, et al., "DINOv2: Learning robust visual features without supervision," *Transactions on Machine Learning Research*, 2024, arXiv:2304.07193; and O. Siméoni, H. V. Vo, M. Seitzer, F. Baldassarre, M. Oquab, et al., "DINOv3," arXiv preprint, 2025, arXiv:2508.10104.
22. L. Zhu, B. Liao, Q. Zhang, X. Wang, W. Liu, and X. Wang, "Vision Mamba: Efficient visual representation learning with bidirectional state space model," *ICML*, 2024, arXiv:2401.09417; and Y. Liu, Y. Tian, Y. Zhao, H. Yu, L. Xie, Y. Wang, Q. Ye, J. Jiao, and Y. Liu, "VMamba: Visual state space model," *NeurIPS*, 2024, arXiv:2401.10166.
23. S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *NeurIPS*, 2015, arXiv:1506.01497; and N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with Transformers," *ECCV*, 2020, arXiv:2005.12872.
24. A. Kirillov, E. Mintun, N. Ravi, et al., "Segment anything," *ICCV*, 2023, arXiv:2304.02643; N. Ravi, V. Gabeur, Y.-T. Hu, R. Hu, et al., "SAM 2: Segment anything in images and videos," *ICLR*, 2025, arXiv:2408.00714; and N. Carion, L. Gustafson, Y.-T. Hu, S. Debnath, et al., "SAM 3: Segment anything with concepts," *ICLR*, 2026, arXiv:2511.16719.
25. A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, et al., "Learning transferable visual models from natural language supervision," *ICML*, 2021. arXiv:2103.00020
26. H. Liu, C. Li, Q. Wu, and Y. J. Lee, "Visual instruction tuning," *NeurIPS*, 2023. arXiv:2304.08485
27. J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *NeurIPS*, 2020. arXiv:2006.11239
28. R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," *CVPR*, 2022, arXiv:2112.10752; and W. Peebles and S. Xie, "Scalable diffusion models with Transformers," *ICCV*, 2023, arXiv:2212.09748.
29. T. Brooks, A. Holynski, and A. A. Efros, "InstructPix2Pix: Learning to follow image editing instructions," *CVPR*, 2023. arXiv:2211.09800
30. Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, 2022. doi:10.1109/TNNLS.2021.3084827 ·arXiv:2004.02806
31. S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah, "Transformers in vision: A survey," *ACM Computing Surveys*, vol. 54, no. 10s, 2022. doi:10.1145/3505244 ·arXiv:2101.01169
32. L. Jing and Y. Tian, "Self-supervised visual feature learning with deep neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, 2021. doi:10.1109/TPAMI.2020.2992393 ·arXiv:1902.06162
33. R. Balestriero, M. Ibrahim, V. Sobal, et al., "A cookbook of self-supervised learning," arXiv preprint, 2023. arXiv:2304.12210
34. M. Awais, M. Naseer, S. Khan, R. M. Anwer, H. Cholakkal, M. Shah, M.-H. Yang, and F. S. Khan, "Foundation models defining a new era in vision: A survey and outlook," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 47, 2025. doi:10.1109/TPAMI.2024.3506283 ·arXiv:2307.13721
35. J. Zhang, J. Huang, S. Jin, and S. Lu, "Vision-language models for vision tasks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, 2024. doi:10.1109/TPAMI.2024.3369699 ·arXiv:2304.00685

36. scikit-learn developers, “sklearn.datasets.load_digits” (a copy of the test set of the UCI optical recognition of handwritten digits dataset). docs
37. S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *ICML*, PMLR 37, pp. 448–456, 2015. PMLR ·arXiv:1502.03167
38. S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, “How does batch normalization help optimization?” *NeurIPS*, vol. 31, 2018. proceedings ·arXiv:1805.11604
39. G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” NIPS 2014 Deep Learning Workshop, arXiv preprint, 2015. arXiv:1503.02531

# DLCV 01A • Training Deep Vision Models in Practice

A practical workflow for training vision networks: data pipelines and augmentation, initialization, learning-rate finding and schedules, batch size, regularization, mixed precision, fine-tuning, debugging and fair experiments.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/training-practice/

## What you'll learn

- Build a training workflow that catches cheap mistakes before expensive runs.
- Prevent data leakage and keep image augmentations consistent with labels.
- Choose learning rates, schedules, batch sizes and gradient accumulation settings.
- Apply regularization, mixed precision and pretrained-model fine-tuning.
- Debug losses and gradients, then compare models with controlled budgets and multiple seeds.

## The big picture

You already know how convolutional and transformer networks for images are built (DLCV 01). This chapter is about the other half of the job: getting one to train well, quickly, and knowing whether a result is real.

> **Explain like I'm five**
>
> Training a network is like cooking a new dish in a big kitchen. First you taste one spoonful to check that the recipe works at all (overfit one batch). You check how hot the oven can go before things burn (the learning-rate range test). You heat up slowly instead of putting the pan straight on full flame (warmup), and you turn the heat down at the end so nothing burns (the decay). And before you tell anyone your new recipe is better, you cook it several times, because one good dinner can be luck (several seeds).

Why spend a chapter on this? Because the recipe often matters as much as the architecture. He et al. combined training refinements, such as learning-rate warmup and cosine decay, with small changes to the network and raised the ImageNet top-1 accuracy of ResNet-50 from 75.3% to 79.29% [1]; DLCV 02 describes a similar result. And training failures are often **silent**: a network with a bug in its data pipeline usually still trains, just worse, so nothing crashes to warn you [2]. The habits in this chapter are how practitioners catch those bugs early.

The theory behind most tools is already covered elsewhere on this site, and this chapter links to it instead of repeating it:

| Topic | Covered in depth in |
|---|---|
| Floating point, gradient checks, a debugging order for NaN values | ML 01C |

| Topic | Covered in depth in |
|---|---|
| SGD, momentum, Adam, AdamW, initialization, normalization layers, gradient clipping, basic schedules | ML 02 |
| Baselines, splits and leakage, significance tests, reproducibility, latency and deployment | ML 11 |
| How batch normalization works; knowledge distillation | DLCV 01 notes |
| Classification recipes: augmentation, loss and optimizer choices, transfer learning, calibration | DLCV 02 |

What this chapter adds is the **workflow**: the order in which to do things, the quick tests that catch mistakes, and the defaults that work for most vision models.

### A training checklist

**Plain version.** Before a large run, verify that a small model can fit one batch. Change one factor at a time.

The list below is the whole chapter in one place. Each item links to the section that explains it.

1. **Fix the evaluation first.** Define the split unit, the metric and a simple baseline before training anything (data pipeline; ML 11).
2. **Look at the data.** Plot a batch *after* the full input pipeline (decoding, resizing, augmentation, normalization), with its labels drawn on top. This catches most pipeline bugs in a minute.
3. **Start from known-good defaults.** A standard architecture, He initialization, a pretrained backbone if one exists, AdamW or SGD with momentum (initialization).
4. **Overfit one batch.** The training loss on a single fixed batch must go to almost zero (debugging).
5. **Find the learning rate** with a short range test, then pick a schedule with warmup and a decay (learning rate).
6. **Set the batch size** from memory, and scale the learning rate with it (batch size).
7. **Train the full model, log everything** (losses, metrics, learning rate, gradient norms), and keep the best checkpoint by validation score.
8. **Regularize only when you see overfitting**, one tool at a time (regularization).
9. **Speed up** with mixed precision once the run is correct (mixed precision).
10. **Compare fairly:** same budget, same data, several seeds, one change per ablation; open the test set once at the end (fair experiments).

### The running example

To show each step with real numbers, the snippets train a small multilayer perceptron (MLP) written in NumPy: 64 inputs, one hidden layer of 32 ReLU units and 10 outputs, trained with SGD and momentum on scikit-learn's $8 \times 8$ handwritten digits [3]. It is not a deep vision model, but every training decision in this chapter applies to it, and each run takes well under a second on a CPU. All numbers called "our toy runs" below come from this code; the figure script `scripts/figures/dl_01a_training_practice.py` reproduces them.

The code below defines it once: data loading with a stratified split, normalization with training-split statistics, a two-layer network with He initialization, backpropagation, and a training loop that accepts any learning-rate schedule. The loop also keeps an exponential moving average of the weights, which the EMA section uses. Later snippets reuse these functions, so run this block first.

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

X, y = load_digits(return_X_y=True)                    # 1,797 images of 8x8 pixels, 10
 ↪  classes
X_tmp, X_te, y_tmp, y_te = train_test_split(X, y, test_size=0.3, stratify=y, random_state=0)
X_tr, X_va, y_tr, y_va = train_test_split(X_tmp, y_tmp, test_size=0.25, stratify=y_tmp,
 ↪  random_state=0)
mu, sd = X_tr.mean(0), X_tr.std(0) + 1e-8              # statistics of the TRAINING split
 ↪  only
X_tr, X_va, X_te = [(a - mu) / sd for a in (X_tr, X_va, X_te)]

def init(seed, sizes=(64, 32, 10)):                    # He initialization for ReLU layers
    rng = np.random.default_rng(seed)
    return [[rng.normal(0, np.sqrt(2 / m), (m, n)), np.zeros(n)]
            for m, n in zip(sizes[:-1], sizes[1:])]

def forward(P, X):
    h = [X]
    for i, (W, b) in enumerate(P):
        z = h[-1] @ W + b
        h.append(np.maximum(z, 0) if i < len(P) - 1 else z)   # ReLU; the last layer gives
         ↪  logits
    return h

def loss_and_grads(P, X, y):
    h = forward(P, X)
    z = h[-1] - h[-1].max(1, keepdims=True)            # stable log-softmax
    logp = z - np.log(np.exp(z).sum(1, keepdims=True))
    loss = -logp[np.arange(len(y)), y].mean()          # cross-entropy
    d = np.exp(logp)                                   # softmax probabilities
    d[np.arange(len(y)), y] -= 1
    d /= len(y)                                        # dLoss/dlogits
    grads = []
    for i in reversed(range(len(P))):
        grads.insert(0, [h[i].T @ d, d.sum(0)])
        if i:
            d = (d @ P[i][0].T) * (h[i] > 0)           # back through the ReLU
    return loss, grads

def evaluate(P, X, y):
    z = forward(P, X)[-1]
    z = z - z.max(1, keepdims=True)
    logp = z - np.log(np.exp(z).sum(1, keepdims=True))
    return -logp[np.arange(len(y)), y].mean(), (z.argmax(1) == y).mean()

def train(schedule, peak, epochs=15, bs=16, momentum=0.9, seed=0, ema_decay=0.99):
```

```
    P = init(seed)
    V = [[np.zeros_like(W), np.zeros_like(b)] for W, b in P]   # momentum buffers
    E = [[W.copy(), b.copy()] for W, b in P]                    # EMA copy of the weights
    rng = np.random.default_rng(seed + 1000)            # data order depends on the seed too
    T, t, log = epochs * int(np.ceil(len(y_tr) / bs)), 0, []
    for _ in range(epochs):
        order = rng.permutation(len(y_tr))
        for i in range(0, len(y_tr), bs):
            idx = order[i:i + bs]
            _, G = loss_and_grads(P, X_tr[idx], y_tr[idx])
            lr = schedule(t, T, peak)
            t += 1
            for (W, b), (gW, gb), v, e in zip(P, G, V, E):   # SGD with momentum, in place
                v[0] = momentum * v[0] + gW
                v[1] = momentum * v[1] + gb
                W -= lr * v[0]
                b -= lr * v[1]
                e[0] += (1 - ema_decay) * (W - e[0])    # e <- decay * e + (1 - decay) * w
                e[1] += (1 - ema_decay) * (b - e[1])
        log.append(evaluate(P, X_tr, y_tr) + evaluate(P, X_va, y_va))
    return P, np.array(log), E   # log columns: train loss, train acc, val loss, val acc
```

The split is 942 training, 315 validation and 540 test images; the test images are not touched in this chapter.

### The data pipeline

**Plain version.** Inspect the tensors produced by the full data pipeline. Their values and labels determine both learning and evaluation validity.

### Splits, leakage and the test set

The test set must be new in the same way deployment data will be new. Split by the unit that will be new (patient, video, camera, site or time period), deduplicate before splitting, fit every preprocessing step on the training split only, and use the validation set for all decisions so that the test set is opened once. ML 11 covers the forms leakage takes and how to measure uncertainty; this chapter assumes those rules.

### Normalization statistics

Networks train best when each input channel has a mean near zero and a spread near one. The standard step is

$$x'_c = \frac{x_c - \mu_c}{\sigma_c},$$

where $x_c$ is a pixel value in channel $c$ (for example red, green or blue), $\mu_c$ and $\sigma_c$ are the mean and standard deviation of channel $c$, and $x'_c$ is the value the network sees. Two practical rules follow.

- **Training from scratch:** compute $\mu_c$ and $\sigma_c$ on the training split only, store them with the model, and apply the same stored values at validation, test and deployment time. The running example does exactly this, per pixel rather than per channel.
- **Fine-tuning a pretrained model:** use the statistics the model was *pretrained* with, not those of your new dataset. The first layer's filters were learned for inputs on that scale.

The most common bug here is a mismatch: images scaled to $[0, 1]$ in training but raw 0–255 values in deployment, or RGB in one place and BGR in another. Both still produce predictions, just bad ones.

### Class imbalance

**Plain version.** Class imbalance can hide rare-class failures behind high overall accuracy. Adjust sampling, loss weights or classifier initialization.

Buda et al. compared remedies for convolutional networks on benchmarks of increasing difficulty [4]. In their experiments oversampling the rare classes was the dominant method in almost all scenarios, it worked best when it removed the imbalance completely, and, unlike for some classical models, it did not cause the networks to overfit; they also recommend thresholding to compensate for the class priors when overall accuracy is what matters [4]. The three levers look like this:

```
import numpy as np

counts = np.array([900, 80, 20])                          # training images per class
labels = np.repeat([0, 1, 2], counts)

w = counts.sum() / (len(counts) * counts)                 # option 1: weight the loss
print("loss weight per class:", np.round(w, 2))

p = 1.0 / counts[labels]                                  # option 2: oversample rare classes
p /= p.sum()
rng = np.random.default_rng(0)
drawn = labels[rng.choice(len(labels), size=3000, p=p)]
print("class share in 3,000 sampled examples:", np.round(np.bincount(drawn) / 3000, 3))

prior = counts / counts.sum()                             # option 3: start the output bias at
 ↪  the prior
b0 = np.log(prior)
print("initial softmax output:", np.round(np.exp(b0) / np.exp(b0).sum(), 3))
```

```
loss weight per class: [ 0.37  4.17 16.67]
class share in 3,000 sampled examples: [0.33  0.338 0.332]
initial softmax output: [0.9  0.08 0.02]
```

Option 1 multiplies each example's loss by $w_k = N/(K\,n_k)$, where $N$ is the number of training examples, $K$ the number of classes and $n_k$ the count of class $k$. Option 2 draws each example with probability proportional to $1/n_k$, so every class appears equally often. Option 3 sets the output

biases to $b_k = \log(n_k/N)$, so the untrained network already predicts the class frequencies; Lin et al. use the same idea in dense detection, starting every location at a foreground probability of about 0.01 [5] (see the focal loss in DLCV 03). Whatever you choose, report per-class metrics (DLCV 02).

### Augmentation as a regularizer

**Plain version.** Augmentation shows the network a slightly different version of each image every time, so it cannot memorize pixels and must learn what stays the same. The golden rule: a transformation is allowed only if the correct label of the new image is known.

That rule decides the augmentation for each task:

| Task | Usually safe | Needs care or is wrong |
|---|---|---|
| Classification of photos | random crops and rescaling, horizontal flips, color jitter, mild blur | vertical flips for upright scenes; flips for text, digits and left/right-sensitive classes |
| Detection | flips, crops, scaling, color jitter, applied to the image **and** the boxes | crops that cut objects: clip the box and drop it if too little is left |
| Segmentation | the same geometry applied to the image **and** the mask | interpolating the mask: resize labels with nearest neighbor, never bilinear |
| Keypoints and pose | geometry applied to the image **and** the points | a horizontal flip must also **swap** left and right keypoint labels (DLCV 05) |
| Restoration and low-level vision | flips, 90° rotations, crops applied identically to input and target | color or blur augmentation that changes the target itself |
| Medical and remote sensing | rotations where orientation carries no meaning | intensity changes that alter diagnostic values |

For dense tasks the key word is **consistency**: every geometric change applied to the pixels must be applied to the labels in the same coordinates. The snippet flips and crops a photo of coins together with a segmentation mask and a bounding box for one coin, then checks that the transformed box equals the box of the transformed mask:

```
import numpy as np
from scipy import ndimage
from skimage import data, filters, measure, segmentation

img = data.coins()                                  # 303 x 384 grayscale photo
fg = segmentation.clear_border(img > filters.threshold_otsu(img))
lab = measure.label(ndimage.binary_fill_holes(fg))
coin = max(measure.regionprops(lab), key=lambda r: r.area)
mask = lab == coin.label                            # segmentation target for one coin
box = tuple(int(v) for v in coin.bbox)              # detection target (y0, x0, y1, x1)
H, W = img.shape

def hflip(img, mask, box):
    y0, x0, y1, x1 = box
    return img[:, ::-1], mask[:, ::-1], (y0, W - x1, y1, W - x0)
```

```
def crop(img, mask, box, top, left, h, w):
    y0, x0, y1, x1 = box                                  # shift, then clip to the crop
    new = (max(y0 - top, 0), max(x0 - left, 0), min(y1 - top, h), min(x1 - left, w))
    return img[top:top + h, left:left + w], mask[top:top + h, left:left + w], new

i2, m2, b2 = crop(*hflip(img, mask, box), top=90, left=0, h=180, w=240)
r, c = np.nonzero(m2)
print("original box:               ", box)
print("box after flip + crop:      ", b2)
print("box of the transformed mask:", (int(r.min()), int(c.min()), int(r.max()) + 1,
 ↪  int(c.max()) + 1))
```

```
original box:                (156, 315, 218, 380)
box after flip + crop:       (66, 4, 128, 69)
box of the transformed mask: (66, 4, 128, 69)
```

Under a horizontal flip of an image of width $W$, a box with columns $x_0$ to $x_1$ moves to columns $W - x_1$ to $W - x_0$: the edges swap roles. Flipping the image but not its labels gives the network contradictory targets (Figure 1, right). Karpathy lists exactly this kind of mistake, flipping images without flipping their labels, among the bugs that make a network train silently worse instead of failing [2]. In practice, use a library that transforms images together with their masks and boxes, such as Albumentations [6], and still verify it once with a check like the one above.

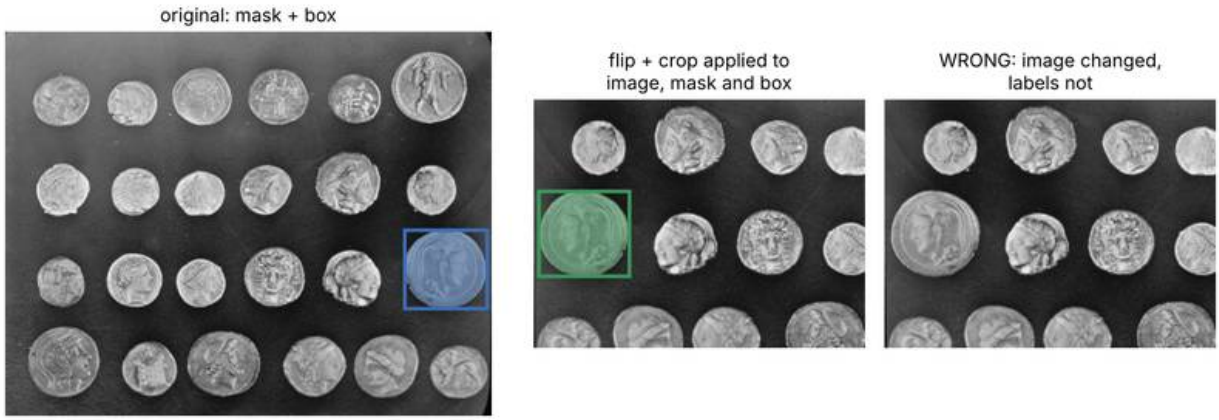



*Figure 1 — Geometric augmentation for dense tasks. A flip and crop must move the mask and the box with the pixels (middle). Transforming only the image (right) leaves annotations that no longer match any object; here the old box falls outside the crop entirely.*

**How strong?** Augmentation strength is a hyperparameter: tune it on the validation set, and expect a strongly augmented model to show a *higher* training loss than validation loss. Shorten and Khoshgoftaar survey the families of image augmentation [7]; DLCV 02 shows the mixing augmentations of modern classification recipes.

### Initialization and normalization in practice

**Plain version.** A good start keeps the signal at a sensible size through every layer, so the first steps of training are spent learning instead of recovering from a bad scale. For standard architectures, the defaults below are well tested; use them unless you have a reason not to.

- **Weights.** Use He initialization for layers followed by ReLU-like activations: draw weights with variance $2/n_{\text{in}}$, where $n_{\text{in}}$ is the number of inputs to a unit, which keeps the activation variance roughly constant across layers [8]. Biases start at zero. ML 02 derives this.
- **Residual blocks.** Initialize the scale $\gamma$ of the last batch-normalization layer in each residual branch to zero. Each block then starts as the identity, the signal travels through the shortcuts, and early training is easier; Goyal et al. found that this helped all their models and especially large-batch training [9], and He et al. include it in their bag of tricks [1].
- **Output layer.** Initialize the classifier bias at the log class prior when classes are imbalanced (option 3 above).
- **Pretrained weights.** When a pretrained backbone exists for your input type, start from it. A new head is the only part initialized from scratch.

**Normalization layers in practice.** Batch normalization (BN) [10] standardizes each channel with the mean and variance of the current mini-batch, then keeps running averages for inference (DLCV 01 note). Two habits avoid most BN problems:

1. **Switch modes.** Training mode uses batch statistics and updates the running averages; evaluation mode uses the stored averages. Forgetting to switch to evaluation mode at test time is a classic bug.
2. **Watch the per-device batch size.** BN statistics are computed over the examples on one device. With only one or two images per device, as is common in detection and segmentation, the statistics become noisy. Group Normalization, which normalizes groups of channels within each example, does not depend on the batch size and stays accurate where BN degrades; Wu and He also describe the common detection practice of *freezing* a pretrained backbone's BN, using its stored statistics as a fixed linear layer [11].

Transformers use a normalization computed within each example, which does not depend on the batch; ML 02 compares the normalization layers.

### Choosing the learning rate

**Plain version.** The learning rate is the size of each step downhill. Too small and training crawls; too large and the loss jumps around or explodes. The best rate also changes during training: larger steps help early, smaller steps help at the end.

Bengio wrote that the learning rate is often the single most important hyperparameter [12]. The procedure here: measure the usable range once, then let a schedule move through it.

### The learning-rate range test

**Plain version.** Run training once while raising the learning rate a little at every step, and watch the loss. Where the loss stops falling and starts to climb, the rate has become too large.

Smith introduced this **learning-rate range test** together with cyclical learning rates: train for a short time while increasing the rate between a low and a high value, and read off the range in which the network still learns [13]. In his later guidance, the rate where the loss is lowest (or the accuracy highest) is the largest usable value, a suitable maximum for a schedule that ramps up and down, while a constant learning rate must be smaller or the network will not begin to converge; for a single-cycle schedule he suggests a minimum 10 to 20 times below the maximum [14].

Smith reads the curve from the validation loss or accuracy; our version uses the training loss of each batch, which costs nothing extra. It increases the rate exponentially from $10^{-4}$ to 10 over at most 200 steps, so that every decade gets equal time, and smooths the noisy batch loss with an exponential moving average:

$$\bar{\ell}_t = \beta\,\bar{\ell}_{t-1} + (1-\beta)\,\ell_t, \qquad \hat{\ell}_t = \frac{\bar{\ell}_t}{1-\beta^{t+1}},$$

where $\ell_t$ is the loss of the batch at step $t$, $\bar{\ell}_t$ the running average (starting from $\bar{\ell}_{-1} = 0$), $\beta = 0.9$ the smoothing factor, and $\hat{\ell}_t$ the bias-corrected average that is plotted; the correction removes the pull toward the zero starting value. The test stops when the smoothed loss exceeds four times its minimum. The code below reuses `init` and `loss_and_grads` from the running example.

```
def lr_range_test(lo=1e-4, hi=10.0, steps=200, bs=16, momentum=0.9, seed=0):
    P = init(seed)
    V = [[np.zeros_like(W), np.zeros_like(b)] for W, b in P]
    rng = np.random.default_rng(seed)
    lrs, smooth, avg = [], [], 0.0
    for t in range(steps):
        lr = lo * (hi / lo) ** (t / (steps - 1))        # exponential sweep
        idx = rng.choice(len(y_tr), bs, replace=False)
        loss, G = loss_and_grads(P, X_tr[idx], y_tr[idx])
        avg = 0.9 * avg + 0.1 * loss                    # smooth the noisy batch loss
        lrs.append(lr)
        smooth.append(avg / (1 - 0.9 ** (t + 1)))       # bias-corrected average
        if not np.isfinite(loss) or smooth[-1] > 4 * min(smooth):
            break                                       # the loss has blown up: stop
        for (W, b), (gW, gb), v in zip(P, G, V):
            v[0] = momentum * v[0] + gW
            v[1] = momentum * v[1] + gb
            W -= lr * v[0]
            b -= lr * v[1]
    return np.array(lrs), np.array(smooth)

lrs, smooth = lr_range_test()
best = lrs[smooth.argmin()]
print(f"lowest smoothed loss {smooth.min():.3f} at lr {best:.3f}; stopped at lr
 ↪ {lrs[-1]:.3f}")
```

```
print(f"suggested peak for a one-cycle or warmup schedule: about {best:.2f}")
print(f"suggested constant learning rate: about {best / 10:.3f}")
```

```
lowest smoothed loss 0.270 at lr 0.138; stopped at lr 0.392
suggested peak for a one-cycle or warmup schedule: about 0.14
suggested constant learning rate: about 0.014
```

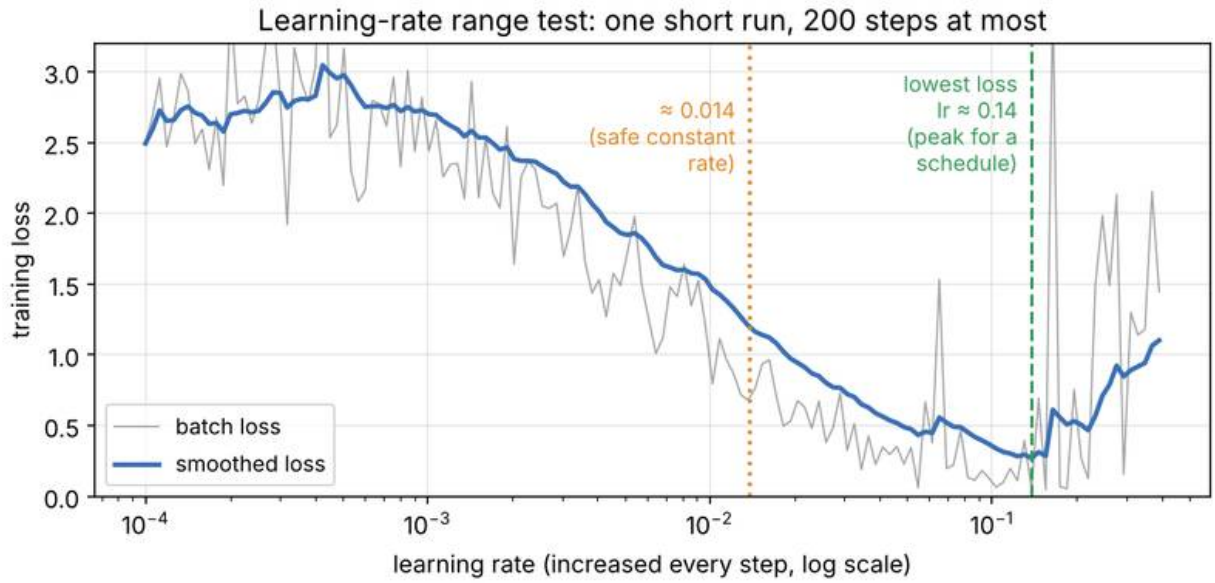



*Figure 2 — A learning-rate range test on the toy MLP (our run). The loss falls once the rate is large enough to make progress, bottoms out near 0.14 and then climbs; the test stopped at 0.39.*

Read Figure 2 from left to right: too small to change anything, then the useful range where the loss falls steeply, then steps that are too large. The code takes a tenth of the minimum as the constant rate; that factor is our rule of thumb, not a value from [14]. A range test costs a few hundred steps, so rerun it whenever you change the model, batch size, optimizer or weight decay. Treat the result as a range, not an exact value: the curve is noisy and depends on the seed.

### Warmup

**Plain version.** Start with a small learning rate and raise it to the target over the first few hundred or thousand steps. At the start, the weights are random and the optimizer's running statistics are empty, so a full-size step can throw the model into a bad region it never leaves.

With a linear warmup over $W$ steps, the rate at step $t < W$ is

$$\eta_t = \eta_{\text{peak}} \frac{t+1}{W},$$

where $\eta_{\text{peak}}$ is the target rate reached at the end of warmup. Goyal et al. showed why the *shape* matters: in large-batch ImageNet training, a constant low rate for the first epochs followed by a jump to the target made the training error spike, while a gradual ramp avoided that spike [9]. Warmup matters most for large batches, large learning rates, adaptive optimizers and Transformers, and it is cheap insurance in fine-tuning. Goyal et al. report that even a constant low-rate warmup was particularly helpful when prototyping detection and segmentation models that fine-tune pre-trained layers together with new ones [9]. Our toy warms up over the first 5% of the steps.

### Schedules: step, cosine and one-cycle

**Plain version.** After warmup, the rate should fall. How it falls matters less than *that* it falls: large steps explore, small steps at the end settle into a good region.

ML 02 introduces step decay, cosine annealing and warmup; this section adds one-cycle and compares all of them on one problem. Write $t$ for the current step, $T$ for the total number of steps and $\eta_{\text{peak}}$ for the largest rate. The four schedules in Figure 3 are:

- **Constant:** $\eta_t = \eta_{\text{peak}}$. Simple, but the final steps are as large as the first ones.
- **Step decay:** multiply the rate by a factor $\gamma$ (often $0.1$) at fixed milestones, for example at 50% and 75% of training. The classic choice for CNNs trained with SGD.
- **Warmup + cosine:** after $W$ warmup steps, $\eta_t = \frac{1}{2}\eta_{\text{peak}}(1 + \cos(\pi q))$ with progress $q = (t - W)/(T - W)$ running from 0 to 1, so the rate glides from $\eta_{\text{peak}}$ to zero. The cosine shape comes from SGDR, which also restarted it periodically [15]; He et al. adopted the plain cosine decay for ImageNet classification, where it improved accuracy over their step-decay baseline [1].
- **One-cycle:** a single long rise from a low rate to $\eta_{\text{peak}}$, then a fall to a value far below the starting rate. Smith and Topin used this "1cycle" policy with a large maximum rate to train some networks much faster than with standard schedules, which they called super-convergence [16]. Our version follows the defaults of PyTorch's `OneCycleLR`: a cosine-shaped rise over the first 30% of steps from $\eta_{\text{peak}}/25$, then a cosine fall to $10^4$ times below the start [17]. That implementation also lowers the momentum while the rate rises, as Smith recommends [14]; our toy keeps momentum fixed.

```
import numpy as np

def constant(t, T, peak):
    return peak

def step_decay(t, T, peak, milestones=(0.5, 0.75), factor=0.1):
    return peak * factor ** sum(t >= m * T for m in milestones)

def warmup_cosine(t, T, peak, warmup=0.05):
    W = max(1, int(warmup * T))
    if t < W:
        return peak * (t + 1) / W                       # linear warmup
    q = (t - W) / max(1, T - W)                         # progress after warmup, 0 -> 1
    return 0.5 * peak * (1 + np.cos(np.pi * q))         # cosine decay to 0
```

```
def one_cycle(t, T, peak, pct=0.3, div=25, final_div=1e4):
    up, lo = max(1, int(pct * T)), peak / div
    if t < up:                                          # rise from peak/div to peak
        return lo + (peak - lo) * 0.5 * (1 - np.cos(np.pi * t / up))
    q = (t - up) / max(1, T - up)                       # fall far below the start
    end = lo / final_div
    return end + (peak - end) * 0.5 * (1 + np.cos(np.pi * q))

T = 900
for f in (constant, step_decay, warmup_cosine, one_cycle):
    print(f"{f.__name__:13s}", [f"{f(t, T, 0.1):.2g}" for t in (0, 20, 270, 450, 700, 899)])
```

```
constant      ['0.1', '0.1', '0.1', '0.1', '0.1', '0.1']
step_decay    ['0.1', '0.1', '0.1', '0.01', '0.001', '0.001']
warmup_cosine ['0.0022', '0.047', '0.084', '0.054', '0.013', '3.4e-07']
one_cycle     ['0.004', '0.0053', '0.1', '0.081', '0.023', '1e-06']
```

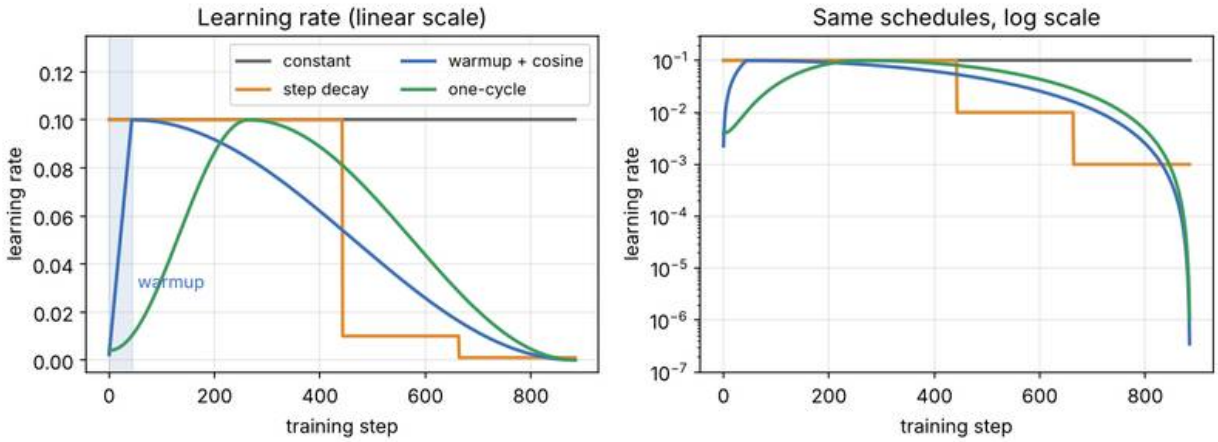



*Figure 3 — The four schedules used in this chapter with a peak of 0.1 over 885 steps (15 epochs of the toy). The log-scale view (right) shows how far cosine and one-cycle fall at the end.*

*Online only: Cosine annealing with warm restarts — https://blog.shyandram.dev/en/learn/deep-learning/training-practice/notes/cosine-annealing-with-warm-restarts/*

### What the schedules do on the toy

The loop below trains the running example once with each schedule at a peak of 0.1, and once more with a constant rate of 0.01. It uses `train` from the running example and the schedule functions just above.

```
for f in (constant, step_decay, warmup_cosine, one_cycle):
    _, log, _ = train(f, peak=0.1)
```

```
    print(f"{f.__name__:13s} peak 0.1   final train loss {log[-1, 0]:8.3f}   val acc {log[-1,
     ↪  3]:.3f}")
_, log, _ = train(constant, peak=0.01)
print(f"{'constant':13s} peak 0.01  final train loss {log[-1, 0]:8.3f}   val acc {log[-1,
 ↪  3]:.3f}")
```

```
constant      peak 0.1   final train loss  995.468   val acc 0.917
step_decay    peak 0.1   final train loss    0.000   val acc 0.940
warmup_cosine peak 0.1   final train loss    0.000   val acc 0.971
one_cycle     peak 0.1   final train loss    0.000   val acc 0.956
constant      peak 0.01  final train loss    0.021   val acc 0.959
```

Figure 4 shows the runs epoch by epoch. Three lessons come out of this toy, and they match the advice above:

1. **A peak near the edge needs a schedule.** At a constant 0.1, close to the range-test minimum, the loss explodes and validation accuracy swings between 81% and 92%. The same peak is fine inside a schedule that ramps up and decays (warmup + cosine reached 97.1%, one-cycle 95.6% in this run).
2. **Decay rescues, but late.** Step decay spends half of training at the unstable rate; once the rate drops, during epoch 8, the loss recovers and validation accuracy reaches 94.0%.
3. **A constant rate can work if it is small.** At 0.01, a tenth of the peak as the range test suggested, the constant schedule trains smoothly (95.9%), just less far in the same budget: its training loss is still 0.021.

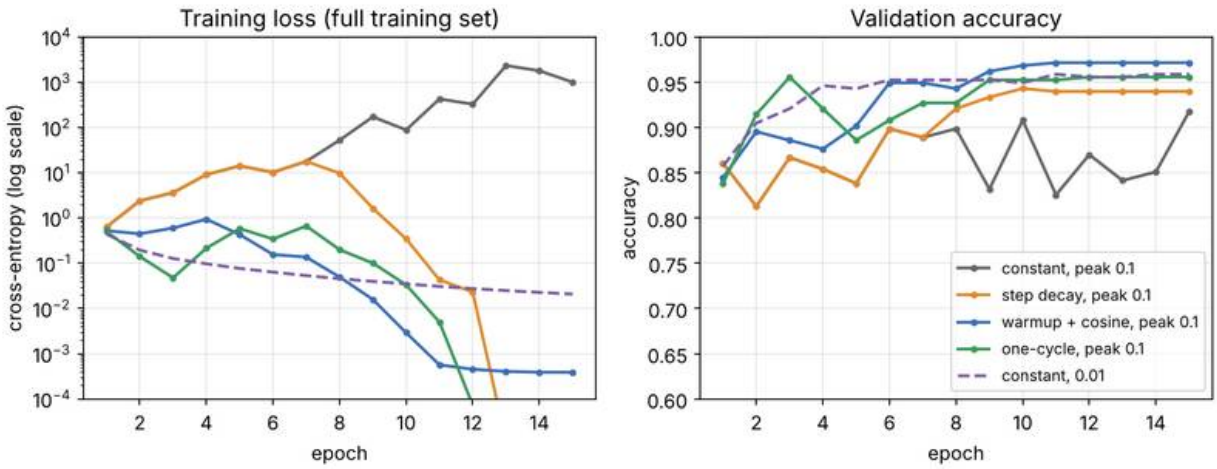



*Figure 4 — The schedules of Figure 3 on the toy MLP, seed 0 (our runs). With a peak of 0.1, the constant rate diverges, step decay recovers after its first drop, and warmup + cosine and one-cycle train stably. The dashed line is a constant rate of 0.01.*

Do not read the ranking between cosine and one-cycle into this figure: it is one seed, and the seed study below shows that their order flips from seed to seed. The robust lesson is the first one.

**Which schedule to pick.** Warmup followed by cosine decay is a safe default; step decay is fine when reproducing an older paper; one-cycle suits short budgets; a constant rate with a short final decay helps when the training length is not known in advance (Modern view). Schedules are defined in *steps*, so recompute $T$ when the epochs, batch size or dataset size change.

### Batch size and gradient accumulation

**Plain version.** A bigger batch gives a less noisy gradient, so each step can be bigger. If you double the batch, you can usually double the learning rate, up to a point. (ML 02 defines batches, iterations and epochs.)

Goyal et al. trained ResNet-50 [18] on ImageNet [19] with mini-batches of up to 8,192 images without losing accuracy, using two rules [9]:

- **Linear scaling rule:** when the mini-batch size is multiplied by $k$, multiply the learning rate by $k$ and keep the other hyperparameters, such as weight decay, unchanged. He et al. state the same rule in the form $\eta = 0.1 \times B/256$ for a ResNet-50 trained with SGD at batch size $B$ [1].
- **Gradual warmup:** ramp the rate from the small-batch value to the scaled value over the first 5 epochs.

The rule is an approximation. It assumes the gradient changes little over the $k$ small steps that one big step replaces, which fails early in training (hence warmup) and beyond some batch size; in their ImageNet experiments that limit was around 8,000 images [9]. For adaptive optimizers the relationship is less simple, so re-run the range test after a large change in batch size.

*Online only: Accurate, Large Minibatch SGD (Goyal et al.) — https://blog.shyandram.dev/en/learn/deep-learning/training-practice/papers/accurate-large-minibatch-sgd-goyal-et-al/*

**Gradient accumulation.** When the batch you want does not fit in memory, run several smaller *micro-batches*, add up their gradients, and take one optimizer step. With a loss that is a mean over examples and $k$ micro-batches of equal size,

$$\nabla_\theta L_{\text{batch}} = \frac{1}{k} \sum_{j=1}^{k} \nabla_\theta L_j,$$

where $L_{\text{batch}}$ is the mean loss over all $k \cdot m$ examples, $L_j$ the mean loss over micro-batch $j$ of $m$ examples, and $\theta$ the parameters. The snippet checks this on the toy (it reuses `init` and `loss_and_grads` from the running example):

```
P = init(0)
xb, yb = X_tr[:32], y_tr[:32]
_, full = loss_and_grads(P, xb, yb)                    # one batch of 32

acc = [[np.zeros_like(W), np.zeros_like(b)] for W, b in P]
k = 4                                                  # 4 micro-batches of 8
for j in range(k):
    _, G = loss_and_grads(P, xb[8 * j:8 * j + 8], yb[8 * j:8 * j + 8])
    for a, (gW, gb) in zip(acc, G):
```

```
        a[0] += gW / k                                  # divide by k: mean of means
        a[1] += gb / k
diff = max(np.abs(a[0] - g[0]).max() for a, g in zip(acc, full))
print(f"largest difference, accumulated vs full batch: {diff:.1e}")
```

```
largest difference, accumulated vs full batch: 1.1e-16
```

The difference is floating-point rounding: the update is the same. Two things break the equivalence in real models. Batch normalization computes its statistics per micro-batch, so accumulation does not give BN a larger batch. And a loss averaged over a variable number of items per image (valid pixels in segmentation, boxes in detection, tokens in a caption) needs the accumulated sum divided by the *total* count, not by $k$.

*Online only: Gradient accumulation when the loss is not a plain mean — https://blog.shyandram.dev/en/learn/deep-learning/training-practice/notes/gradient-accumulation-when-the-loss-is-not-a-plain-mean/*

### Regularization in practice

**Plain version.** Regularizers make memorizing harder so that the network learns patterns that generalize. They all pull in the same direction, so add them one at a time and only when the validation curve shows overfitting.

Smith and Topin stress a point that is easy to miss: large learning rates regularize too, so a high peak needs less of the other kinds, and all forms (large learning rates, small batch sizes, weight decay, dropout) must be balanced together [16]. When you change the schedule, re-tune weight decay and augmentation, and vice versa.

**Weight decay.** Weight decay shrinks every weight a little at each step. With plain SGD this equals adding an L2 penalty to the loss; with adaptive optimizers such as Adam [20] it does not, because the penalty's gradient gets rescaled per parameter. Loshchilov and Hutter's AdamW decouples the decay from the gradient step [21]; ML 02 derives the difference. Two practical rules:

- In PyTorch, `torch.optim.AdamW` applies decoupled decay, while the `weight_decay` argument of `torch.optim.Adam` adds an L2 term to the gradient unless `decoupled_weight_decay=True` is set [22]. Use decoupled decay with Transformers and modern CNN recipes.
- Do not decay biases and the scale and shift of normalization layers. He et al. list this "no bias decay" among their refinements [1]; DLCV 02 shows the two parameter groups in code.

**Dropout and stochastic depth.** Dropout zeroes random activations during training [23]; stochastic depth skips whole residual blocks at random, with a skip probability that grows with depth [24]. Dropout is common in classifier heads and Transformer blocks, stochastic depth in deep residual networks and vision Transformers. Both change behavior between training and evaluation mode.

**Early stopping and checkpoint selection.** Evaluate on the validation set at regular intervals, keep the best checkpoint, and stop when the score has not improved for a while (the "patience"). In

Prechelt's study on multilayer perceptrons, slower stopping criteria gave small improvements in generalization at a much higher training cost [25]. The Tuning Playbook recommends saving checkpoints regularly and selecting the best one retrospectively [26].

### EMA of the weights

**Plain version.** Keep a slowly moving average of the weights next to the weights themselves, and evaluate the average. The last few noisy steps cancel out, much as averaging several photos of the same scene removes noise.

DLCV 02 lists the EMA among the parts of a modern classification recipe; this section is about choosing its window. After every optimizer step, update

$$\bar{\theta} \leftarrow \mu\,\bar{\theta} + (1-\mu)\,\theta,$$

where $\theta$ are the current weights, $\bar{\theta}$ their exponential moving average (EMA), and $\mu \in [0, 1)$ the decay. The average effectively covers about the last $1/(1-\mu)$ steps: about 100 steps for $\mu = 0.99$, about 1,000 for $\mu = 0.999$. Averaging the iterates of stochastic gradient descent is an old idea in optimization [27], and stochastic weight averaging (SWA) brought an equal-weight version to deep learning [28]. Morales-Brotons et al. studied the EMA of weights systematically: EMA models generalized better than the last iterate and needed less learning-rate decay, and batch normalization limited how long the averaging window could be unless its statistics were recomputed [29].

The window must be short compared with the run. Our toy makes this concrete (it reuses `train`, `constant` and `evaluate` from above):

```
for decay in (0.99, 0.999):
    raw, avg = [], []
    for s in range(5):
        P, log, E = train(constant, peak=0.05, seed=s, ema_decay=decay)
        raw.append(evaluate(P, X_va, y_va)[1])
        avg.append(evaluate(E, X_va, y_va)[1])
    print(f"decay {decay}: window ~{1 / (1 - decay):.0f} steps   "
          f"val acc raw {np.mean(raw):.3f}   EMA {np.mean(avg):.3f}")
print("steps in the whole run:", 15 * int(np.ceil(len(y_tr) / 16)))
```

```
decay 0.99: window ~100 steps   val acc raw 0.963   EMA 0.963
decay 0.999: window ~1000 steps   val acc raw 0.963   EMA 0.945
steps in the whole run: 885
```

On this small, easy problem the EMA brings no gain at $\mu = 0.99$, and at $\mu = 0.999$ it is *worse*: its window of about 1,000 steps is longer than the whole 885-step run, so the average still carries weight from the poor early iterates. EMA pays off in long runs with noisy updates, which is where it is used in practice.

*Online only: EMA of weights: choosing the decay — https://blog.shyandram.dev/en/learn/deep-learning/training-practice/notes/ema-of-weights-choosing-the-decay/*

### Mixed precision and loss scaling

**Plain version.** Doing most arithmetic with 16-bit numbers halves memory traffic and uses fast hardware units, but 16 bits cannot hold everything. Keep a few sensitive quantities in 32 bits, and scale the loss so that tiny gradients do not round to zero.

Micikevicius et al. made float16 training work with two techniques: a float32 master copy of the weights that accumulates the updates, and loss scaling, which multiplies the loss by a factor $S$ before backpropagation and divides the gradients by $S$ before the update [30]. Both problems are easy to reproduce:

```
import numpy as np

g = np.float32(2e-8)                                  # a small, real gradient value
print("float16 gradient:          ", np.float16(g))   # underflows to zero
S = np.float32(1024)                                  # loss scale
g16 = np.float16(g * S)                               # scaled: representable in float16
print("scaled, then unscaled:     ", np.float32(g16) / S)
print("float16 weight 1 + 1e-4:   ", np.float16(1) + np.float16(1e-4))   # update lost
print("float32 weight 1 + 1e-4:   ", np.float32(1) + np.float32(1e-4))
```

```
float16 gradient:           0.0
scaled, then unscaled:      2.0023435e-08
float16 weight 1 + 1e-4:    1.0
float32 weight 1 + 1e-4:    1.0001
```

The first line is **underflow**: $2 \times 10^{-8}$ is below the smallest positive float16 value, so it becomes zero, and scaling by 1,024 moves it back into range. The third line is a **lost update**: near 1.0, float16 numbers are spaced too far apart to register a change of $10^{-4}$, which is why the weights themselves stay in float32.

In PyTorch, automatic mixed precision combines `torch.autocast`, which runs each operation in a suitable precision, with `torch.amp.GradScaler`, which performs dynamic loss scaling; its bfloat16 example on the CPU uses autocast alone [31]. bfloat16 keeps the 8 exponent bits of float32, and therefore its range, so it rarely needs loss scaling (ML 01C). Make the run correct in float32 first, then switch on mixed precision and check that the loss curve barely changes; if NaNs appear only in mixed precision, look for operations that overflow in 16 bits. ML 01C explains the formats in detail.

### Transfer learning and fine-tuning strategies

**Plain version.** A pretrained network already knows edges, textures and object parts. Fine-tuning teaches it your task without destroying that knowledge: change the new parts quickly and the old parts gently.

DLCV 02 defines linear probing and full fine-tuning; this section is about choosing between them and making fine-tuning gentle. Yosinski et al. measured how general the features of each layer

are: early layers learn features that transfer across datasets, later layers are more task-specific, and transferability falls as the source and target tasks drift apart, though transferred features beat random ones even for distant tasks [32]. That is the reason behind the strategies below, ordered from cheapest to most flexible:

| Strategy | What trains | When it fits |
|---|---|---|
| Linear probe | only a new linear head on frozen features | very little data; features already good for the task; also to *measure* feature quality |
| Partial fine-tuning | the head and the last stages | moderate data; target close to the pretraining domain |
| Full fine-tuning | everything, with a small learning rate for the backbone | enough data or a large domain gap |
| Linear probe, then fine-tune (LP-FT) | first the head alone, then everything | when robustness to distribution shift matters |
| Parameter-efficient fine-tuning | small added modules, such as low-rank updates (LoRA [33]) | very large models; many tasks sharing one backbone |

Three tools make full fine-tuning gentler:

- **Discriminative learning rates.** Give each layer group its own rate, smaller for earlier layers. Howard and Ruder introduced this as *discriminative fine-tuning*, together with *gradual unfreezing* (train the last layer first, then unfreeze one more layer group per epoch) [34]. In vision recipes the backbone often gets a rate 10 to 100 times smaller than the head (DLCV 02).
- **LP-FT.** Kumar et al. showed that full fine-tuning with a randomly initialized head can *distort* good pretrained features: while the random head is being fitted, the backbone changes for training-like inputs but much less for different-looking ones. Training the linear head first, then fine-tuning everything (LP-FT), kept the in-distribution gain of fine-tuning and the out-of-distribution robustness of linear probing in their experiments [35].
- **Warmup and BN.** A short warmup protects the pretrained weights from the first large gradients of the new head, and with small per-device batches the backbone's BN layers are often frozen (above).

*Online only: Discriminative learning rates and gradual unfreezing — https://blog.shyandram.dev/en/learn/deep-learning/training-practice/notes/discriminative-learning-rates-and-gradual-unfreezing/*

*Online only: Fine-Tuning can Distort Pretrained Features (Kumar et al.) — https://blog.shyandram.dev/en/learn/deep-learning/training-practice/papers/fine-tuning-can-distort-pretrained-features-kumar-et-al/*

### Debugging a training run

**Plain version.** Most training bugs do not crash; they make the model a little worse. Debugging means running cheap tests that a correct pipeline must pass, in an order that isolates the cause (Figure 5).

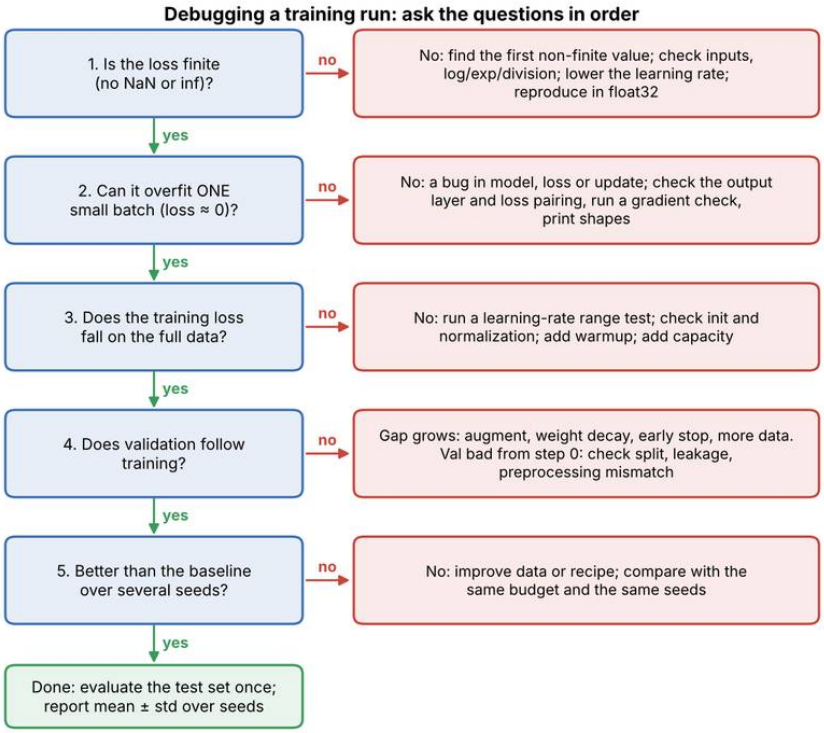



*Figure 5 — A debugging order. Each question assumes the ones above it are answered yes, so a failure points to a small set of causes.*

### Overfit one batch first

**Plain version.** Take one small batch and train on it over and over. A correct model and training loop can memorize a handful of images, so the loss must fall to nearly zero. If it cannot, something is broken, and you have found that out in seconds instead of days.

Karpathy's recipe for training networks puts this test early: overfit a single small batch, and before that, check that the loss at initialization is what it should be [2]. For $K$ balanced classes and a well-initialized final layer, the first predictions are close to uniform, so the cross-entropy starts near $\ln K$ ($\ln 10 \approx 2.30$ for our digits). The snippet checks that, then runs the one-batch test on 16 digits twice: once correctly, and once with a bug that is easy to make, applying softmax in the model *and* inside the loss. It reuses `init` and `forward` from the running example.

```
def overfit_one_batch(double_softmax=False, steps=300, lr=0.1, seed=0):
    P = init(seed)
    xb, yb = X_tr[:16], y_tr[:16]                        # ONE fixed batch of 16 images
    for step in range(steps):
        h = forward(P, xb)
        z = h[-1]
        if double_softmax:                               # BUG: the "logits" are already
         ↪  probabilities
```

```
            z = np.exp(z - z.max(1, keepdims=True))
            z = z / z.sum(1, keepdims=True)
        s = z - z.max(1, keepdims=True)
        p = np.exp(s) / np.exp(s).sum(1, keepdims=True)
        loss = -np.log(p[np.arange(16), yb]).mean()
        d = p.copy()
        d[np.arange(16), yb] -= 1
        d /= 16
        if double_softmax:                              # chain rule through the extra
         ↪ softmax
            d = z * (d - (d * z).sum(1, keepdims=True))
        for i in reversed(range(len(P))):
            gW, gb = h[i].T @ d, d.sum(0)
            if i:
                d = (d @ P[i][0].T) * (h[i] > 0)
            P[i][0] -= lr * gW
            P[i][1] -= lr * gb
    return loss

print(f"loss at init:          {evaluate(init(0), X_tr, y_tr)[0]:.4f}   (ln 10 =
 ↪ {np.log(10):.4f})")
print(f"correct loss:          {overfit_one_batch():.4f}")
print(f"double-softmax bug:    {overfit_one_batch(double_softmax=True):.4f}")
print(f"floor of the bug:      {-np.log(np.e / (np.e + 9)):.4f}")
```

```
loss at init:          2.8204   (ln 10 = 2.3026)
correct loss:          0.0036
double-softmax bug:    1.4661
floor of the bug:      1.4612
```

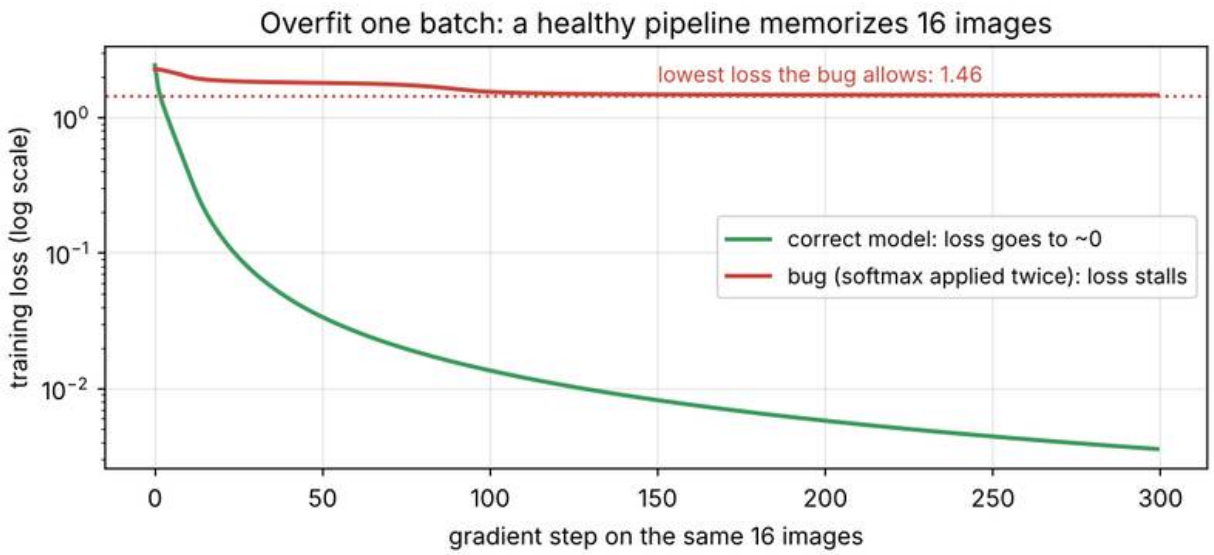



*Figure 6 — Overfitting one batch (our run). The correct pipeline memorizes 16 images; the double-softmax bug cannot go below 1.46, however long it trains.*

The initial loss is somewhat above $\ln 10$ because the He-initialized output layer produces logits of moderate size, so the first predictions are not exactly uniform; that is harmless here. A value far above $\ln K$ means the network starts out confidently wrong, and scaling the last layer's initial weights down fixes it. The bug has a hard floor. After the first softmax, the "logits" passed to the loss lie between 0 and 1, so the best the second softmax can do is give the true class $e^1$ against $e^0$ for each of the nine others: a probability of $e/(e+9) \approx 0.23$ and a loss of 1.46. On the full dataset this bug does not crash and still produces a usable classifier, so it can go unnoticed; the one-batch test exposes it in seconds.

A passing test proves less than it seems. Zhang et al. showed that standard image classifiers can fit random labels, and even random noise images, perfectly [36], so memorizing one batch says that the model, the loss and the update can learn, not that the labels are correct. Mislabeled or misaligned data must be caught by looking at it (checklist item 2).

### Reading loss curves

**Plain version.** A loss curve is the main record of a run: its shape tells you which part of the pipeline to suspect. Plot training and validation curves together, on a log scale for the loss.

ML 02 explains the four basic shapes (flat, overfitting, converged, healthy). A few more patterns are typical of vision training:

- **Stuck exactly at $\ln K$.** The network outputs uniform probabilities: the gradient is not reaching it (a detached tensor, frozen parameters, a learning rate of zero) or the labels are constant.
- **Loss spikes that recover.** The learning rate is near the edge of stability, as in Figure 4, or a few batches contain corrupted images or labels. Log the indices of the batches with the largest loss and look at them.
- **Validation better than training.** Normal with strong augmentation or dropout, because the training loss is measured on harder, augmented inputs.
- **Validation accuracy flat while validation loss rises.** The network becomes overconfident on the examples it gets wrong; the ranking is fine, but the probabilities are not (calibration in DLCV 02).
- **A drop at every milestone.** Expected with step decay. If the curve is still falling steeply at the end of a cosine schedule, the budget is too short.
- **Good validation, bad deployment.** Suspect a preprocessing mismatch, or *shortcut learning*: the network relies on a cue that the validation set shares with the training set but deployment data lacks [37] (DLCV 02).

### Gradient norms, dead units and NaNs

**Plain version.** Log a few health numbers every few hundred steps: the size of each layer's gradient, the fraction of units that never activate, and whether anything has become NaN. When training goes wrong, these numbers usually go wrong first.

The snippet computes two of them after three epochs at three constant learning rates. It reuses `train`, `forward`, `loss_and_grads` and `constant` from above.

```
def health(P, X, y):
    _, G = loss_and_grads(P, X, y)
    h = forward(P, X)[1]                                  # hidden ReLU activations
    dead = (h.max(0) == 0).mean()                         # units that are zero for every input
    norms = [np.linalg.norm(gW) for gW, _ in G]
    return norms, dead

for peak in (0.01, 0.1, 0.3):
    P, log, _ = train(constant, peak=peak, epochs=3)
    norms, dead = health(P, X_tr, y_tr)
    print(f"lr {peak:<5} train loss {log[-1, 0]:9.3g}  grad norms "
          f"{norms[0]:9.3g} {norms[1]:9.3g}  dead units {dead:.0%}")
```

```
lr 0.01  train loss     0.127  grad norms    0.0664    0.0496  dead units 0%
lr 0.1   train loss      3.61  grad norms      1.44      2.11  dead units 0%
lr 0.3   train loss   2.1e+08  grad norms  1.02e+04  1.31e+04  dead units 0%
```

The gradient norms grow by more than five orders of magnitude between the healthy and the diverging run, long before the loss becomes NaN, so logging them is cheap insurance. Gradient clipping (ML 02) is a sensible guard, but it hides a too-large learning rate rather than fixing it. No units die here, because this toy fails by explosion. A **dead ReLU** outputs zero for every input, so it gets no gradient and never recovers; many dead units, typically after a huge step or with strongly negative biases, mean lost capacity.

When the loss becomes NaN or infinite, find the *first* operation that produced a non-finite value rather than clipping everything: check inputs, then the forward pass, the loss, the gradients and the update size, and reproduce in float32 before blaming mixed precision. ML 01C gives this order in detail, and its section on checking derivatives shows how to test a hand-written backward pass.

### Reproducibility and fair experiments

**Plain version.** A result is only useful if it can be repeated and if the comparison behind it is fair. Fix the seeds, log what you did, and run each setting several times, because random chance alone can move a score by more than the improvement you hope to show.

### Seeds, determinism and logging

Seed every random number generator: Python's, NumPy's, the framework's and the data-loader workers'. Even then, runs can differ: the PyTorch documentation states that completely reproducible results are not guaranteed across releases, commits or platforms, or between CPU and GPU. Within one setup it recommends `torch.use_deterministic_algorithms(True)`, disabling cuDNN benchmarking, and seeding data-loader workers through `worker_init_fn`

and a `generator`, and it warns that deterministic operations are often slower [38]. Log enough to rerun the experiment: code version, configuration, data version and split identifiers, seeds, environment and hardware, and the curves of loss, metrics, learning rate and gradient norm (ML 11).

### How much do seeds matter?

The seed changes the initialization and the order of the data, and therefore the result. The snippet trains two configurations on the toy, A (warmup + cosine) and B (one-cycle), each with five seeds, and compares them pair by pair (it reuses `train` and the schedule functions):

```
seeds = range(5)
A = [train(warmup_cosine, peak=0.1, seed=s)[1][-1, 3] for s in seeds]
B = [train(one_cycle, peak=0.1, seed=s)[1][-1, 3] for s in seeds]
print("A (warmup + cosine):", np.round(A, 3))
print("B (one-cycle):      ", np.round(B, 3))
d = np.array(B) - np.array(A)                          # paired: same seed, same data order
print(f"A mean {np.mean(A):.3f} +- {np.std(A, ddof=1):.3f}   B mean {np.mean(B):.3f} +-
 ↪ {np.std(B, ddof=1):.3f}")
print(f"paired difference B - A: mean {d.mean():+.3f}, range {d.min():+.3f} to
 ↪ {d.max():+.3f}")
```

```
A (warmup + cosine): [0.971 0.962 0.956 0.956 0.959]
B (one-cycle):       [0.956 0.952 0.952 0.978 0.956]
A mean 0.961 +- 0.007   B mean 0.959 +- 0.011
paired difference B - A: mean -0.002, range -0.016 to +0.022
```

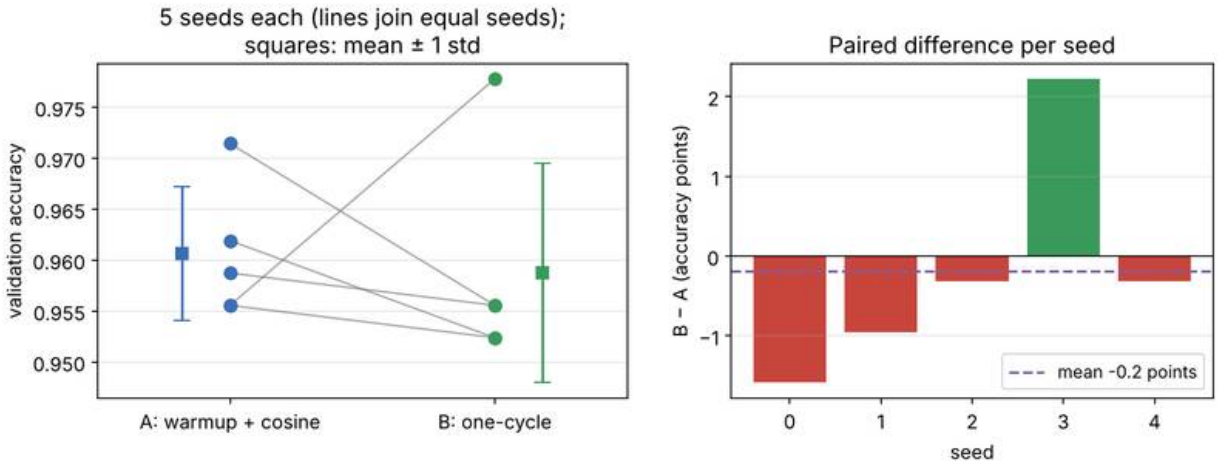



*Figure 7 — Seed variance on the toy (our runs, validation set of 315 images). A single seed could support either conclusion: seed 3 says B wins by 2.2 points, seed 0 says A wins by 1.6. Over five seeds the two are indistinguishable.*

With only seed 3 you would report that one-cycle wins by 2.2 points; with seed 0, that it loses by 1.6. Over five seeds the mean difference is 0.2 points with a spread far larger than that: no detectable difference on this toy (and one validation image is worth about 0.3 points here). Report the mean and standard deviation over seeds, compare configurations **paired** on the same seeds and splits, and use the confidence intervals and paired tests of ML 11 before claiming an improvement. Published large-scale results do this too: Goyal et al. report every ImageNet error as a mean and standard deviation over five trials [9].

### Baselines, ablations and budgets

A fair experiment changes one thing and holds everything else fixed:

- **Baselines.** Always include a simple baseline and the previous best recipe, trained by you in the same pipeline. A new method compared with a baseline from an old paper mixes the effect of the method with the effect of the recipe (DLCV 02).
- **Equal tuning budget.** Give every method the same number of trials for its hyperparameters. A method tuned for a week against a baseline run once wins by tuning, not by design.
- **Equal training budget.** Compare at the same number of steps or the same compute; a longer schedule alone can explain a gain.
- **Ablations.** Remove one component at a time from the full method, with several seeds each, and report the drop. An ablation that changes two things at once cannot tell you which one mattered.
- **One test evaluation.** Make every decision on validation data; evaluate the final choice on the test set once.

The Deep Learning Tuning Playbook organizes this into an incremental strategy: decide on the goal of each round of experiments, separate the hyperparameters being studied from those that must be re-tuned for each setting ("nuisance" hyperparameters) and those held fixed, and adopt a change only when the evidence is strong [26].

### Modern view

**Plain version.** The core habits of this chapter are stable. Recent work changes how training methods are compared, how much of the schedule must be fixed in advance, and how large pretrained models are adapted.

### What the surveys and guides say

Practical training knowledge lives more in reviews and guides than in single papers. Bengio's chapter in *Neural Networks: Tricks of the Trade* collects recommendations for gradient-based training, from the learning rate and mini-batch size to early stopping and hyperparameter search [12]. The Deep Learning Tuning Playbook covers choosing the architecture, optimizer, batch size and

initial configuration, an incremental, scientific tuning process, how long to train, and pipeline details such as checkpoint selection and experiment tracking [26]. Shorten and Khoshgoftaar organize image augmentation into geometric and color transformations, kernel filters, mixing images, random erasing, feature-space and adversarial augmentation, generative and style-transfer augmentation, and meta-learned policies, and also discuss test-time augmentation [7]. Xin et al. divide parameter-efficient fine-tuning of vision models into addition-based, partial-based, unified-based and multi-task tuning, and propose a unified benchmark [39].

Comparing training methods fairly is itself hard. Dahl et al. argue that the community cannot reliably identify training-algorithm improvements, because it is unclear when training is complete, results are sensitive to workload details, and methods that need tuning are hard to compare fairly; they introduced the AlgoPerf benchmark, which measures time to reach a target result on fixed hardware across several workloads [40].

### The state of the art, 2023–2026

- **Benchmarked training algorithms.** The first AlgoPerf competition drew 18 submissions from 10 teams. In the ruleset where submitters provide a search space, the winner used non-diagonal preconditioning and beat popular methods such as Adam even on wall-clock time; in the ruleset with no tuning allowed, the winner was based on Schedule-Free AdamW [41].
- **Fewer schedules to tune.** Defazio et al.'s Schedule-Free approach removes the need to know the stopping step in advance by unifying scheduling and iterate averaging, without new hyperparameters beyond those of momentum optimizers [42]. A simpler alternative is a constant learning rate followed by a short "cooldown" decay, named warmup-stable-decay (WSD) in MiniCPM [43]. Hägele et al. found that this scales as predictably as a cosine schedule, that a cooldown over a modest fraction of training is enough, and that weight averaging improves results along the trajectory at no extra training cost; their experiments are on language models, so verify the transfer to vision workloads yourself [44].
- **Weight averaging everywhere.** Beyond the EMA study above [29], averaging is also used in fine-tuning: WiSE-FT interpolates between the zero-shot and fine-tuned weights of a pretrained model and improves robustness under distribution shift [45], and model soups average several fine-tuned models to improve accuracy without extra inference cost [46].
- **Fine-tuning foundation models.** With very large pretrained backbones, full fine-tuning can be replaced by linear probing, LP-FT [35], or parameter-efficient methods such as LoRA [33], [39]. The practical questions shift from "which architecture" to "which pretrained model, and how much of it to adapt".

### Open problems

1. **Hyperparameters that transfer.** Best learning rates, warmup lengths and weight decays still change with model size, batch size and dataset; the range test and the linear scaling rule are approximations, not laws.
2. **Fair, affordable comparison.** Tuning budgets differ between papers, and seeds are often too few to separate methods. Benchmarks such as AlgoPerf address this for training algorithms

[40], but no comparable standard exists for most vision tasks.
3. **Determinism at scale.** Bit-exact reproducibility is not guaranteed across hardware and software versions [38], which makes small regressions hard to diagnose in large training runs.
4. **Augmentation for dense and specialized tasks.** How to choose augmentation automatically for detection, segmentation, medical and remote-sensing images, where geometry and intensity carry label information, is still an open question.
5. **Fine-tuning without forgetting.** Adapting a foundation model to a narrow task while keeping its robustness out of distribution is still an active research question [35], [45].

### Key takeaways

- Inspect a processed batch, check the initial loss and overfit one batch before scaling up.
- Split by the deployment unit, fit preprocessing on training data, and transform spatial labels with the image.
- Use a learning-rate range test, warmup and decay; recheck the schedule when batch size changes.
- Accumulation matches a larger batch only under compatible loss weighting and batch-dependent operations; normalize by the actual sample, pixel or box count.
- Tune interacting regularizers together and select checkpoints using validation data.
- Fine-tune pretrained features gently and check normalization behavior with small batches.
- Report multiple seeds, matched budgets and isolated ablations; reserve the test set for final evaluation.

### Exercises

1. **Range test, then schedule.** Rerun the range test with batch sizes 8, 16 and 64 (change `bs`). How does the learning rate of the minimum move? Then train with `warmup_cosine` at each batch size, using a peak taken from its own range test, and compare with the linear scaling rule.

   **Hint**

   In our runs (seeds 0 and 1) the minimum sat at about 0.15–0.16 for batch 8, 0.13–0.14 for batch 16 and 0.35 for batch 64: it moves up with the batch size, but not in exact proportion, and a short test is noisy. Remember that $T$ in steps shrinks as the batch grows at a fixed number of epochs, so the larger batches also get fewer updates.

2. **A one-batch test that passes for the wrong reason.** Shuffle `y_tr[:16]` (so labels no longer match images) and run `overfit_one_batch`. What happens to the loss, and what does that tell you about what the test can and cannot detect? Which check from the checklist would have caught the shuffled labels?

   **Hint**

The loss still goes to nearly zero (0.0056 in our run with the labels permuted by `np.random.default_rng(0)`), because a network can memorize any 16 labels, as Zhang et al. showed for random labels at scale [36]. The test checks the model, the loss and the update, not the data. Looking at a batch with its labels drawn on it (item 2) catches misaligned labels; so does a validation accuracy near chance.

3. **Flip a keypoint.** Extend the `hflip` function so it also transforms a list of keypoints given as `(x, y, name)` with names such as `left_eye` and `right_eye`. Write a check that fails if a flipped `left_eye` is still called `left_eye`.

   **Hint**

   The coordinates follow $x \to W - 1 - x$ for pixel centers (or $W - x$ for continuous coordinates, matching the box convention used here), $y$ unchanged. Then swap every left/right name pair with a lookup table. Applying the flip twice must return the original keypoints exactly, which is a good test.

4. **Accumulation with ignored pixels.** A segmentation loss averages the per-pixel cross-entropy over pixels whose label is not "ignore". Micro-batch 1 has 1,000 valid pixels with mean loss 0.5; micro-batch 2 has 3,000 valid pixels with mean loss 0.9. What is the mean loss of the combined batch, and what does averaging the two micro-batch means give instead?

   **Hint**

   The combined mean is $(1000 \cdot 0.5 + 3000 \cdot 0.9)/4000 = 0.8$. Averaging the means gives $(0.5 + 0.9)/2 = 0.7$, which underweights the micro-batch with more valid pixels. The same weighting applies to the gradients: accumulate sums and divide by the total count.

5. **How many seeds?** Using `train`, run configurations A and B with 10 seeds each instead of 5. Does the mean difference change sign? Estimate the standard error of the paired mean difference, $s_d/\sqrt{n}$, where $s_d$ is the standard deviation of the per-seed differences and $n$ the number of seeds, and decide whether a 0.5-point improvement could be detected with 10 seeds.

   **Hint**

   In our run with seeds 0–9 the sign does change: the mean difference B − A becomes about +0.6 points, against −0.2 points with five seeds. With $s_d \approx 1.3$ points, the standard error is $1.3/\sqrt{10} \approx 0.4$ points, so a 0.5-point improvement is only about one standard error away from zero: not convincing. Larger validation sets and more seeds both help; ML 10 shows paired bootstrap intervals.

6. **Fine-tuning plan.** You have 2,000 labeled X-ray images, a backbone pretrained on natural photos and one GPU that fits 8 images per batch. Write a training plan: normalization statistics, augmentations (which ones are unsafe?), how to handle batch normalization, the fine-tuning strategy, the learning-rate procedure and how you will decide that a change helped.

   **Hint**

Use the pretrained normalization statistics; avoid intensity augmentations that change diagnostic values, and be careful with flips if laterality matters; freeze the backbone BN statistics (or use group normalization in new layers) because 8 images per batch give noisy statistics; start with a linear probe, then LP-FT with smaller rates for early layers and a short warmup; run a range test for the head; split by patient; compare variants over several seeds with paired intervals.

## References


1. T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie and M. Li, "Bag of tricks for image classification with convolutional neural networks," in *Proc. IEEE/CVF CVPR*, 2019. doi
2. A. Karpathy, "A recipe for training neural networks," blog post, 2019. link
3. scikit-learn developers, "load_digits," *scikit-learn documentation*, version 1.9.1. link
4. M. Buda, A. Maki and M. A. Mazurowski, "A systematic study of the class imbalance problem in convolutional neural networks," *Neural Networks*, vol. 106, 2018. doi
5. T.-Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár, "Focal loss for dense object detection," in *Proc. IEEE ICCV*, 2017. doi
6. A. Buslaev, V. I. Iglovikov, E. Khvedchenya, A. Parinov, M. Druzhinin and A. A. Kalinin, "Albumentations: Fast and flexible image augmentations," *Information*, vol. 11, no. 2, 2020. doi
7. C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, 2019. doi
8. K. He, X. Zhang, S. Ren and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE ICCV*, 2015. doi
9. P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia and K. He, "Accurate, large minibatch SGD: Training ImageNet in 1 hour," arXiv:1706.02677 (technical report), 2017. arXiv
10. S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. ICML*, PMLR 37, 2015. link
11. Y. Wu and K. He, "Group normalization," in *Proc. ECCV*, 2018. doi
12. Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade*, 2nd ed., LNCS 7700, Springer, 2012. doi
13. L. N. Smith, "Cyclical learning rates for training neural networks," in *Proc. IEEE WACV*, 2017. doi
14. L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1 — learning rate, batch size, momentum, and weight decay," US Naval Research Laboratory Technical Report 5510-026, arXiv:1803.09820, 2018. arXiv
15. I. Loshchilov and F. Hutter, "SGDR: Stochastic gradient descent with warm restarts," in *Proc. ICLR*, 2017. arXiv
16. L. N. Smith and N. Topin, "Super-convergence: Very fast training of neural networks using large learning rates," in *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, Proc. SPIE 11006, 2019. doi
17. PyTorch contributors, "torch.optim.lr_scheduler.OneCycleLR," *PyTorch 2.15 documentation*, 2026. link
18. K. He, X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE CVPR*, 2016. doi

19. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE CVPR*, 2009. doi
20. D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICLR*, 2015. arXiv
21. I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *Proc. ICLR*, 2019. arXiv
22. PyTorch contributors, "torch.optim.Adam," *PyTorch 2.15 documentation*, 2026. link
23. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, 2014. link
24. G. Huang, Y. Sun, Z. Liu, D. Sedra and K. Q. Weinberger, "Deep networks with stochastic depth," in *Proc. ECCV*, 2016. doi
25. L. Prechelt, "Early stopping — but when?," in *Neural Networks: Tricks of the Trade*, 2nd ed., LNCS 7700, Springer, 2012. doi
26. V. Godbole, G. E. Dahl, J. Gilmer, C. J. Shallue and Z. Nado, "Deep learning tuning playbook," Google Research, GitHub repository, 2023. link
27. B. T. Polyak and A. B. Juditsky, "Acceleration of stochastic approximation by averaging," *SIAM Journal on Control and Optimization*, vol. 30, no. 4, 1992. doi
28. P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov and A. G. Wilson, "Averaging weights leads to wider optima and better generalization," in *Proc. UAI*, 2018. arXiv
29. D. Morales-Brotons, T. Vogels and H. Hendrikx, "Exponential moving average of weights in deep learning: Dynamics and benefits," *Transactions on Machine Learning Research*, 2024. arXiv
30. P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh and H. Wu, "Mixed precision training," in *Proc. ICLR*, 2018. arXiv
31. PyTorch contributors, "Automatic mixed precision package — torch.amp," *PyTorch 2.15 documentation*, 2026. link
32. J. Yosinski, J. Clune, Y. Bengio and H. Lipson, "How transferable are features in deep neural networks?," in *Advances in Neural Information Processing Systems 27*, 2014. link
33. E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang and W. Chen, "LoRA: Low-rank adaptation of large language models," in *Proc. ICLR*, 2022. arXiv
34. J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," in *Proc. ACL*, 2018. doi
35. A. Kumar, A. Raghunathan, R. Jones, T. Ma and P. Liang, "Fine-tuning can distort pretrained features and underperform out-of-distribution," in *Proc. ICLR*, 2022. arXiv
36. C. Zhang, S. Bengio, M. Hardt, B. Recht and O. Vinyals, "Understanding deep learning requires rethinking generalization," in *Proc. ICLR*, 2017. arXiv
37. R. Geirhos, J.-H. Jacobsen, C. Michaelis, R. Zemel, W. Brendel, M. Bethge and F. A. Wichmann, "Shortcut learning in deep neural networks," *Nature Machine Intelligence*, vol. 2, 2020. doi
38. PyTorch contributors, "Reproducibility," *PyTorch 2.15 documentation*, 2026. link
39. Y. Xin, J. Yang, S. Luo, Y. Du, Q. Qin, K. Cen, Y. He, Z. Zhang, B. Fu, X. Yang, G. Zhai, M.-H. Yang and X. Liu, "Parameter-efficient fine-tuning for pre-trained vision models: A survey and benchmark," arXiv:2402.02242, 2024. arXiv
40. G. E. Dahl, F. Schneider, Z. Nado, N. Agarwal, C. S. Sastry, P. Hennig, S. Medapati, R. Eschenhagen, P. Kasimbeg, D. Suo et al., "Benchmarking neural network training algorithms," arXiv:

2306.07179, 2023. arXiv
41. P. Kasimbeg, F. Schneider, R. Eschenhagen, J. Bae, C. S. Sastry, M. Saroufim, B. Feng, L. Wright, E. Z. Yang, Z. Nado, S. Medapati, P. Hennig, M. Rabbat and G. E. Dahl, "Accelerating neural network training: An analysis of the AlgoPerf competition," in *Proc. ICLR*, 2025. arXiv
42. A. Defazio, X. A. Yang, H. Mehta, K. Mishchenko, A. Khaled and A. Cutkosky, "The road less scheduled," in *Advances in Neural Information Processing Systems 37*, 2024. link
43. S. Hu, Y. Tu, X. Han, C. He, G. Cui, X. Long, Z. Zheng, Y. Fang, Y. Huang, W. Zhao et al., "MiniCPM: Unveiling the potential of small language models with scalable training strategies," in *Proc. Conference on Language Modeling (COLM)*, 2024. arXiv
44. A. Hägele, E. Bakouch, A. Kosson, L. Ben Allal, L. Von Werra and M. Jaggi, "Scaling laws and compute-optimal training beyond fixed training durations," in *Advances in Neural Information Processing Systems 37*, 2024. link
45. M. Wortsman, G. Ilharco, J. W. Kim, M. Li, S. Kornblith, R. Roelofs, R. Gontijo Lopes, H. Hajishirzi, A. Farhadi, H. Namkoong and L. Schmidt, "Robust fine-tuning of zero-shot models," in *Proc. IEEE/CVF CVPR*, 2022. doi
46. M. Wortsman, G. Ilharco, S. Y. Gadre, R. Roelofs, R. Gontijo-Lopes, A. S. Morcos, H. Namkoong, A. Farhadi, Y. Carmon, S. Kornblith and L. Schmidt, "Model soups: Averaging weights of multiple fine-tuned models improves accuracy without increasing inference time," in *Proc. ICML*, PMLR 162, 2022. link
47. PyTorch contributors, "Automatic mixed precision examples," *PyTorch 2.15 documentation*, 2026. link
48. PyTorch contributors, "DistributedDataParallel," PyTorch 2.15.0 source documentation, 2026. Official source.

# DLCV 01B • Building Blocks and How to Compose Them: Classification and Restoration Networks

A catalogue of the blocks deep vision networks are built from, the rules for composing them into classifiers and into image-restoration networks, side-by-side comparisons of matched models, and how to explore a design space without fooling yourself.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/building-blocks/

## What you'll learn

- Identify a network's stem, stages, blocks, neck and head.
- Track how blocks mix spatial and channel information, and estimate their costs.
- Explain why classification and restoration use resolution, normalization and skips differently.
- Compare matched network designs and diagnose composition errors with small experiments.
- Read ablations under controlled training and compute budgets.

## The big picture

A vision network looks complicated in a paper figure, but it is built from a small number of block types, repeated and connected. Two networks that use the same blocks can behave very differently because they are **composed** differently: where the resolution changes, where the skips go, where normalization sits, how the output is formed. This chapter is about those composition decisions.

> **Explain like I'm five**
>
> Think of a toolbox: saws, brushes, glue. A furniture maker uses them to turn a big log into one small chair leg: cut away most of the wood, keep only what matters. A picture restorer uses the same tools very differently: the painting must come back the same size, every brushstroke in place, just cleaner. A classifier is the furniture maker: it throws away pixels until only the answer ("cat") is left. A restoration network is the restorer: it must give back every pixel. Same tools, opposite plans.

**Why it matters.** Practical work is rarely inventing a block. It is choosing a backbone, adapting it to a new task, or deciding which of a paper's ten changes actually helped. A classifier backbone used unchanged for denoising throws away detail, and a block that helped in one paper may not help in your design space.

**Where the basics are.** The blocks themselves are taught elsewhere on this site, and this chapter links there instead of repeating them: convolution, receptive field and residual blocks in ML 03 and DLCV 01; the Transformer block, Swin and efficient attention in ML 05; the classification backbones in DLCV 02; encoder–decoders and ASPP in DLCV 04; restoration backbones and upsampling in DLCV 09 and DLCV 09A. All code here uses NumPy and SciPy only, runs on a CPU in seconds, and uses synthetic data or `skimage.data`.

### Anatomy of a network

**Plain version.** A vision network usually has a stem that extracts features, a body of repeated blocks and a task head. Follow resolution changes through all three.

**Precise version.** Write a feature map as a tensor of shape $C \times H \times W$, where $C$ is the number of channels (the **width**) and $H \times W$ the spatial size. The common parts are:

- **Stem**: the first one or two layers. ResNet uses a $7 \times 7$ convolution with stride 2 and a $3 \times 3$ max-pool with stride 2, so the body starts at $1/4$ of the input resolution [1]. ConvNeXt replaces this with a "patchify" $4 \times 4$ convolution with stride 4 [2]; ViT cuts the image into $16 \times 16$ patches [3]. Restoration networks use a single $3 \times 3$ convolution at full resolution, which SwinIR calls **shallow feature extraction** [4].
- **Stage**: a run of blocks that share one resolution and one width. A **block** is the repeated unit (a residual block, a Transformer block).
- **Downsampling layer** (transition): halves $H$ and $W$, usually while doubling $C$. It can be a strided convolution inside the first block of a stage (ResNet), a separate $2 \times 2$ convolution with stride 2 (ConvNeXt, NAFNet), patch merging (Swin), or pixel unshuffle (Restormer).
- **Neck**: an optional module that mixes features from several stages, such as the Feature Pyramid Network (FPN) used by detectors [5].
- **Head**: global average pooling and a linear layer for classification; a final convolution, and for super-resolution an upsampler, for restoration.

The **output stride** of a stage is the ratio between the input size and the stage's feature-map size. ResNet's four stages have output strides 4, 8, 16 and 32 [1]; so do Swin's [6]. Networks that keep one resolution and width throughout, like ViT, are **isotropic**; those with stages at decreasing resolution are **hierarchical**.

ResNet also states the most copied rule of macro design: when the feature map is halved, the number of filters is doubled "so as to preserve the time complexity per layer" [1]. A $3 \times 3$ convolution with $C$ input and output channels on an $H \times W$ map costs $9HWC^2$ multiply-accumulates (MACs); after halving the map and doubling the width it costs $9 \cdot \frac{H}{2} \cdot \frac{W}{2} \cdot (2C)^2 = 9HWC^2$, the same. Figure 1 shows the three macro-architectures this chapter compares.

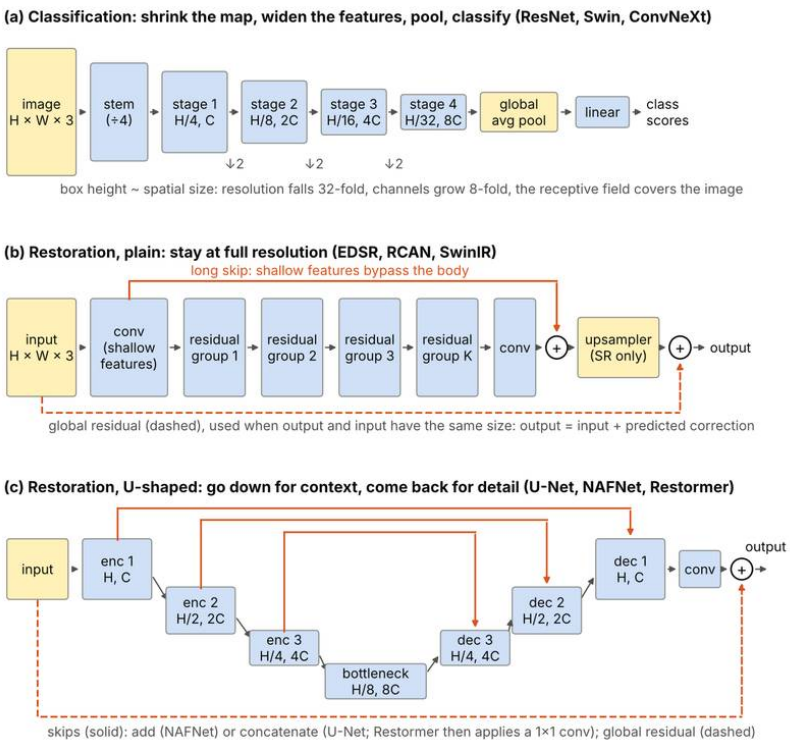



*Figure 1 — Three macro-architectures. (a) A classifier is a funnel: resolution falls, width grows, and global pooling removes the spatial grid. (b) A plain restorer keeps full resolution and adds long and global skips. (c) A U-shaped restorer goes down for context and comes back for detail.*

## The block catalogue

**Plain version.** Blocks mix information across space and channels. Skips, normalization and nonlinearities control how that mixing affects the signal and its cost.

The MetaFormer paper makes this split explicit: a Transformer block is a **token mixer** (attention) followed by a **channel mixer** (the MLP), each in a residual branch with normalization, and replacing attention by a simple spatial pooling operator still gave competitive accuracy on ImageNet [7], which suggests that the general structure carries much of the performance [8]. Figure 2 shows the blocks of this chapter in that common format.

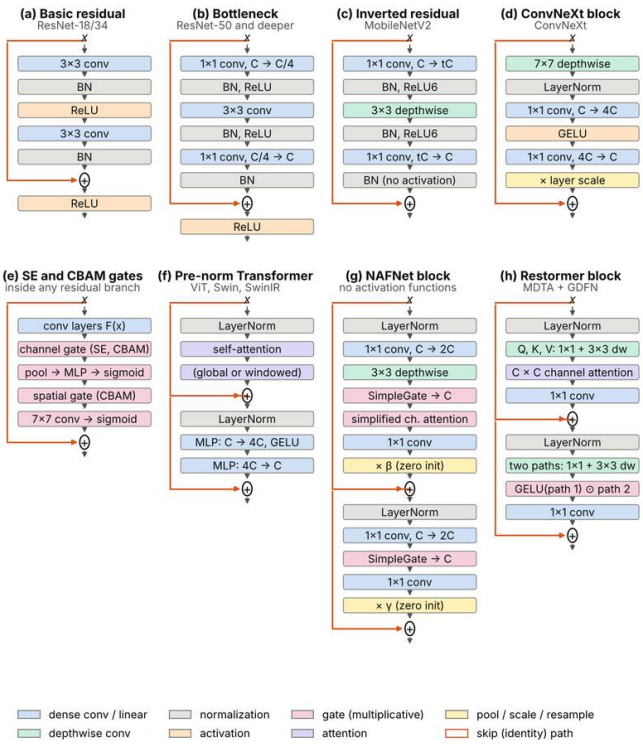



*Figure 2 — The block catalogue, drawn in one format. Colors mark the kind of operation; orange arrows are identity skips. (a)–(d) are classification blocks, (e) are attention add-ons, (f) is the Transformer block, (g)–(h) are restoration blocks.*

The table gives each block one plain line and says where its basics are taught.

| Block | One plain line | Used in | Basics |
|---|---|---|---|
| Residual (basic) | Learn a correction $F(x)$ and add it to the input | ResNet-18/34, EDSR | DLCV 01 |
| Bottleneck | Squeeze channels with $1 \times 1$, filter, expand back | ResNet-50 and deeper | DLCV 02 |
| Depthwise separable | Filter each channel alone, then mix channels with $1 \times 1$ | MobileNet [9] | DLCV 02 |
| Inverted residual | Expand, filter depthwise, project back linearly; skip between thin ends | MobileNetV2 [10] | ML 03 |
| ConvNeXt block | Large depthwise kernel, then a Transformer-style MLP | ConvNeXt | DLCV 02 |
| SE / CBAM | Reweight channels (and positions) by a gate computed from pooled statistics | SENet, RCAN, NAFNet | DLCV 02 note |
| Self-attention | Every token looks at every other token | ViT | ML 05 |

| Block | One plain line | Used in | Basics |
|---|---|---|---|
| Window attention | Attention inside local windows that shift between layers | Swin, SwinIR | ML 05 |
| Channel (transposed) attention | Attention between channels, so cost grows linearly with pixels | Restormer | this chapter |
| Gated block | Multiply two feature halves instead of applying an activation | NAFNet, Restormer's GDFN | this chapter |

### Residual connections and residual scaling

**Plain version.** A residual block adds a learned correction to its input. Check normalization, activation placement and the correction's initial scale.

**Precise version.** The general form is

$$y = x + \alpha F(x),$$

where $x$ is the block input, $F$ the residual branch (convolutions, attention, normalization), $y$ the output and $\alpha$ a scale on the branch. Three placements of the nonlinearity are common:

- **Post-activation** (original ResNet): $y = \mathrm{ReLU}(x + F(x))$, with BN inside $F$ [1].
- **Pre-activation**: BN and ReLU move inside the branch, before each convolution, so the shortcut is a pure identity; He et al. found this easier to train at very large depth [11]. The pre-norm Transformer block, $y = x + \mathrm{MHA}(\mathrm{LN}(x))$, is the same idea (ML 05).
- **No activation after the addition**: EDSR, MobileNetV2, ConvNeXt, NAFNet and Restormer all end the block with the addition.

Several designs control $\alpha$ explicitly. EDSR found training numerically unstable once the number of feature maps grew large, and fixed it with a constant residual scaling of 0.1 in its final model of 32 blocks with 256 channels [12]. ConvNeXt uses Layer Scale [13], a learnable per-channel $\alpha$, initialized to $10^{-6}$ [2]. NAFNet's code multiplies each branch by learnable per-channel parameters initialized to zero, so every block starts as the identity [14]; zero-initializing the last BN scale of a ResNet branch does the same (DLCV 01A). The shared reason: a deep stack of identity blocks is easy to optimize, and each block can then grow its correction.

Skips also come at two scales. **Short skips** wrap one block; **long skips** wrap a group of blocks or the whole body. RCAN's "residual in residual" structure uses both: residual groups with long skip connections, each containing blocks with short skip connections [15].

Training and inference can also use different block graphs: RepVGG [54] trains with additional affine branches and converts them into one convolution for deployment.

*Online only: RepVGG: training with branches, deploying a plain network — https://blog.shyandram.dev/en/learn/deep-learning/building-blocks/papers/repvgg-training-with-branches-deploying-a-plain-network/*

### Bottlenecks, inverted residuals and depthwise separable convolutions

**Plain version.** Spatial filtering and channel mixing need not happen in one expensive layer. Bottlenecks run the expensive $3 \times 3$ on fewer channels; depthwise convolutions filter each channel alone and leave mixing to cheap $1 \times 1$ layers. These blocks change cost by large factors, so it pays to count.

**Precise version.** A dense $k \times k$ convolution from $a$ to $b$ channels has $k^2ab$ weights and, at stride 1, costs $k^2ab$ MACs per output pixel. A depthwise $k \times k$ convolution on $a$ channels has $k^2a$ of each. The ResNet bottleneck reduces $C$ to $C/4$, filters, and expands back [1]. The MobileNetV2 **inverted residual** does the opposite: it expands $C$ to $tC$ with a $1 \times 1$ convolution (expansion factor $t = 6$ in the paper's main experiments), filters with a $3 \times 3$ depthwise convolution, and projects back with a **linear** $1 \times 1$ convolution, without an activation, because the authors argue that a ReLU in the narrow layer destroys information [10]. The paper gives its cost as $h \cdot w \cdot d' \cdot t\,(d' + k^2 + d'')$ multiply-adds for an $h \times w$ input with $d'$ input and $d''$ output channels [10]. The snippet counts weights and MACs per output pixel for the blocks of Figure 2 at $C = 64$, on a $128 \times 128$ map, and checks the inverted residual against that formula.

```
# Weights (no biases or norm parameters) and MACs per output pixel of common blocks,
# all with C input and C output channels at stride 1. A dense k x k conv from a to b
# channels has k*k*a*b weights and costs k*k*a*b MACs per pixel; a depthwise one k*k*a.
def blocks(C, n_tokens):
    g = int(2.66 * C)                                  # Restormer GDFN hidden width
    return {                                           # name: (weights, MACs per pixel)
        'basic residual (2 x 3x3)':     (18 * C * C, 18 * C * C),
        'bottleneck (1x1, 3x3, 1x1)':   (2 * C * C // 4 + 9 * (C // 4) ** 2,) * 2,
        'depthwise separable 3x3':      (9 * C + C * C,) * 2,
        'inverted residual, t = 6':     (12 * C * C + 54 * C,) * 2,
        'ConvNeXt (7x7 dw, 4x MLP)':    (49 * C + 8 * C * C,) * 2,
        'NAFNet block':                 (7 * C * C + 18 * C, 6 * C * C + 18 * C),  # SCA acts
        ↪  on a pooled vector
        'SE add-on, r = 16':            (2 * C * C // 16, C),                      # one
        ↪  multiply per pixel
        'ViT block, global attention':  (12 * C * C, 12 * C * C + 2 * n_tokens * C),
        'Swin block, 8 x 8 windows':    (12 * C * C, 12 * C * C + 2 * 64 * C),
        'Restormer block (MDTA+GDFN)':  (4 * C * C + 27 * C + 3 * g * C + 18 * g,
                                         4 * C * C + 27 * C + 2 * C * C // 4 + 3 * g * C + 18
                                         ↪  * g),  # 4 heads
    }

C, side = 64, 128
for name, (p, m) in blocks(C, side * side).items():
    print(f"{name:30s} {p:7d} weights  {m:8d} MACs/pixel")

# Check the inverted residual against MobileNetV2's formula h*w*d'*t*(d' + k^2 + d'') per
↪  pixel.
t, k = 6, 3
assert blocks(C, 1)['inverted residual, t = 6'][1] == C * t * (C + k * k + C)
```

```
basic residual (2 x 3x3)         73728 weights     73728 MACs/pixel
```

```
bottleneck (1x1, 3x3, 1x1)        4352 weights      4352 MACs/pixel
depthwise separable 3x3           4672 weights      4672 MACs/pixel
inverted residual, t = 6         52608 weights     52608 MACs/pixel
ConvNeXt (7x7 dw, 4x MLP)        35904 weights     35904 MACs/pixel
NAFNet block                     29824 weights     25728 MACs/pixel
SE add-on, r = 16                  512 weights        64 MACs/pixel
ViT block, global attention      49152 weights   2146304 MACs/pixel
Swin block, 8 x 8 windows        49152 weights     57344 MACs/pixel
Restormer block (MDTA+GDFN)      53812 weights     55860 MACs/pixel
```

Four lessons come out of these numbers.

1. **At equal width, the blocks differ by more than 10 times in cost.** The bottleneck and the separable block are about 16 to 17 times cheaper than the basic block. Real networks therefore never compare blocks at equal width: MobileNetV2 uses an inverted residual on thin tensors, ResNet-50 uses bottlenecks on wide ones. Compare blocks at equal cost, not at equal $C$.
2. **Global attention over pixels is out of reach for restoration.** With every pixel of a $128 \times 128$ map as a token, the ViT block costs about 37 times more per pixel than a Swin block with $8 \times 8$ windows, and the gap grows with image size. This is why restoration Transformers use windows (SwinIR) or attention across channels (Restormer); see ML 05 for the general cost argument.
3. **Attention add-ons are cheap in MACs.** An SE gate adds $C^2/8$ weights and one multiply per channel and pixel; its cost is memory traffic, not arithmetic.
4. **MACs are not latency.** Depthwise layers do few operations per byte moved, so they can run slower than their MAC count suggests; ML 03 and the DLCV 02 efficiency section discuss this. Measure on the target hardware.

Whether a cheaper block is also a better one depends on the design space. RegNet's population study found that its best models used neither a bottleneck nor an inverted bottleneck, and that depthwise convolution did even worse than the inverted bottleneck in its setting [16], while ConvNeXt found the inverted bottleneck slightly better within its own sequence of changes [2]. Both can be true: the comparison depends on the other choices held fixed.

### Attention inside a CNN: SE and CBAM

**Plain version.** A gate looks at a summary of the whole feature map and decides which channels (or positions) to turn up or down for this particular image. It is a cheap way to give a convolutional network some content-dependent, image-wide behavior.

**Precise version.** Squeeze-and-excitation (SE) computes a channel gate $s = \sigma(W_2\, \delta(W_1\, \bar{u}))$ from the vector $\bar{u}$ of channel means, where $W_1$ and $W_2$ are the weights of a two-layer MLP that shrinks the dimension by a ratio $r$ and restores it, $\delta$ is a ReLU and $\sigma$ the sigmoid, and multiplies each channel $u_c$ of the feature map by $s_c$; the formula and code are in the DLCV 02 note [17]. CBAM extends it in two ways [18]: its channel gate pools each channel by both average and maximum and passes both vectors through one shared MLP, and a second, **spatial** gate pools across channels and applies a $7 \times 7$ convolution, so the block can also turn whole regions up or down. CBAM's

ablations found the sequential channel-then-spatial order better than running the two gates in parallel [18].

*Online only: CBAM's channel and spatial gates — https://blog.shyandram.dev/en/learn/deep-learning/building-blocks/notes/cbams-channel-and-spatial-gates/*

**Where it goes.** Both SE and CBAM sit on the residual branch, before the addition: SE applies the gate to the non-identity branch "before summation with the identity branch" [17], and CBAM applies its module to the convolution outputs of each ResNet block [18]. RCAN brought channel attention into super-resolution [15]; NAFNet later simplified it to $X \cdot W\,\mathrm{pool}(X)$, a single linear map of the pooled vector with no ReLU or sigmoid, and lost nothing in its ablation [19]. The Modern view's attention survey places SE in the "channel attention" family and CBAM in "channel and spatial" [20].

### Token mixers: self-, window- and channel attention

**Plain version.** Attention mixes information between tokens with weights computed from the content, instead of fixed weights as in a convolution. The design question is **which tokens talk to which**: all of them (global), those in the same local window, or, in Restormer's twist, the channels instead of the pixels.

**Precise version.** Global multi-head self-attention over $N$ tokens of width $C$ costs about $2N^2C$ MACs for the scores and the weighted sum, on top of $4NC^2$ for the projections; window attention restricts it to $M \times M$ windows, giving $2M^2NC$ (ML 05) [6]. Restormer's **multi-Dconv head transposed attention** (MDTA) instead computes attention between channels [21]:

$$\hat{X} = W_p\,\mathrm{Attention}(\hat{Q},\hat{K},\hat{V}) + X, \qquad \mathrm{Attention}(\hat{Q},\hat{K},\hat{V}) = \hat{V}\cdot\mathrm{Softmax}(\hat{K}\cdot\hat{Q}/\alpha),$$

where $X$ is the input feature map and $\hat{X}$ the output, $Q = W_d^Q W_p^Q Y$ (and likewise $K, V$) is computed from the layer-normalized input $Y$ by a $1 \times 1$ convolution $W_p^Q$ followed by a $3 \times 3$ depthwise convolution $W_d^Q$, the hats denote reshaped matrices $\hat{Q} \in \mathbb{R}^{HW\times C}$, $\hat{K} \in \mathbb{R}^{C\times HW}$, $\hat{V} \in \mathbb{R}^{HW\times C}$, $\alpha$ is a learnable temperature, and $W_p$ without a superscript is the output $1 \times 1$ convolution. The attention map $\hat{K}\hat{Q}$ has size $C \times C$ instead of $HW \times HW$, so the cost grows linearly with the number of pixels, and every output pixel still depends on the whole image through the channel statistics. The depthwise convolutions put local context back in. As in ordinary multi-head attention, the channels are split into heads with separate maps.

Restormer's feed-forward part (GDFN) is covered with the gates below. The trade-off between the mixers returns in the restoration sections: window attention mixes space strongly but locally, channel attention reaches globally at low cost, and a comparative study argued that Restormer's spatial mapping is weaker than SwinIR's for super-resolution [22].

### Normalization: BN, LN, GN, and when to leave it out

**Plain version.** A normalization layer rescales features to a standard size and then lets the network learn its preferred scale. The layers differ in **which numbers share statistics**. In classification the

choice mostly affects training stability; in restoration the wrong choice can change the brightness and contrast of the output.

**Precise version.** All the common layers compute $y = \gamma(x-\mu)/\sqrt{\sigma^2+\epsilon}+\beta$ over some set of values (ML 02 gives the template). For a mini-batch of shape $N \times C \times H \times W$:

| Layer | Statistics computed over | Depends on the batch? |
|---|---|---|
| BatchNorm (BN) [23] | $N, H, W$ for each channel | yes; running averages at test time |
| LayerNorm (LN) [24] | the features of one sample (for images, implementations differ: all of $C, H, W$, or $C$ at each position) | no |
| InstanceNorm (IN) [25] | $H, W$ for each sample and channel | no |
| GroupNorm (GN) [26] | $H, W$ and a group of channels, for each sample | no; $G = 1$ gives LN, $G = C$ gives IN |

BN is the default in classification CNNs and works well with large batches. Its known failure is small batches: GN's authors report a ResNet-50 on ImageNet with 10.6% lower error than its BN counterpart at a batch size of 2 [26]. What BN computes, and why training and inference differ, is in the DLCV 01 note.

**Why restorers dropped BN.** EDSR removed BN from its residual blocks, arguing that because BN normalizes the features it removes "range flexibility" from the network; the change improved accuracy and cut training memory by about 40% compared with SRResNet, the residual SR network of the SRGAN paper [27], [12]. ESRGAN added that BN tends to introduce artifacts when training and test statistics differ, especially in deeper networks trained with a GAN loss, and removed it as well [28]. This is not a law: DnCNN combined residual learning with BN for Gaussian denoising and reported that the two help each other [29]. The snippet shows the mechanism with a fixed, untrained residual branch $r(x)$ (convolution, normalization, ReLU, $1 \times 1$ convolution) and the global residual $y = x + r(x)$. The same mid-gray image $A$ is restored inside two different batches.

```
import numpy as np
from scipy.ndimage import gaussian_filter, correlate

rng = np.random.default_rng(0)

def image(level, size=32):
    """A smooth synthetic image with mean brightness `level` (values roughly in [0, 1])."""
    t = gaussian_filter(rng.standard_normal((size, size)), 2)
    return level + 0.15 * t / t.std()

C = 4
W1 = rng.normal(0.1, 0.2, (C, 3, 3))          # 3x3 conv, 1 -> C channels (fixed random
 ↪ weights)
w2 = rng.normal(0.0, 0.05, C)                 # 1x1 conv, C -> 1 channel

def features(batch):                          # (N, H, W) -> (N, C, H, W)
    return np.stack([[correlate(x, W1[c], mode='nearest') for c in range(C)] for x in batch])
```

```
def branch(batch, norm, running=None):
    """Residual branch: conv -> normalize -> ReLU -> 1x1 conv."""
    f = features(batch)
    if norm == 'bn-train':                      # per channel, over the whole batch
        mu, var = f.mean((0, 2, 3), keepdims=True), f.var((0, 2, 3), keepdims=True)
    elif norm == 'bn-eval':                     # per channel, stored training statistics
        mu, var = running
    else:                                       # 'instance': per sample and channel
        mu, var = f.mean((2, 3), keepdims=True), f.var((2, 3), keepdims=True)
    h = np.maximum((f - mu) / np.sqrt(var + 1e-5), 0)
    return np.tensordot(w2, h, axes=(0, 1))    # (N, H, W)

train = np.stack([image(rng.uniform(0.3, 0.7)) for _ in range(64)])
f = features(train)
running = (f.mean((0, 2, 3), keepdims=True), f.var((0, 2, 3), keepdims=True))

A = image(0.5)
dark = np.stack([image(0.15) for _ in range(7)])
bright = np.stack([image(0.85) for _ in range(7)])
for norm in ['bn-train', 'bn-eval', 'instance']:
    for name, others in [('dark', dark), ('bright', bright)]:
        batch = np.concatenate([A[None], others])
        out = A + branch(batch, norm, running)[0]          # global residual: y = x + r(x)
        print(f"{norm:8s}  A with 7 {name:6s} images: output mean {out.mean():.3f}")

# Per-sample normalization forgets each image's level and contrast:
B = 0.5 * A + 0.3
r_A, r_B = branch(A[None], 'instance')[0], branch(B[None], 'instance')[0]
print("instance-norm branch, A vs 0.5*A + 0.3: max |r_A - r_B| =", np.abs(r_A -
 ↪  r_B).max().round(4))
print("outputs with the global residual: means", (A + r_A).mean().round(3), (B +
 ↪  r_B).mean().round(3))
```

```
bn-train  A with 7 dark   images: output mean 0.629
bn-train  A with 7 bright images: output mean 0.482
bn-eval   A with 7 dark   images: output mean 0.509
bn-eval   A with 7 bright images: output mean 0.509
instance  A with 7 dark   images: output mean 0.521
instance  A with 7 bright images: output mean 0.521
instance-norm branch, A vs 0.5*A + 0.3: max |r_A - r_B| = 0.0005
outputs with the global residual: means 0.521 0.575
```

In training mode, BN gives the same image (input mean 0.492) an output brightness of 0.629 or 0.482 depending only on its batch companions (Figure 3). In evaluation mode the batch no longer matters, but the stored statistics come from other images, so the network behaves differently at test time from how it was trained: the mismatch EDSR and ESRGAN describe. Per-sample normalization is batch-independent but blind in the other direction: the branch gives the same output for $A$ and for $0.5A + 0.3$ (up to $\epsilon$), so only the identity path carries the image's own level and contrast. That is why restorers that do normalize (NAFNet, Restormer and SwinIR use LayerNorm) keep it **inside** residual branches with a global residual [19], [21], [4]. The weights are random and untrained; training can partly compensate, but the batch dependence is structural.

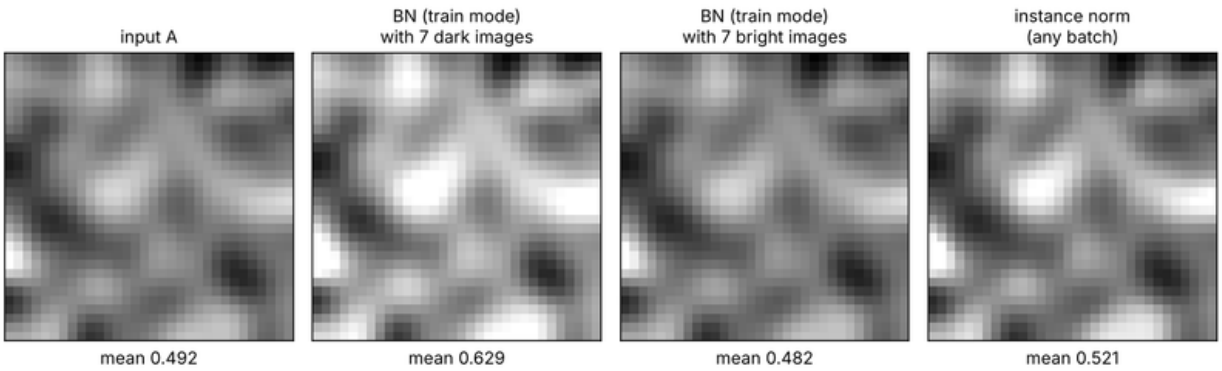



*Figure 3 — The same image through the same untrained residual branch. With batch normalization in training mode its output brightness depends on the other images in the batch; with per-sample normalization it does not.*

NAFNet's ablation shows the other side: adding LayerNorm to its plain block let it train with a 10 times larger learning rate and raised PSNR [30] by 0.44 dB on the SIDD denoising benchmark [31] and by 3.39 dB on the GoPro deblurring benchmark [32], [19]. Normalization helps restorers too, as long as it does not take away the information the output needs.

### Activations and gates

**Plain version.** A nonlinearity is what lets stacked layers do more than one linear map. The classic choice applies a fixed function to every number (ReLU, GELU). A gate instead **multiplies** two learned features, so one feature decides how much of the other passes. Modern restoration blocks use gates and, in NAFNet's case, nothing else.

**Precise version.** ReLU, GELU and the SwiGLU feed-forward block [33] are defined in ML 02. The gated linear unit (GLU) of Dauphin et al. computes $f(X) \odot \sigma(g(X))$ with two linear maps $f, g$ and a sigmoid $\sigma$ [34]. NAFNet's **SimpleGate** drops the sigmoid and the second map [19]:

$$\text{SimpleGate}(X) = X_1 \odot X_2, \qquad X = [X_1; X_2],$$

where $X$ is a feature map with $2C$ channels, $X_1$ and $X_2$ are its first and second $C$ channels, and $\odot$ is element-wise multiplication. The product of two linear functions of the input is quadratic, so the block is nonlinear without any activation function; the authors note that GELU itself, $x\,\Phi(x)$ with $\Phi$ the standard normal CDF [35], is a special case of a gate. Restormer's GDFN uses $\phi(W_d^1 W_p^1 \text{LN}(X)) \odot W_d^2 W_p^2 \text{LN}(X)$ with $\phi$ the GELU, two paths of $1 \times 1$ and $3 \times 3$ depthwise convolutions, and a channel expansion $\gamma = 2.66$ [21]. In NAFNet's ablation, replacing GELU by SimpleGate gave +0.08 dB on SIDD and +0.41 dB on GoPro [19].

### Upsampling: transposed convolution, pixel shuffle and resize-convolution

**Plain version.** Upsampling makes a feature map larger. There are three common ways: let each input pixel "paint" a learned patch into the output (transposed convolution), compute several val-

ues per input pixel and rearrange them (pixel shuffle), or enlarge first and convolve afterwards (resize-convolution). The first two can easily paint a regular checkerboard pattern; the third cannot.

**Precise version.** The three methods and the exact pixel-shuffle index rule are in DLCV 09 and DLCV 09A [36]. What matters for composition is a counting argument. With stride $s$, the output pixels fall into $s^2$ **phases** (their row and column positions modulo $s$). In a transposed convolution, the output at a given phase receives only some of the kernel's taps. Its value on a flat input equals the sum of those taps, the **phase sum**. If the phase sums differ, a flat input comes out as a periodic pattern. One cause, which Odena et al. call **uneven overlap**, occurs when the kernel size is not divisible by the stride: the phases then receive different numbers of taps [37]. Even with divisible sizes, random weights give unequal phase sums, so a freshly initialized layer still makes a pattern; Aitken et al. list these two causes and strided operations in the loss as the three sources of checkerboard artifacts [38]. A sub-pixel convolution has the same freedom, since each phase has its own filter. A resize-convolution (nearest-neighbor upsampling, then an ordinary convolution) applies the same full kernel at every output pixel, so all phases see the same sum. The snippet upsamples a flat $16 \times 16$ input by 2.

```
import numpy as np
from scipy.ndimage import correlate

rng = np.random.default_rng(0)

def transposed_conv(x, w, s):
    """Each input pixel 'paints' a scaled copy of the k x k kernel w; copies are s pixels
     ↪  apart."""
    k, (H, W) = w.shape[0], x.shape
    out = np.zeros((s * (H - 1) + k, s * (W - 1) + k))
    for i in range(H):
        for j in range(W):
            out[s * i:s * i + k, s * j:s * j + k] += x[i, j] * w
    return out

def resize_conv(x, w, s):
    """Nearest-neighbour upsampling by s, then an ordinary 'same' convolution with w."""
    return correlate(np.kron(x, np.ones((s, s))), w, mode='nearest')

def subpixel_conv(x, ws, s):
    """s*s ordinary convolutions whose outputs are interleaved (pixel shuffle)."""
    out = np.zeros((s * x.shape[0], s * x.shape[1]))
    for n, w in enumerate(ws):
        out[n // s::s, n % s::s] = correlate(x, w, mode='nearest')
    return out

x = np.ones((16, 16))                          # flat input: a good upsampler returns a flat
 ↪  image
cases = {
    'transposed k=3, s=2, uniform w': transposed_conv(x, np.ones((3, 3)), 2)[4:-4, 4:-4],
    'transposed k=4, s=2, uniform w': transposed_conv(x, np.ones((4, 4)), 2)[4:-4, 4:-4],
    'transposed k=4, s=2, random w ': transposed_conv(x, rng.uniform(0.5, 1.5, (4, 4)),
     ↪  2)[4:-4, 4:-4],
```

```
    'sub-pixel, 4 random 3x3 kernels': subpixel_conv(x, rng.uniform(0.5, 1.5, (4, 3, 3)),
    ↪  2)[4:-4, 4:-4],
    'resize + conv, random 3x3 w   ': resize_conv(x, rng.uniform(0.5, 1.5, (3, 3)), 2)[4:-4,
    ↪  4:-4],
}
for name, y in cases.items():
    phases = [y[a::2, b::2].mean() for a in (0, 1) for b in (0, 1)]     # the four 2x2
    ↪  sub-pixel phases
    print(f"{name}: phase means {np.round(phases, 2)}  relative spread {np.ptp(y) /
    ↪  y.mean():.2f}")
```

```
transposed k=3, s=2, uniform w: phase means [4. 2. 2. 1.]  relative spread 1.30
transposed k=4, s=2, uniform w: phase means [4. 4. 4. 4.]  relative spread 0.00
transposed k=4, s=2, random w : phase means [4.04 3.22 5.01 3.85]  relative spread 0.44
sub-pixel, 4 random 3x3 kernels: phase means [ 8.71 10.13  9.49  8.91]  relative spread 0.15
resize + conv, random 3x3 w   : phase means [7.86 7.86 7.86 7.86]  relative spread 0.00
```

With $k = 3$ and $s = 2$, one phase receives 4 taps, two receive 2 and one receives 1, even with uniform weights: the $4:2:2:1$ pattern is built into the shapes. With $k = 4$ every phase receives 4 taps, so uniform weights give a flat output, but random weights do not. Only resize-convolution is flat for any weights. Figure 4 shows the same four upsamplers on a flat input and on a photograph. Training can push the phase sums together, and Odena et al. report their best results with nearest-neighbor resize-convolution [37]; for pixel shuffle, initializing all phases from one kernel removes the pattern at the start [38], as noted in DLCV 09A.

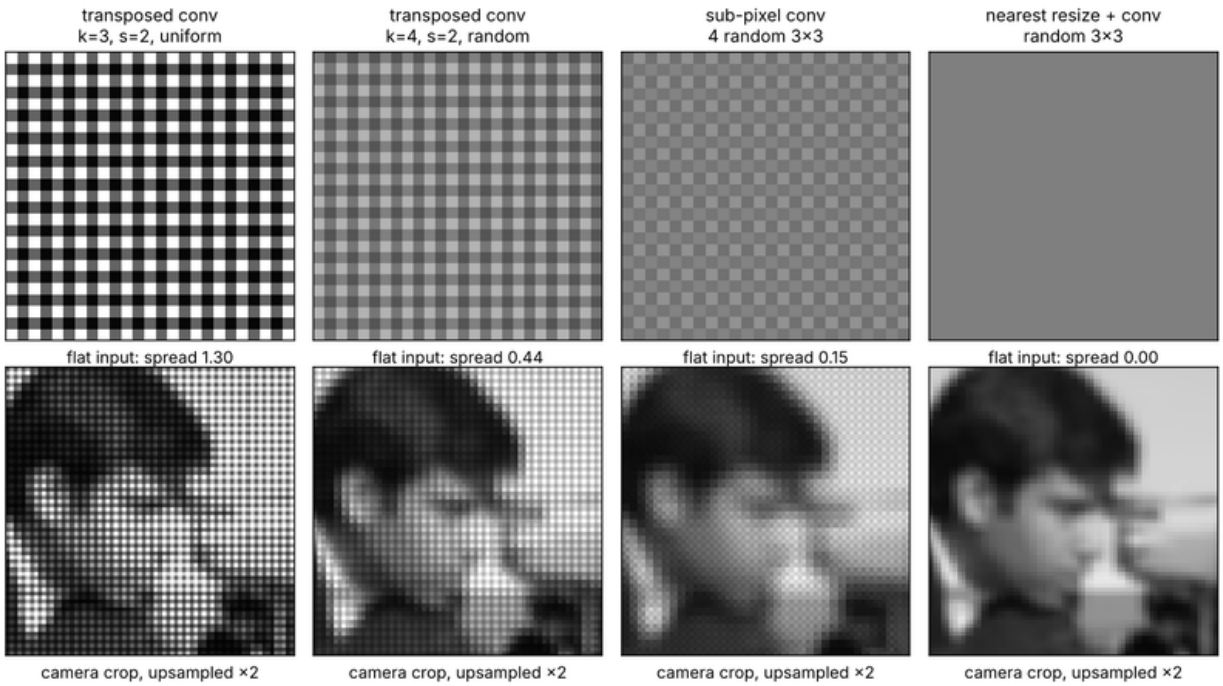



*Figure 4 — Checkerboard artifacts from untrained upsamplers. Top: a flat input (relative spread printed*

*below). Bottom: a camera crop upsampled by 2 with the same kernels, rescaled to unit average gain. Only resize-convolution is free of the pattern by construction.*

*Online only: Phase sums of a transposed convolution in 1-D — https://blog.shyandram.dev/en/learn/deep-learning/building-blocks/notes/phase-sums-of-a-transposed-convolution-in-1-d/*

Downsampling has matching choices: strided convolution (ResNet), pooling, a separate $2 \times 2$ convolution with stride 2 (ConvNeXt, NAFNet [14]), patch merging (Swin) and pixel unshuffle (Restormer [21]). ConvNeXt reports that its separate $2 \times 2$ downsampling layers made training diverge until it added normalization layers wherever the spatial resolution changes [2].

### Skip connections: add or concatenate

**Plain version.** A skip connection brings an earlier feature map to a later layer. It can be **added** (the two maps must have the same shape, and the result keeps the width) or **concatenated** (the maps are stacked along channels, and the next layer decides how to combine them). Concatenation is more flexible and more expensive.

**Precise version.** Concatenation followed by a $1 \times 1$ convolution contains addition as a special case: with encoder map $e$ and decoder map $d$, each with $C$ channels, the $1 \times 1$ convolution with weight matrix $[I \;\; I]$ (two $C \times C$ identity blocks side by side) maps $[e; d]$ to $e + d$. So concatenation can learn anything addition can, and more, at the cost of a wider layer and of storing both maps. The snippet checks the identity and counts the cost of the first convolution after each merge in a 4-level U-Net decoder with widths 32 to 256, on a $256 \times 256$ input.

```
import numpy as np

rng = np.random.default_rng(0)

# 1) Concatenation followed by a 1x1 convolution contains addition as a special case.
C, H, W = 8, 5, 5
enc, dec = rng.standard_normal((C, H, W)), rng.standard_normal((C, H, W))
cat = np.concatenate([enc, dec])                        # (2C, H, W)
W_fuse = np.hstack([np.eye(C), np.eye(C)])              # a 1x1 conv is a (C_out x C_in)
 ↪  matrix
print("concat + 1x1 conv [I I] == add:", np.allclose(np.tensordot(W_fuse, cat, 1), enc +
 ↪  dec))

# 2) What each choice costs in the decoder of a 4-level U-Net on a 256 x 256 input.
widths, size = [32, 64, 128, 256], 256
totals = {'add': [0, 0], 'concat': [0, 0], 'concat + 1x1': [0, 0]}
for level, c in enumerate(widths):
    hw = (size // 2 ** level) ** 2                      # pixels at this level
    first = {'add': 9 * c * c,                          # first 3x3 conv after the merge
             'concat': 9 * 2 * c * c,
             'concat + 1x1': 2 * c * c + 9 * c * c}     # 1x1 halves channels, then 3x3
    for k, p in first.items():
        totals[k][0] += p
        totals[k][1] += p * hw
for k, (p, m) in totals.items():
```

```
    print(f"{k:13s} merge-layer weights {p:7d}   MACs {m / 1e9:.2f} G")
```

```
concat + 1x1 conv [I I] == add: True
add           merge-layer weights  783360   MACs 2.42 G
concat        merge-layer weights 1566720   MACs 4.83 G
concat + 1x1  merge-layer weights  957440   MACs 2.95 G
```

Concatenating doubles the merge layer's weights and MACs; a $1 \times 1$ fusion first, as Restormer does, recovers most of the saving. The choice differs across well-known designs:

- **U-Net** concatenates the (cropped) encoder map with the decoder map after a $2 \times 2$ up-convolution that halves the channels [39].
- **FPN** adds: the coarser map is upsampled by 2 with nearest-neighbor interpolation and added to a $1 \times 1$-projected backbone map, and a $3 \times 3$ convolution on each merged map reduces the aliasing of the upsampling [5] (DLCV 03).
- **Restormer** concatenates and then halves the channels with a $1 \times 1$ convolution at all levels except the top [21].
- **NAFNet**'s code adds the encoder map to the upsampled decoder map [14].
- **DenseNet** concatenates the outputs of all earlier layers in a block [40] (DLCV 02 note).

### Multi-scale modules: FPN, ASPP and the U-Net

**Plain version.** Objects and degradations come at many sizes. Multi-scale modules let one network look at several scales at once: by building a pyramid of feature maps (FPN), by probing one map with filters of several sizes (ASPP), or by going down and back up (U-Net).

These are composition patterns more than blocks, and each is taught where it is used: FPN in DLCV 03 [5]; atrous spatial pyramid pooling (ASPP), parallel atrous convolutions at several rates, in DLCV 04 [41]; the U-Net in DLCV 01 and the IP 10 paper guide [39]. One rule connects them: **context comes from going coarse, detail comes from a path that stayed fine**. FPN's lateral connections, U-Net's skips and the low-level skip of the segmentation decoders in DLCV 03B are all that second path.

### Composing for classification

**Plain version.** A classifier must answer one question about the whole image. So it shrinks the image quickly, spends its computation on small, wide feature maps where each position already sees a large region, and finally throws away the spatial grid altogether.

The principles, with the reason for each:

1. **Downsample early with a stem.** The first layers run on the most pixels, so the stem cuts the resolution by 4 before the body starts (ResNet's $7 \times 7$ stride-2 convolution plus max-pool [1]; ConvNeXt's $4 \times 4$ stride-4 patchify layer, which cost nothing in accuracy in its roadmap [2]).

2. **Halve the resolution and double the width at each stage.** This keeps the cost per layer roughly constant (see the anatomy section) and grows the receptive field geometrically.
3. **Put most blocks in the third stage.** ResNet-50 uses (3, 4, 6, 3) blocks per stage; ConvNeXt changed this to (3, 3, 9, 3), following Swin's 1:1:3:1 ratio, and gained accuracy [2]. The third stage, at output stride 16, has a large receptive field and still enough positions.
4. **Pool globally, then classify linearly.** Global average pooling makes the output invariant to where the object is and lets the network accept any input size. The ViT paper found that a class token and global average pooling work similarly well, but need different learning rates [3].
5. **Put expensive global mixing where tokens are few.** Global attention is affordable in late stages; hybrid designs put convolutions early and attention late (ML 05). Cheap gates like SE can go in every block.
6. **Batch normalization is fine** when batches are large; use GN or LN when they are not.

**How fast does the receptive field grow?** The recurrence of ML 03 covers strided stacks; the snippet also handles upsampling by walking back from one output unit and tracking the interval of inputs that reach it. It compares ResNet-50 with two restorers on a 256 × 256 input: a plain full-resolution body of the EDSR-baseline shape (16 residual blocks of two 3 × 3 convolutions, 64 channels, plus head and tail) and a 4-level U-Net with widths 32 to 512.

```
def analyze(layers, H=256):
    """Exact 1-D receptive field of one output unit, plus the multiply-accumulates (MACs)
    of a square H x H input. A layer is ('conv', k, stride, c_in, c_out) or ('up', f),
    where 'up' is nearest-neighbour upsampling by f (the convolution after it pays the
    ↪  cost)."""
    macs, size = 0, H
    for layer in layers:                                # forward: track the map size and cost
        if layer[0] == 'conv':
            _, k, s, cin, cout = layer
            size //= s
            macs += size * size * k * k * cin * cout
        else:
            size *= layer[1]
    rf = 0
    for p0 in range(16):                                # a few output positions (alignment
    ↪  matters)
        lo = hi = p0
        for layer in reversed(layers):                  # backward: which inputs reach unit p0?
            if layer[0] == 'conv':
                _, k, s, _, _ = layer
                pad = (k - 1) // 2
                lo, hi = s * lo - pad, s * hi - pad + k - 1
            else:
                lo, hi = lo // layer[1], hi // layer[1]
        rf = max(rf, hi - lo + 1)
    return rf, macs, size

# Classification: ResNet-50 (stride on the 3x3 conv of each first block, max-pool as a k=3,
↪  s=2 layer).
resnet = [('conv', 7, 2, 3, 64), ('conv', 3, 2, 64, 0)]        # stem conv, then max-pool (no
↪  MACs)
cin = 64
```

```
for width, blocks, stride in [(64, 3, 1), (128, 4, 2), (256, 6, 2), (512, 3, 2)]:
    for b in range(blocks):
        s = stride if b == 0 else 1
        resnet += [('conv', 1, 1, cin, width), ('conv', 3, s, width, width),
                   ('conv', 1, 1, width, 4 * width)]
        if b == 0:
            resnet += [('conv', 1, 1, cin, 4 * width)]          # projection shortcut (cost
             ↪ only)
        cin = 4 * width

# Restoration A: plain full-resolution body, 64 channels, 32 convs of 3x3
 ↪ (EDSR-baseline-like).
plain = [('conv', 3, 1, 3, 64)] + [('conv', 3, 1, 64, 64)] * 32 + [('conv', 3, 1, 64, 3)]

# Restoration B: 4-level U-Net, 2 convs per level, widths double as resolution halves.
w = [32, 64, 128, 256, 512]
unet = [('conv', 3, 1, 3, w[0])]
for i in range(4):
    unet += [('conv', 3, 1, w[i], w[i])] * 2 + [('conv', 2, 2, w[i], w[i + 1])]
unet += [('conv', 3, 1, w[4], w[4])] * 2
for i in reversed(range(4)):
    unet += [('up', 2), ('conv', 1, 1, w[i + 1], w[i]),
             ('conv', 3, 1, 2 * w[i], w[i]), ('conv', 3, 1, w[i], w[i])]   # concat skip
              ↪ doubles c_in
unet += [('conv', 3, 1, w[0], 3)]

for name, net in [('ResNet-50', resnet), ('plain 64ch x 34', plain), ('U-Net 4 levels',
 ↪ unet)]:
    rf, macs, size = analyze(net)
    print(f"{name:16s} RF {rf:4d} px   output {size:3d}x{size:<3d}   {macs / 1e9:6.2f}
     ↪ GMACs")
```

```
ResNet-50        RF  427 px   output   8x8       5.34 GMACs
plain 64ch x 34  RF   69 px   output 256x256    77.54 GMACs
U-Net 4 levels   RF  206 px   output 256x256    14.47 GMACs
```

The U-Net's skips are left out because its deepest path sets the receptive field (a brute-force check over all paths gives the same 206). Figure 5 plots the growth. ResNet-50 reaches 427 pixels for 5.3 GMACs but returns an $8 \times 8$ map; the plain restorer grows by 2 pixels per layer, 69 pixels for 77.5 GMACs; the U-Net reaches 206 pixels at full output resolution for less than a fifth of that compute. These are theoretical receptive fields; the effective one, measured by gradients, is smaller and concentrated in the center [42] (DLCV 01).

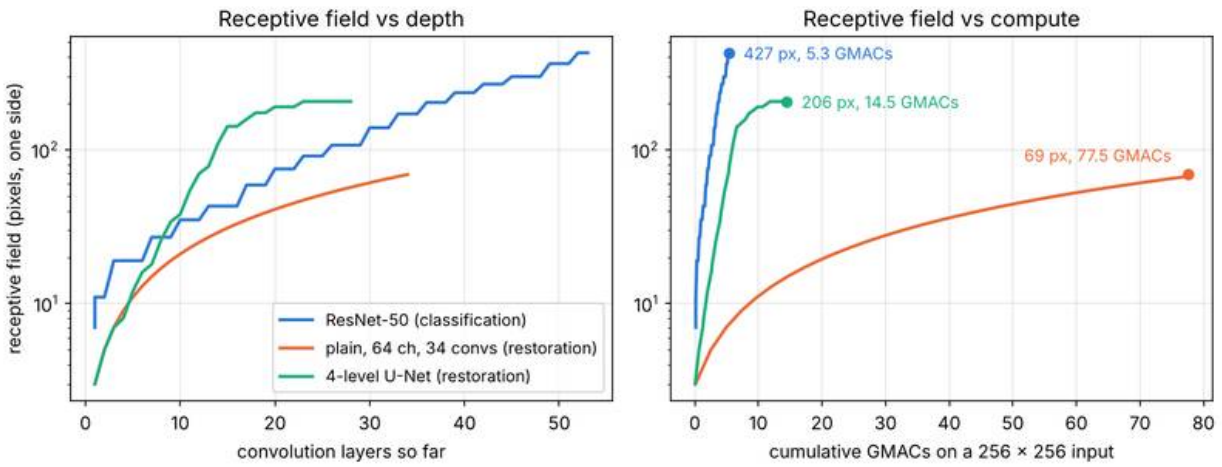



*Figure 5 — Receptive field (one side, theoretical) as a function of depth and of compute, for the three networks of the snippet. Downsampling buys receptive field cheaply; a full-resolution body pays for every pixel of it.*

*Online only: Receptive fields through upsampling and skips — https://blog.shyandram.dev/en/learn/deep-learning/building-blocks/notes/receptive-fields-through-upsampling-and-skips/*

## Composing for restoration

**Plain version.** A restoration network returns an image of the same size (or larger, for super-resolution) that is almost the input. So it keeps the resolution, or comes back to it with skips; it predicts only the correction; it avoids layers that change the intensity range; and it buys context without paying full resolution for every layer.

The principles:

1. **Keep the resolution, or come back to it.** The comparative study behind X-Restormer sorts restoration networks into three architectures: U-shaped encoder–decoders (such as Restormer and NAFNet), plain residual-in-residual networks that keep the original size (RCAN, SwinIR) and multi-stage progressive networks, which process the image in several sub-networks one after another [22]. It describes the plain design as favorable for reconstructing high-frequency detail but expensive, and the U-shaped one as handling large inputs with a receptive field that grows quickly through downsampling [22]. Figure 1(b) and (c) show the two main shapes.
2. **Learn the residual.** The output is the input plus a predicted correction, $\hat{x} = y + R(y)$ for degraded input $y$ and network $R$. VDSR predicted the residual between the high-resolution image and its interpolated input [43]; DnCNN predicts the noise [29]; SwinIR uses $I_{\text{HQ}} = H(I_{\text{LQ}}) + I_{\text{LQ}}$ for tasks without upsampling [4]; Restormer outputs a residual image $R$ and returns $\hat{I} = I + R$ [21]; NAFNet's code adds the input to the last convolution's output [14]. The correction is small and centered on zero, so no capacity is spent copying the image.
3. **Add a long skip for low frequencies.** SwinIR sends its shallow features directly to the reconstruction module, $H_{\text{REC}}(F_0 + F_{\text{DF}})$, because shallow features mainly contain low frequencies and the deep features can then focus on the lost high frequencies [4]. EDSR and RCAN use

long skips around the body or around groups [12], [15].

4. **Preserve the intensity range.** No BN in the body (EDSR, ESRGAN), normalization only inside residual branches (NAFNet, Restormer, SwinIR), as shown in the normalization section [12], [28].
5. **Match the receptive field to the degradation, and pay for it wisely.** The X-Restormer study argues that super-resolution depends mostly on local information and strong spatial mapping, that U-shaped downsampling and upsampling help denoising, and that motion deblurring and dehazing need large-range or global information [22]. The receptive-field snippet shows the price: a full-resolution body grows its field by a few pixels per layer at full cost, downsampling grows it fast and cheaply, and channel attention or state-space scans reach the whole image at linear cost [21], [44].
6. **Design the output layer for the target.** The last layer is a plain convolution (SwinIR's reconstruction for denoising, Restormer's final convolution, NAFNet's ending layer [4], [21], [14]); a ReLU or sigmoid there would clip the correction. For super-resolution, the upsampler goes at the end so the body runs at low resolution (DLCV 09A).
7. **Plan for full-size test images.** Restorers train on crops and test on whole images. A U-Net needs sizes divisible by $2^{\text{levels}}$, which NAFNet's code handles by padding [14]. Global statistics, such as channel attention's pooling, cover a crop in training but the whole image at test time; NAFNet notes that this degrades performance and adopts a remedy from earlier work [19], and Restormer trains on progressively larger patches [21].

*Online only: A Comparative Study of Image Restoration Networks for General Backbone Network Design (X-Restormer)* — *https://blog.shyandram.dev/en/learn/deep-learning/building-blocks/papers/a-comparative-study-of-image-restoration-networks-for-general-backbone-network-design-x-restormer/*

### Side by side: what changed and why

**Plain version.** The fastest way to see composition at work is to put a classification model next to the restoration model derived from it and list the differences. Each difference has a reason from the two lists above.

### ResNet versus EDSR and NAFNet

| Component | ResNet [1] | EDSR [12] | NAFNet [19], [14] | Why it changed |
|---|---|---|---|---|
| Stem | 7 × 7 stride 2 + max-pool | one 3 × 3 conv | one 3 × 3 conv | keep every pixel |
| Resolution | 4 stages, output stride 32 | constant (low-resolution input size) | U-shape, 2×2 stride-2 conv down, 1 × 1 conv + pixel shuffle up | detail versus context |
| Block | conv-BN-ReLU-conv-BN, add, ReLU | conv-ReLU-conv, ×0.1 (large model), add | LN, 1 × 1, 3 × 3 dw, SimpleGate, SCA, 1 × 1, ×β, add; then a gated MLP, ×γ, add | range preservation; cheap gating |

| Component | ResNet [1] | EDSR [12] | NAFNet [19], [14] | Why it changed |
|---|---|---|---|---|
| Normalization | BN | none | LayerNorm inside the branches | see the normalization section |
| Skips | short only | short + long skip around the body | short + add-skips between levels + global residual | low frequencies bypass the body |
| Head | global average pool + FC | upsampler (conv + pixel shuffle) + conv | $3 \times 3$ conv + input | output is an image |
| Loss | cross-entropy | L1 | PSNR loss in the ablations | regression targets |

EDSR's own table lists the step from SRResNet: 16 blocks of 64 filters for both baseline models, BN removed, L1 instead of L2, and for the full model 32 blocks of 256 filters with residual scaling 0.1, growing from 1.5M to 43M parameters [12].

### Swin versus SwinIR

| Component | Swin [6] | SwinIR [4] | Why it changed |
|---|---|---|---|
| Input | $4 \times 4$ patch partition and linear embedding | $3 \times 3$ conv (shallow features) at full resolution | one token per pixel |
| Hierarchy | patch merging between 4 stages | none: 6 residual Swin Transformer blocks (RSTB) at one resolution | restoration needs every pixel |
| Window | $M = 7$ | $M = 8$ (7 for JPEG artifact removal) | JPEG [45] codes $8 \times 8$ blocks; the authors observed a drop with 8 |
| Block group | Swin layers | 6 Swin layers + a $3 \times 3$ conv + a residual connection per RSTB | the conv adds translation equivariance; the residual helps aggregation |
| Head | global average pool + linear | conv (+ pixel shuffle for SR) on $F_0 + F_{\mathrm{DF}}$ | low frequencies bypass the body |
| Width | 96 channels and up | 180 channels, 6 heads | balance accuracy and size |

SwinIR's ablation reports that the residual connection inside each RSTB improves PSNR [30] by 0.16 dB, and that a $1 \times 1$ convolution there brings little improvement, possibly because it cannot extract local neighboring information as a $3 \times 3$ convolution does [4]. The JPEG window size is a good example of a composition choice that interacts with the data: the authors suggest the drop with $M = 8$ may come from JPEG's $8 \times 8$ block partitions.

### ViT versus Restormer

| Component | ViT [3] | Restormer [21] | Why it changed |
|---|---|---|---|
| Tokens | 16 × 16 patches | every pixel, after a conv embedding | no information is thrown away |
| Token mixer | global self-attention over patches, $N \times N$ map | MDTA: channel attention, $C \times C$ map, with depthwise convs | linear cost in pixels, global reach |
| Channel mixer | MLP, 4× expansion, GELU | GDFN: gated, depthwise convs, expansion 2.66 | gating and local context |
| Shape | isotropic, one resolution | 4-level U-shape, pixel-unshuffle down, pixel-shuffle up, concat + 1 × 1 skips, refinement stage | multi-scale at manageable cost |
| Head | class token + MLP | conv to a residual image, added to the input | output is an image |
| Training | fixed resolution | progressive patch sizes, starting at 128 × 128 | learn from larger regions |

Restormer's configuration uses `[4, 6, 6, 8]` blocks, `[1, 2, 4, 8]` heads and `[48, 96, 192, 384]` channels from level 1 to level 4, plus 4 blocks in the refinement stage [21]. Both columns use LayerNorm in pre-norm residual branches: the Transformer block itself survived; what changed is the token mixer and the macro-architecture around it.

### Exploring a design space

**Plain version.** A design question ("does block X help?") has no answer in general; it has an answer for a given set of other choices, data and budget. So change one thing at a time, keep the budget fixed, look at more than one task or seed, and when you can, compare populations of models rather than single winners.

### Reading an ablation

NAFNet's paper is a model of a readable ablation. Starting from a plain block of convolution, ReLU and shortcut in a U-Net, under a fixed budget of 16 GMACs, the authors add one change at a time and report PSNR on SIDD [31] (real-camera denoising) and GoPro [32] (motion deblurring) [19]:

| Step | SIDD (dB) | GoPro (dB) |
|---|---|---|
| PlainNet | 39.29 | 28.51 |
| + LayerNorm (and a 10× larger learning rate) | 39.73 | 31.90 |
| ReLU → GELU | 39.71 | 32.11 |
| + channel attention | 39.85 | 32.35 |
| GELU → SimpleGate | 39.93 | 32.76 |
| CA → simplified CA | 39.96 | 32.85 |

What to check when you read such a table:

- **Is the budget fixed?** Here it is (16 GMACs), so a gain is not just "more compute".
- **Are the steps cumulative?** Yes: each row includes the previous changes, so the table measures each change **given** the earlier ones, not in isolation. A different order could give different increments.
- **Do the tasks agree?** Not always: GELU is neutral on SIDD and helps on GoPro. A change that matters for one degradation may not matter for another.
- **Is the training recipe confounded with the block?** The LayerNorm row also changes the learning rate, because LayerNorm made the larger rate stable. The authors say so explicitly; many papers do not.
- **How large is the noise?** A single run per row gives no error bar. Differences of a few hundredths of a dB should be read as "no loss", which is how the paper reads the last row.

*Online only: Simple Baselines for Image Restoration (NAFNet) — https://blog.shyandram.dev/en/learn/deep-learning/building-blocks/papers/simple-baselines-for-image-restoration-nafnet/*

ConvNeXt's roadmap reads the same way for classification: a ResNet-50 went from 76.1% to 78.8% ImageNet top-1 with a modern training recipe alone, and then to 82.0% through a sequence of architectural changes, the largest single one adding less than the recipe [2]. That is why architectures trained with different recipes cannot be compared (DLCV 01A).

*Online only: The ConvNeXt roadmap step by step — https://blog.shyandram.dev/en/learn/deep-learning/building-blocks/notes/the-convnext-roadmap-step-by-step/*

### Comparing populations: design spaces

RegNet replaces "which single model is best?" with "which **design space** produces good models more often?" [16]. A design space is a set of models defined by a few parameters (number of blocks per stage, widths, bottleneck ratio, group width). The tool is the **error empirical distribution function**:

$$F(e) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}[e_i < e],$$

where $e_i$ is the error of the $i$-th of $n$ models sampled from the space and trained, $\mathbf{1}[\cdot]$ is 1 when its argument is true and 0 otherwise, and $F(e)$ is the fraction of models with error below $e$. A space whose curve lies higher is better as a whole, a far more robust comparison than two hand-tuned models. RegNet sampled $n = 500$ models at 360 to 400 million FLOPs, trained each for 10 epochs, and kept constraints that did not hurt the curve (shared bottleneck ratio and group width) or improved it (widths and depths increasing across stages) [16]. The best models had about 20 blocks across compute regimes, used no bottleneck, and benefited from SE [16].

*Online only: Designing Network Design Spaces (RegNet) — https://blog.shyandram.dev/en/learn/deep-learning/building-blocks/papers/designing-network-design-spaces-regnet/*

*Online only: RegNet's quantized linear widths — https://blog.shyandram.dev/en/learn/deep-learning/building-blocks/notes/regnets-quantized-linear-widths/*

### A practical protocol

1. Fix the budget (MACs, parameters, memory or measured latency) and the training recipe before comparing blocks.
2. Change one thing at a time, and keep a log of every run, including the ones that did not help.
3. Evaluate on more than one task, dataset or seed; report the spread.
4. Confirm at the target model size; a block that helps a tiny model may not help a large one.
5. For architecture searches, think in terms of the three dimensions of the NAS survey: the search space, the search strategy and how performance is estimated [46]. Cheap estimates such as short training (RegNet's 10 epochs) are fine for ranking populations but should be confirmed with full training.

## Common composition mistakes

**Plain version.** Most architecture bugs are not wrong blocks; they are right blocks in the wrong place. The list gives the symptom, the cause and the fix.

- **BN in a restoration body.** Symptom: batch-dependent brightness or color shifts, or test-time artifacts. Fix: remove BN, or normalize per sample inside residual branches with a global residual.
- **Small per-device batches with BN** (detection, segmentation, high resolution). Fix: GroupNorm, or freeze the pretrained BN statistics (DLCV 01A).
- **Transposed convolution with a kernel size not divisible by the stride.** Symptom: checkerboard texture. Fix: resize-convolution, or pixel shuffle with a matched initialization.
- **No global residual in a restorer.** The network must learn to copy the image before it can correct it. Fix: predict the correction.
- **A ReLU or sigmoid at the output.** Corrections cannot be negative, or highlights saturate. Fix: a linear last layer; clip only for display and evaluation.
- **A receptive field smaller than the degradation.** Symptom: large blur or haze remains. Fix: downsampling levels, dilation, or global mixing; compute the receptive field first.
- **Downsampling too early in a restorer**, or a classification backbone used unchanged: fine detail is lost. Fix: a full-resolution path (skips, a refinement stage).
- **Global attention at full resolution** runs out of memory. Use windows, channel attention or downsampling.
- **Input sizes not divisible by $2^{\text{levels}}$ in a U-Net**, or odd sizes with concatenated skips. Symptom: shape errors or misaligned skips. Fix: pad and crop, as NAFNet's code does [14].
- **Counting parameters as speed.** Depthwise and concatenation-heavy designs can be slow despite few parameters or MACs. Measure latency and memory.
- **Comparing designs trained with different recipes or budgets**: the recipe alone can be worth more than any single block [2].
- **Ignoring interactions with the data**, such as SwinIR's window size and JPEG's blocks [4].

## Modern view

### What the surveys say

**Khan et al.** [47] review CNN architectures and sort their innovations into seven categories: spatial exploitation, depth, multi-path, width, feature-map exploitation, channel boosting and attention. The taxonomy is useful here because most categories are composition choices (depth, width, multi-path connections) rather than new layer types.

**Guo et al.** [20] survey attention in computer vision and classify it by what it selects: channel attention (what to attend, e.g. SE), spatial attention (where, including self-attention), temporal attention (when), branch attention (which branch), and the two hybrids channel-and-spatial (e.g. CBAM) and spatial-and-temporal. Their framing of attention as "dynamic weight adjustment" based on the input matches the gate view of this chapter.

**Khan et al.** [48] review Transformers in vision, from the basics of self-attention and large-scale pretraining to applications in recognition, detection, segmentation, generation, video and low-level vision, compare the advantages and limitations of popular designs, and end with open research directions.

**Elsken, Metzen and Hutter** [46] organize neural architecture search along three dimensions: the search space, the search strategy and the performance-estimation strategy. Their survey is the natural next step after the manual design-space exploration of this chapter.

**Chen et al.** [22] is a comparative study rather than a survey: it benchmarks five restoration backbones, among them SwinIR, Restormer and NAFNet, on five tasks, classifies them by architecture (U-shaped, plain residual-in-residual, multi-stage) and core operator (convolution, spatial self-attention, transposed self-attention), and finds that networks strong on one task often fall short on another.

### The state of the art, 2023–2026

- **Convolutions and attention converge on one block shape.** ConvNeXt V2 adds a Global Response Normalization layer and masked-autoencoder pretraining to ConvNeXt [49]; the MetaFormer view treats the token mixer as one interchangeable part [8].
- **Pretraining matters as much as the block.** Battle of the Backbones compared many pretrained backbones across tasks in more than 1,500 training runs and found that convolutional networks pretrained with supervision on large datasets still performed best on most of the tasks it considered, while self-supervised backbones were competitive in matched comparisons [50].
- **Restoration mixes operators.** HAT combines channel attention with window self-attention and adds overlapping cross-attention between windows [51]; X-Restormer replaces half of Restormer's channel-attention blocks with spatial attention blocks to improve its spatial mapping [22]; MambaIR and MambaIRv2 replace attention with state-space scans that reach the whole image at linear cost, adding local enhancement and channel attention, and in v2 non-causal "attentive" modeling [44], [52].

### Open problems

1. **General restoration backbones.** No single composition is best on all degradations; the comparative study frames task generality as the open target [22].
2. **Fair comparisons.** Architecture, recipe and pretraining are entangled, and population-level studies like RegNet's need hundreds of training runs [16], which is costly for high-resolution restoration models.
3. **Receptive field in practice.** Theoretical fields are easy to compute; how much context a trained restorer actually uses is harder to measure, and attribution analyses suggest it can be less than the design allows [51], [42].
4. **Cost models.** MACs, memory traffic and latency disagree, especially for depthwise convolutions, gating and attention; design rules still depend on hardware.
5. **Train–test shift inside the architecture.** Global statistics (pooling, normalization, attention) behave differently on crops and full images, and the remedies in use change the training or test procedure rather than the architecture [19], [21].

*Online only: Simple Baselines for Image Restoration (NAFNet) — https://blog.shyandram.dev/en/learn/deep-learning/building-blocks/papers/simple-baselines-for-image-restoration-nafnet/*

### Key takeaways

- Track resolution changes and skip paths before comparing individual blocks.
- Compare spatial/channel mixing at matched cost; measure latency separately from parameters and MACs.
- Classifiers commonly downsample and pool; restorers must retain or reconstruct spatial detail and intensity.
- Residual learning and long skips preserve a direct path for the input; check the output layer's allowed range.
- Batch-statistic normalization introduces batch dependence; per-sample normalization can still remove useful intensity information.
- Check upsampling phase balance, skip alignment and padding before blaming the training data for artifacts.
- Separate block effects from recipe changes; compare design families as well as individual winners.

### Exercises

1. **Halve and double.** A stage runs $3 \times 3$ convolutions with 128 channels on a $56 \times 56$ map. (a) How many MACs does one such layer cost? (b) The next stage halves the map and doubles the width; what does one layer cost now? (c) What happens to the cost if the width is tripled instead?

#### Hint

(a) $9 \cdot 56^2 \cdot 128^2 = 462{,}422{,}016$ MACs. (b) $9 \cdot 28^2 \cdot 256^2$ is the same number. (c) With $3 \cdot 128 = 384$ channels the cost is $9 \cdot 28^2 \cdot 384^2$, that is 9/4 of (a): width enters squared, area linearly.

2. **Block budget.** Using the `blocks` function, find the expansion factor $t$ at which an inverted residual with $C$ input and output channels costs the same as a basic residual block, for $C = 64$. Then find the width $C'$ at which an inverted residual with $t = 6$ costs the same as a basic block with $C = 64$. What does this say about comparing blocks "at equal width"?

   **Hint**

   Set $t(2C^2 + 9C) = 18C^2$; for $C = 64$ this gives $t = 18 \cdot 64/(2 \cdot 64 + 9) \approx 8.4$. For the second question solve $12C'^2 + 54C' = 73{,}728$, which gives $C' \approx 76$. The inverted residual is not "cheap" at equal width; it is cheap because MobileNets run it on thin tensors.

3. **BN with a batch of one.** In the normalization snippet, restore $A$ alone (a batch of size 1) with `'bn-train'`. Compare the output mean with the `'instance'` result and explain the relation between BN at batch size 1 and instance normalization. Then add 32 random mid-gray images to the batch and see how close training mode comes to evaluation mode.

   **Hint**

   With $N = 1$, BN's statistics over $(N, H, W)$ are exactly the per-sample, per-channel statistics, so the result equals instance normalization (0.521). With many images from the training distribution, the batch statistics approach the running statistics and the output approaches the evaluation-mode value.

4. **Make a transposed convolution flat.** For $k = 4$, $s = 2$, write the condition on the 16 kernel weights under which a flat input gives a flat output (away from the borders). Construct such a kernel from a random $2 \times 2$ kernel, check it with the upsampling snippet, and relate it to resize-convolution.

   **Hint**

   The four phase sums $\sum_{m,n} w[a + 2m, b + 2n]$ for $a, b \in \{0, 1\}$ must be equal. One construction is `w = np.kron(v, np.ones((2, 2)))` for any $2 \times 2$ kernel `v`: every phase then receives each entry of `v` exactly once, so all phase sums equal the sum of `v`. Resize-convolution has the same property for every kernel, which is why it is flat.

5. **Receptive field of a design.** Modify the U-Net in the receptive-field snippet to three levels instead of four, and to four levels with three convolutions per level. Which change buys more receptive field per GMAC? Then replace the 32 middle layers of the plain network by 16 layers with dilation 2 (extend `analyze` with a dilation entry). What is the new receptive field, and what problem from DLCV 04 does this design risk?

   **Hint**

   A dilated layer with kernel $k$ and dilation $d$ spans $d(k-1)+1$ inputs, so replace `k - 1` by `d * (k - 1)` in the backward step. Removing a level removes the coarsest, cheapest and most receptive-field-rich part of the U-Net; adding convolutions per level adds cost at every resolution. Stacking layers with the same dilation leaves holes in the receptive field (gridding).

## References


1. K. He, X. Zhang, S. Ren and J. Sun, “Deep Residual Learning for Image Recognition,” in *Proc. IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2016. CVF ·arXiv
2. Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell and S. Xie, “A ConvNet for the 2020s,” in *Proc. IEEE/CVF CVPR*, 2022. CVF ·arXiv
3. A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer et al., “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale,” in *International Conference on Learning Representations* (*ICLR*), 2021. arXiv
4. J. Liang, J. Cao, G. Sun, K. Zhang, L. Van Gool and R. Timofte, “SwinIR: Image Restoration Using Swin Transformer,” in *Proc. IEEE/CVF International Conference on Computer Vision Workshops* (*ICCVW*), 2021. CVF ·arXiv
5. T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan and S. Belongie, “Feature Pyramid Networks for Object Detection,” in *Proc. IEEE CVPR*, 2017. CVF ·arXiv
6. Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin and B. Guo, “Swin Transformer: Hierarchical Vision Transformer Using Shifted Windows,” in *Proc. IEEE/CVF International Conference on Computer Vision* (*ICCV*), 2021. CVF ·arXiv
7. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy et al., “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015. doi
8. W. Yu, M. Luo, P. Zhou, C. Si, Y. Zhou, X. Wang, J. Feng and S. Yan, “MetaFormer Is Actually What You Need for Vision,” in *Proc. IEEE/CVF CVPR*, 2022. CVF ·arXiv
9. A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and H. Adam, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” arXiv:1704.04861, 2017. arXiv
10. M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” in *Proc. IEEE/CVF CVPR*, 2018. CVF ·arXiv
11. K. He, X. Zhang, S. Ren and J. Sun, “Identity Mappings in Deep Residual Networks,” in *Proc. European Conference on Computer Vision* (*ECCV*), LNCS, Springer, 2016. doi ·arXiv
12. B. Lim, S. Son, H. Kim, S. Nah and K. M. Lee, “Enhanced Deep Residual Networks for Single Image Super-Resolution,” in *Proc. IEEE CVPR Workshops* (*CVPRW*), 2017. CVF ·arXiv
13. H. Touvron, M. Cord, A. Sablayrolles, G. Synnaeve and H. Jégou, “Going Deeper With Image Transformers,” in *Proc. IEEE/CVF ICCV*, 2021. CVF
14. L. Chen, X. Chu, X. Zhang and J. Sun, NAFNet official implementation, `basicsr/models/archs/NAFNet_arch.py`, GitHub (megvii-research/NAFNet), accessed 2026. code
15. Y. Zhang, K. Li, K. Li, L. Wang, B. Zhong and Y. Fu, “Image Super-Resolution Using Very Deep Residual Channel Attention Networks,” in *Proc. ECCV*, 2018. CVF ·arXiv
16. I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He and P. Dollár, “Designing Network Design Spaces,” in *Proc. IEEE/CVF CVPR*, 2020. CVF ·arXiv
17. J. Hu, L. Shen and G. Sun, “Squeeze-and-Excitation Networks,” in *Proc. IEEE/CVF CVPR*, 2018. CVF ·arXiv

18. S. Woo, J. Park, J.-Y. Lee and I. S. Kweon, “CBAM: Convolutional Block Attention Module,” in *Proc. ECCV*, 2018. CVF ·arXiv

19. L. Chen, X. Chu, X. Zhang and J. Sun, “Simple Baselines for Image Restoration,” in *Proc. ECCV*, LNCS, Springer, 2022. doi ·arXiv

20. M.-H. Guo, T.-X. Xu, J.-J. Liu, Z.-N. Liu, P.-T. Jiang, T.-J. Mu, S.-H. Zhang, R. R. Martin, M.-M. Cheng and S.-M. Hu, “Attention Mechanisms in Computer Vision: A Survey,” *Computational Visual Media*, vol. 8, no. 3, pp. 331–368, 2022. doi ·arXiv

21. S. W. Zamir, A. Arora, S. Khan, M. Hayat, F. S. Khan and M.-H. Yang, “Restormer: Efficient Transformer for High-Resolution Image Restoration,” in *Proc. IEEE/CVF CVPR*, 2022. CVF · arXiv

22. X. Chen, Z. Li, Y. Pu, Y. Liu, J. Zhou, Y. Qiao and C. Dong, “A Comparative Study of Image Restoration Networks for General Backbone Network Design,” in *Proc. ECCV 2024*, LNCS, Springer, 2025. doi ·arXiv

23. S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” in *Proc. International Conference on Machine Learning (ICML)*, PMLR 37, 2015. PMLR

24. J. L. Ba, J. R. Kiros and G. E. Hinton, “Layer Normalization,” arXiv:1607.06450, 2016. arXiv

25. D. Ulyanov, A. Vedaldi and V. Lempitsky, “Instance Normalization: The Missing Ingredient for Fast Stylization,” arXiv:1607.08022, 2016. arXiv

26. Y. Wu and K. He, “Group Normalization,” in *Proc. ECCV*, 2018. CVF ·arXiv

27. C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang and W. Shi, “Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network,” in *Proc. IEEE CVPR*, 2017. CVF

28. X. Wang, K. Yu, S. Wu, J. Gu, Y. Liu, C. Dong, Y. Qiao and C. C. Loy, “ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks,” in *Proc. ECCV Workshops*, 2018. CVF · arXiv

29. K. Zhang, W. Zuo, Y. Chen, D. Meng and L. Zhang, “Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising,” *IEEE Transactions on Image Processing*, vol. 26, no. 7, pp. 3142–3155, 2017. doi ·arXiv

30. Q. Huynh-Thu and M. Ghanbari, “Scope of Validity of PSNR in Image/Video Quality Assessment,” *Electronics Letters*, vol. 44, no. 13, pp. 800–801, 2008. doi

31. A. Abdelhamed, S. Lin and M. S. Brown, “A High-Quality Denoising Dataset for Smartphone Cameras,” in *Proc. IEEE/CVF CVPR*, 2018. CVF

32. S. Nah, T. H. Kim and K. M. Lee, “Deep Multi-Scale Convolutional Neural Network for Dynamic Scene Deblurring,” in *Proc. IEEE CVPR*, 2017. CVF

33. N. Shazeer, “GLU Variants Improve Transformer,” arXiv:2002.05202, 2020. arXiv

34. Y. N. Dauphin, A. Fan, M. Auli and D. Grangier, “Language Modeling with Gated Convolutional Networks,” in *Proc. ICML*, PMLR 70, 2017. PMLR

35. D. Hendrycks and K. Gimpel, “Gaussian Error Linear Units (GELUs),” arXiv:1606.08415, 2016. arXiv

36. W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert and Z. Wang, "Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network," in *Proc. IEEE CVPR*, 2016. CVF ·arXiv

37. A. Odena, V. Dumoulin and C. Olah, "Deconvolution and Checkerboard Artifacts," *Distill*, vol. 1, no. 10, 2016. Distill ·doi

38. A. Aitken, C. Ledig, L. Theis, J. Caballero, Z. Wang and W. Shi, "Checkerboard Artifact Free Sub-Pixel Convolution: A Note on Sub-Pixel Convolution, Resize Convolution and Convolution Resize," arXiv:1707.02937, 2017. arXiv

39. O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Proc. Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, LNCS, Springer, pp. 234–241, 2015. doi ·arXiv

40. G. Huang, Z. Liu, L. van der Maaten and K. Q. Weinberger, "Densely Connected Convolutional Networks," in *Proc. IEEE CVPR*, 2017. CVF

41. L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy and A. L. Yuille, "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 834–848, 2018. doi ·arXiv

42. W. Luo, Y. Li, R. Urtasun and R. Zemel, "Understanding the Effective Receptive Field in Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2016. NeurIPS ·arXiv

43. J. Kim, J. K. Lee and K. M. Lee, "Accurate Image Super-Resolution Using Very Deep Convolutional Networks," in *Proc. IEEE CVPR*, 2016. CVF ·arXiv

44. H. Guo, J. Li, T. Dai, Z. Ouyang, X. Ren and S.-T. Xia, "MambaIR: A Simple Baseline for Image Restoration with State-Space Model," in *Proc. ECCV 2024*, LNCS, Springer, 2025. doi ·arXiv

45. ITU-T, "Recommendation T.81: Information technology – Digital compression and coding of continuous-tone still images – Requirements and guidelines" (the JPEG standard), 1992. ITU

46. T. Elsken, J. H. Metzen and F. Hutter, "Neural Architecture Search: A Survey," *Journal of Machine Learning Research*, vol. 20, 2019. JMLR ·arXiv

47. A. Khan, A. Sohail, U. Zahoora and A. S. Qureshi, "A Survey of the Recent Architectures of Deep Convolutional Neural Networks," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5455–5516, 2020. doi ·arXiv

48. S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan and M. Shah, "Transformers in Vision: A Survey," *ACM Computing Surveys*, vol. 54, no. 10s, 2022. doi ·arXiv

49. S. Woo, S. Debnath, R. Hu, X. Chen, Z. Liu, I. S. Kweon and S. Xie, "ConvNeXt V2: Co-Designing and Scaling ConvNets With Masked Autoencoders," in *Proc. IEEE/CVF CVPR*, 2023. CVF ·arXiv

50. M. Goldblum, H. Souri, R. Ni, M. Shu, V. Prabhu, G. Somepalli, P. Chattopadhyay, M. Ibrahim et al., "Battle of the Backbones: A Large-Scale Comparison of Pretrained Models across Computer Vision Tasks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv

51. X. Chen, X. Wang, J. Zhou, Y. Qiao and C. Dong, "Activating More Pixels in Image Super-Resolution Transformer," in *Proc. IEEE/CVF CVPR*, 2023. CVF ·arXiv

52. H. Guo, Y. Guo, Y. Zha, Y. Zhang, W. Li, T. Dai, S.-T. Xia and Y. Li, "MambaIRv2: Attentive State Space Restoration," in *Proc. IEEE/CVF CVPR*, 2025. CVF ·arXiv

53. A. Araujo, W. Norris and J. Sim, "Computing Receptive Fields of Convolutional Neural Networks," *Distill*, vol. 4, no. 11, 2019. Distill ·doi

54. X. Ding, X. Zhang, N. Ma, J. Han, G. Ding and J. Sun, "RepVGG: Making VGG-style ConvNets Great Again," in *Proc. IEEE/CVF CVPR*, pp. 13733–13742, 2021. CVF paper

55. S. Zagoruyko and N. Komodakis, "DiracNets: Training Very Deep Neural Networks Without Skip-Connections," arXiv:1706.00388, 2017. arXiv

56. X. Ding, Y. Guo, G. Ding and J. Han, "ACNet: Strengthening the Kernel Skeletons for Powerful CNN via Asymmetric Convolution Blocks," in *Proc. IEEE/CVF ICCV*, 2019. CVF

57. H. Zhao, J. Shi, X. Qi, X. Wang and J. Jia, "Pyramid Scene Parsing Network," in *Proc. IEEE CVPR*, 2017. CVF

58. M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth and B. Schiele, "The Cityscapes Dataset for Semantic Urban Scene Understanding," in *Proc. IEEE CVPR*, 2016. CVF

59. E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan and Q. V. Le, "AutoAugment: Learning Augmentation Strategies From Data," in *Proc. IEEE/CVF CVPR*, 2019. CVF

60. H. Zhang, M. Cisse, Y. N. Dauphin and D. Lopez-Paz, "mixup: Beyond Empirical Risk Minimization," in *Proc. ICLR*, 2018. arXiv

# DLCV 01C • Model Compression and Efficient Inference

How to make a trained vision model smaller and faster without fooling yourself: what cost means and how to measure it, pruning, quantization, knowledge distillation, low-rank factorization, designing and searching for small architectures, and why a paper's speed-up is not your device's speed-up.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/model-compression/

## What you'll learn

- Distinguish model size, operations, memory traffic, latency and throughput.
- Compare unstructured, structured and hardware-supported sparse pruning.
- Choose quantization scales, calibration and PTQ or QAT.
- Explain distillation, low-rank factorization and efficient architecture design.
- Check task sensitivity and measure the exported model on its target device.

## The big picture

A trained network is usually bigger and slower than it needs to be for one particular job on one particular device. **Model compression** is the set of techniques that remove that slack after (or during) training, and **efficient inference** is the engineering that makes the smaller model actually run faster on real hardware. The two are different, and most disappointments in this area come from confusing them.

> **Explain like I'm five**
>
> Imagine moving from a big house into a small flat. You can throw away things you never use (pruning). You can fold your clothes more tightly so the same clothes take less space (quantization). You can ask a friend who already lives in a small flat to show you which few things really matter, and copy their choices (distillation). Or you can buy foldable furniture in the first place (designing a small network). But the removal company charges by the weight of the boxes and the number of trips, not by how neatly you wrote the list. In the same way, a computer cares about how many bytes it must carry and whether it has the right tools to open the boxes, not about how small your model looks on paper.

**Why it matters.** Most deployed vision models run under a budget (a phone camera, a 60-frames-per-second factory line, a battery-powered robot), and compression decides whether a model fits it. It also changes the model: a quantized or pruned network is a different function that can fail where the original did not. Hence one rule for the whole chapter: **measure the cost on the target device and the accuracy after compression, on deployment data**.

**Where the basics are.** This chapter builds on material taught elsewhere on the site and links to it instead of repeating it: counting FLOPs, memory accounting, the latency protocol, the affine quantization formula and export runtimes in ML 11; the logit-distillation loss with temperature, with a tested gradient check, in the DLCV 01 Supplementary notes; depthwise-separable convolutions and block costs in DLCV 02 and DLCV 01B; speed measurement for detectors in DLCV

03A. All code here uses NumPy and scikit-learn on a CPU, runs in seconds, and uses scikit-learn's bundled digits dataset or synthetic data. **Every accuracy number produced by our code is a toy result** on an $8 \times 8$-pixel digit task; it illustrates a mechanism, not the size of the effect on a real vision model.

The table is a map of the chapter.

| Technique | What it shrinks | What it needs | Where a speed-up can come from |
|---|---|---|---|
| Unstructured pruning | number of nonzero weights | fine-tuning; sparse kernels to run fast | sparse kernels, or compressed storage when memory-bound |
| Structured pruning | channels, heads, blocks | fine-tuning | smaller dense tensors: works on any runtime |
| Quantization | bits per weight and activation | calibration data, or quantization-aware training | integer or low-precision kernels; fewer bytes moved |
| Distillation | the architecture (a smaller student) | a teacher and a training run | the student is smaller by design |
| Low-rank factorization | one big layer into two thin ones | fine-tuning, usually | fewer MACs, if the thin layers run efficiently |
| Efficient design and search | the architecture itself | a search or design budget | blocks chosen to suit the hardware |

### What "cost" means

**Plain version.** Choose the constrained resource before compressing: storage, runtime memory, latency or throughput. Fewer parameters or operations do not guarantee faster inference.

### Six numbers that are not the same

Six numbers describe what running a network costs:

- **Parameters** $P$: the number of learned values. With $b$ bits each, the weights take $P\,b/8$ bytes on disk and in memory. Pruning and quantization reduce this.
- **Multiply–accumulates (MACs)** or **FLOPs**: the arithmetic per image. One MAC is one multiplication plus one addition, usually counted as 2 FLOPs; libraries disagree, so state which you report. A $k \times k$ convolution from $C_{\text{in}}$ to $C_{\text{out}}$ channels on an $H' \times W'$ output costs $k^2 C_{\text{in}} C_{\text{out}} H' W'$ MACs (ML 11).
- **Memory traffic**: the bytes moved between main memory and the arithmetic units. Weights, input activations and output activations all have to be read or written at least once.
- **Peak memory**: the largest amount of memory alive at one time during a forward pass (weights plus the activations still needed). For high-resolution restoration networks it can be dominated by activations rather than weights.
- **Latency**: wall-clock time for one input, end to end, on the target device. This is what a camera pipeline or a user feels.

- **Throughput**: images per second when many are processed together. Large batches raise throughput but not single-image latency.

Dehghani et al. showed that these cost indicators can disagree and that reporting only one of them can lead to wrong conclusions about which model is more efficient [1]. Compression makes the disagreement worse, because each technique attacks a different number: pruning attacks parameters and MACs, quantization attacks bytes, and neither automatically attacks latency.

### Arithmetic intensity and the roofline

**Plain version.** A layer can be limited by arithmetic or memory access. Reducing computation helps little when moving data is the bottleneck.

**Precise version.** The **roofline model** of Williams, Waterman and Patterson [2] bounds the attainable speed of a computation by

$$\text{attainable speed} = \min(\pi,\ \beta \cdot I), \qquad I = \frac{\text{operations}}{\text{bytes moved}},$$

where $\pi$ is the peak arithmetic rate of the device (here in MACs per second), $\beta$ its memory bandwidth (bytes per second) and $I$ the **arithmetic intensity** of the computation (MACs per byte). The intensity at which the two limits meet, $I^* = \pi/\beta$, is the **ridge point**: below it the layer is **memory-bound**, above it **compute-bound**. A simple time estimate for one layer is

$$t \approx \max\left(\frac{\text{MACs}}{\pi},\ \frac{\text{bytes}}{\beta}\right).$$

The snippet applies this to a few layers on a **made-up** accelerator (4 TMAC/s, 100 GB/s; real devices differ), assuming every weight, input and output value crosses memory exactly once. Real kernels reuse caches and overlap transfers, so treat the numbers as an idealized sketch.

```
# A made-up accelerator: 4 TMAC/s peak compute, 100 GB/s memory bandwidth.
PEAK_MACS, BANDWIDTH = 4e12, 100e9

def layer(kind, C=64, H=56, W=56, batch=1):
    """MACs and bytes moved, if weights, inputs and outputs each cross memory once."""
    if kind == "conv 3x3":
        weights, macs = 9 * C * C, 9 * C * C * H * W
    elif kind == "depthwise 3x3":
        weights, macs = 9 * C, 9 * C * H * W
    elif kind == "pointwise 1x1":
        weights, macs = C * C, C * C * H * W
    elif kind == "linear 2048x2048":
        weights, macs, C, H, W = 2048 * 2048, 2048 * 2048, 2048, 1, 1
    acts = 2 * C * H * W                                  # one input and one output tensor
    return batch * macs, weights + batch * acts

for kind, batch, nbytes in [("conv 3x3", 1, 4), ("depthwise 3x3", 1, 4), ("pointwise 1x1", 1,
 ↪  4),
```

```
                          ("depthwise 3x3", 1, 1), ("linear 2048x2048", 1, 4), ("linear
                           ↪  2048x2048", 64, 4)]:
    macs, values = layer(kind, batch=batch)
    traffic = values * nbytes
    t_compute, t_memory = macs / PEAK_MACS, traffic / BANDWIDTH
    bound = "compute" if t_compute > t_memory else "memory"
    print(f"{kind:17s} batch {batch:2d} {8 * nbytes:2d}-bit: {macs / 1e6:7.1f} M MACs, "
          f"{macs / traffic:5.1f} MACs/byte, ~{max(t_compute, t_memory) * 1e6:6.1f} us
           ↪  ({bound}-bound)")
print(f"ridge point: {PEAK_MACS / BANDWIDTH:.0f} MACs/byte")
```

Output:

```
conv 3x3          batch  1 32-bit:   115.6 M MACs,  65.9 MACs/byte, ~  28.9 us
 ↪  (compute-bound)
depthwise 3x3     batch  1 32-bit:     1.8 M MACs,   1.1 MACs/byte, ~  16.1 us (memory-bound)
pointwise 1x1     batch  1 32-bit:    12.8 M MACs,   7.9 MACs/byte, ~  16.2 us (memory-bound)
depthwise 3x3     batch  1  8-bit:     1.8 M MACs,   4.5 MACs/byte, ~   4.0 us (memory-bound)
linear 2048x2048  batch  1 32-bit:     4.2 M MACs,   0.2 MACs/byte, ~ 167.9 us (memory-bound)
linear 2048x2048  batch 64 32-bit:   268.4 M MACs,  15.1 MACs/byte, ~ 178.3 us (memory-bound)
ridge point: 40 MACs/byte
```

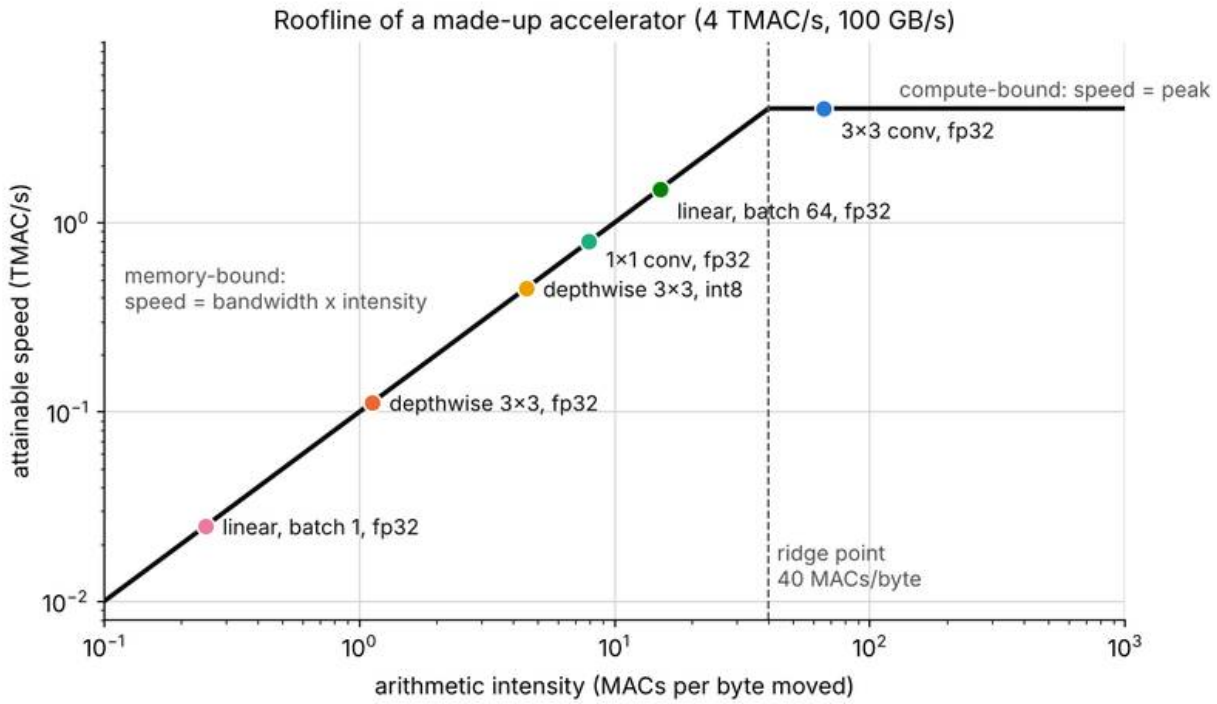



*Figure 1 — The roofline of a made-up accelerator with the layers from the snippet. Everything left of the ridge point is limited by memory traffic, so cutting its arithmetic barely helps; cutting its bytes (fewer bits, larger batches) moves it up the slope.*

Three lessons recur later. First, the depthwise layer has 64 times fewer MACs than the dense $3 \times 3$ convolution but is estimated to be only about 1.8 times faster, because it is memory-bound (the arithmetic behind the warning in DLCV 01B that MACs are not latency). Second, storing that layer in 8 bits instead of 32 cuts its estimated time by 4 without changing a single MAC: for memory-bound layers, quantization helps by moving fewer bytes. Third, the batch-64 linear layer does 64 times the work of the batch-1 layer in about the same time, because its weights are read once and reused, so throughput at a large batch says little about latency at batch 1.

### Measuring honestly

Follow the latency protocol in ML 11 (fixed and reported device, runtime, precision, batch size, resolution and threads; warm-up; percentiles; synchronization on GPUs) and the end-to-end checklist in DLCV 03A. Compression adds three rules of its own:

1. **Compare against a fair dense baseline**: a dense model of the same size, trained with the same budget. Liu et al. found that, for the structured pruning methods they examined, training the pruned architecture from scratch matched or beat fine-tuning the pruned weights [3]; Blalock et al. reviewed 81 pruning papers and found that inconsistent baselines and metrics made most comparisons impossible [4].
2. **Measure accuracy after export, on the runtime you ship.** The exported, quantized, fused graph is a different numerical program from the one you trained.
3. **Report a trade-off curve, not one point.** Accuracy against latency (and memory) for several compression levels shows whether a method moves the whole curve or just picks a different point on it.

### Pruning

**Plain version.** Pruning removes selected weights or channels. Scattered zeros need sparse kernels for speed; removing channels yields smaller dense operations whose latency still needs testing.

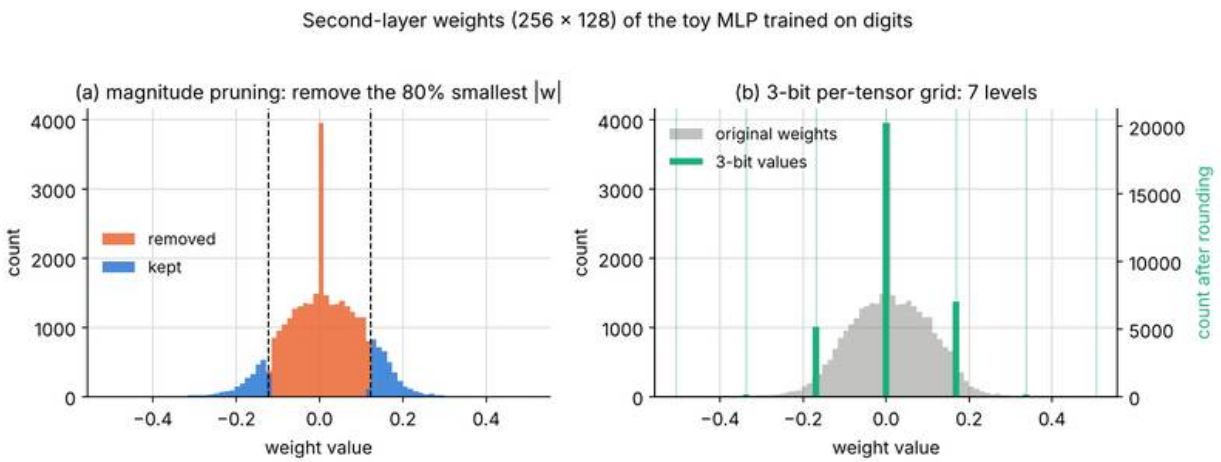


*Figure 2 — What two compressions do to one weight matrix of the toy MLP used throughout this chapter. (a) Magnitude pruning removes the middle of the distribution. (b) A 3-bit grid with one scale for the whole tensor: about 62% of the weights round to exactly zero and 99% land on the three central levels, so most of the grid is spent on a few large weights.*

### What gets removed

Write the parameters of a network as a vector $\boldsymbol{\theta} \in \mathbb{R}^N$. Pruning chooses a binary **mask** $\mathbf{m} \in \{0, 1\}^N$ and uses $\mathbf{m} \odot \boldsymbol{\theta}$ instead of $\boldsymbol{\theta}$, where $\odot$ is elementwise multiplication. The **sparsity** is the fraction of zeros,

$$s = 1 - \frac{\|\mathbf{m}\|_0}{N},$$

where $\|\mathbf{m}\|_0$ counts the nonzero entries of the mask. What differs between methods is the **granularity** of the mask:

- **Unstructured** (fine-grained): any single weight may be removed. This gives the most freedom in choosing what to remove, but the result is a dense-shaped tensor full of scattered zeros. Li et al. point out that such irregular sparsity does not reduce the cost of convolutional layers much unless sparse convolution libraries are available [5].
- **Structured** (coarse-grained): whole filters, channels, attention heads or blocks are removed. Removing output channel $j$ of a convolution also removes input channel $j$ of the next layer, so both tensors shrink and stay dense. Li et al. prune whole filters together with their connecting feature maps for exactly this reason: the result runs with ordinary dense matrix libraries [5].
- **Semi-structured**: a fixed small pattern that hardware can exploit. NVIDIA's sparse Tensor Cores support **2:4 sparsity**, in which every group of four consecutive weights has at least two zeros; the paper reports twice the math throughput of the dense units for this pattern [6].

*Online only: N:M semi-structured sparsity — https://blog.shyandram.dev/en/learn/deep-learning/model-compression/notes/nm-semi-structured-sparsity/*

### Which weights: magnitude and other criteria

The simplest and still most common criterion is **magnitude**: remove the weights with the smallest $|\theta_i|$. It can be applied per layer (remove the same fraction of each layer) or globally (one threshold for the whole network; layers with many small weights then lose more). For structured pruning, the magnitude of a group is used instead: Li et al. rank filters by the sum of absolute weights ($\ell_1$ norm) of each filter [5]. **Network slimming** adds an $\ell_1$ penalty on the scale factors $\gamma$ of the batch-normalization layers during training, which pushes the scales of unimportant channels towards zero, and then removes the channels with the smallest $|\gamma|$ [7]. Many other criteria exist (gradient-based, second-order, data-driven); Hoefler et al. survey them [8].

Structured pruning has one trap that unstructured pruning does not: **coupled channels**, such as the channels on both sides of a residual addition, which must be removed together. Fang et al.'s DepGraph finds such groups automatically from the computational graph [9].

*Online only: Pruning coupled channels — https://blog.shyandram.dev/en/learn/deep-learning/model-compression/notes/pruning-coupled-channels/*

### Prune, fine-tune, repeat

Removing weights changes the function, so pruning is followed by training. Han et al. described the standard three-step recipe: train the network to learn which connections are important, prune the unimportant ones, and retrain the remaining weights; on ImageNet they reported reducing AlexNet's parameters from 61 million to 6.7 million and VGG-16's from 138 million to 10.3 million without accuracy loss [10]. Their Deep Compression pipeline added quantization and Huffman coding on top of pruning [11].

Pruning is often spread over several smaller steps, each followed by some training, instead of one large cut. Zhu and Gupta's **gradual** schedule raises the sparsity from zero to its target during training, quickly at first and more slowly near the end [12]. They also compared large-sparse models with small-dense models of the same memory footprint and found the large-sparse ones consistently more accurate across the architectures they tested [12].

*Online only: The gradual pruning schedule — https://blog.shyandram.dev/en/learn/deep-learning/model-compression/notes/the-gradual-pruning-schedule/*

### Code: unstructured and structured pruning of a small MLP

The snippet trains a 64–256–128–10 MLP on scikit-learn's digits, then prunes it in two ways: by removing the smallest weights in every layer, and by removing whole hidden units (the smallest $\ell_1$ norm of incoming weights, in both hidden layers). Each pruned model is fine-tuned for 10 epochs, re-applying the mask after every optimizer step. At the end, the most heavily pruned structured model is sliced into a genuinely smaller dense network.

```
import copy, warnings
import numpy as np
from sklearn.datasets import load_digits
from sklearn.exceptions import ConvergenceWarning
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
warnings.filterwarnings("ignore", category=ConvergenceWarning)

X, y = load_digits(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(X / 16.0, y, test_size=0.3, random_state=0, stratify=y)
base = MLPClassifier(hidden_layer_sizes=(256, 128), max_iter=300, random_state=0).fit(Xtr,
 ↪  ytr)
print(f"dense: {sum(w.size for w in base.coefs_)} weights, test accuracy {base.score(Xte,
 ↪  yte):.3f}")

def fine_tune(net, masks, epochs=10):
    order_rng = np.random.default_rng(0)
    for _ in range(epochs):
        order = order_rng.permutation(len(Xtr))
```

```
        for i in range(0, len(Xtr), 128):
            net.partial_fit(Xtr[order[i:i + 128]], ytr[order[i:i + 128]])   # one Adam step
            for w, m in zip(net.coefs_, masks):
                w *= m                                                      # keep pruned
                 ↪  weights at zero

def unstructured_masks(net, sparsity):          # smallest |w| in each layer -> 0
    return [np.abs(w) > np.quantile(np.abs(w), sparsity) for w in net.coefs_]

def structured_masks(net, keep):                # drop hidden units with the smallest L1 norm
 ↪  of incoming weights
    masks = [np.ones_like(w) for w in net.coefs_]
    for l in range(len(net.coefs_) - 1):
        norms = np.abs(net.coefs_[l]).sum(axis=0)
        dead = np.argsort(norms)[: int(round((1 - keep) * norms.size))]
        masks[l][:, dead] = 0                   # incoming weights of the unit ...
        masks[l + 1][dead, :] = 0               # ... and its outgoing weights
    return masks

for name, make, levels in [("remove weights", unstructured_masks, [0.8, 0.9, 0.95]),
                           ("keep units", structured_masks, [0.5, 0.25, 0.125])]:
    for level in levels:
        net = copy.deepcopy(base)
        masks = make(net, level)
        for w, m in zip(net.coefs_, masks):
            w *= m
        before = net.score(Xte, yte)
        fine_tune(net, masks)
        kept = int(sum(m.sum() for m in masks))
        print(f"{name} {level:6.1%}: {kept:5d} weights left, "
              f"accuracy {before:.3f} -> {net.score(Xte, yte):.3f} after fine-tuning")

# The last structured model is a genuinely smaller dense network: slice out the live units.
alive = [masks[l].any(axis=0) for l in range(2)]
small_W = [net.coefs_[0][:, alive[0]], net.coefs_[1][alive[0]][:, alive[1]],
 ↪  net.coefs_[2][alive[1]]]
small_b = [net.intercepts_[0][alive[0]], net.intercepts_[1][alive[1]], net.intercepts_[2]]
h = Xte
for l, (w, b) in enumerate(zip(small_W, small_b)):
    h = h @ w + b if l == 2 else np.maximum(h @ w + b, 0)
print("sliced shapes:", [w.shape for w in small_W],
      "| same predictions:", bool((h.argmax(1) == net.predict(Xte)).all()))
```

Output (toy results):

```
dense: 50432 weights, test accuracy 0.980
remove weights  80.0%: 10087 weights left, accuracy 0.909 -> 0.974 after fine-tuning
remove weights  90.0%:  5044 weights left, accuracy 0.563 -> 0.954 after fine-tuning
remove weights  95.0%:  2523 weights left, accuracy 0.170 -> 0.885 after fine-tuning
keep units  50.0%: 17024 weights left, accuracy 0.887 -> 0.981 after fine-tuning
keep units  25.0%:  6464 weights left, accuracy 0.550 -> 0.980 after fine-tuning
keep units  12.5%:  2720 weights left, accuracy 0.402 -> 0.957 after fine-tuning
sliced shapes: [(64, 32), (32, 16), (16, 10)] | same predictions: True
```

What to read from it. **Fine-tuning does most of the work**: removing 90% of the weights drops accuracy to 0.563, and ten epochs bring it back to 0.954. **Structured pruning produced a real network** of shape 64–32–16–10 that any library runs as ordinary dense matrices. And at about 2,500 remaining weights the small dense network (0.957) beat the large sparse one (0.885), the opposite of Zhu and Gupta's finding on large models [12]: our network is so over-parameterized for an $8 \times 8$ digit task that making it narrower costs almost nothing. Do not generalize from this toy in either direction.

### The lottery ticket hypothesis

**Plain version.** If a pruned network can be this small, why not train the small network directly? Frankle and Carbin found that, in their experiments, a randomly reinitialized pruned network trained worse, but the same sparse network **reset to its original initial weights** trained as well as the full one. They proposed that large networks contain such lucky sparse subnetworks ("winning tickets") from the start.

The tickets are found by **iterative magnitude pruning**: train, prune the smallest-magnitude weights, reset the survivors to their initial values, and repeat; on small MNIST and CIFAR-10 networks this consistently found tickets at 10–20% of the original size or less [13]. At ImageNet scale, the weights had to be reset to their values from early in training rather than to initialization [14], and Liu et al. reported that with a well-tuned learning rate the ticket initialization brought no gain over random initialization in their experiments [3]. The practical lesson is modest: the hypothesis is about which sparse networks are trainable, not a cheaper way to compress (finding a ticket costs several full trainings).

*Online only: The Lottery Ticket Hypothesis (Frankle and Carbin, ICLR 2019) — https://blog.shyandram.dev/en/learn/deep-learning/model-compression/papers/the-lottery-ticket-hypothesis-frankle-and-carbin-iclr-2019/*

## Quantization

**Plain version.** A network trained in 32-bit floating point does not need that precision to make good predictions. Quantization stores weights, and often activations, as small integers (8 bits, sometimes 4) together with a few scale factors. That cuts memory by 4 to 8 times and, on hardware with integer arithmetic, also speeds up computation. The difficulty is that a small grid of values has to cover both the many small numbers and the few very large ones.

### Uniform affine quantization

The most common scheme, because it permits efficient fixed-point arithmetic [15], maps a real value $x$ to an integer $q$ and back [16]:

$$q = \operatorname{clip}\Big(\operatorname{round}\Big(\frac{x}{s}\Big) + z,\ q_{\min},\ q_{\max}\Big), \qquad \hat{x} = s\,(q - z),$$

where $s > 0$ is the **scale** (the step between neighbouring representable values), $z$ the integer **zero point** (the integer that represents the real value 0 exactly), $[q_{\min}, q_{\max}]$ the integer range of a $b$-bit format (for example $[0, 255]$ for unsigned 8-bit or $[-127, 127]$ for a symmetric signed format), round rounding to the nearest integer, clip clamping into the range, and $\hat{x}$ the dequantized approximation of $x$. For a real range $[x_{\min}, x_{\max}]$ that contains 0, a common choice is

$$s = \frac{x_{\max} - x_{\min}}{q_{\max} - q_{\min}}, \qquad z = q_{\min} - \operatorname{round}\left(\frac{x_{\min}}{s}\right).$$

**Symmetric** quantization fixes $z = 0$ and uses a signed range. Exact representation of 0 matters because zero padding and ReLU outputs produce many exact zeros [15]. Two kinds of error arise: **rounding error**, at most $s/2$ for values inside the range, and **clipping error** for values outside it. A larger range reduces clipping but makes $s$, and therefore the rounding error, larger [15]. ML 11 has a tested implementation and the error-versus-bits curve.

### Why integer kernels can factor out the scales

**Plain version.** Integers are only useful if the expensive part of a layer, the long sum of products, can run entirely in integers. With one scale per weight row and one per activation tensor, the scales can be pulled out of the sum and applied once at the end.

For one output $y_i = \sum_j W_{ij} x_j$ of a linear layer (a convolution is the same sum over a patch), quantize the weights symmetrically with one scale $s_{w,i}$ per output channel, $W_{ij} \approx s_{w,i}\, q^w_{ij}$, and the inputs affinely with scale $s_x$ and zero point $z_x$, $x_j \approx s_x(q^x_j - z_x)$. Then

$$y_i \approx s_{w,i}\, s_x \sum_j q^w_{ij}\,(q^x_j - z_x),$$

where the sum involves only integers and is accumulated in a 32-bit integer register, and the single product $s_{w,i} s_x$ is applied once per output. This is why per-channel scales for weights are cheap (one factor per output channel), while per-channel scales for **activations** are not: they would sit inside the sum and require rescaling the accumulator for each input channel [15]. It is also why a common choice is symmetric weights and asymmetric activations: an asymmetric weight zero point would add a data-dependent term to every output [15]. Biases are typically kept as 32-bit integers [16].

```
import numpy as np

rng = np.random.default_rng(0)
W = rng.normal(0, 0.2, size=(16, 256))              # a layer: 16 outputs, 256 inputs
x = rng.normal(0.3, 1.0, size=256)                  # input activations (not centred at 0)

# Weights: symmetric int8, one scale per output channel (zero point 0).
s_w = np.abs(W).max(axis=1) / 127
W_q = np.round(W / s_w[:, None]).astype(np.int8)

# Activations: affine uint8 with a scale and an integer zero point.
lo, hi = min(x.min(), 0.0), x.max()
```

```
s_x = (hi - lo) / 255
z_x = int(np.round(-lo / s_x))
x_q = np.clip(np.round(x / s_x) + z_x, 0, 255).astype(np.uint8)
print("activation scale:", round(s_x, 5), "| zero point:", z_x)

# Integer-only inner products, accumulated in int32, then one rescale per output.
acc = W_q.astype(np.int32) @ (x_q.astype(np.int32) - z_x)
y_int = s_w * s_x * acc
y_ref = W @ x
print("max |integer result - float result|:", np.abs(y_int - y_ref).max().round(4))
print("relative error:", (np.linalg.norm(y_int - y_ref) / np.linalg.norm(y_ref)).round(4))
print("largest |accumulator|:", np.abs(acc).max(), "| int16 limit:", 2**15 - 1)
```

Output:

```
activation scale: 0.01967 | zero point: 117
max |integer result - float result|: 0.0495
relative error: 0.0083
largest |accumulator|: 83167 | int16 limit: 32767
```

The integer path reproduces the float result to within about 1%. The accumulator reaches 83,167, far beyond what 16 bits can hold, which is why accelerators keep 32-bit accumulators even when inputs are 8-bit [15]. Before the next layer, the 32-bit result is converted back to 8 bits ("requantized") with the next layer's scale.

*Online only: Requantization with a fixed-point multiplier — https://blog.shyandram.dev/en/learn/deep-learning/model-compression/notes/requantization-with-a-fixed-point-multiplier/*

### Granularity and range setting

**Granularity** is how many values share one scale: one per tensor, one per output channel, or one per small group of consecutive weights. Finer granularity fits the grid to each group's own range. **Range setting** (or calibration) is how $[x_{\min}, x_{\max}]$ is chosen [15]:

- **Min-max**: the observed minimum and maximum. No clipping, but one outlier stretches the grid for everyone.
- **Percentile**: ignore a small fraction of the most extreme values.
- **MSE**: search for the range that minimizes $\|x - \hat{x}\|^2$ on representative data, trading clipping against rounding.

Weights are known in advance and need no data. Activations depend on the input, so their ranges are estimated by running a small **calibration set** of unlabeled images through the network; Nagel et al. note that 500 to 1,000 unlabeled images are usually enough [15].

*Online only: Choosing the clipping range: min-max, percentile and MSE — https://blog.shyandram.dev/en/learn/deep-learning/model-compression/notes/choosing-the-clipping-range-min-max-percentile-and-mse/*

### What breaks: outliers, folded batch norm and activations

**Plain version.** Quantization fails when one shared scale must cover values of very different sizes. The large values force a coarse grid, and the many small values all round to the same few integers.

The classic vision example is MobileNetV2. Nagel et al. showed that the weight ranges of different output channels of one of its depthwise-separable layers differ so much that one set of quantization parameters cannot serve the whole tensor; with batch normalization folded and 8-bit per-tensor quantization, the ImageNet top-1 accuracy of their model fell from 71.72% to 0.12% [17]. Their fix needs no data or training: because $\mathrm{ReLU}(c\,t) = c\,\mathrm{ReLU}(t)$ for $c > 0$, an output channel of one layer can be scaled down and the matching input channel of the next layer scaled up without changing the function (**cross-layer equalization**). After replacing ReLU6 by ReLU, which has this property, equalization and bias absorption brought the model to 70.92% at 8 bits [17].

Our toy reproduces the mechanism. We take the trained MLP and rescale one hidden unit by 100 (and the next layer's weights by $1/100$). The float network computes exactly the same function, but its activation and weight ranges now contain one huge channel.

```
import warnings
import numpy as np
from sklearn.datasets import load_digits
from sklearn.exceptions import ConvergenceWarning
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
warnings.filterwarnings("ignore", category=ConvergenceWarning)

X, y = load_digits(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(X / 16.0, y, test_size=0.3, random_state=0, stratify=y)
net = MLPClassifier(hidden_layer_sizes=(256, 128), max_iter=300, random_state=0).fit(Xtr,
 ↪  ytr)

def q_weight(w, bits, per_channel):     # symmetric: one scale per output unit, or one per
 ↪  tensor
    qmax = 2 ** (bits - 1) - 1
    m = np.abs(w).max(axis=0, keepdims=True) if per_channel else np.abs(w).max()
    return np.clip(np.round(w * qmax / m), -qmax, qmax) * m / qmax

def q_act(a, hi, bits):                 # post-ReLU activations: grid on [0, hi], zero point
 ↪  0
    s = hi / (2 ** bits - 1)
    return np.clip(np.round(a / s), 0, 2 ** bits - 1) * s

def run(Ws, bs, X, wbits=None, abits=None, his=None, per_channel=True, record=None):
    h = X
    for l, (w, b) in enumerate(zip(Ws, bs)):
        h = h @ (q_weight(w, wbits, per_channel) if wbits else w) + b
        if l < len(Ws) - 1:
            h = np.maximum(h, 0)
            if record is not None:
                record.append(h)
            if abits:
```

```
                h = q_act(h, his[l], abits)
    return h

acc = lambda logits: (logits.argmax(1) == yte).mean()
for c in [1, 100]:
    Ws = [w.copy() for w in net.coefs_]
    bs = [b.copy() for b in net.intercepts_]
    Ws[0][:, 0] *= c; bs[0][0] *= c; Ws[1][0, :] /= c   # same function: ReLU(c*t) =
     ↪  c*ReLU(t)
    calib = []
    run(Ws, bs, Xtr[:100], record=calib)                 # 100 training images calibrate the
     ↪  ranges
    minmax = [a.max() for a in calib]
    pct99 = [np.percentile(a, 99) for a in calib]
    print(f"unit 0 scaled by {c}: float {acc(run(Ws, bs, Xte)):.3f} | "
          f"W8A8 {acc(run(Ws, bs, Xte, 8, 8, minmax)):.3f} | "
          f"W8A4 {acc(run(Ws, bs, Xte, 8, 4, minmax)):.3f} | "
          f"W8A4 clipped {acc(run(Ws, bs, Xte, 8, 4, pct99)):.3f} | "
          f"W4 per-tensor {acc(run(Ws, bs, Xte, 4, per_channel=False)):.3f} | "
          f"W4 per-channel {acc(run(Ws, bs, Xte, 4)):.3f}")
```

Output (toy results; "W8A4" means 8-bit weights and 4-bit activations):

```
unit 0 scaled by 1: float 0.980 | W8A8 0.980 | W8A4 0.978 | W8A4 clipped 0.978 | W4
 ↪  per-tensor 0.983 | W4 per-channel 0.980
unit 0 scaled by 100: float 0.980 | W8A8 0.980 | W8A4 0.163 | W8A4 clipped 0.980 | W4
 ↪  per-tensor 0.102 | W4 per-channel 0.978
```

The two networks are the same function, yet with one outlier channel, 4-bit min-max activations collapse to 0.163 and 4-bit per-tensor weights to 0.102. Clipping the activation range at the 99th percentile (0.980) and per-unit weight scales (0.978) both fix it. Here clipping happens to cost nothing; when the outlier channel carries real information it does, which is why methods that move or isolate outliers exist.

The same pattern, at larger scale, drives quantization research for Transformers: Dettmers et al. found systematic outlier feature dimensions in large language models and kept them in 16-bit [18], and SmoothQuant moves the difficulty from activations to weights with an equivalent per-channel rescaling, the same family of idea as cross-layer equalization [19].

### Post-training quantization and quantization-aware training

**Plain version.** You can quantize a finished model with a little calibration data (fast, no labels), or you can train the model while it "feels" the rounding, so that it learns weights that survive it (slower, more accurate at low bit widths).

**Post-training quantization (PTQ)** takes a trained float model and converts it using only a small calibration set. Nagel et al.'s recommended pipeline is: cross-layer equalization, symmetric weight and asymmetric activation quantizers (per-channel weights if the hardware supports it),

MSE-based weight ranges, then **AdaRound** (a learned choice between rounding each weight up or down [20]) if calibration data is available, analytical bias correction otherwise, and finally activation range setting [15].

**Quantization-aware training (QAT)** inserts "fake quantization" (quantize, then immediately dequantize) into the forward pass and trains as usual. The difficulty is that round has zero gradient almost everywhere. The **straight-through estimator** (STE) [21] treats rounding as the identity in the backward pass. For the symmetric quantizer $\hat{x} = s\ \mathrm{clip}(\mathrm{round}(x/s), q_{\min}, q_{\max})$ this gives

$$\frac{\partial \hat{x}}{\partial x} \approx \begin{cases} 1, & q_{\min} \le x/s \le q_{\max}, \\ 0, & \text{otherwise}, \end{cases}$$

so gradients pass unchanged through values inside the range and stop at clipped values [15]. The float "shadow" weights are updated, and the forward pass always uses their quantized version.

When to use which: in the white paper's experiments PTQ was sufficient for 8-bit weights and activations (within 0.7% of float for every model tested), while lower bit widths, and 4-bit activations in particular, needed QAT; networks with depthwise-separable layers remained the hardest even with QAT [15].

*Online only: Beyond round-to-nearest: AdaRound, BRECQ and LSQ — https://blog.shyandram.dev/en/learn/deep-learning/model-compression/notes/beyond-round-to-nearest-adaround-brecq-and-lsq/*

### Mixed precision and other number formats

Not all layers are equally sensitive, so **mixed-precision quantization** gives different layers different bit widths: HAWQ chooses them from second-order (Hessian) information [22], HAQ from latency and energy feedback of a hardware simulator, finding that the best policies differ between edge and cloud accelerators [23]. Integers are not the only option: 8-bit floating-point encodings (E4M3 and E5M2) [24] spend bits on an exponent, which tolerates a wider range of magnitudes at the cost of fewer steps.

Do not confuse this with **mixed-precision training** (DLCV 01A), which keeps a 32-bit master copy of the weights and computes in 16-bit to speed up training. Mixed-precision quantization is about the deployed model.

## Knowledge distillation

**Plain version.** Train a small **student** to imitate a large **teacher**, not only the correct labels. The teacher's full output (which wrong classes it considers plausible) and its intermediate features carry more information per example than a label does, so the student learns more from the same images.

The response-based loss of Hinton et al., with temperature $T$ and the $T^2$ factor, is derived and tested in the DLCV 01 Supplementary notes and summarized in ML 11 [25]; we do not repeat it. In one line: the student minimizes

$(1-\alpha)\,\mathrm{CE}(y, \sigma(\mathbf{z})) + \alpha\, T^2\, \mathrm{CE}(\sigma(\mathbf{v}/T), \sigma(\mathbf{z}/T))$, where $\mathbf{z}$ and $\mathbf{v}$ are the student and teacher logits,

$y$ the label, $\sigma$ the softmax, CE cross-entropy and $\alpha \in [0,1]$ the weight of the soft term. Hinton et al. show that matching logits, as Ba and Caruana did by regressing them with an L2 loss [26], is the high-temperature limit of this loss [25].

### Code: distilling the toy MLP into an 8-unit student

The teacher is the 50,432-weight MLP from the pruning snippet. The student has one hidden layer of 8 units (592 weights) and is trained by full-batch gradient descent, once on labels only and twice with the distillation loss.

```
import warnings
import numpy as np
from sklearn.datasets import load_digits
from sklearn.exceptions import ConvergenceWarning
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
warnings.filterwarnings("ignore", category=ConvergenceWarning)

X, y = load_digits(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(X / 16.0, y, test_size=0.3, random_state=0, stratify=y)
teacher = MLPClassifier(hidden_layer_sizes=(256, 128), max_iter=300, random_state=0).fit(Xtr,
 ↪  ytr)
h = Xtr
for l, (w, b) in enumerate(zip(teacher.coefs_, teacher.intercepts_)):   # teacher logits
    h = h @ w + b if l == 2 else np.maximum(h @ w + b, 0)
z_teacher = h

def softmax(z, T=1.0):
    e = np.exp((z - z.max(axis=1, keepdims=True)) / T)
    return e / e.sum(axis=1, keepdims=True)

def train_student(T, alpha, hidden=8, steps=500, lr=0.2, seed=0):
    """64 -> 8 -> 10 ReLU network, full-batch gradient descent on the distillation loss."""
    rng = np.random.default_rng(seed)
    W1, b1 = rng.normal(0, np.sqrt(2 / 64), (64, hidden)), np.zeros(hidden)
    W2, b2 = rng.normal(0, np.sqrt(1 / hidden), (hidden, 10)), np.zeros(10)
    Y, p_T = np.eye(10)[ytr], softmax(z_teacher, T)
    for _ in range(steps):
        a = np.maximum(Xtr @ W1 + b1, 0)
        z = a @ W2 + b2
        # gradient of (1-alpha)*CE(y, softmax(z)) + alpha*T^2*CE(p_T, softmax(z/T)) w.r.t. z
        g = ((1 - alpha) * (softmax(z) - Y) + alpha * T * (softmax(z, T) - p_T)) / len(Xtr)
        ga = (g @ W2.T) * (a > 0)
        W2 -= lr * a.T @ g;  b2 -= lr * g.sum(0)
        W1 -= lr * Xtr.T @ ga; b1 -= lr * ga.sum(0)
    return (np.maximum(Xte @ W1 + b1, 0) @ W2 + b2).argmax(1)

print(f"teacher: {sum(w.size for w in teacher.coefs_)} weights, accuracy {teacher.score(Xte,
 ↪  yte):.3f}")
for T in (1, 4):
    print(f"mean top probability of the teacher on training images at T = {T}: "
          f"{softmax(z_teacher, T).max(axis=1).mean():.3f}")
```

```
for name, T, alpha in [("labels only", 1, 0.0), ("distill, T = 1", 1, 0.5), ("distill, T =
↪ 4", 4, 0.5)]:
    accs = [(train_student(T, alpha, seed=s) == yte).mean() for s in range(5)]
    print(f"student (592 weights), {name:14s}: accuracy {np.mean(accs):.3f} +/-
    ↪ {np.std(accs):.3f} over 5 seeds")
```

Output (toy results):

```
teacher: 50432 weights, accuracy 0.980
mean top probability of the teacher on training images at T = 1: 0.998
mean top probability of the teacher on training images at T = 4: 0.787
student (592 weights), labels only   : accuracy 0.936 +/- 0.025 over 5 seeds
student (592 weights), distill, T = 1: accuracy 0.936 +/- 0.025 over 5 seeds
student (592 weights), distill, T = 4: accuracy 0.954 +/- 0.006 over 5 seeds
```

Distillation at $T = 4$ raised the student's mean accuracy from 0.936 to 0.954 and made it much less dependent on the random seed. At $T = 1$ it changed nothing, and the second line says why: on its own training images the teacher is almost certain (mean top probability 0.998), so its soft targets are practically the labels. The temperature is what exposes the teacher's opinion about the wrong classes. The effect is small and specific to this toy; it is the mechanism, not the number, that transfers.

### Feature and attention distillation

The teacher's intermediate representations can be imitated too. **FitNets** train a thin, deep student so that one of its middle layers (the "guided" layer), passed through a small learned **regressor**, predicts a middle layer of the teacher (the "hint" layer); the regressor is needed because the two layers have different widths, and the authors use a convolutional regressor to keep its size manageable [27]. **Attention transfer** imitates where the teacher's features are active rather than the features themselves: for a feature map $A \in \mathbb{R}^{C\times H\times W}$ it forms a spatial map

$$F(A) = \sum_{c=1}^{C} |A_c|^p,$$

where $A_c$ is channel $c$ and $p \geq 1$ an exponent (the paper also studies $p = 1$ and a max over channels), and trains the student to match the $\ell_2$-normalized maps of the teacher at several depths [28]. Because $F(A)$ has no channel dimension, teacher and student widths need not match. Gou et al. organize the field by what is transferred (responses, features, relations between examples or layers) and how (offline from a fixed teacher, online, or self-distillation) [29].

### What decides whether distillation helps

- **Teacher and student must see the same input.** Beyer et al. found that distillation works best when teacher and student receive the identical augmented view of each image ("consistent"

teaching), when augmentation is aggressive, and when training is very long ("patient"); inconsistent setups plateaued lower in their experiments [30].

- **A better teacher is not always a better teacher.** Cho and Hariharan found that larger, more accurate teachers often did not produce better students, attributed this to mismatched capacity, found multi-step distillation through intermediate-size models ineffective, and reported that stopping the teacher's training early mitigated the problem [31].
- **Distillation is a training method, not a compression of an existing model**: it needs a training run, data, and a student architecture chosen in advance. DeiT, for example, trains a vision Transformer with an extra distillation token that learns from a convolutional teacher (DLCV 02) [32].

### Low-rank factorization

**Plain version.** A big weight matrix often contains much less information than its size suggests. Writing it as the product of two thin matrices replaces one large layer by two small ones, with far fewer weights and operations.

Any $m \times n$ matrix $W$ has a singular value decomposition $W = U\Sigma V^\top$, where $U$ $(m \times m)$ and $V$ $(n \times n)$ have orthonormal columns and $\Sigma$ is diagonal with the singular values $\sigma_1 \geq \sigma_2 \geq \ldots \geq 0$. Keeping the $r$ largest gives the best rank-$r$ approximation in the Frobenius norm (ML 01A):

$$W \approx AB, \qquad A = U_{:,1:r}\,\Sigma_{1:r,1:r} \in \mathbb{R}^{m\times r}, \qquad B = V^\top_{:,1:r} \in \mathbb{R}^{r\times n},$$

where $U_{:,1:r}$ is the first $r$ columns of $U$ and $r$ the chosen **rank**. The layer $x \mapsto Wx$ becomes two layers $x \mapsto A(Bx)$ with $r(m+n)$ weights instead of $mn$, which saves weights and MACs only if $r < mn/(m+n)$. For a convolution with kernel $W \in \mathbb{R}^{C_{\text{out}}\times C_{\text{in}}\times k\times k}$, reshape it to a $C_{\text{out}} \times (C_{\text{in}}k^2)$ matrix and factorize: $B$ becomes a $k\times k$ convolution with $r$ output channels and $A$ a $1\times 1$ convolution back to $C_{\text{out}}$ channels. Denton et al. exploited such linear structure in convolutional filters to speed up convolutional layers about twofold on CPU and GPU with accuracy within 1% [33]; Jaderberg et al. built filter bases that are rank-1 in the spatial domain (a $k \times 1$ followed by a $1 \times k$ filter) [34]; Zhang et al. extended low-rank acceleration to very deep networks for classification and detection, minimizing the reconstruction error of each layer's nonlinear responses rather than of its weights [35].

```
import warnings
import numpy as np
from sklearn.datasets import load_digits
from sklearn.exceptions import ConvergenceWarning
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
warnings.filterwarnings("ignore", category=ConvergenceWarning)

X, y = load_digits(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(X / 16.0, y, test_size=0.3, random_state=0, stratify=y)
net = MLPClassifier(hidden_layer_sizes=(256, 128), max_iter=300, random_state=0).fit(Xtr,
 ↪  ytr)
(W1, W2, W3), (b1, b2, b3) = net.coefs_, net.intercepts_
```

```
U, S, Vt = np.linalg.svd(W2, full_matrices=False)       # W2 is 256 x 128
energy = np.cumsum(S**2) / np.sum(S**2)
for r in [4, 8, 16, 32, 128]:
    A, B = U[:, :r] * S[:r], Vt[:r]                     # W2 ~ A @ B: a 256->r and an r->128
     ↪  layer
    h = np.maximum(np.maximum(Xte @ W1 + b1, 0) @ A @ B + b2, 0)
    acc = ((h @ W3 + b3).argmax(1) == yte).mean()
    print(f"rank {r:3d}: {A.size + B.size:5d} weights instead of {W2.size} "
          f"({energy[r - 1]:.0%} of the squared singular values), accuracy {acc:.3f}")
```

Output (toy results):

```
rank   4:  1536 weights instead of 32768 (29% of the squared singular values), accuracy 0.620
rank   8:  3072 weights instead of 32768 (47% of the squared singular values), accuracy 0.978
rank  16:  6144 weights instead of 32768 (58% of the squared singular values), accuracy 0.981
rank  32: 12288 weights instead of 32768 (72% of the squared singular values), accuracy 0.981
rank 128: 49152 weights instead of 32768 (100% of the squared singular values), accuracy
 ↪  0.980
```

Rank 8 keeps the accuracy with less than a tenth of the layer's weights, although it keeps only 47% of the squared singular values. The Frobenius error of the weights is a poor guide to the error of the task: what matters is what the layer does to the inputs it actually sees, which is why Zhang et al. minimize response error instead [35]. The last line shows the break-even rule: at full rank the factorized form has more weights than the original, because $128 > 256 \cdot 128/384 \approx 85$. Splitting one layer into two also adds a memory round trip for the intermediate result, so on a memory-bound device the latency saving is smaller than the MAC saving.

### The three toys side by side

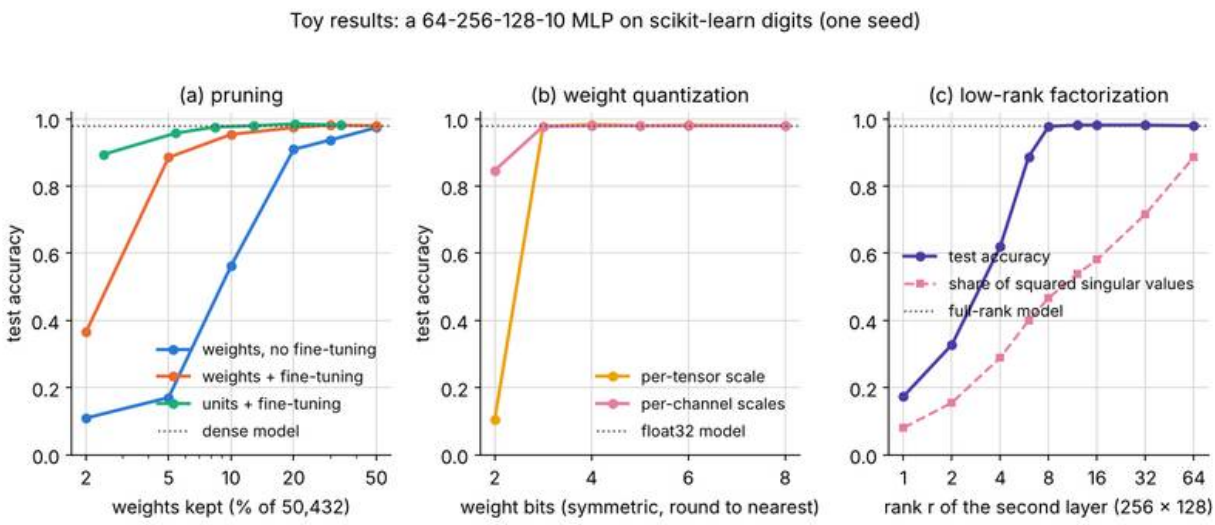


*Figure 3 — The three toy compressions on one network (one seed; toy results). (a) Fine-tuning rescues pruning, and on this over-parameterized toy removing whole units holds up best. (b) Weights survive down to 3 bits; at 2 bits only per-channel scales keep the model usable. (c) Accuracy saturates at rank 8 long before the weight approximation is good.*

### Designing small from the start

**Plain version.** Instead of shrinking a big network after training, you can choose an architecture that is small and hardware-friendly from the beginning, either by hand (efficient blocks) or by search. In practice the two approaches are combined: a well-designed small network is then quantized, and sometimes pruned or distilled.

The main hand-designed families are covered elsewhere and only listed here: **depthwise-separable convolutions** and inverted residuals (MobileNet [36], MobileNetV2 [37]; derivation and cost in DLCV 02 and ML 03), **compound scaling** of depth, width and resolution (EfficientNet [38]; DLCV 02), and the block-by-block cost arithmetic in DLCV 01B. The roofline view from the start of this chapter is the right lens for all of them: a block is efficient on a device if it does enough arithmetic per byte for that device and has fast kernels there.

### Re-parameterization: train rich, deploy plain

**Structural re-parameterization** trains a block with several parallel linear branches and merges them into one convolution for deployment. RepVGG trains blocks with a $3 \times 3$ convolution, a $1 \times 1$ convolution and an identity branch, each followed by batch normalization, and deploys a plain stack of $3 \times 3$ convolutions and ReLUs [39]. The folding is derived and checked numerically in the DLCV 03A Supplementary notes and, for enhancement networks, in the DLCV 09C Supplementary notes.

Re-parameterization interacts with quantization. The YOLOv6 report states that, because of its heavy use of re-parameterized blocks, earlier PTQ techniques failed to produce good results; in its version 1.0, YOLOv6-S fell from 42.4 to 35.0 COCO val AP under 8-bit PTQ, and to 40.9 when trained with RepOptimizer, which narrowed the activation distribution [40]. The merged kernel adds branches with very different ranges, which is exactly the situation per-tensor quantization handles worst.

### Neural architecture search

**Plain version.** Neural architecture search (NAS) lets an algorithm choose layer types, widths and kernel sizes, scoring candidates by accuracy and, in its hardware-aware form, by measured latency on the target device.

Elsken et al. describe every NAS method by three choices: the **search space** (which architectures are allowed), the **search strategy** (how candidates are proposed) and the **performance estimation strategy** (how a candidate is scored without training it fully) [41]. Three ideas matter for compression:

- **Weight sharing makes search affordable.** The original reinforcement-learning approach trained each proposed architecture separately [42]; ENAS lets all candidates share the weights of one large graph [43], DARTS relaxes the discrete choice into a continuous one optimized by gradient descent [44], and Once-for-All trains one network from which sub-networks of different depth, width, kernel size and resolution are extracted for different devices without retraining [45].
- **Search for measured latency, not FLOPs.** MnasNet puts latency measured on real phones into the objective [46]; ProxylessNAS searches directly on the target task and hardware [47]; FBNet's differentiable search is motivated by the same observation that FLOPs do not always reflect latency [48].
- **Keep a random baseline.** Li and Talwalkar found random search with early stopping at least as good as ENAS on the two benchmarks they used [49].

### Deployment: from a small model to a fast one

**Plain version.** A compressed model becomes fast only after a compiler or runtime turns it into a few large, fused operations with kernels that support its number format and sparsity pattern. Speed claims from papers often assume support that your device does not have.

### Operator fusion and batch-norm folding

Each separate operation reads its input from memory and writes its output back. **Operator fusion** merges consecutive operations (convolution, bias, batch normalization, activation) into one kernel so that intermediate results stay in fast on-chip memory; from the roofline viewpoint, it raises arithmetic intensity by removing memory round trips. The most common fusion is **batch-norm folding**: at inference, batch normalization [50] with learned scale $\gamma_c$, shift $\beta_c$, running mean $\mu_c$ and variance $\sigma_c^2$ is an affine map per output channel $c$, so it can be absorbed into the preceding convolution,

$$W'_c = \frac{\gamma_c}{\sqrt{\sigma_c^2 + \epsilon}} W_c, \qquad b'_c = \frac{\gamma_c}{\sqrt{\sigma_c^2 + \epsilon}} (b_c - \mu_c) + \beta_c,$$

where $W_c$ and $b_c$ are the kernel and bias of output channel $c$ and $\epsilon$ a small constant. A tested NumPy version is in the DLCV 03A Supplementary notes. For compression, notice the factor $\gamma_c / \sqrt{\sigma_c^2 + \epsilon}$: it differs from channel to channel, so folding can spread the per-channel weight ranges apart, which is one reason MobileNetV2 was hard to quantize per tensor [17]. On device, batch normalization is folded before quantization, so the folded weights are what the integer kernel sees, and QAT has to simulate the folding correctly [15].

### Integer kernels and runtime support

Quantization speeds up inference only if the runtime has integer (or low-precision) kernels for every operation in the graph. ML 11 shows a weight-only int8 model that is four times smaller but

*slower*, because it converts weights back to float on every call. In a real deployment, check: does the target support per-channel weight scales (not all hardware does [15]); which operators fall back to float (each fallback adds a dequantize and a quantize); and whether the runtime keeps the graph in integers between layers. The export and runtime tools themselves (ONNX, ONNX Runtime, TensorRT, `torch.export`) are described in ML 11.

### Why paper speed-ups shrink on devices

- **FLOPs are not the bottleneck.** A method that removes 90% of the MACs of memory-bound layers barely changes their time (Figure 1).
- **Sparsity needs kernels.** Unstructured sparsity speeds up only with sparse kernels for that pattern on that device [5], [8]; 2:4 sparsity has dedicated hardware on some GPUs [6]; structured pruning needs nothing special.
- **Batch size.** Throughput at a large batch is not latency at batch 1 (the linear-layer example above).
- **The rest of the pipeline.** Pre-processing, post-processing such as NMS (DLCV 03A), resizing and data transfer do not shrink when the network does.
- **Unsupported operators.** A single operator without a low-precision kernel can force conversions that cost more than they save.

### Compressing vision models: what is sensitive

**Plain version.** Different parts of a vision model react very differently to compression. Layers with few weights per channel, outputs that must be numerically precise (box coordinates, pixel values), and activations whose range changes from image to image are the usual weak points.

### Classification backbones

Networks built from depthwise-separable layers are the hardest common backbones to quantize. With 4-bit weights and 8-bit activations, per-tensor PTQ cost MobileNetV2 and EfficientNet-lite about 2.5 and 4.2 points of ImageNet top-1 in Nagel et al.'s experiments, which they attribute likely to the depthwise-separable convolutions; per-channel scales recovered 2.8 points for EfficientNet-lite [15]. Layers whose weight ranges differ strongly across channels are the first candidates for per-channel scales or higher precision, and a per-layer sensitivity test (quantize one layer at a time and measure the drop, as in Exercise 5) finds the others quickly.

### Detectors

Detectors predict continuous box coordinates and scores, and both suffer from coarse grids. In the white paper, EfficientDet-D1 reached 40.08 COCO mAP in float and 38.29 with 8-bit PTQ, but only 0.31 with 4-bit per-tensor weights (35.08 per-channel); QAT brought the 4-bit-weight results to 35.34 and 36.75 [15]. Re-parameterized detectors such as YOLOv6 need quantization-friendly

training (above) [40]. Because strict-IoU accuracy is what coarse box coordinates hurt first, check AP at high IoU thresholds after quantizing, not only AP at IoU 0.5 (DLCV 03A).

Distillation also needs care for detectors. Wang et al. found that applying ordinary distillation to a detector gave only minor gains, and instead had the student imitate the teacher's features at locations near objects, estimated from the anchors; this improved students on KITTI, Pascal VOC and COCO in their experiments [51].

### Dense prediction and restoration

For super-resolution, denoising and enhancement the output is an image, and every quantization error can become visible. Two properties make these networks harder than classifiers. First, many restoration networks remove batch normalization; Li et al. found that this leaves activations with a much larger dynamic range, which breaks quantization schemes that assume normalized activations, and proposed learning a max scale per layer (PAMS) [52]. Second, Hong et al. observed that in super-resolution networks each channel has a different activation distribution, which also changes with the input image, and proposed channel-wise distribution-aware quantization (DAQ) [53]. So calibrate on images like the deployment content (dark scenes for a low-light model), evaluate at full resolution with the metrics of DLCV 09, and inspect smooth gradients, where banding appears first. The NTIRE efficient super-resolution challenges rank runtime, FLOPs and parameters in separate tracks under a fixed PSNR threshold [54] (DLCV 09A); DLCV 09C covers efficient enhancement networks.

### Vision Transformers and diffusion models

In vision Transformers, the activations after softmax and GELU are far from Gaussian, which PTQ4ViT handles with two ranges per such tensor [55]; Liu et al. add a ranking loss so that quantized attention keeps the order of the attention scores [56]. Diffusion models add a time dimension: the same denoising network runs at many noise levels, and its activation distributions change with the time step. Q-Diffusion identifies this, together with bimodal activations in shortcut layers, as the key difficulty and uses time-step-aware calibration [57]; PTQ4DM studies the same multi-time-step calibration problem [58]. Large segmentation foundation models are compressed by distillation: MobileSAM distills the large image encoder of the Segment Anything Model [59] into a small one ("decoupled distillation") [60], and EfficientSAM pre-trains small encoders to reconstruct SAM's features [61].

## Modern view

### What the surveys say

- **General compression.** Cheng et al. divide the field into four categories: parameter pruning and quantization, low-rank factorization, transferred/compact convolutional filters, and knowledge distillation, and discuss each category's advantages and drawbacks [62]. Menghani organizes

efficient deep learning into five areas: compression techniques, learning techniques (such as distillation), automation (hyper-parameter optimization and architecture search), efficient architectures, and infrastructure (software and hardware), and adds a practitioner's guide [63].

- **Quantization.** Gholami et al. survey quantization from the basics (uniform and non-uniform quantizers, range calibration, granularity, fine-tuning) to advanced topics (integer-only inference, mixed precision, hardware-aware and distillation-assisted quantization, extreme low-bit quantization) [64]; Nagel et al.'s white paper is the practical companion, with tested PTQ and QAT pipelines and a debugging workflow [15].
- **Pruning and sparsity.** Hoefler et al. distill more than 300 papers on sparsity for inference and training, including how to obtain real speed-ups on hardware [8]. Cheng, Zhang and Shi organize pruning by universal or specific speed-up, when to prune, how to prune, and fusion with other compression techniques, and cover pruning of large language and multimodal models [65]. Blalock et al. remain the cautionary reference on how pruning results are reported [4].
- **Distillation.** Gou et al. categorize distillation by knowledge type, training scheme, teacher–student architecture and algorithm [29]; Wang and Yoon focus on vision and on why distillation works [66].
- **Architecture search.** Elsken et al. provide the three-dimension taxonomy used above [41]; White et al., writing after more than 1,000 NAS papers, give a taxonomy of search spaces, algorithms and speed-up techniques together with benchmarks and best practices [67].

*Online only: A White Paper on Neural Network Quantization (Nagel et al., 2021) — https://blog.shyandram.dev/en/learn/deep-learning/model-compression/papers/a-white-paper-on-neural-network-quantization-nagel-et-al-2021/*

### The state of the art, 2023–2026

- **Low-bit weights for very large models.** Weight-only quantization to 3–4 bits with error-compensating rounding (GPTQ [68]) or activation-aware protection of salient channels (AWQ [69]) became standard for large Transformers; ML 11 introduces both. SmoothQuant made 8-bit weights and activations practical for language models by migrating outliers from activations to weights [19].
- **Generative vision models at 4 bits.** After Q-Diffusion [57], SVDQuant quantizes both weights and activations of large diffusion models to 4 bits by moving activation outliers into the weights and absorbing the weight outliers with a small high-precision low-rank branch, fused into the low-bit kernel to avoid extra memory traffic; it combines two techniques of this chapter, low-rank factorization and quantization [70].
- **Floating-point 8-bit formats** (E4M3, E5M2) as an alternative to int8 for both training and inference [24].
- **Distilled foundation models.** Small students of large vision foundation models, such as MobileSAM and EfficientSAM for segmentation [60], [61], and the distilled DINOv2 backbones discussed in DLCV 02.
- **Structural pruning for any graph.** DepGraph-style dependency analysis makes channel pruning applicable to CNNs, Transformers and other architectures with one method [9].
- **Challenge-driven efficient restoration**, where runtime, FLOPs and parameters are scored sep-

arately [54].

### Open problems

- **Predicting device speed.** FLOPs and parameters still dominate papers, but only measured latency on the target matters, and it depends on kernels, memory and batch size [1].
- **Fair comparisons.** Baselines of equal size and training budget, multiple seeds and full trade-off curves are still often missing [4], [3].
- **Low-bit activations.** 4-bit weights are increasingly routine; 4-bit activations, especially in networks with depthwise layers, attention and time-varying distributions, still cost accuracy [15], [57].
- **What compression changes beyond accuracy.** Average accuracy can hide changes on rare classes, shifted inputs or calibration; measuring these after compression is still not standard practice.
- **Compression of restoration and generative models**, where the error is visible in the output and no single metric captures it.

### Key takeaways

- Define the actual deployment budget and measure end-to-end cost on the target device.
- Fewer MACs may not help memory-bound layers; inspect bytes moved and available kernels.
- Unstructured pruning needs sparse support for speed; structured pruning creates smaller dense layers but still needs measurement.
- Start with PTQ, inspect calibration and outliers, and use QAT when the required precision loses too much accuracy.
- Distillation depends on teacher information, matched input views and training; a larger teacher is not automatically better.
- Validate compressed models after export, including sensitive outputs; compare against equally well-trained small dense models.

### Exercises

1. **Roofline arithmetic.** On the made-up accelerator of the first snippet (4 TMAC/s, 100 GB/s), estimate the time of a $3 \times 3$ convolution with 128 input and 128 output channels on a $28 \times 28$ map in 16-bit precision. Is it compute- or memory-bound? Then find the smallest batch size at which a $1024 \times 1024$ fp32 linear layer becomes compute-bound.

   **Hint**

   The convolution does $9 \cdot 128 \cdot 128 \cdot 784 \approx 115.6$ M MACs and moves $147{,}456 + 2 \cdot 128 \cdot 784 = 348{,}160$ values, or about 0.70 MB at 2 bytes each, so its intensity is about 166 MACs per byte: compute-bound, about 28.9 µs. For the linear layer at batch $B$, intensity is $B \cdot 1024^2 / (4\,(1024^2 + 2048B))$; setting it to 40 gives $B \geq 233$. Adapt the `layer` function from the snippet to check.

2. **Large-sparse or small-dense?** In the pruning snippet, the narrow dense network beat the sparse one at about 2,500 weights, unlike Zhu and Gupta's results on large models. Design an experiment on the toy that would make the comparison fairer (think about what each model was allowed to train for and how it was initialized), and predict how the result would change if the original network were already narrow, for example 64–32–16–10.

**Hint**

Give both models the same total number of optimizer steps, tune their learning rates separately, and add a third model: the narrow architecture trained from scratch (the baseline that Liu et al. argue for). A network that is already narrow has little redundancy to remove, so structured pruning should lose accuracy faster, and the unstructured model, which can keep important weights anywhere, may then win.

3. **Quantize by hand.** Quantize the values $-0.8, 0, 0.3, 2.4$ to 4-bit unsigned integers with the affine scheme and a min-max range. Compute $s$, $z$, every $q$ and every $\hat{x}$. Why is the error for $-0.8$ not zero even though it is the end of the range, and could it be larger than $s/2$?

**Hint**

$s = 3.2/15 \approx 0.2133$ and $z = 0 - \text{round}(-0.8/0.2133) = -\text{round}(-3.75) = 4$, so $q = 0, 4, 5, 15$ and $\hat{x} \approx -0.853, 0, 0.213, 2.347$. Rounding $z$ to an integer shifts the whole grid so that 0 is exact; the grid then spans $[-0.853, 2.347]$ instead of $[-0.8, 2.4]$, so neither end value lies on a grid point (both errors are about 0.053, below $s/2 \approx 0.107$). A value above $2.347 + s/2$ or below $-0.853 - s/2$ would be clipped, and its error could exceed $s/2$.

4. **Temperature and weight in distillation.** Rerun the distillation snippet with $T \in \{2, 8\}$ and $\alpha \in \{0.5, 0.9\}$. Which settings help the 8-unit student, and why might a very high temperature help less than a moderate one? Use the gradient of the soft term, $\alpha T\,(\sigma(\mathbf{z}/T) - \sigma(\mathbf{v}/T))$, in your answer.

**Hint**

One run gave 0.948 and 0.953 for $T = 2$ ($\alpha = 0.5$ and 0.9), but 0.937 and 0.916 for $T = 8$, against 0.936 with labels only. At a very high temperature the soft targets approach a uniform distribution and say less about the correct class, and with $\alpha = 0.9$ the label term has little weight left; the factor $\alpha T$ also changes the effective step size, so the learning rate interacts with $T$. Compare differences with the spread over seeds before drawing conclusions.

5. **Per-layer sensitivity.** Using the functions `q_weight` and `run` from the outlier snippet, quantize the weights of **one layer at a time** to 2 bits (per channel) while keeping the others in float, and rank the three layers by the accuracy drop. Then quantize only the most robust two layers to 2 bits and keep the most sensitive one at 8 bits. What weight memory does this mixed-precision model need compared with all-8-bit?

**Hint**

Pass a list of quantized and unquantized weight matrices to `run` instead of using its `wbits` argument. The layers have $64 \cdot 256$, $256 \cdot 128$ and $128 \cdot 10$ weights; memory is $\sum_{\ell} P_{\ell}\, b_{\ell}/8$ bytes plus one scale per output unit. This is the idea behind sensitivity-based mixed precision such as HAWQ, with measured accuracy instead of Hessian information.

## References


1. M. Dehghani, A. Arnab, L. Beyer, A. Vaswani and Y. Tay, "The efficiency misnomer," ICLR, 2022. arXiv
2. S. Williams, A. Waterman and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009. doi
3. Z. Liu, M. Sun, T. Zhou, G. Huang and T. Darrell, "Rethinking the value of network pruning," ICLR, 2019. arXiv
4. D. Blalock, J. J. Gonzalez Ortiz, J. Frankle and J. Guttag, "What is the state of neural network pruning?," MLSys, 2020. arXiv
5. H. Li, A. Kadav, I. Durdanovic, H. Samet and H. P. Graf, "Pruning filters for efficient ConvNets," ICLR, 2017. arXiv
6. A. Mishra, J. Albericio Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu and P. Micikevicius, "Accelerating sparse deep neural networks," arXiv:2104.08378, 2021. arXiv
7. Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan and C. Zhang, "Learning efficient convolutional networks through network slimming," ICCV, 2017. arXiv
8. T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden and A. Peste, "Sparsity in deep learning: pruning and growth for efficient inference and training in neural networks," *Journal of Machine Learning Research*, vol. 22, no. 241, pp. 1–124, 2021. paper
9. G. Fang, X. Ma, M. Song, M. B. Mi and X. Wang, "DepGraph: towards any structural pruning," CVPR, 2023. doi arXiv
10. S. Han, J. Pool, J. Tran and W. J. Dally, "Learning both weights and connections for efficient neural networks," NeurIPS (NIPS), 2015. arXiv
11. S. Han, H. Mao and W. J. Dally, "Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding," ICLR, 2016. arXiv
12. M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," arXiv:1710.01878, 2017. arXiv
13. J. Frankle and M. Carbin, "The lottery ticket hypothesis: finding sparse, trainable neural networks," ICLR, 2019. arXiv
14. J. Frankle, G. K. Dziugaite, D. M. Roy and M. Carbin, "Linear mode connectivity and the lottery ticket hypothesis," ICML, 2020. arXiv
15. M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen and T. Blankevoort, "A white paper on neural network quantization," arXiv:2106.08295, 2021. arXiv
16. B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," CVPR, 2018. doi arXiv

17. M. Nagel, M. van Baalen, T. Blankevoort and M. Welling, "Data-free quantization through weight equalization and bias correction," ICCV, 2019. arXiv
18. T. Dettmers, M. Lewis, Y. Belkada and L. Zettlemoyer, "LLM.int8(): 8-bit matrix multiplication for transformers at scale," NeurIPS, 2022. arXiv
19. G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth and S. Han, "SmoothQuant: accurate and efficient post-training quantization for large language models," ICML, 2023. arXiv
20. M. Nagel, R. A. Amjad, M. van Baalen, C. Louizos and T. Blankevoort, "Up or down? Adaptive rounding for post-training quantization," ICML, 2020. arXiv
21. Y. Bengio, N. Léonard and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," arXiv:1308.3432, 2013. arXiv
22. Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney and K. Keutzer, "HAWQ: Hessian aware quantization of neural networks with mixed-precision," ICCV, 2019. arXiv
23. K. Wang, Z. Liu, Y. Lin, J. Lin and S. Han, "HAQ: hardware-aware automated quantization with mixed precision," CVPR, 2019. arXiv
24. P. Micikevicius et al., "FP8 formats for deep learning," arXiv:2209.05433, 2022. arXiv
25. G. Hinton, O. Vinyals and J. Dean, "Distilling the knowledge in a neural network," NIPS Deep Learning Workshop, 2014; arXiv:1503.02531. arXiv
26. L. J. Ba and R. Caruana, "Do deep nets really need to be deep?," NeurIPS (NIPS), 2014. arXiv
27. A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta and Y. Bengio, "FitNets: hints for thin deep nets," ICLR, 2015. arXiv
28. S. Zagoruyko and N. Komodakis, "Paying more attention to attention: improving the performance of convolutional neural networks via attention transfer," ICLR, 2017. arXiv
29. J. Gou, B. Yu, S. J. Maybank and D. Tao, "Knowledge distillation: a survey," *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021. doi
30. L. Beyer, X. Zhai, A. Royer, L. Markeeva, R. Anil and A. Kolesnikov, "Knowledge distillation: a good teacher is patient and consistent," CVPR, 2022. arXiv
31. J. H. Cho and B. Hariharan, "On the efficacy of knowledge distillation," ICCV, 2019. arXiv
32. H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles and H. Jégou, "Training data-efficient image transformers & distillation through attention," ICML, 2021. arXiv
33. R. Denton, W. Zaremba, J. Bruna, Y. LeCun and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," NeurIPS (NIPS), 2014. arXiv
34. M. Jaderberg, A. Vedaldi and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," BMVC, 2014. arXiv
35. X. Zhang, J. Zou, K. He and J. Sun, "Accelerating very deep convolutional networks for classification and detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 10, pp. 1943–1955, 2016. doi arXiv
36. A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and H. Adam, "MobileNets: efficient convolutional neural networks for mobile vision applications," arXiv:1704.04861, 2017. arXiv
37. M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L.-C. Chen, "MobileNetV2: inverted residuals and linear bottlenecks," CVPR, 2018. arXiv
38. M. Tan and Q. V. Le, "EfficientNet: rethinking model scaling for convolutional neural networks," ICML, 2019. arXiv
39. X. Ding, X. Zhang, N. Ma, J. Han, G. Ding and J. Sun, "RepVGG: making VGG-style ConvNets

great again," CVPR, 2021. arXiv
40. C. Li et al., "YOLOv6: a single-stage object detection framework for industrial applications," arXiv:2209.02976, 2022. arXiv
41. T. Elsken, J. H. Metzen and F. Hutter, "Neural architecture search: a survey," *Journal of Machine Learning Research*, vol. 20, pp. 1–21, 2019. arXiv
42. B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," ICLR, 2017. arXiv
43. H. Pham, M. Y. Guan, B. Zoph, Q. V. Le and J. Dean, "Efficient neural architecture search via parameter sharing," ICML, 2018. arXiv
44. H. Liu, K. Simonyan and Y. Yang, "DARTS: differentiable architecture search," ICLR, 2019. arXiv
45. H. Cai, C. Gan, T. Wang, Z. Zhang and S. Han, "Once-for-all: train one network and specialize it for efficient deployment," ICLR, 2020. arXiv
46. M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard and Q. V. Le, "MnasNet: platform-aware neural architecture search for mobile," CVPR, 2019. arXiv
47. H. Cai, L. Zhu and S. Han, "ProxylessNAS: direct neural architecture search on target task and hardware," ICLR, 2019. arXiv
48. B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia and K. Keutzer, "FBNet: hardware-aware efficient ConvNet design via differentiable neural architecture search," CVPR, 2019. doi arXiv
49. L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," UAI, 2019. arXiv
50. S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift," ICML, 2015. arXiv
51. T. Wang, L. Yuan, X. Zhang and J. Feng, "Distilling object detectors with fine-grained feature imitation," CVPR, 2019. arXiv
52. H. Li, C. Yan, S. Lin, X. Zheng, Y. Li, B. Zhang, F. Yang and R. Ji, "PAMS: quantized super-resolution via parameterized max scale," ECCV, 2020. arXiv
53. C. Hong, H. Kim, S. Baik, J. Oh and K. M. Lee, "DAQ: channel-wise distribution-aware quantization for deep image super-resolution networks," WACV, 2022. arXiv
54. B. Ren, Y. Li, N. Mehta, R. Timofte et al., "The ninth NTIRE 2024 efficient super-resolution challenge report," CVPR Workshops, 2024. doi arXiv
55. Z. Yuan, C. Xue, Y. Chen, Q. Wu and G. Sun, "PTQ4ViT: post-training quantization for vision transformers with twin uniform quantization," ECCV, 2022. doi arXiv
56. Z. Liu, Y. Wang, K. Han, S. Ma and W. Gao, "Post-training quantization for vision transformer," NeurIPS, 2021. arXiv
57. X. Li, Y. Liu, L. Lian, H. Yang, Z. Dong, D. Kang, S. Zhang and K. Keutzer, "Q-Diffusion: quantizing diffusion models," ICCV, 2023. doi arXiv
58. Y. Shang, Z. Yuan, B. Xie, B. Wu and Y. Yan, "Post-training quantization on diffusion models," CVPR, 2023. arXiv
59. A. Kirillov et al., "Segment anything," ICCV, 2023. doi arXiv
60. C. Zhang, D. Han, Y. Qiao, J. U. Kim, S.-H. Bae, S. Lee and C. S. Hong, "Faster segment anything: towards lightweight SAM for mobile applications," arXiv:2306.14289, 2023. arXiv
61. Y. Xiong et al., "EfficientSAM: leveraged masked image pretraining for efficient segment any-

thing," CVPR, 2024. doi arXiv
62. Y. Cheng, D. Wang, P. Zhou and T. Zhang, "Model compression and acceleration for deep neural networks: the principles, progress, and challenges," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 126–136, 2018. doi arXiv
63. G. Menghani, "Efficient deep learning: a survey on making deep learning models smaller, faster, and better," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–37, 2023. doi arXiv
64. A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney and K. Keutzer, "A survey of quantization methods for efficient neural network inference," book chapter in *Low-Power Computer Vision*; arXiv:2103.13630, 2021. arXiv
65. H. Cheng, M. Zhang and J. Q. Shi, "A survey on deep neural network pruning: taxonomy, comparison, analysis, and recommendations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 12, pp. 10558–10578, 2024. doi arXiv
66. L. Wang and K.-J. Yoon, "Knowledge distillation and student-teacher learning for visual intelligence: a review and new outlooks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 6, pp. 3048–3068, 2022. doi arXiv
67. C. White, M. Safari, R. Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey and F. Hutter, "Neural architecture search: insights from 1000 papers," arXiv:2301.08727, 2023. arXiv
68. E. Frantar, S. Ashkboos, T. Hoefler and D. Alistarh, "GPTQ: accurate post-training quantization for generative pre-trained transformers," ICLR, 2023. arXiv
69. J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan and S. Han, "AWQ: activation-aware weight quantization for LLM compression and acceleration," MLSys, 2024. arXiv
70. M. Li, Y. Lin, Z. Zhang, T. Cai, X. Li, J. Guo, E. Xie, C. Meng, J.-Y. Zhu and S. Han, "SVDQuant: absorbing outliers by low-rank components for 4-bit diffusion models," ICLR, 2025. arXiv
71. Y. Li, R. Gong, X. Tan, Y. Yang, P. Hu, Q. Zhang, F. Yu, W. Wang and S. Gu, "BRECQ: pushing the limit of post-training quantization by block reconstruction," ICLR, 2021. arXiv
72. S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy and D. S. Modha, "Learned step size quantization," ICLR, 2020. arXiv

# DLCV 02 • Deep Image Classification

How deep networks decide what an image shows: losses and metrics, datasets, the CNN and transformer architectures and why they were designed that way, modern training recipes, pre-training, robustness, calibration and interpretability.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/image-classification/

## What you'll learn

- Match single-label and multi-label tasks to outputs, losses and metrics.
- Compare CNN and Transformer backbones by design and cost.
- Separate architecture gains from training-recipe gains.
- Choose linear probing, fine-tuning or zero-shot transfer.
- Evaluate class imbalance, calibration, distribution shift and deployment efficiency.

## The big picture

Image classification asks one question of a whole image: *which of these labels fits?* It is the oldest deep-learning benchmark task and still the place where most backbone networks are born. A detector, a segmenter or a restoration network usually starts from a backbone that was first trained, or at least evaluated, as an image classifier.

> **Explain like I'm five**
>
> Think of a child learning animals from a picture book. At first the child only sees blobs. With many pages, the child learns to spot ears, then faces, then "that's a cat". A deep classifier does the same in layers: early layers notice edges and colors, middle layers notice parts like eyes and wheels, and the last layer votes for a label. Training is the parent saying "no, that's a dog" thousands of times. The rest of this tutorial is about building a better picture book (data), a better brain (architecture), a better way of teaching (training recipe), and better exams (evaluation), because a child who only ever saw cats on sofas may decide that "sofa" means "cat".

This tutorial assumes you know what a convolution, a CNN layer and backpropagation are. The classical side of classification (Bayes classifiers, perceptrons, a small CNN written in NumPy, softmax and cross-entropy derived by hand) is in IP 13. The broad history of deep learning for images is in DLCV 01. Here we go deep on classification itself. For a refresher on activation functions and optimizers, see ML 02 ·Neural Networks. A good single survey of the CNN side of this story is Li et al. [1].

**Related chapters.** DLCV 02A turns classification into comparison: face recognition, embeddings and margin-based losses for open-set problems. DLCV 01B compares how classification and restoration networks compose the same building blocks.

## The classification problem

**Plain version.** A classifier produces class scores. The loss compares their probabilities with labels and updates the network to improve its predictions.

### Single-label, multi-label and fine-grained

Let $\mathbf{x} \in \mathbb{R}^{H\times W\times 3}$ be an RGB image and $\mathcal{Y} = \{1, \dots, K\}$ a set of $K$ classes. A network $f_\theta$ with parameters $\theta$ maps the image to a vector of **logits** $\mathbf{z} = f_\theta(\mathbf{x}) \in \mathbb{R}^K$, one unnormalized score per class.

- **Single-label (multi-class)** classification assumes exactly one correct class $y \in \mathcal{Y}$. The logits are turned into a probability vector with the softmax,

$$p_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \qquad k = 1, \dots, K,$$

where $p_k$ is the model's estimate of $P(y = k \mid \mathbf{x})$. Training minimizes the **cross-entropy**, the negative log-probability of the true class, averaged over a mini-batch:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{B}\sum_{i=1}^{B} \log p_{y_i}(\mathbf{x}_i),$$

where $B$ is the batch size and $y_i$ the label of image $\mathbf{x}_i$. The gradient with respect to the logits is $\partial\mathcal{L}/\partial\mathbf{z} = \mathbf{p} - \mathbf{e}_y$ (per image), with $\mathbf{e}_y$ the one-hot vector of the true class; DIP Chapter 13 derives it.

- **Multi-label** classification allows any subset of labels (a street photo can contain "car", "person" and "bicycle"). Each class gets an independent sigmoid $\sigma(z_k) = 1/(1+e^{-z_k})$ and a binary cross-entropy, $-\sum_k [t_k \log \sigma(z_k) + (1 - t_k)\log(1 - \sigma(z_k))]$, where $t_k \in \{0, 1\}$ says whether class $k$ is present. Metrics change too: mean average precision over classes replaces accuracy.
- **Fine-grained** classification separates visually similar subordinate classes, such as 200 bird species or 1,000 aircraft variants. The loss is the same as single-label, but the signal lives in small parts (a beak shape, a wing bar), classes are often long-tailed, and label noise from non-expert annotators is common. iNaturalist [6] is the standard large fine-grained benchmark.

### Metrics: top-1, top-5 and beyond

**Top-1 accuracy** is the fraction of images whose highest-scoring class is the true one. **Top-$k$ accuracy** counts an image as correct if the true class is among the $k$ highest scores. The ImageNet challenge reported top-5 error because many images contain several objects, and some classes

are nearly indistinguishable [2]. Top-5 hides real mistakes, so modern papers mostly report top-1. For imbalanced data, report **balanced accuracy** (the mean of per-class recalls) or per-class results, because a classifier can score high accuracy by ignoring rare classes.

```
import torch
import torch.nn.functional as F

logits = torch.tensor([[2.0, 1.0, 0.1, -1.0],    # 2 images, 4 classes
                       [0.5, 2.5, 2.4, 0.0]])
y = torch.tensor([0, 2])

p = logits.softmax(dim=1)
ce = F.cross_entropy(logits, y)                       # mean of -log p[i, y_i]
ce_ls = F.cross_entropy(logits, y, label_smoothing=0.1)
top1 = (logits.argmax(1) == y).float().mean()
top2 = (logits.topk(2, dim=1).indices == y[:, None]).any(1).float().mean()
print(p.round(decimals=3))
print(f"CE {ce:.3f}  CE+LS {ce_ls:.3f}  top-1 {top1:.2f}  top-2 {top2:.2f}")

# multi-label: one independent sigmoid per class, binary cross-entropy
targets = torch.tensor([[1., 0., 1., 0.], [0., 1., 1., 0.]])
print(f"BCE {F.binary_cross_entropy_with_logits(logits, targets):.3f}")
```

The second image is a near-tie between classes 1 and 2. Top-1 counts it as wrong; top-2 counts it as right. That one example is the whole argument for and against top-5.

### The confusion matrix and the metrics built on it

**Plain version.** Accuracy summarizes correctness; a confusion matrix shows which classes are confused. Choose additional metrics to expose the errors that matter.

### The binary case: TP, FP, FN, TN

Take one class as **positive** (say "sick") and the rest as **negative**. Each prediction lands in one of four cells:

| | Predicted positive | Predicted negative |
|---|---|---|
| **Actually positive** | true positive (TP) | false negative (FN) |
| **Actually negative** | false positive (FP) | true negative (TN) |

**Explain like I'm five**

A smoke alarm is a classifier. When there is a fire and it rings, that is a true positive. When you burn toast and it rings, that is a false positive: annoying, but nobody is hurt. When there is a fire and it stays quiet, that is a false negative: the dangerous one. When nothing happens and it stays quiet, that is a true negative. A good alarm is not just "right most of the time" (it is quiet almost every day, so that is easy). It must ring for fires and stay quiet for toast. Different metrics weigh those two goals differently.

With $N = \mathrm{TP} + \mathrm{FP} + \mathrm{FN} + \mathrm{TN}$ test examples, the standard ratios are

$$\text{accuracy} = \frac{\mathrm{TP}+\mathrm{TN}}{N}, \quad \text{precision} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}, \quad \text{recall (TPR)} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}},$$
$$\text{specificity (TNR)} = \frac{\mathrm{TN}}{\mathrm{TN}+\mathrm{FP}}, \quad \mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP}+\mathrm{TN}} = 1 - \mathrm{TNR}.$$

- **Precision** answers "when the model says positive, how often is it right?" It matters when a false alarm is costly: a spam filter that hides real mail, an automatic product recall.
- **Recall**, also called **sensitivity** or the **true positive rate (TPR)**, answers "of all real positives, how many did we catch?" It matters when a miss is costly: cancer screening, a defect detector on a production line.
- **Specificity**, the **true negative rate (TNR)**, is recall for the negative class. The **false positive rate (FPR)** is its complement, the fraction of negatives wrongly flagged.

Precision and recall pull against each other: lowering the decision threshold flags more items, which raises recall and usually lowers precision. Single numbers that combine them are:

$$F_1 = \frac{2\,\mathrm{P}\,\mathrm{R}}{\mathrm{P}+\mathrm{R}} = \frac{2\,\mathrm{TP}}{2\,\mathrm{TP}+\mathrm{FP}+\mathrm{FN}}, \qquad F_\beta = \frac{(1+\beta^2)\,\mathrm{P}\,\mathrm{R}}{\beta^2\,\mathrm{P}+\mathrm{R}},$$

where P is precision, R recall and $\beta > 0$ says how many times more recall matters than precision ($F_2$ favours recall, $F_{0.5}$ favours precision). $F_1$ is the harmonic mean, so it is high only when *both* parts are high. Note what $F_1$ ignores: true negatives never appear in it.

Two further summaries use all four cells:

$$\text{balanced accuracy} = \frac{\mathrm{TPR}+\mathrm{TNR}}{2}, \qquad \mathrm{MCC} = \frac{\mathrm{TP}\cdot\mathrm{TN} - \mathrm{FP}\cdot\mathrm{FN}}{\sqrt{(\mathrm{TP}+\mathrm{FP})(\mathrm{TP}+\mathrm{FN})(\mathrm{TN}+\mathrm{FP})(\mathrm{TN}+\mathrm{FN})}}.$$

Balanced accuracy is the mean of the per-class recalls, so a classifier that ignores one class scores 0.5 however rare that class is. The **Matthews correlation coefficient** (MCC) is the Pearson correlation between the true and predicted binary labels: $+1$ is perfect, 0 is no better than chance, $-1$ is perfectly inverted. Chicco and Jurman [57] argue that it should be preferred over accuracy and $F_1$ for binary evaluation because it is high only when the classifier does well in all four cells. Sokolova and Lapalme [58] give a systematic comparison of these measures and of how each one reacts to changes in the confusion matrix.

**Why accuracy misleads at 99:1.** Suppose 10 of 1,000 scans are sick. A "model" that always answers *healthy* scores 99.0% accuracy. Model B flags 30 scans and catches 8 of the 10 sick ones; it scores *lower* accuracy (97.6%) yet is the only useful one:

```
import numpy as np
from sklearn.metrics import (accuracy_score, balanced_accuracy_score, confusion_matrix,
                             f1_score, matthews_corrcoef, precision_score, recall_score)

# 1,000 scans, 10 sick (label 1). Model A says "healthy" to everyone.
# Model B flags 30 scans and catches 8 of the 10 sick ones.
y = np.r_[np.ones(10), np.zeros(990)].astype(int)
```

```
pred_a = np.zeros(1000, dtype=int)
pred_b = np.r_[np.ones(8), np.zeros(2), np.ones(22), np.zeros(968)].astype(int)

for name, p in [("all-negative", pred_a), ("model B", pred_b)]:
    tn, fp, fn, tp = confusion_matrix(y, p).ravel()
    print(f"{name:12s} TP={tp} FP={fp} FN={fn} TN={tn}  "
          f"acc={accuracy_score(y, p):.3f} bal-acc={balanced_accuracy_score(y, p):.3f} "
          f"P={precision_score(y, p, zero_division=0):.3f} R={recall_score(y, p):.3f} "
          f"F1={f1_score(y, p, zero_division=0):.3f} MCC={matthews_corrcoef(y, p):.3f}")
# all-negative TP=0 FP=0 FN=10 TN=990  acc=0.990 bal-acc=0.500 P=0.000 R=0.000 F1=0.000
 ↪  MCC=0.000
# model B      TP=8 FP=22 FN=2 TN=968  acc=0.976 bal-acc=0.889 P=0.267 R=0.800 F1=0.400
 ↪  MCC=0.454
```

Balanced accuracy, $F_1$ and MCC all rank model B first; accuracy ranks it last. Rules of thumb: use accuracy only when classes are roughly balanced and all errors cost the same; report precision and recall (or $F_\beta$ with a $\beta$ chosen from the real costs) when one kind of error matters more; use balanced accuracy or MCC as a single imbalance-aware summary.

### The multi-class confusion matrix

With $K$ classes the confusion matrix $C \in \mathbb{N}^{K\times K}$ has entry $C_{ij}$ = number of test images whose **true** class is $i$ (row) and whose **predicted** class is $j$ (column); this is the convention of scikit-learn [62]. The diagonal holds the correct predictions. Dividing each row by its sum gives the **row-normalized** matrix, whose diagonal is the per-class recall and whose off-diagonal entries show *which* classes are confused. Figure 1 shows one for a logistic regression trained on only 200 of scikit-learn's $8 \times 8$ digits: most mistakes are $3 \to 8$, $4 \to 1$ and $7 \to 9$, pairs that share strokes.

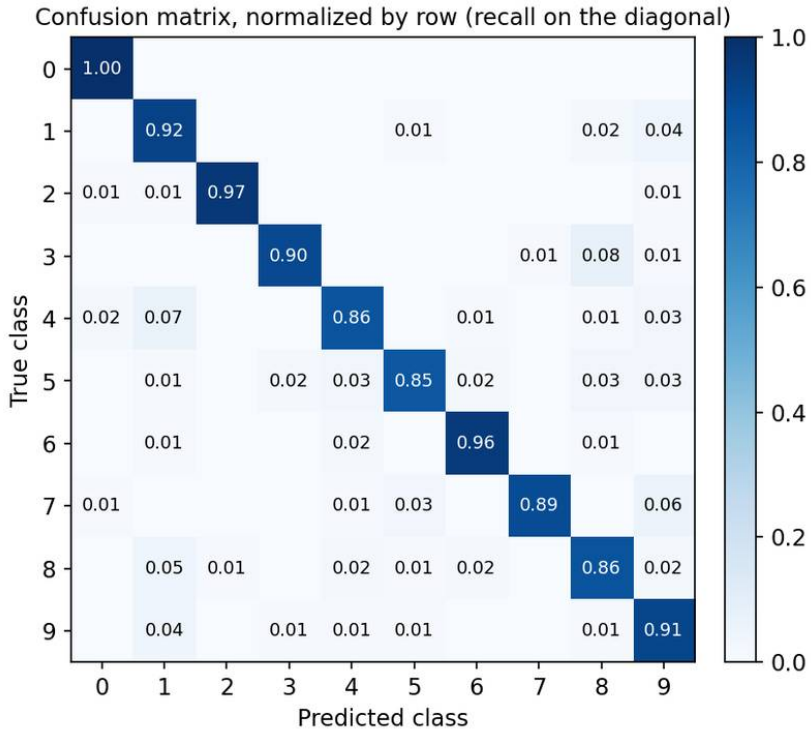



*Figure 1 — Row-normalized confusion matrix of a logistic regression trained on 200 digits and tested on 899 (accuracy 91.1%). Each row sums to 1; the diagonal is the recall of each class, and the bright off-diagonal cells name the confusions worth fixing.*

Treating each class $k$ in turn as "positive" and all others as "negative" gives per-class counts: $\mathrm{TP}_k = C_{kk}$, $\mathrm{FP}_k = \sum_{i \neq k} C_{ik}$ (the rest of column $k$), $\mathrm{FN}_k = \sum_{j \neq k} C_{kj}$ (the rest of row $k$). Hence per-class precision $\mathrm{P}_k = C_{kk} / \sum_i C_{ik}$ and recall $\mathrm{R}_k = C_{kk} / \sum_j C_{kj}$. There are three common ways to turn $K$ per-class numbers into one:

- **Macro average**: the plain mean over classes, $\mathrm{F}_1^{\text{macro}} = \frac{1}{K} \sum_k \mathrm{F}_{1,k}$. Every class counts equally, so rare classes weigh as much as common ones.
- **Micro average**: sum $\mathrm{TP}_k$, $\mathrm{FP}_k$, $\mathrm{FN}_k$ over classes first, then compute the metric once. In single-label multi-class problems every error is one FP (for the predicted class) and one FN (for the true class), so micro precision = micro recall = micro $F_1$ = accuracy.
- **Weighted average**: the mean of per-class values weighted by each class's **support** $n_k$ (number of true examples). It sits close to micro and hides rare-class failures almost as well.

On an imbalanced test set the averages can differ a lot. If a model is good on a common class and poor on rare ones, micro averaging (dominated by the common class) gives a high score and macro averaging a much lower one; that gap is the signal. Report macro when every class matters (rare diseases, rare species); report micro/accuracy when every *image* matters equally.

*Online only: A worked macro-versus-micro example — https://blog.shyandram.dev/en/learn/deep-learning/image-classification/notes/a-worked-macro-versus-micro-example/*

**Cohen's kappa** [59] is accuracy corrected for chance: it compares the observed agreement between predictions and labels with the agreement a classifier would reach by guessing each class at its observed rate. It is 1 for perfect agreement and 0 at chance level, and it is common when a model is compared with human annotators.

*Online only: Cohen's kappa: agreement beyond chance — https://blog.shyandram.dev/en/learn/deep-learning/image-classification/notes/cohens-kappa-agreement-beyond-chance/*

```
from sklearn.datasets import load_digits
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (average_precision_score, classification_report,
                             confusion_matrix, precision_recall_fscore_support,
                              ↪  roc_auc_score)
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize

X, y = load_digits(return_X_y=True)
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.5, stratify=y, random_state=0)
clf = LogisticRegression(max_iter=2000).fit(X_tr[:200], y_tr[:200])   # only 200 labels
pred, prob = clf.predict(X_te), clf.predict_proba(X_te)

cm = confusion_matrix(y_te, pred)                     # rows = true, columns = predicted
cm_norm = confusion_matrix(y_te, pred, normalize="true")   # each row sums to 1 (Figure 1)
print(classification_report(y_te, pred, digits=3))    # per-class P, R, F1, support +
 ↪  averages
for avg in ["macro", "micro", "weighted"]:
    p, r, f, _ = precision_recall_fscore_support(y_te, pred, average=avg)
    print(f"{avg:8s} P={p:.3f} R={r:.3f} F1={f:.3f}")
print(f"ROC-AUC (one-vs-rest, macro) {roc_auc_score(y_te, prob, multi_class='ovr'):.4f}")
Y = label_binarize(y_te, classes=range(10))           # one binary column per class
print(f"mean AP over classes {average_precision_score(Y, prob, average='macro'):.4f}")
# macro    P=0.914 R=0.911 F1=0.912
# micro    P=0.911 R=0.911 F1=0.911
# weighted P=0.915 R=0.911 F1=0.912
# ROC-AUC (one-vs-rest, macro) 0.9958
# mean AP over classes 0.9742
```

Here macro and micro agree because the digit classes are balanced; on an imbalanced test set they would not.

### Threshold-free evaluation: ROC and precision–recall curves

Every metric above needs a decision rule (argmax, or "positive if the score exceeds $t$"). A scoring classifier can be evaluated at *all* thresholds at once. Sweep $t$ from high to low and record a point per threshold:

- The **ROC curve** plots TPR against FPR. Its area, **ROC-AUC**, equals the probability that a randomly chosen positive gets a higher score than a randomly chosen negative; 0.5 is chance, 1.0 is perfect separation. Fawcett [54] is the standard introduction.

- The **precision–recall (PR) curve** plots precision against recall. Its summary, **average precision** (AP), is the precision averaged over the recall levels at which a new positive is retrieved,

$$\mathrm{AP} = \sum_n (\mathrm{R}_n - \mathrm{R}_{n-1})\,\mathrm{P}_n,$$

where $\mathrm{P}_n$ and $\mathrm{R}_n$ are precision and recall at the $n$-th threshold, ordered by decreasing score (this is scikit-learn's definition [62]; benchmarks such as PASCAL VOC [61] have used interpolated variants, so check which one a paper reports).

**Why PR under heavy imbalance.** FPR divides by the number of negatives. When negatives outnumber positives 99 to 1, even thousands of false alarms barely move FPR, so the ROC curve looks excellent while precision is poor. Precision divides by the number of *flagged* items, so false alarms hit it directly, and the random baseline of AP equals the positive rate rather than 0.5. Davis and Goadrich [55] proved that a curve dominates in ROC space if and only if it dominates in PR space, and showed that optimizing ROC-AUC does not optimize AP; Saito and Rehmsmeier [56] showed with simulations and a literature review that PR plots are more informative than ROC plots on imbalanced data. Figure 2 makes this concrete: two classifiers with the same ROC-AUC (0.95) on a task with 1% positives differ clearly in AP (0.45 against 0.36).

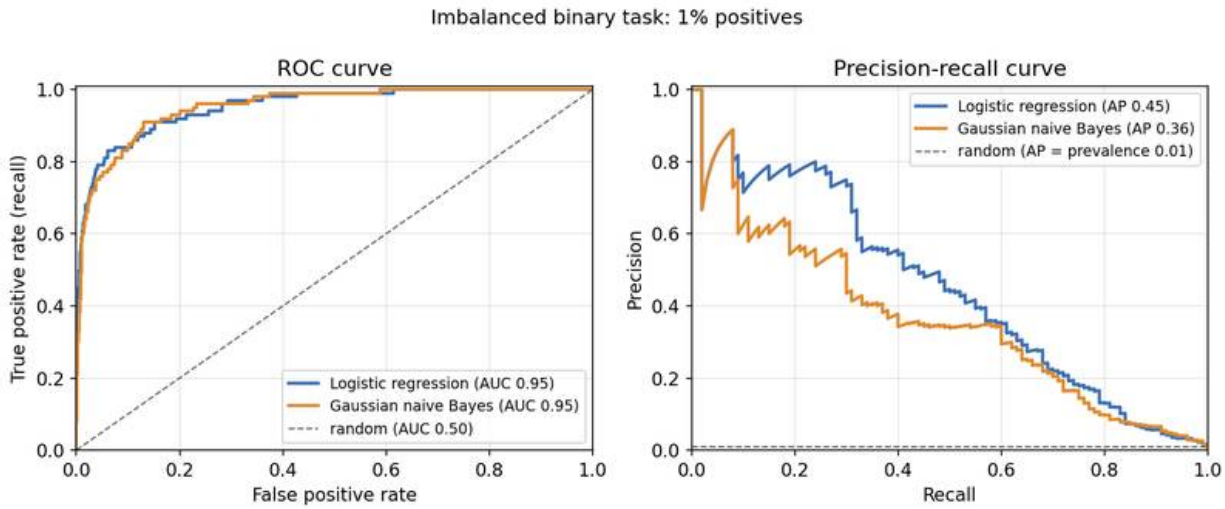



*Figure 2 — ROC (left) and precision–recall (right) curves for two classifiers on a synthetic task with 1% positives (100 of 10,000 test points). Both have ROC-AUC 0.95, yet logistic regression keeps much higher precision at every recall level (AP 0.45 against 0.36). An all-negative classifier would score 99% accuracy here.*

For $K > 2$ classes, both curves are computed **one-vs-rest**: class $k$ is positive, all others negative, using the score $p_k$. The $K$ areas are then macro-averaged (`roc_auc_score(..., multi_class="ovr")`), or micro-averaged by pooling all $(k, \text{image})$ pairs.

### Multi-label metrics

In multi-label classification each label is its own yes/no problem, so the standard summary is the **average precision of each class**, averaged over classes into **mAP**, as in the PASCAL VOC classification task [61]. Thresholded metrics either compute a binary metric per label and average it over labels (macro or micro, as above), or score each image's predicted label set and average over images.

*Online only: Multi-label metrics in detail — https://blog.shyandram.dev/en/learn/deep-learning/image-classification/notes/multi-label-metrics-in-detail/*

### Calibration: are the probabilities honest?

Accuracy only checks the argmax. **Calibration** checks whether the confidence means what it says: among all predictions made with confidence 0.8, about 80% should be right. Formally, a classifier is calibrated if $P(\hat{y} = y \mid \hat{p} = c) = c$ for every confidence $c$, where $\hat{y}$ is the predicted class and $\hat{p} = \max_k p_k$ its probability. The usual summary is the **expected calibration error**: split predictions into $M$ confidence bins $B_m$ and compute

$$\mathrm{ECE} = \sum_{m=1}^{M} \frac{|B_m|}{n} \left| \mathrm{acc}(B_m) - \mathrm{conf}(B_m) \right|,$$

where $n$ is the number of test images, $\mathrm{acc}(B_m)$ the accuracy inside bin $m$ and $\mathrm{conf}(B_m)$ the mean confidence there. Guo et al. [47] found that modern deep networks, trained long with cross-entropy, tend to be **overconfident**, and that a single scalar fix works remarkably well. We return to it in the robustness section.

## Datasets and why they matter

**Plain version.** Benchmarks support comparison under fixed conditions. Keep training and evaluation data separate, and check whether benchmark gains transfer beyond that test set.

### Training datasets

| Dataset | Images | Classes | Resolution | Role |
|---|---|---|---|---|
| MNIST [3] | 60k train / 10k test | 10 digits | 28×28 gray | sanity check, teaching |
| CIFAR-10 / CIFAR-100 [4] | 50k train / 10k test | 10 / 100 (20 superclasses) | 32×32 RGB | small-scale architecture and recipe studies |
| ImageNet-1k (ILSVRC 2012) [2] | about 1.28M train / 50k val | 1,000 | variable, usually trained at 224×224 | the standard backbone benchmark |

| Dataset | Images | Classes | Resolution | Role |
|---|---|---|---|---|
| ImageNet-21k [5] | roughly 13–14M (versions differ) | about 21k | variable | larger-scale supervised pre-training |
| iNaturalist [6] | hundreds of thousands, by year | thousands of species | variable | fine-grained, long-tailed |

MNIST is solved and now only catches bugs. CIFAR is small enough to run many controlled experiments, which is why regularization papers love it. ImageNet-1k became the reference after the 2012 challenge [2], [10]: architectures are usually reported as "top-1 on ImageNet-1k val at 224×224", and almost every pre-trained backbone in a model zoo has an ImageNet number attached. ImageNet-21k is the larger hierarchy from which ImageNet-1k was drawn; Ridnik et al. [5] made pre-training on it practical with a cleaned, semantic-hierarchy-aware recipe.

### Evaluation benchmarks

**Plain version.** Training data teaches; evaluation data examines. A model that only ever sat one exam may have learned that exam. So modern papers grade a backbone on a *battery* of test sets, each built to probe one thing: the original distribution, a fresh sample of it, corrupted or stylized images, unusual viewpoints, fine-grained categories, small labelled budgets or rare classes.

The evaluation sets you will meet most often fall into five groups:

- **In distribution.** The ImageNet-1k validation set [2] is the de facto test set; **ImageNet-ReaL** [8] re-labels the same images with multiple labels per image.
- **A fresh sample. ImageNetV2** [7] repeats the original collection process to build new test images.
- **Distribution shift. ImageNet-C** [42] adds synthetic corruptions, **ImageNet-R** [43] contains renditions such as art and cartoons, **ImageNet-A** [44] collects natural photos that fool a ResNet-50, **ImageNet-Sketch** [63] contains sketches, and **ObjectNet** [64] shows objects at unusual rotations, backgrounds and viewpoints.
- **Long tail and fine grain. ImageNet-LT** [73] trains on a long-tailed subset and tests on balanced data; iNaturalist [6], CUB-200-2011 [65] and Stanford Cars [66] test fine-grained recognition.
- **Transfer.** Suites such as the 12 datasets of Kornblith et al. [33] and **VTAB** [72] measure how well frozen or fine-tuned features adapt to small, specialized tasks.

Two practical warnings. Several fine-grained benchmarks report **mean per-class accuracy** instead of plain accuracy, so do not mix the two in one comparison. And image–text models such as CLIP [40] are often evaluated **zero-shot**, without any training on the target data: prompt wording then becomes a hyperparameter that must not be tuned on the test set, and web-scale pre-training data may already contain the test images.

*Online only: Evaluation benchmarks in detail — https://blog.shyandram.dev/en/learn/deep-learning/image-classification/notes/evaluation-benchmarks-in-detail/*

### How reliable are the numbers?

Three lines of work changed how we read ImageNet accuracy. Rebuilding the test set by the original procedure (ImageNetV2 [7]) lowered every model's accuracy but largely kept their ranking. Re-labelling studies [8], [9] showed that single labels and label errors can change which model looks best. And every dataset has its own bias in viewpoint and background, so test on independently collected data whenever possible.

*Online only: How reliable are ImageNet numbers? — https://blog.shyandram.dev/en/learn/deep-learning/image-classification/notes/how-reliable-are-imagenet-numbers/*

> **Read a benchmark number as a claim about a benchmark**
>
> "81.0% top-1" means: on these 50,000 particular validation images, with this preprocessing and this label set. Report the resolution, the number of crops, whether extra data or distillation was used, and the variance across seeds when differences are under a point.

### Evaluation protocol

A benchmark number is only as honest as the protocol behind it.

- **Three splits, three jobs.** Fit parameters on the **training** set. Choose hyperparameters, early stopping, temperature, thresholds and prompts on a **validation** set (or by cross-validation inside the training data). Touch the **test** set once, at the end. When a benchmark only publishes a validation split, as with ImageNet-1k, hold out part of the training set for tuning.
- **Never tune on the test set.** Every decision made after looking at test results (an extra augmentation, another epoch, a different checkpoint) fits the test set a little. Repeated over a field's worth of papers, this is **test-set reuse**, or adaptive overfitting to a benchmark. ImageNetV2 [7] found that ranking was largely preserved on a fresh sample, which is reassuring, but the same study also shows that absolute numbers do not transfer and that a fresh test set is the only direct check.
- **Report variation.** Train with several random seeds (at least three, more for small datasets) and report mean $\pm$ standard deviation, or a confidence interval. With 10,000 test images, ImageNetV2 [7] quotes 95% intervals of about $\pm 1\%$; a 0.3-point gain on such a set is not a result.
- **Report the whole recipe.** Resolution, number of crops, extra data, distillation, which variant of a metric (top-1 or mean per-class accuracy, which AP definition) and which version of a label set (original or ReaL).

## CNN architectures and the reasons behind them

**Plain version.** Each famous CNN fixed one specific problem of its predecessors: training too slow, too many parameters, too hard to optimize when deep, too expensive on a phone. Knowing the problem each one solved is more useful than memorizing their layer tables.

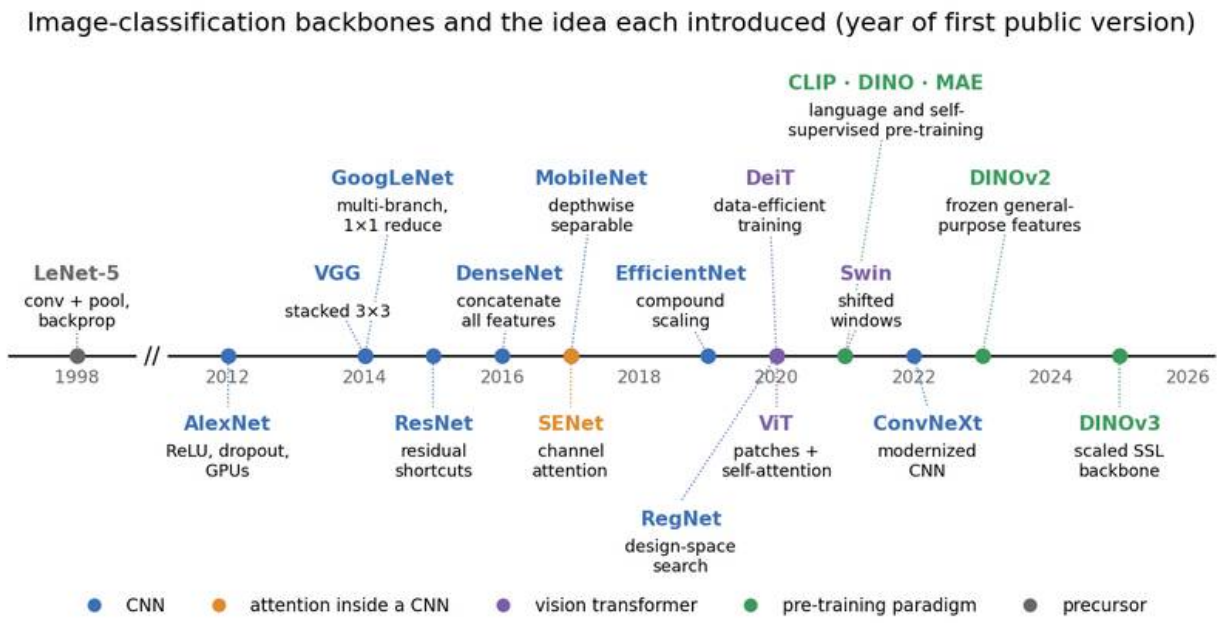



*Figure 3 — Image-classification backbones and the main idea each introduced. Dates are the year of the first public version (usually arXiv). Colors separate pure CNNs, attention inside CNNs, vision transformers and pre-training paradigms.*

### AlexNet: depth becomes trainable

Before 2012, the best ImageNet systems used hand-crafted features (SIFT, Fisher vectors) with linear classifiers. AlexNet [10] was a five-convolution, three-fully-connected network trained end to end on ImageNet. Three ingredients made it work at that scale. **ReLU** activations, $\max(0, z)$, do not saturate for positive inputs, so gradients do not vanish the way they do with tanh, and the paper reports several times faster training. **Dropout** in the large fully connected layers fought overfitting; those layers held most of the parameters. **GPUs** made it feasible: the model was split across two GPUs. Heavy data augmentation (random crops and flips, color perturbation) was the fourth, often forgotten, ingredient. The design lesson: with enough data and compute, learned features beat engineered ones.

### VGG: small kernels, stacked

VGG [11] asked a cleaner question: what happens if we only use $3 \times 3$ convolutions and go deeper (16–19 weight layers)? Two stacked $3 \times 3$ layers see a $5 \times 5$ region; three see $7 \times 7$. For $C$ input and output channels,

$$\underbrace{2 \cdot 3^2 C^2}_{\text{two } 3\times3} = 18C^2 \;<\; \underbrace{5^2 C^2}_{\text{one } 5\times5} = 25C^2,$$

ignoring biases. So the stack has fewer parameters for the same receptive field, and it inserts

an extra nonlinearity between the two layers. VGG's uniform design made it a favourite feature extractor for years. Its weakness is cost: the fully connected head and the wide early layers make it heavy.

### GoogLeNet / Inception: width, branches and $1 \times 1$ bottlenecks

GoogLeNet [12] attacked cost instead. Each **Inception module** runs $1\times1$, $3\times3$, $5\times5$ convolutions and pooling in parallel and concatenates the outputs, so the next layer can choose the right scale. Running big kernels on many channels would be expensive, so $1\times1$ **convolutions** first reduce the channel count. A $1 \times 1$ convolution is a per-pixel linear map across channels: it mixes channels without looking at neighbours, and it is the cheapest way to change width. GoogLeNet also replaced the large fully connected head with **global average pooling**. The follow-up Inception-v3 paper [13] factorized large kernels into stacks or into $1 \times n$ and $n \times 1$ pairs, and introduced **label smoothing**, which we meet again in the training section.

### ResNet: learning residuals

**The problem.** Stacking more plain layers eventually made networks *worse*, even on the training set. This "degradation" is an optimization problem, not overfitting: a deeper network could in principle copy a shallower one and set the extra layers to identity, yet SGD did not find that solution.

**The idea.** He et al. [14] let a block learn a residual $\mathcal{F}$ and add its input back:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}; \{W_i\}) + \mathbf{x},$$

where $\mathbf{x}$ is the block input, $\mathcal{F}$ is two or three convolution layers with weights $W_i$, and $\mathbf{y}$ the output. If the best mapping is close to identity, the block only has to push $\mathcal{F}$ toward zero, which is easy. When the shapes differ (more channels, stride 2), the shortcut becomes a $1 \times 1$ projection.

**Why it trains.** Because every block adds its input back, the output of a deep stack equals the input of any earlier block plus a sum of residuals. The gradient of the loss therefore has a direct path through the shortcuts to every block, so it cannot vanish through a long product of small factors. A follow-up paper [15] also found that keeping the shortcut a *pure* identity (moving BN and ReLU into the residual branch, "pre-activation") makes even deeper networks easier to train.

*Online only: Why residual networks train: the unrolled gradient — https://blog.shyandram.dev/en/learn/deep-learning/image-classification/notes/why-residual-networks-train-the-unrolled-gradient/*

*Optional: Batch normalization (a supplementary note in DLCV 01). What the BN layer used throughout this page computes, and why training and inference use different statistics.*

**The bottleneck.** For deeper ResNets (50 layers and up) each block becomes $1\times1$ (reduce) $\to 3\times3$ $\to 1\times1$ (expand), shown in Figure 4(c). At 256 channels, a basic block of two $3\times3$ convolutions has $2\cdot9\cdot256^2 \approx 1.18$M weights; a bottleneck that reduces to 64 channels has $256\cdot64+9\cdot64^2+64\cdot256 \approx$ 70k. The expensive $3 \times 3$ runs on a thin tensor, which is what makes 50–152 layers affordable.

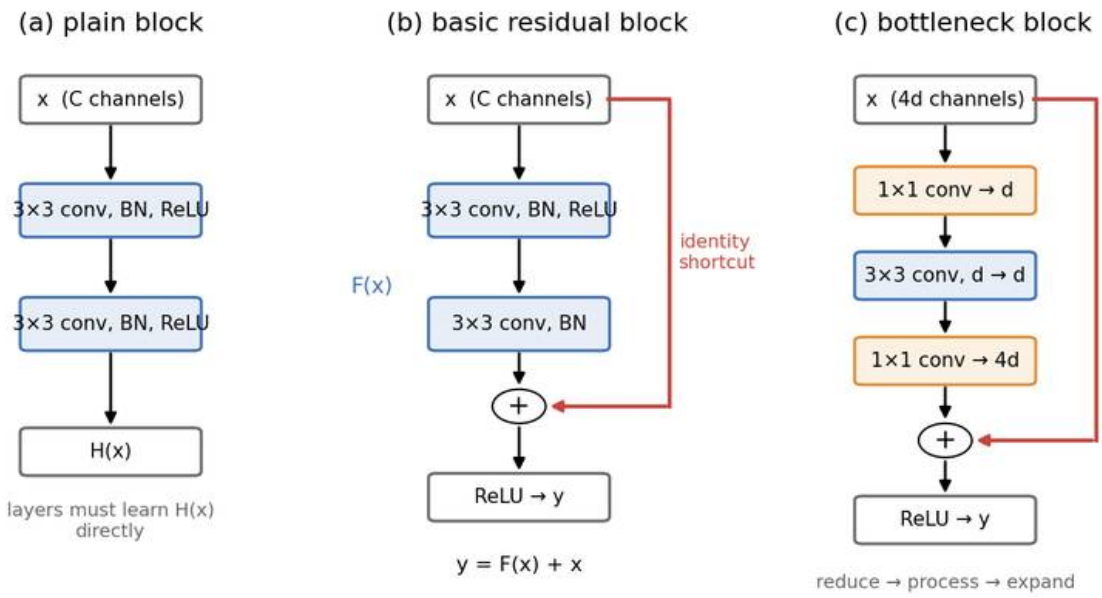



*Figure 4 — (a) A plain block must learn the full mapping $H(x)$. (b) A residual block learns $F(x) = H(x) - x$ and adds the input back through an identity shortcut. (c) The bottleneck block runs the 3×3 convolution on a reduced width d and expands back to 4d.*

The basic block used in this tutorial's experiments, written in PyTorch:

```
import torch.nn as nn
import torch.nn.functional as F

class BasicBlock(nn.Module):
    def __init__(self, cin, cout, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(cin, cout, 3, stride, 1, bias=False)
        self.bn1 = nn.BatchNorm2d(cout)
        self.conv2 = nn.Conv2d(cout, cout, 3, 1, 1, bias=False)
        self.bn2 = nn.BatchNorm2d(cout)
        self.short = nn.Identity()
        if stride != 1 or cin != cout:   # projection shortcut when shapes change
            self.short = nn.Sequential(nn.Conv2d(cin, cout, 1, stride, bias=False),
             ↪  nn.BatchNorm2d(cout))

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        return F.relu(out + self.short(x))   # y = F(x) + x
```

Two later CNN designs refine these ideas. **DenseNet** [16] gives each layer in a block the concatenated outputs of *all* earlier layers, which reuses features with few parameters but costs memory. **RegNet** [19] studied whole populations of ResNet-like networks and found that good ones follow simple rules for how width grows from stage to stage.

*Online only: Two more CNN designs: DenseNet and RegNet — https://blog.shyandram.dev/en/learn/deep-learning/image-classification/notes/two-more-cnn-designs-densenet-and-regnet/*

### MobileNet: factorize the convolution

MobileNet [17] targets phones. A standard $K \times K$ convolution from $M$ to $N$ channels on an $H \times W$ map costs $HWK^2MN$ multiply–adds. A **depthwise separable** convolution splits it into a **depthwise** $K \times K$ convolution that filters each channel separately ($HWK^2M$) and a **pointwise** $1 \times 1$ convolution that mixes channels ($HWMN$). The cost ratio is

$$\frac{HWK^2M + HWMN}{HWK^2MN} = \frac{1}{N} + \frac{1}{K^2},$$

so with $K = 3$ and many output channels the separable version is roughly 8–9 times cheaper. The bet is that spatial filtering and channel mixing do not need to be learned jointly. That bet held up so well that depthwise convolutions reappear in EfficientNet and ConvNeXt.

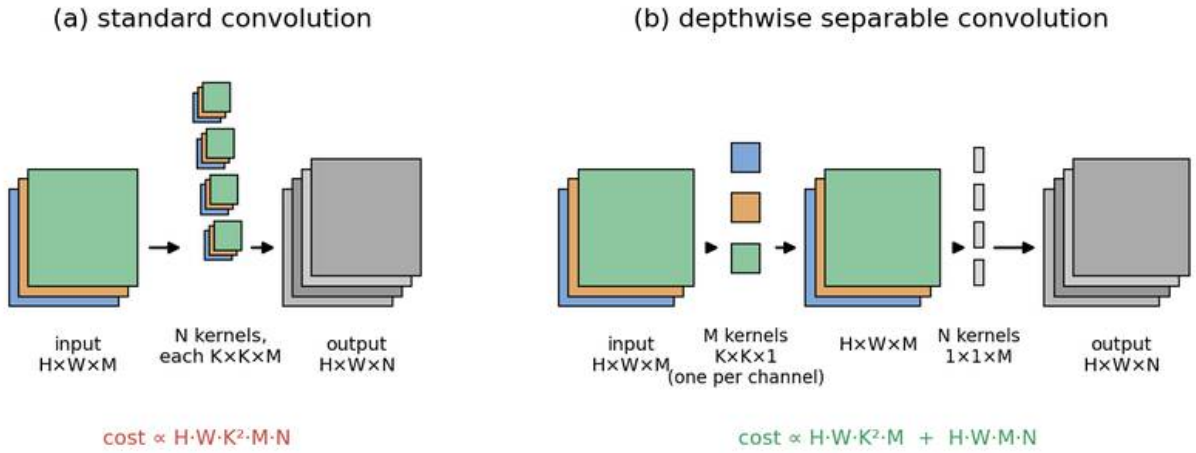



*Figure 5 — Standard convolution filters and mixes channels in one step. Depthwise separable convolution filters each channel alone (depthwise) and then mixes channels with 1×1 kernels (pointwise).*

```
import torch.nn as nn

def n_params(m):
    return sum(p.numel() for p in m.parameters())

M, N, K = 128, 128, 3
standard = nn.Conv2d(M, N, K, padding=1, bias=False)
separable = nn.Sequential(nn.Conv2d(M, M, K, padding=1, groups=M, bias=False),   # depthwise
                          nn.Conv2d(M, N, 1, bias=False))                        #
                           ↪  pointwise
print("standard:", n_params(standard), " separable:", n_params(separable),
      f" ratio {n_params(separable) / n_params(standard):.3f} vs 1/N + 1/K^2 = {1/N +
       ↪  1/K**2:.3f}")
```

```
# standard: 147456  separable: 17536  ratio 0.119 vs 1/N + 1/K^2 = 0.119
```

### EfficientNet: scale depth, width and resolution together

Given a good small network, how should we make it bigger? Deeper, wider, or higher input resolution? Tan and Le [18] argued these interact: a higher resolution image needs more depth (larger receptive field) and more width (finer patterns). **Compound scaling** grows all three with one coefficient $\phi$:

$$d = \alpha^\phi, \quad w = \beta^\phi, \quad r = \gamma^\phi, \qquad \text{s.t. } \alpha\beta^2\gamma^2 \approx 2, \;\; \alpha, \beta, \gamma \geq 1,$$

where $d$, $w$, $r$ multiply depth, width and resolution. FLOPs scale roughly as $d\,w^2 r^2$, so each unit increase of $\phi$ roughly doubles the cost. The constants are found by a small grid search on the base network (itself found by architecture search, using inverted-bottleneck blocks with depthwise convolutions and squeeze-and-excitation). The idea, "scale in a balanced way", outlived the specific constants.

### ConvNeXt: a CNN modernized step by step

After vision transformers took over the leaderboards, Liu et al. [20] asked how much of their advantage came from attention and how much from everything else. They started from a ResNet-50 trained with a transformer-style recipe and changed one thing at a time: stage ratios like Swin's, a "patchify" stem (a $4 \times 4$ stride-4 convolution), depthwise convolutions with larger $7 \times 7$ kernels, an inverted bottleneck, fewer activations and normalizations, GELU and LayerNorm. The result, ConvNeXt, matched Swin-style transformers of similar cost on ImageNet and downstream tasks. The lesson: architecture families matter less than the many small design and training choices that come with them.

## Attention and transformers

**Plain version.** A convolution only looks at a small neighbourhood with fixed weights. Attention lets each position decide, based on content, which other positions to listen to. Vision transformers apply this to image patches.

Attention first entered CNNs in a light form. The **squeeze-and-excitation** (**SE**) block [21] summarizes each channel of a feature map by its global average, and a tiny network turns these summaries into weights between 0 and 1 that rescale the channels. The network can then emphasise, say, "fur" channels when the whole image looks like an animal. SE blocks are part of EfficientNet and many mobile networks.

*Online only: Squeeze-and-excitation: attention inside a CNN — https://blog.shyandram.dev/en/learn/deep-learning/image-classification/notes/squeeze-and-excitation-attention-inside-a-cnn/*

### ViT: an image as a sequence of patches

The Vision Transformer [22] removes convolutions almost entirely. An image $\mathbf{x} \in \mathbb{R}^{H\times W\times C}$ is cut into $N = HW/P^2$ non-overlapping $P\times P$ patches. Each patch is flattened and linearly projected to a $D$-dimensional **token**. A learnable **class token** is prepended and learnable **position embeddings** are added:

$$\mathbf{z}_0 = [\,\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \ldots;\ \mathbf{x}_p^N\mathbf{E}\,] + \mathbf{E}_{\text{pos}},$$

where $\mathbf{x}_p^i \in \mathbb{R}^{P^2C}$ is the $i$-th flattened patch, $\mathbf{E} \in \mathbb{R}^{P^2C\times D}$ the patch embedding and $\mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1)\times D}$ the position embeddings. A stack of $L$ standard transformer encoder layers (multi-head self-attention and MLP, each with LayerNorm and a residual connection) processes the sequence, and a small head reads the final class token to produce logits. Self-attention for one head is

$$\text{Attn}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\Big(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\Big)\mathbf{V},$$

with $\mathbf{Q} = \mathbf{Z}\mathbf{W}_Q$, $\mathbf{K} = \mathbf{Z}\mathbf{W}_K$, $\mathbf{V} = \mathbf{Z}\mathbf{W}_V$ linear projections of the token matrix $\mathbf{Z}$, and $d$ the key dimension. The attention matrix is $(N+1)\times(N+1)$: every patch can attend to every other patch from the first layer, so the receptive field is global immediately.

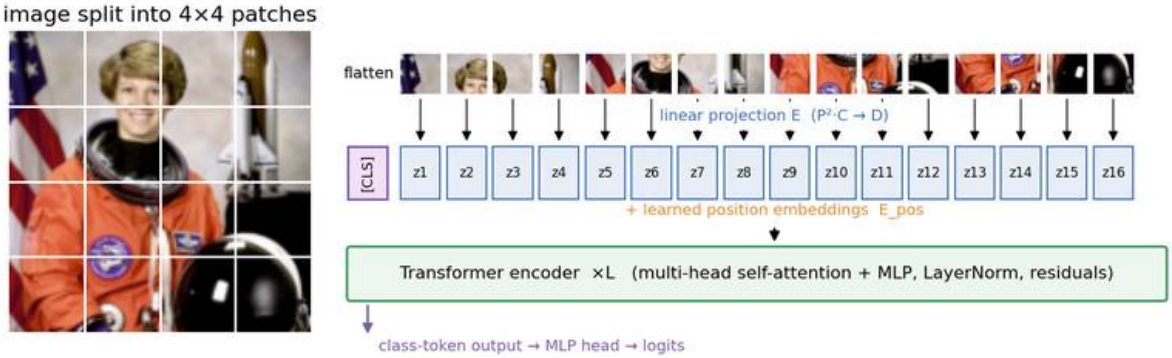



*Figure 6 — The ViT input pipeline. Patches become tokens through one shared linear projection (equivalently, a $P\times P$ convolution with stride $P$). A class token and learned position embeddings are added, and the class token's final state is classified.*

```
import torch
import torch.nn as nn

B, C, H, W, P, D = 2, 3, 32, 32, 8, 64
img = torch.randn(B, C, H, W)

# 1) patchify: (B, C, H, W) -> (B, N, P*P*C) with N = (H/P)*(W/P)
patches = img.unfold(2, P, P).unfold(3, P, P)                   # (B, C, H/P, W/P, P, P)
patches = patches.permute(0, 2, 3, 1, 4, 5).reshape(B, -1, C * P * P)
N = patches.shape[1]
```

```
# 2) linear patch embedding + class token + learned position embedding
embed = nn.Linear(C * P * P, D)          # same as nn.Conv2d(C, D, kernel_size=P, stride=P)
cls = nn.Parameter(torch.zeros(1, 1, D))
pos = nn.Parameter(torch.randn(1, N + 1, D) * 0.02)
z = torch.cat([cls.expand(B, -1, -1), embed(patches)], dim=1) + pos   # (B, N+1, D)

# 3) one head of scaled dot-product self-attention
Wq, Wk, Wv = (nn.Linear(D, D, bias=False) for _ in range(3))
q, k, v = Wq(z), Wk(z), Wv(z)
A = (q @ k.transpose(1, 2) / D ** 0.5).softmax(dim=-1)        # (B, N+1, N+1), rows sum to 1
out = A @ v
print("tokens:", N + 1, " attention:", tuple(A.shape), " class-token output:", tuple(out[:,
 ↪  0].shape))
```

**Why ViT needs data.** A CNN has built-in assumptions, called **inductive biases**: locality (a pixel mostly relates to its neighbours) and translation equivariance (the same filter everywhere). ViT has almost none; apart from patch extraction and position embeddings, it must learn spatial structure from data. The ViT paper [22] reports that, trained on ImageNet-1k alone, ViTs fell behind comparable ResNets, but pre-trained on much larger datasets (ImageNet-21k or the 300M-image JFT) they matched or beat them. Weak priors cost data at small scale and stop limiting the model at large scale. The same paper also tried **hybrid** models, feeding the feature map of a ResNet stem into the transformer instead of raw patches; hybrids helped at small compute budgets and the gap closed as models grew.

### DeiT: training ViTs on ImageNet alone

DeiT [23] showed that the "ViT needs huge data" conclusion was partly a statement about recipes. With strong augmentation (RandAugment, Mixup, CutMix, random erasing), stochastic depth, repeated augmentation and AdamW, a ViT-B trained only on ImageNet-1k reached 83.1% top-1. DeiT also added a **distillation token**: a second learnable token, like the class token, whose output is trained to match a teacher network's predictions (they found a convnet teacher worked best), while the class token is trained on the true label. This is a transformer-specific form of knowledge distillation [51]. At test time the two heads' outputs are combined.

*Optional: Knowledge distillation (a supplementary note in DLCV 01). The distillation loss and the role of the temperature.*

### Swin: windows, shifts and a hierarchy

Global attention costs grow quadratically with the number of tokens, which is a problem for high-resolution inputs and for dense tasks. Swin [24] computes attention only inside non-overlapping $M \times M$ windows, and shifts the window grid by half a window in alternating layers so information crosses window borders. Token count is reduced between stages by merging $2 \times 2$ neighbouring patches, giving a CNN-like pyramid with strides 4, 8, 16, 32. For an $h \times w$ token map with $C$ channels, the paper gives

$$\Omega(\text{MSA}) = 4hwC^2 + 2(hw)^2C, \qquad \Omega(\text{W-MSA}) = 4hwC^2 + 2M^2hwC,$$

so windowed attention is linear in the number of tokens $hw$ for a fixed window size $M$. Swin brought back two CNN priors, locality and hierarchy, and became a general-purpose backbone for detection and segmentation as well as classification. The broad landscape of vision transformers is surveyed by Khan et al. [25].

## Training recipes

**Plain version.** The same network can gain or lose several points of accuracy depending on how it is trained. A modern recipe is a bundle of small tricks that each make memorizing harder or optimization smoother.

### Data augmentation

- **Random resized crop** samples a crop covering a random fraction of the image area and a random aspect ratio, then resizes it to the training resolution. It teaches scale and position invariance and is the default for ImageNet training. **Horizontal flips** are added when they are label-preserving (they are for most photos, not for text or digits).
- **Mixup** [26] blends two images and their labels: $\tilde{\mathbf{x}} = \lambda \mathbf{x}_i + (1-\lambda)\mathbf{x}_j$ and $\tilde{\mathbf{y}} = \lambda \mathbf{y}_i + (1-\lambda)\mathbf{y}_j$, with $\lambda \sim \text{Beta}(\alpha, \alpha)$ and $\mathbf{y}$ one-hot. It encourages linear behaviour between training examples.
- **CutMix** [27] pastes a random rectangle from image $j$ into image $i$ and mixes the labels in proportion to the pasted area. Unlike Mixup, every pixel stays natural, and the network must recognize objects from partial views.
- **RandAugment** [28] replaces learned augmentation policies with two numbers: apply $N$ random operations (rotate, shear, posterize, contrast, ...) at a shared magnitude $M$. Its paper reports that this tiny search space matches or beats earlier automated policies, and that the right strength depends on model and dataset size.

```
import numpy as np
import torch
import torch.nn.functional as F

def mixup(x, y, num_classes, alpha=0.2):
    lam = float(np.random.beta(alpha, alpha))
    idx = torch.randperm(x.size(0))
    y1 = F.one_hot(y, num_classes).float()
    return lam * x + (1 - lam) * x[idx], lam * y1 + (1 - lam) * y1[idx]

def cutmix(x, y, num_classes, alpha=1.0):
    lam = float(np.random.beta(alpha, alpha))
    idx = torch.randperm(x.size(0))
    H, W = x.shape[-2:]
    h, w = int(H * np.sqrt(1 - lam)), int(W * np.sqrt(1 - lam))
    cy, cx = np.random.randint(H), np.random.randint(W)
    y0, y1_, x0, x1_ = max(cy - h // 2, 0), min(cy + h // 2, H), max(cx - w // 2, 0), min(cx
     ↪  + w // 2, W)
```

```
    x = x.clone()
    x[..., y0:y1_, x0:x1_] = x[idx][..., y0:y1_, x0:x1_]
    lam = 1 - (y1_ - y0) * (x1_ - x0) / (H * W)            # actual area kept
    y1 = F.one_hot(y, num_classes).float()
    return x, lam * y1 + (1 - lam) * y1[idx]

def soft_ce(logits, soft_targets):
    return -(soft_targets * logits.log_softmax(1)).sum(1).mean()
```

### Label smoothing

Cross-entropy with one-hot targets keeps pushing the true logit up forever; the loss only reaches zero at infinite margin. **Label smoothing** [13] replaces the target with

$$\mathbf{q} = (1 - \varepsilon)\,\mathbf{e}_y + \frac{\varepsilon}{K}\mathbf{1},$$

where $\varepsilon$ (often 0.1) is the smoothing strength, $\mathbf{e}_y$ the one-hot vector and $\mathbf{1}$ the all-ones vector. The optimum is now a finite logit gap, which regularizes the network and usually reduces overconfidence. In PyTorch it is one argument: `F.cross_entropy(logits, y, label_smoothing=0.1)`.

### Optimizer, schedule and regularizers

- **AdamW** [29] decouples weight decay from the adaptive gradient step: instead of adding $\lambda\theta$ to the gradient (which Adam then rescales per parameter), it shrinks the weights directly, $\theta \leftarrow \theta - \eta(\hat{\mathbf{m}}/(\sqrt{\hat{\mathbf{v}}} + \epsilon) + \lambda\theta)$, where $\eta$ is the learning rate, $\hat{\mathbf{m}}, \hat{\mathbf{v}}$ Adam's moment estimates and $\lambda$ the decay. It is the default for transformers and for modern CNN recipes. Biases and normalization parameters are usually excluded from decay.
- **Warmup + cosine schedule.** Large learning rates are unstable in the first steps, when normalization statistics and Adam's moment estimates are still poor, so the learning rate rises linearly for a few epochs. It then follows a half cosine down to near zero, $\eta_t = \frac{1}{2}\eta_{\max}(1 + \cos(\pi t/T))$ with $t$ the step after warmup and $T$ the remaining steps, an idea popularized by SGDR [30].
- **Stochastic depth** [31] randomly skips whole residual branches during training (the block becomes identity), with a skip probability that grows with depth. It shortens the effective depth during training and acts like an ensemble of shallower networks.
- **EMA of weights** keeps an exponential moving average $\bar{\theta} \leftarrow \mu\bar{\theta} + (1 - \mu)\theta$ (with $\mu$ close to 1, e.g. 0.999) and evaluates $\bar{\theta}$. It smooths out the noise of the last SGD steps.

```
import math
import torch

def warmup_cosine(step, total, warmup):
    if step < warmup:
        return (step + 1) / warmup                          # linear warmup
    p = (step - warmup) / max(1, total - warmup)
```

```
    return 0.5 * (1 + math.cos(math.pi * p))            # cosine decay to 0

model = torch.nn.Linear(10, 3)
decay = [p for n, p in model.named_parameters() if p.ndim > 1]      # weights
no_decay = [p for n, p in model.named_parameters() if p.ndim <= 1]  # biases, norm scales
opt = torch.optim.AdamW([{"params": decay, "weight_decay": 0.05},
                         {"params": no_decay, "weight_decay": 0.0}], lr=1e-3)
total = 1000
sched = torch.optim.lr_scheduler.LambdaLR(opt, lambda s: warmup_cosine(s, total, warmup=50))
ema = torch.optim.swa_utils.AveragedModel(model,
 ↪  multi_avg_fn=torch.optim.swa_utils.get_ema_multi_avg_fn(0.999))
for step in range(total):
    loss = model(torch.randn(16, 10)).pow(2).mean()
    opt.zero_grad(); loss.backward(); opt.step(); sched.step(); ema.update_parameters(model)
```

### A small experiment

To see the pieces working together, we trained a 78k-parameter ResNet ("TinyResNet": a stem, three basic blocks with 16, 32 and 64 channels, global average pooling) on scikit-learn's $8 \times 8$ digits, upsampled to $32 \times 32$ so that the strided stages have room to work. To make overfitting visible we used only **100 training images**, 500 images for calibration and 1,197 for testing, with AdamW (weight decay 0.05), batch size 32, 40 epochs and two epochs of warmup before a cosine decay. One run used no augmentation. The other used random rotation ($\pm 15°$), scaling and shifts (a random-resized-crop-like affine; no flips, since a flipped digit is a different symbol) together with label smoothing 0.1. All code is in `scripts/figures/dl_02_image_classification.py` and runs on a CPU in about a minute.

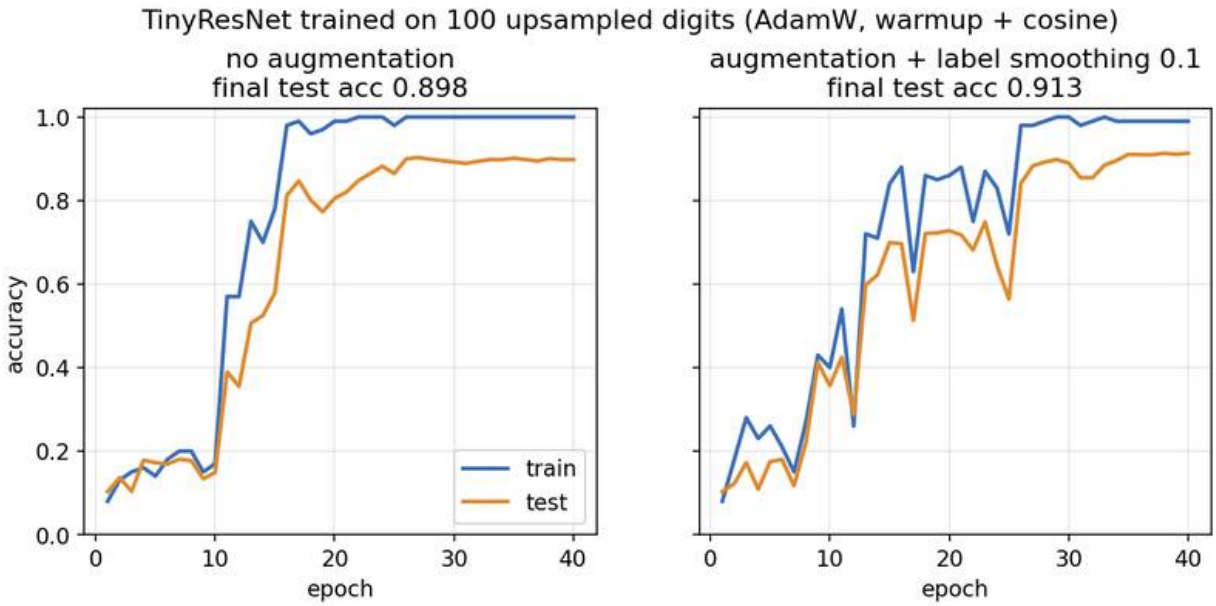



*Figure 7 — Training curves on 100 training digits. Without augmentation (left) the network memorizes*

*the training set and reaches 89.8% test accuracy. With augmentation and label smoothing (right) it reaches 91.3%. The gap is modest on such an easy task; on natural images the effect of the recipe is much larger.*

The no-augmentation model reaches 100% training accuracy after about twenty epochs: with 100 images, memorization is easy. The augmented model sees a slightly different version of each digit every epoch, learns more slowly, and generalizes a little better (91.3% against 89.8%). Do not read too much into 1.5 points on one seed; the point is the mechanism.

#### "ResNet strikes back"

The most striking evidence for recipes comes from Wightman, Touvron and Jégou [32]. They kept the 2015 ResNet-50 architecture fixed and retrained it with a modern recipe: random resized crops and flips, RandAugment, Mixup and CutMix, the LAMB optimizer with a cosine schedule, a binary cross-entropy loss and longer training. The vanilla ResNet-50 reached **80.4% top-1** on ImageNet at $224 \times 224$ without extra data or distillation, against 76.1% for the standard torchvision weights they compare with. Two consequences follow. First, many "new architecture beats ResNet-50" claims compared a new model with a modern recipe against an old model with an old recipe. Second, when you compare architectures, train them with the same, tuned recipe, as ConvNeXt [20] and RegNet [19] did.

### Transfer learning and fine-tuning

**Plain version.** Training a backbone from scratch needs a lot of data. Instead, start from a network already trained on a big dataset and adapt it to your task. You can either keep its features frozen and train only a new last layer, or gently retrain everything.

Let $g_\phi$ be a pre-trained backbone that maps an image to a feature vector $\mathbf{h} = g_\phi(\mathbf{x}) \in \mathbb{R}^D$, and $\mathbf{W} \in \mathbb{R}^{K \times D}$ a new linear head.

- **Linear probe.** Freeze $\phi$ and train only $\mathbf{W}$ (a logistic regression on fixed features). It is cheap, needs few labels, and is the standard way to *measure* how good a representation is.
- **Full fine-tuning.** Update $\phi$ and $\mathbf{W}$ together, usually with a learning rate for the backbone 10–100 times smaller than for the head, a short warmup, and sometimes layer-wise learning-rate decay (smaller rates for earlier layers). It adapts the features to the new domain and usually wins when there is enough labelled data.
- In between lie partial fine-tuning (only the last stages), and parameter-efficient methods that add small trainable modules to a frozen backbone.

Kornblith et al. [33] measured how ImageNet accuracy predicts transfer performance. For fine-tuning, better ImageNet models generally transferred better, with a strong correlation. For fixed features, the correlation was weaker and depended on regularization choices: some settings that improved ImageNet accuracy (for example, label smoothing and dropout) produced *worse* frozen features for other tasks. A good classifier and a good feature extractor are related, but not the same thing.

```
import torch
import torch.nn as nn

backbone = nn.Sequential(nn.Conv2d(3, 32, 3, 2, 1), nn.ReLU(), nn.Conv2d(32, 64, 3, 2, 1),
 ↪  nn.ReLU(),
                          nn.AdaptiveAvgPool2d(1), nn.Flatten())          # stand-in for a
                           ↪  pre-trained model
head = nn.Linear(64, 5)

# linear probe: freeze the backbone, train only the head
for p in backbone.parameters():
    p.requires_grad = False
probe_opt = torch.optim.AdamW(head.parameters(), lr=1e-3)

# full fine-tuning: unfreeze, use a smaller learning rate for the backbone than for the head
for p in backbone.parameters():
    p.requires_grad = True
ft_opt = torch.optim.AdamW([{"params": backbone.parameters(), "lr": 1e-5},
                            {"params": head.parameters(), "lr": 1e-3}], weight_decay=0.05)
```

## Self-supervised and foundation-model classifiers

**Plain version.** Labels are expensive; images are not. Self-supervised learning invents a task whose answer is in the image itself (are these two crops from the same photo? what was under this mask?), learns features by solving it, and only then uses a few labels.

### Contrastive learning: SimCLR and MoCo

Contrastive methods pull together the embeddings of two augmented **views** of the same image and push apart views of different images. In SimCLR [34], each view must pick out its partner among all the other views in the batch, a classification task whose "wrong answers" are the other images; strong augmentation (especially cropping plus color distortion) and large batches turned out to be essential. MoCo [35] gets many negatives without huge batches by keeping a queue of embeddings produced by a slowly updated copy of the encoder.

*Online only: Contrastive learning in detail: SimCLR and MoCo — https://blog.shyandram.dev/en/learn/deep-learning/image-classification/notes/contrastive-learning-in-detail-simclr-and-moco/*

### DINO and DINOv2: self-distillation

DINO [36] drops explicit negatives. A student network sees several crops (including small local ones) and is trained to match the output distribution of a teacher that sees global crops. The teacher is an EMA of the student; centering and sharpening of the teacher outputs prevent collapse to a constant. With ViTs, DINO features showed striking properties: the class token's attention maps segment objects without any segmentation labels, and a simple $k$-NN classifier on frozen

features works well. DINOv2 [38] scaled the approach with an automatically curated dataset and a large ViT distilled into smaller ones, producing general-purpose **frozen features** that work across classification, segmentation and depth without fine-tuning. DINOv3 [39] continued the scaling and introduced "Gram anchoring" to keep dense patch features from degrading during long training.

### MAE: masked image modeling

MAE [37] borrows the masked-language-model idea. It masks a large fraction of patches (75% in the paper), feeds only the visible patches to a ViT encoder, and asks a light decoder to reconstruct the missing pixels. Because the encoder skips masked tokens, pre-training is cheap. MAE features are less linearly separable than contrastive ones (linear probes are weaker), but they **fine-tune** very well. This is a recurring pattern: different pre-training objectives favour different adaptation methods.

### CLIP: zero-shot classification with language

CLIP [40] trains an image encoder $g$ and a text encoder $h$ on hundreds of millions of image–caption pairs with a symmetric contrastive loss, so that matching pairs have high cosine similarity. A classifier is then built **without any training images**: write a prompt per class, such as `a photo of a dog.`, embed it, and predict

$$\hat{y} = \arg\max_k \frac{g(\mathbf{x})^\top h(t_k)}{\|g(\mathbf{x})\| \, \|h(t_k)\|},$$

where $t_k$ is the prompt for class $k$. The text embeddings act as the weights of a linear classifier, computed from words. The paper reports that zero-shot CLIP matched the accuracy of a ResNet-50 trained on ImageNet's 1.28M labelled images, and that averaging several prompt templates ("a photo of a ...", "a blurry photo of a ...") helps. Zero-shot accuracy depends strongly on prompt wording, on whether the class names are visually meaningful, and on how well the web data covers the domain.

## Robustness and evaluation pitfalls

**Plain version.** A classifier that scores 90% on its test set may fail badly when the photos come from a different camera, when someone adds invisible noise on purpose, or when the "easy clue" it learned (backgrounds, watermarks) is missing. And even when it is right, its confidence may be wrong.

### Distribution shift: ImageNet-C, -R and -A

Hendrycks and colleagues built three widely used shifted test sets for ImageNet classifiers. **ImageNet-C** [42] applies 15 algorithmic corruptions (noise, blur, weather, digital artefacts)

at five severities and summarizes the error relative to a baseline (mean corruption error). **ImageNet-R** [43] collects renditions (art, cartoons, sculptures, sketches, toys) of 200 ImageNet classes and tests whether a model understands shape and concept rather than photographic texture. **ImageNet-A** [44] contains natural, unmodified photos that were filtered to fool a ResNet-50; standard models do very poorly on it. These sets reveal different weaknesses, and improving one does not guarantee improving the others. Large-scale pre-training (as in CLIP and DINOv2) tends to improve robustness to such natural shifts much more than architecture changes at fixed data.

### Adversarial examples: FGSM

Goodfellow et al. [45] explained adversarial examples by the **linearity** of networks in high dimensions: a tiny change to every input dimension, aligned with the weights, adds up to a large change in the output. The **fast gradient sign method** perturbs an input in one step:

$$\mathbf{x}_{\text{adv}} = \text{clip}\left(\mathbf{x} + \varepsilon \, \text{sign}(\nabla_{\mathbf{x}} \mathcal{L}(f_\theta(\mathbf{x}), y))\right),$$

where $\varepsilon$ bounds the change per pixel ($\ell_\infty$ norm), $\mathcal{L}$ is the loss and clip keeps pixels in the valid range.

```
import torch.nn.functional as F

def fgsm(model, x, y, eps):
    x = x.clone().requires_grad_(True)
    F.cross_entropy(model(x), y).backward()
    return (x + eps * x.grad.sign()).clamp(-1, 1).detach()
```

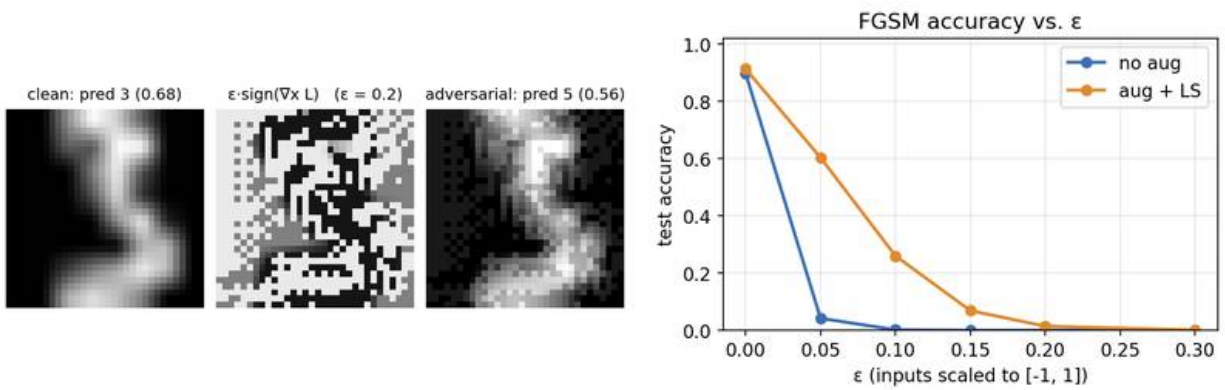



*Figure 8 — FGSM on the TinyResNet models (inputs in $[-1, 1]$). At $\varepsilon = 0.05$ test accuracy falls from 89.8% to 4.2% for the model trained without augmentation and from 91.3% to 60.3% for the model trained with augmentation and label smoothing; at $\varepsilon = 0.2$ both are near zero.*

Two cautions about Figure 8. First, the augmented model looks more robust at small $\varepsilon$, but label smoothing is known to flatten the loss surface around training points, which can weaken one-step gradient attacks without giving real robustness; stronger iterative attacks are the proper test.

Second, $\varepsilon = 0.05$ on a $[-1, 1]$ scale is visible on an 8×8 digit; on natural images, imperceptible perturbations already fool standard ImageNet models.

### Shortcut learning

Geirhos et al. [46] collect many failures under one name: **shortcut learning**. A network minimizes the training loss with whatever cue is easiest, which is often not the cue we intended: background instead of animal, hospital tag instead of pathology, texture instead of shape. Shortcuts look fine on an i.i.d. test set, because the same shortcut exists there. They fail under distribution shift. The prescription is to test out of distribution on purpose, design tests that break the suspected shortcut, and use interpretability tools such as Grad-CAM (below) to check what the model attends to.

### Calibration and temperature scaling

**Temperature scaling** [47] divides all logits by one scalar $T > 0$ before the softmax, $\mathbf{p} = \text{softmax}(\mathbf{z}/T)$. $T$ is fitted by minimizing the negative log-likelihood on a held-out **calibration** split (never the test set). $T > 1$ softens overconfident predictions; $T < 1$ sharpens underconfident ones. Because dividing by $T$ does not change the argmax, accuracy is unchanged.

```
import torch
import torch.nn.functional as F

def fit_temperature(logits, y, iters=300):
    """Learn one scalar T > 0 on a held-out calibration split."""
    log_t = torch.zeros(1, requires_grad=True)
    opt = torch.optim.Adam([log_t], lr=0.05)
    for _ in range(iters):
        loss = F.cross_entropy(logits / log_t.exp(), y)
        opt.zero_grad(); loss.backward(); opt.step()
    return log_t.exp().item()

def ece(probs, y, n_bins=10):
    conf, pred = probs.max(1)
    correct = (pred == y).float()
    edges = torch.linspace(0, 1, n_bins + 1)
    total = 0.0
    for lo, hi in zip(edges[:-1], edges[1:]):
        m = (conf > lo) & (conf <= hi)
        if m.any():
            total += m.float().mean() * (correct[m].mean() - conf[m].mean()).abs()
    return float(total)

# toy check: an over-confident "model" = true logits multiplied by 3
torch.manual_seed(0)
true_logits = torch.randn(4000, 10)
y = torch.distributions.Categorical(logits=true_logits).sample()
over = 3 * true_logits
T = fit_temperature(over[:2000], y[:2000])
print(f"T = {T:.2f}  ECE {ece(over[2000:].softmax(1), y[2000:]):.3f} -> "
```

```
      f"{ece((over[2000:] / T).softmax(1), y[2000:]):.3f}")
# T = 2.90  ECE 0.345 -> 0.018
```

In this toy check the logits are three times too large, and the fitted temperature (2.90) recovers that factor. On our real TinyResNet (the model trained without augmentation), the miscalibration went the *other* way.

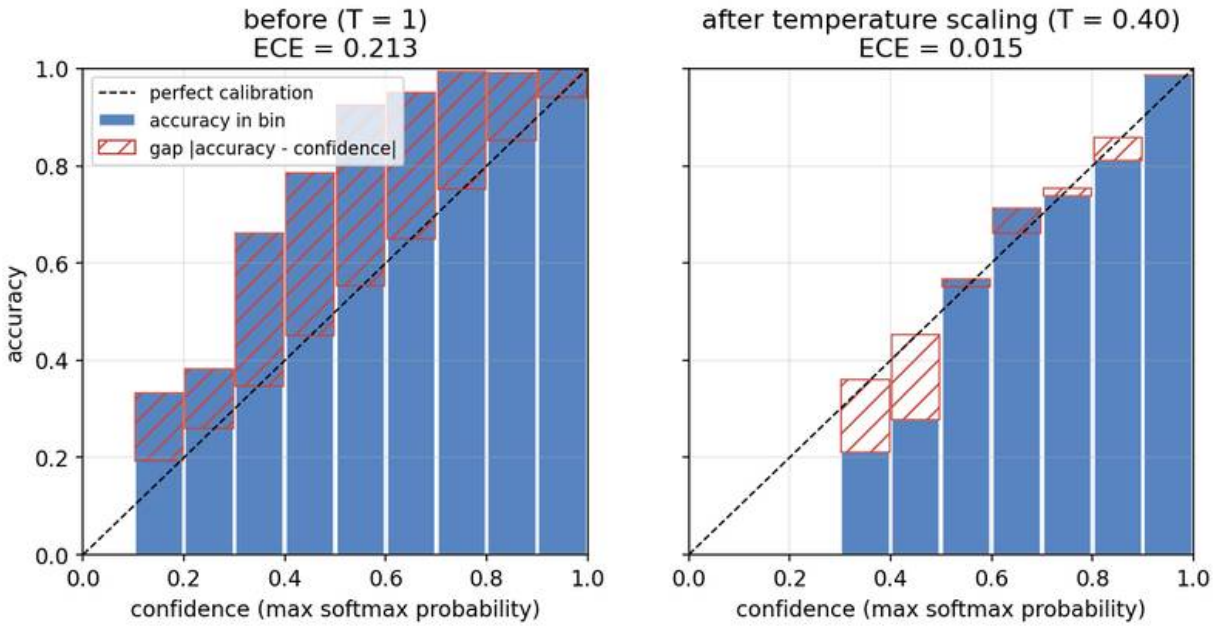



*Figure 9 — Reliability diagrams for the TinyResNet trained without augmentation, on 1,197 test digits. Before scaling (left), accuracy exceeds confidence in most bins: the model is underconfident (ECE 0.213). A temperature $T = 0.40$ fitted on 500 separate calibration images brings ECE to 0.015 and test NLL from 0.515 to 0.298, without changing any prediction.*

Why underconfident? Our tiny model was trained briefly with strong weight decay, which keeps logits small. Guo et al. [47] observed overconfidence in large, long-trained modern networks, and identified depth, width, batch normalization and *reduced* weight decay as contributing factors. Miscalibration has no fixed sign; measure it. Also note that calibration is itself fragile under distribution shift: a temperature fitted on in-distribution data does not guarantee calibrated confidence on ImageNet-C-style shifts.

## Efficiency

**Plain version.** A model must fit the device and the time budget. Parameter count, compute, and actual speed are three different measurements, and only the last one is what users feel.

- **Parameters** determine model size on disk and in memory. A float32 parameter takes 4 bytes.

- **FLOPs** (or multiply–accumulates, MACs; one MAC is usually counted as two FLOPs) measure arithmetic per image. For a convolution with $K \times K$ kernels, $C_{\text{in}}$ input and $C_{\text{out}}$ output channels on an $H' \times W'$ output map, MACs $= H'W'K^2C_{\text{in}}C_{\text{out}}$. Compute grows with the square of resolution.
- **Latency and throughput** are measured on target hardware. Depthwise convolutions have few FLOPs but low arithmetic intensity, so they often run slower on GPUs than their FLOP count suggests; DenseNet's concatenations cost memory bandwidth. Always benchmark.
- **Quantization** stores weights and activations as 8-bit integers instead of 32-bit floats. Jacob et al. [50] described integer-only inference with quantization-aware training, which simulates rounding during training so the network learns to tolerate it. Post-training quantization with a small calibration set is the cheap alternative.
- **Distillation** [51] trains a small student to match a large teacher's soft predictions, as DeiT did with its distillation token.

## Interpretability: CAM and Grad-CAM

**Plain version.** A heat map over the image shows which regions pushed the score for a class up. It is a sanity check, not a proof of reasoning.

**Class activation mapping** (CAM) [48] works for networks that end with global average pooling and one linear layer. If $A^k$ is the $k$-th feature map of the last convolutional layer and $w_k^c$ the weight connecting it to class $c$, then the class score is $\sum_k w_k^c \cdot \text{mean}(A^k)$, and the map $\sum_k w_k^c A^k$ shows where that score comes from. **Grad-CAM** [49] generalizes this to any differentiable architecture by replacing the weights with spatially averaged gradients:

$$\alpha_k^c = \frac{1}{Z}\sum_{i,j}\frac{\partial y^c}{\partial A_{ij}^k}, \qquad L_{\text{Grad-CAM}}^c = \text{ReLU}\Big(\sum_k \alpha_k^c A^k\Big),$$

where $y^c$ is the logit of class $c$, $A_{ij}^k$ the activation of channel $k$ at position $(i,j)$, and $Z$ the number of positions. The ReLU keeps only regions with a positive influence on the class. The coarse map ($8 \times 8$ in our network) is upsampled to the image size.

```
import torch.nn.functional as F

def grad_cam(model, x, target=None):
    model.eval()
    feats = model.features(x)                          # (1, K, h, w) last conv stage
    feats.retain_grad()
    logits = model.fc(feats.mean(dim=(2, 3)))
    c = logits.argmax(1).item() if target is None else target
    logits[0, c].backward()
    alpha = feats.grad.mean(dim=(2, 3), keepdim=True)  # channel importance weights
    cam = F.relu((alpha * feats).sum(1, keepdim=True))
    cam = F.interpolate(cam, size=x.shape[-2:], mode="bilinear", align_corners=False)
    return (cam / (cam.max() + 1e-8))[0, 0].detach(), c
```

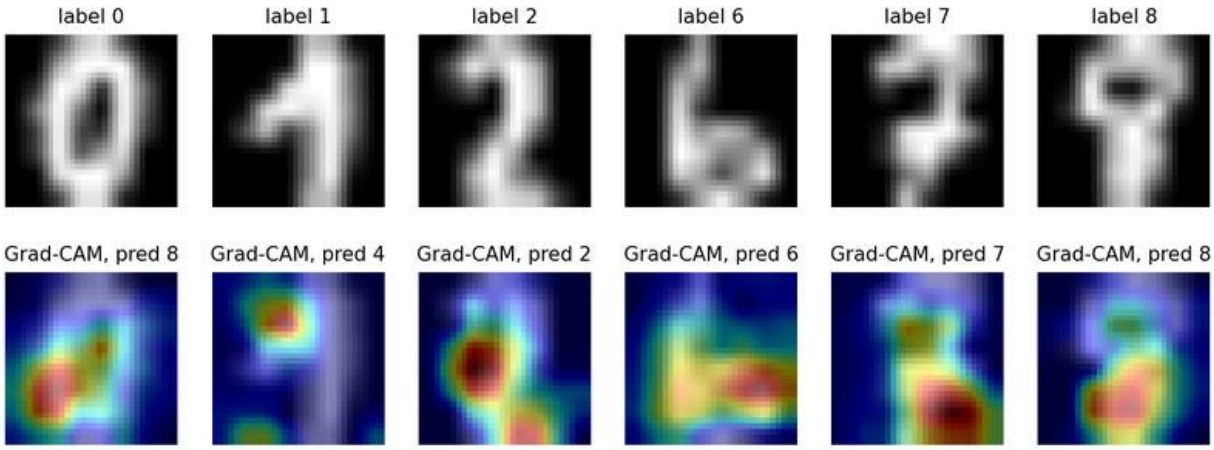



*Figure 10 — Grad-CAM for the predicted class on the first test example of six digits, using the augmented TinyResNet. Two are misclassified: the 0 is called an 8, with heat on the lower-left part of the loop, and the 1 is called a 4, with heat on the hook at its upper left. The maps point to where the misleading evidence was found.*

Explanations are most useful exactly when the model is wrong, as in Figure 10: they show which strokes looked like the wrong class. But treat saliency maps with care. They are coarse (limited by the last feature map's resolution), and they explain one class score for one input, not the model's general reasoning.

## Modern view

**Surveys worth reading.** Li et al. [1] review CNNs from basic building blocks to advanced models, with experiments that compare activation functions, losses and optimizers. Khan et al. [25] survey vision transformers and stress their reliance on large-scale pre-training and the cost of quadratic attention. Gui et al. [41] divide self-supervised learning into context-based, contrastive, generative and hybrid methods. On benchmark reliability, the ImageNetV2 [7], re-labelling [8] and label-error [9] studies together argue for multiple test sets, multi-label evaluation and reporting variance.

*Online only: Guide to the Modern view surveys — https://blog.shyandram.dev/en/learn/deep-learning/image-classification/papers/guide-to-the-modern-view-surveys/*

**The 2023–2026 picture.** Five trends define current practice.

1. **Pre-train once, adapt cheaply.** Most new classifiers are not trained from scratch. They start from a large backbone trained with self-supervision (DINOv2 [38], DINOv3 [39]), image–text contrast (CLIP [40] and open reproductions), or masked modeling (MAE [37]), and adapt it with a linear probe, a $k$-NN rule, or a short fine-tune.
2. **Zero-shot and open-vocabulary classification.** Image–text models turn the label set into text. The class list can change at test time without retraining, which blurs the line between classification and retrieval.

3. **Scaling laws for vision.** Zhai et al. [52] scaled ViTs to billions of parameters and characterized how error falls with compute, data and model size, including the observation that larger models are more sample-efficient in few-shot transfer. Cherti et al. [53] fitted reproducible power laws for open CLIP training on public data, and found that the scaling behaviour depends on the training data distribution and the downstream task.
4. **Architectures converged.** ConvNeXt [20] and Swin [24] show that convolutions and attention, given the same recipe and scale, land in a similar place. The choice is now driven by hardware, resolution and downstream tasks (dense prediction favours hierarchical designs) more than by top-1 accuracy.
5. **Evaluation moved beyond ImageNet-1k top-1.** Robustness suites (ImageNet-C/R/A [42]–[44] and ImageNetV2 [7]), calibration, and transfer to many downstream datasets are now standard parts of a backbone paper.

**Open problems.** Robustness to natural distribution shift still lags clean accuracy, and gains from robust training methods are often specific to the shift they target. Shortcut learning [46] is hard to detect without counterfactual tests. Calibration under shift is unsolved. Long-tailed and fine-grained recognition remain hard because the rare classes have the fewest examples. Benchmark saturation and label noise make small accuracy differences on ImageNet-1k uninformative. And the dominant foundation models are trained on web-scale data whose composition, licensing and biases are hard to audit, which matters directly for what a "zero-shot" classifier will and will not recognize.

## Key takeaways

- Use softmax for mutually exclusive labels and independent sigmoid outputs for multi-label tasks.
- Compare backbones under matched data, training and compute; architecture names alone do not explain score differences.
- Pretraining supplies reusable features; linear probing evaluates them, while fine-tuning changes them.
- Inspect confusion matrices and per-class results; state whether averages are macro or micro.
- Fit calibration on held-out data and evaluate both confidence and accuracy.
- Test shifted data, shortcuts and efficiency separately; report multiple seeds and tune only on validation data.

## Exercises

1. **Receptive fields and parameters.** Show that three stacked $3 \times 3$ convolutions (stride 1) have a $7 \times 7$ receptive field. With $C$ input and output channels and no biases, compare their parameter count with a single $7 \times 7$ convolution.

**Hint**

Each $3 \times 3$ layer adds 2 pixels to the receptive field: $3 \to 5 \to 7$. Parameters: $3 \cdot 9C^2 = 27C^2$ versus $49C^2$, about 45% fewer, plus two extra nonlinearities.

2. **Bottleneck arithmetic.** A ResNet bottleneck has $4d$ input channels and reduces to $d$. Count the weights (no biases) of the $1 \times 1$–$3 \times 3$–$1 \times 1$ branch as a function of $d$, and compare with a basic block of two $3 \times 3$ convolutions at width $4d$. What is the ratio?

   **Hint**

   Bottleneck: $4d \cdot d + 9d^2 + d \cdot 4d = 17d^2$. Basic block at width $4d$: $2 \cdot 9 \cdot 16d^2 = 288d^2$. The ratio is $17/288 \approx 0.059$, about 17 times fewer weights.

3. **Label smoothing optimum.** For one example with $K$ classes and smoothing $\varepsilon$, assume all wrong-class logits are equal. Show that the cross-entropy with smoothed targets is minimized when the logit gap between the true class and each wrong class is $\log\left((1 - \varepsilon + \varepsilon/K) \,/\, (\varepsilon/K)\right)$. Evaluate it for $K = 1000$, $\varepsilon = 0.1$.

   **Hint**

   The minimum of cross-entropy over $\mathbf{p}$ is at $\mathbf{p} = \mathbf{q}$. So $p_y = 1 - \varepsilon + \varepsilon/K$ and every other $p_k = \varepsilon/K$, and the logit gap is the log of their ratio: $\log(0.9001/0.0001) \approx 9.1$. Without smoothing the optimal gap is infinite.

4. **ViT token count.** A ViT with $16 \times 16$ patches receives a $224 \times 224$ image. How many tokens enter the encoder? If the image is fine-tuned at $384 \times 384$, how many tokens are there, by what factor does the attention-matrix size grow, and what must be done with the position embeddings?

   **Hint**

   $(224/16)^2 = 196$ patches plus one class token: 197. At 384: $24^2 + 1 = 577$. The attention matrix grows by $(577/197)^2 \approx 8.6$. The learned position embeddings form a $14 \times 14$ grid; they are interpolated (2-D) to $24 \times 24$, which is what the ViT paper does when fine-tuning at higher resolution.

5. **Temperature does not change accuracy.** Prove that for any $T > 0$, $\arg\max_k \mathrm{softmax}(\mathbf{z}/T)_k = \arg\max_k z_k$. Then, using `scripts/figures/dl_02_image_classification.py`, fit the temperature on the *test* set instead of the calibration split and compare the reported ECE. Why is that number optimistic?

   **Hint**

   Division by a positive constant and the exponential are both strictly increasing, and the softmax denominator is shared. Fitting $T$ on the test set uses test labels to tune a parameter, so the reported ECE is no longer an honest estimate of performance on new data.

6. **Build a shortcut.** Modify the digit experiment so that in the training set every "3" has a bright pixel in the top-left corner, but the test set does not. Train the TinyResNet, report test accuracy on the 3s, and look at Grad-CAM maps for a few training 3s. What do you expect, and how would you detect this problem if you did not know the shortcut was there?

   **Hint**

   The network can classify training 3s from the corner pixel alone, so test accuracy on 3s drops while training accuracy stays perfect, and Grad-CAM tends to highlight the corner. Without knowing the shortcut, look for heat maps that fall on background, evaluate on independently collected data, and run counterfactual tests (erase or move suspicious regions and see whether predictions change).

7. **Macro versus micro F1.** A three-class model is tested on 100 images. Rows are true classes, columns predictions: class A (90 images) → 88 A, 1 B, 1 C; class B (5 images) → 3 A, 2 B, 0 C; class C (5 images) → 4 A, 0 B, 1 C. Compute per-class precision, recall and $F_1$, then macro $F_1$ and micro $F_1$. Which number would you put in a report about detecting the rare classes B and C, and why?

   **Hint**

   Column sums are 95, 3 and 2. Precision: $88/95 = 0.926$, $2/3 = 0.667$, $1/2 = 0.500$. Recall: $88/90 = 0.978$, $2/5 = 0.400$, $1/5 = 0.200$. Using $F_1 = 2\text{TP}/(2\text{TP} + \text{FP} + \text{FN})$: $176/185 = 0.951$, $4/8 = 0.500$, $2/7 = 0.286$. Macro $F_1 = (0.951 + 0.500 + 0.286)/3 = 0.579$. Micro $F_1$ equals accuracy, $91/100 = 0.910$. Micro is dominated by class A; macro exposes that B and C are mostly missed, so macro (plus per-class recall) is the honest number for rare classes. You can check the result with `precision_recall_fscore_support(..., average="macro")`.

## References


1. Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022. doi:10.1109/TNNLS.2021.3084827
2. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, et al., "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, pp. 211–252, 2015. doi:10.1007/s11263-015-0816-y
3. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. doi:10.1109/5.726791
4. A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," technical report, University of Toronto, 2009. dataset page
5. T. Ridnik, E. Ben-Baruch, A. Noy, and L. Zelnik-Manor, "ImageNet-21K Pretraining for the Masses," *NeurIPS*, 2021. arXiv
6. G. Van Horn, O. Mac Aodha, Y. Song, Y. Cui, C. Sun, A. Shepard, et al., "The iNaturalist Species Classification and Detection Dataset," *CVPR*, 2018. arXiv
7. B. Recht, R. Roelofs, L. Schmidt, and V. Shankar, "Do ImageNet Classifiers Generalize to ImageNet?," *ICML*, PMLR vol. 97, 2019. paper

8. L. Beyer, O. J. Hénaff, A. Kolesnikov, X. Zhai, and A. van den Oord, "Are We Done with ImageNet?," arXiv:2006.07159, 2020. arXiv
9. C. G. Northcutt, A. Athalye, and J. Mueller, "Pervasive Label Errors in Test Sets Destabilize Machine Learning Benchmarks," *NeurIPS*, 2021. arXiv
10. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems 25* (*NIPS*), 2012. paper
11. K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *ICLR*, 2015. arXiv
12. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," *CVPR*, 2015. doi:10.1109/CVPR.2015.7298594
13. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," *CVPR*, 2016. doi:10.1109/CVPR.2016.308
14. K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *CVPR*, 2016. doi:10.1109/CVPR.2016.90
15. K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," *ECCV*, 2016. arXiv
16. G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," *CVPR*, 2017. arXiv
17. A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," arXiv:1704.04861, 2017. arXiv
18. M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," *ICML*, 2019. arXiv
19. I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár, "Designing Network Design Spaces," *CVPR*, 2020. arXiv
20. Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A ConvNet for the 2020s," *CVPR*, 2022. arXiv
21. J. Hu, L. Shen, S. Albanie, G. Sun, and E. Wu, "Squeeze-and-Excitation Networks," arXiv:1709.01507, 2017 (journal version in *IEEE TPAMI*). arXiv
22. A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *ICLR*, 2021. arXiv
23. H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training Data-Efficient Image Transformers & Distillation through Attention," *ICML*, PMLR vol. 139, 2021. paper
24. Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows," *ICCV*, 2021. doi:10.1109/ICCV48922.2021.00986
25. S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah, "Transformers in Vision: A Survey," *ACM Computing Surveys*, 2022. arXiv
26. H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "mixup: Beyond Empirical Risk Minimization," *ICLR*, 2018. arXiv
27. S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo, "CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features," *ICCV*, 2019. arXiv

28. E. D. Cubuk, B. Zoph, J. Shlens, and Q. V. Le, “RandAugment: Practical Automated Data Augmentation with a Reduced Search Space,” *CVPR Workshops*, 2020. doi:10.1109/CVPRW50498.2020.00359
29. I. Loshchilov and F. Hutter, “Decoupled Weight Decay Regularization,” *ICLR*, 2019. arXiv
30. I. Loshchilov and F. Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts,” *ICLR*, 2017. arXiv
31. G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, “Deep Networks with Stochastic Depth,” *ECCV*, 2016. doi:10.1007/978-3-319-46493-0_39
32. R. Wightman, H. Touvron, and H. Jégou, “ResNet Strikes Back: An Improved Training Procedure in timm,” arXiv:2110.00476, 2021. arXiv
33. S. Kornblith, J. Shlens, and Q. V. Le, “Do Better ImageNet Models Transfer Better?,” *CVPR*, 2019. arXiv
34. T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, “A Simple Framework for Contrastive Learning of Visual Representations,” *ICML*, 2020. arXiv
35. K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, “Momentum Contrast for Unsupervised Visual Representation Learning,” *CVPR*, 2020. arXiv
36. M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin, “Emerging Properties in Self-Supervised Vision Transformers,” *ICCV*, 2021. doi:10.1109/ICCV48922.2021.00951
37. K. He, X. Chen, S. Xie, Y. Li, P. Dollár, and R. Girshick, “Masked Autoencoders Are Scalable Vision Learners,” *CVPR*, 2022. doi:10.1109/CVPR52688.2022.01553
38. M. Oquab, T. Darcet, T. Moutakanni, H. Vo, M. Szafraniec, V. Khalidov, et al., “DINOv2: Learning Robust Visual Features without Supervision,” *Transactions on Machine Learning Research*, 2024. arXiv
39. O. Siméoni, H. V. Vo, M. Seitzer, F. Baldassarre, M. Oquab, C. Jose, et al., “DINOv3,” arXiv:2508.10104, 2025. arXiv
40. A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, et al., “Learning Transferable Visual Models From Natural Language Supervision,” *ICML*, PMLR vol. 139, 2021. paper
41. J. Gui, T. Chen, J. Zhang, Q. Cao, Z. Sun, H. Luo, and D. Tao, “A Survey on Self-Supervised Learning: Algorithms, Applications, and Future Trends,” *IEEE Transactions on Pattern Analysis and Machine Intelligence* (arXiv:2301.05712, 2023). arXiv
42. D. Hendrycks and T. Dietterich, “Benchmarking Neural Network Robustness to Common Corruptions and Perturbations,” *ICLR*, 2019. arXiv
43. D. Hendrycks, S. Basart, N. Mu, S. Kadavath, F. Wang, E. Dorundo, et al., “The Many Faces of Robustness: A Critical Analysis of Out-of-Distribution Generalization,” *ICCV*, 2021. arXiv
44. D. Hendrycks, K. Zhao, S. Basart, J. Steinhardt, and D. Song, “Natural Adversarial Examples,” *CVPR*, 2021. arXiv
45. I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and Harnessing Adversarial Examples,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2015. arXiv
46. R. Geirhos, J.-H. Jacobsen, C. Michaelis, R. Zemel, W. Brendel, M. Bethge, and F. A. Wichmann, “Shortcut Learning in Deep Neural Networks,” *Nature Machine Intelligence*, 2020. arXiv
47. C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, “On Calibration of Modern Neural Networks,” *ICML*, 2017. arXiv
48. B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, “Learning Deep Features for Dis-

criminative Localization," *CVPR*, 2016. doi:10.1109/CVPR.2016.319
49. R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization," *ICCV*, 2017. doi:10.1109/ICCV.2017.74
50. B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, et al., "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," *CVPR*, 2018. doi:10.1109/CVPR.2018.00286
51. G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," arXiv: 1503.02531, 2015. arXiv
52. X. Zhai, A. Kolesnikov, N. Houlsby, and L. Beyer, "Scaling Vision Transformers," *CVPR*, 2022. arXiv
53. M. Cherti, R. Beaumont, R. Wightman, M. Wortsman, G. Ilharco, C. Gordon, et al., "Reproducible Scaling Laws for Contrastive Language-Image Learning," *CVPR*, 2023. arXiv
54. T. Fawcett, "An Introduction to ROC Analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006. doi:10.1016/j.patrec.2005.10.010
55. J. Davis and M. Goadrich, "The Relationship Between Precision-Recall and ROC Curves," *Proc. 23rd International Conference on Machine Learning* (*ICML*), pp. 233–240, 2006. doi:10.1145/1143844.1143874
56. T. Saito and M. Rehmsmeier, "The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets," *PLOS ONE*, vol. 10, no. 3, e0118432, 2015. doi:10.1371/journal.pone.0118432
57. D. Chicco and G. Jurman, "The Advantages of the Matthews Correlation Coefficient (MCC) over F1 Score and Accuracy in Binary Classification Evaluation," *BMC Genomics*, vol. 21, article 6, 2020. doi:10.1186/s12864-019-6413-7
58. M. Sokolova and G. Lapalme, "A Systematic Analysis of Performance Measures for Classification Tasks," *Information Processing & Management*, vol. 45, no. 4, pp. 427–437, 2009. doi:10.1016/j.ipm.2009.03.002
59. J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960. doi:10.1177/001316446002000104
60. M.-L. Zhang and Z.-H. Zhou, "A Review on Multi-Label Learning Algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 8, pp. 1819–1837, 2014. doi:10.1109/TKDE.2013.39
61. M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes (VOC) Challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, 2010. doi:10.1007/s11263-009-0275-4
62. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, et al., "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. paper
63. H. Wang, S. Ge, E. P. Xing, and Z. C. Lipton, "Learning Robust Global Representations by Penalizing Local Predictive Power," *NeurIPS*, 2019 (introduces ImageNet-Sketch). arXiv
64. A. Barbu, D. Mayo, J. Alverio, W. Luo, C. Wang, D. Gutfreund, J. Tenenbaum, and B. Katz, "ObjectNet: A Large-Scale Bias-Controlled Dataset for Pushing the Limits of Object Recognition Models," *NeurIPS*, 2019. dataset page
65. C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie, "The Caltech-UCSD Birds-200-2011

Dataset," Technical Report CNS-TR-2011-001, California Institute of Technology, 2011. dataset page
66. J. Krause, M. Stark, J. Deng, and L. Fei-Fei, "3D Object Representations for Fine-Grained Categorization," *IEEE International Conference on Computer Vision Workshops* (*ICCVW*), pp. 554–561, 2013 (introduces Stanford Cars). doi:10.1109/ICCVW.2013.77
67. S. Maji, E. Rahtu, J. Kannala, M. Blaschko, and A. Vedaldi, "Fine-Grained Visual Classification of Aircraft," arXiv:1306.5151, 2013. arXiv
68. O. M. Parkhi, A. Vedaldi, A. Zisserman, and C. V. Jawahar, "Cats and Dogs," *CVPR*, 2012 (introduces Oxford-IIIT Pets). dataset page
69. M.-E. Nilsback and A. Zisserman, "Automated Flower Classification over a Large Number of Classes," *Indian Conference on Computer Vision, Graphics and Image Processing* (*ICVGIP*), 2008 (introduces Oxford Flowers-102). dataset page
70. L. Bossard, M. Guillaumin, and L. Van Gool, "Food-101 – Mining Discriminative Components with Random Forests," *ECCV*, 2014. dataset page
71. L. Fei-Fei, R. Fergus, and P. Perona, "One-Shot Learning of Object Categories," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2006 (the reference given for Caltech-101). dataset page
72. X. Zhai, J. Puigcerver, A. Kolesnikov, et al., "A Large-scale Study of Representation Learning with the Visual Task Adaptation Benchmark," arXiv:1910.04867, 2019. arXiv
73. Z. Liu, Z. Miao, X. Zhan, J. Wang, B. Gong, and S. X. Yu, "Large-Scale Long-Tailed Recognition in an Open World," *CVPR*, 2019 (introduces ImageNet-LT). arXiv

# DLCV 02A • Face Recognition and Metric Learning

How deep networks recognize people they never saw in training: verification and identification in closed and open sets, detection and landmark alignment, normalized embeddings, contrastive and triplet losses, margin-based softmax losses (SphereFace, CosFace, ArcFace), TAR at a fixed FAR, ROC and DET curves, CMC and FPIR, datasets and their withdrawal, demographic differentials, privacy and presentation attacks.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/face-metric-learning/

## What you'll learn

- Distinguish verification, closed-set identification and open-set identification.
- Trace detection, landmark alignment, normalized embeddings and similarity scoring.
- Compare pair-based losses and angular-margin classification losses.
- Choose thresholds and metrics for verification or gallery search.
- Evaluate dataset suitability, demographic differences, privacy and presentation attacks.

## The big picture

A classifier picks one of a fixed list of answers. A face recognition system cannot work that way: the people it will meet next year are not in its training set. Instead it learns a way to **measure how alike two faces are**, and then decides with a threshold.

> **Explain like I'm five**
>
> Imagine a guard at a school gate who has never met most of the visitors. The guard cannot memorize everyone in the world. Instead, each visitor shows a photo card, and the guard asks one question: "Does the face in front of me look enough like the face on this card?" Over many years the guard has learned which differences do not matter (a new haircut, glasses, a shadow) and which do (the shape of the eyes and nose). Face recognition models learn the same skill: turn every face into a short list of numbers, so that two photos of the same person give similar lists and photos of different people give different ones.

This idea, **metric learning**, is much wider than faces. The same recipe (learn an embedding, compare by distance, decide by a threshold) powers person re-identification (DLCV 07), image retrieval, speaker verification and the contrastive self-supervised methods of ML 06. Faces are where the recipe was pushed hardest, because the scale is extreme (millions of identities, error rates of one in a million) and the stakes are high (phone unlocking, border control, police searches).

Face recognition is also a technology with real social costs: errors fall unevenly on different groups, training data was often collected without consent, and some uses are restricted by law. This chapter treats those topics as part of the engineering, factually and with verified sources.

Two surveys frame the field: Wang and Deng's review of deep face recognition [1] and NIST's report on demographic effects in commercial algorithms [2]. We return to both in the Modern view.

**Related chapters.** The softmax, cross-entropy, ROC and calibration basics are in DLCV 02. Re-identification, the triplet loss for people and the CMC metric are in DLCV 07. Contrastive learning and InfoNCE are in ML 06, and image–text embeddings (CLIP) in ML 10. The classical view of nearest-neighbor matching and of open-set rejection is in PR 04 and PR 10.

## Recognition as comparison: the open-set problem

**Plain version.** Verification asks whether two faces belong to the same person; identification searches stored identities. Both can compare people unseen during training.

### Verification, identification, closed and open set

Let $x$ be a face image and $f_\theta(x) \in \mathbb{R}^d$ its embedding, produced by a network with parameters $\theta$. A comparison of two images $x_a, x_b$ produces a **similarity score**

$$s(x_a, x_b) = \cos \angle(f_\theta(x_a), f_\theta(x_b)) = \frac{f_\theta(x_a)^\top f_\theta(x_b)}{\|f_\theta(x_a)\| \, \|f_\theta(x_b)\|} \in [-1, 1],$$

where $\|\cdot\|$ is the Euclidean norm. People who have stored reference images are **enrolled**; their stored embeddings (one or several per person) form the **gallery**, and a new image to be recognized is a **probe**. A fixed **threshold** $t$ turns a score into a decision.

| Task | Question | Output | Typical use |
|---|---|---|---|
| Verification (1:1) | Is probe $x$ the claimed person $c$? | accept if $s(x, g_c) \geq t$, else reject | phone unlock, passport gate |
| Closed-set identification (1:N) | Which of the $N$ enrolled people is $x$? | $\arg\max_n s(x, g_n)$ | benchmarks; a known, closed group |
| Open-set identification (1:N) | Is $x$ enrolled, and if so, who? | best match if its score $\geq t$, else "not enrolled" | watch lists, photo search |
| Classification (DLCV 02) | Which training class is $x$? | one of the classes seen in training | not used for faces at test time |

Here $g_n$ is the gallery embedding of person $n$ and $N$ is the gallery size. Figure 1 draws the three recognition tasks. The important line is the last one: the classes of a face classifier are the *training* identities, and at test time almost every probe belongs to someone else. NIST's description of how deployed systems work makes the same point: a face recognition algorithm has no built-in notion of a particular person; it extracts a feature vector from people it has usually never seen and compares two vectors to produce a score [2].

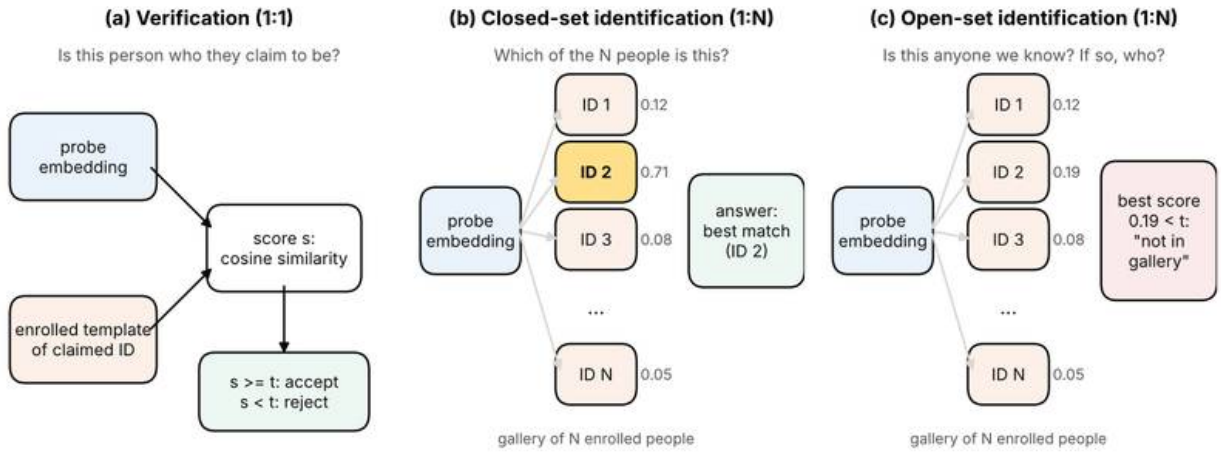



*Figure 1 — (a) Verification compares a probe with one claimed identity. (b) Closed-set identification always returns the best match. (c) Open-set identification also needs a threshold, because the person may not be enrolled at all.*

**Closed versus open set** is a property of the evaluation, not of the network. A closed-set identification test guarantees that every probe has a mate in the gallery, so "pick the best match" is always right in principle. An open-set test includes probes of people who are not enrolled, and then a threshold is unavoidable. Most real searches are open-set. The same distinction appears for generic classifiers as open-set recognition in PR 10; for faces it is the normal case, not an edge case.

### Why not just train a classifier?

**Plain version.** A classifier has outputs only for training identities. Recognition instead reuses its learned embedding to compare new identities.

A standard classifier trained with softmax and cross-entropy on $C$ identities already learns a useful penultimate layer. DeepFace [6], one of the first deep systems, trained a network on an identity-labeled dataset of more than 4,000 people and showed that the learned representation generalizes to faces in unconstrained photos; the output layer itself is not used for new people. FaceNet [7] instead trained the embedding directly with a triplet loss. The two lines (classification-style training and pair-based training) then merged into the margin-based softmax losses of 2017–2019 that we study below.

What makes a good embedding for open-set use? Two properties, stated by SphereFace as the ideal criterion: the **largest distance within a class** should be smaller than the **smallest distance between classes** [12]. A plain classifier only needs each training sample to fall on the correct side of a decision boundary; it does not care whether the classes are compact. Margins are how we ask for compactness.

### The face recognition pipeline

**Plain version.** Detect the face, locate landmarks, align its crop by rotation and scale, then compute the embedding.

1. **Detection** finds a box around each face.
2. **Landmark localization** finds a few keypoints, usually five: the two eye centers, the nose tip and the two mouth corners.
3. **Alignment** warps the image so that the landmarks match a fixed template, and crops a small, fixed-size image (for example $112 \times 112$ pixels).
4. **Embedding** maps the crop to a vector, usually $L_2$-normalized to unit length.
5. **Comparison** computes cosine similarities against one template (verification) or a gallery (identification) and applies a threshold.

### Detection and landmarks

Face detection is object detection specialized to one class, and the general ideas of DLCV 03 apply. The classical milestone was the Viola–Jones cascade, summarized in DLCV 03's classical detectors note. Two deep detectors are common front ends in the face-recognition literature, and both predict the five landmarks together with the box:

- **MTCNN** [4] is a cascade of three convolutional networks that predict face boxes and landmark positions from coarse to fine, trained with an online hard-sample mining strategy.
- **RetinaFace** [5] is a single-stage detector with multi-task heads; its authors manually annotated five landmarks on the WIDER FACE dataset and showed that this extra supervision helps detect hard faces.

These are not historical details: CosFace aligned its faces with MTCNN landmarks [13], and the ArcFace journal version with RetinaFace landmarks [16]. Detector and landmark errors propagate into everything that follows, so a recognition result is always a result *of the whole pipeline*.

### Alignment by a similarity transform

**Plain version.** We want every face to look straight at the camera, at the same size and position. A similarity transform (rotate, scale uniformly, shift) is just flexible enough to do this without distorting the face's shape.

A **similarity transform** maps a point $\mathbf{p} \in \mathbb{R}^2$ to

$$T(\mathbf{p}) = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \mathbf{p} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}, \qquad a = \sigma \cos\phi,\ b = \sigma \sin\phi,$$

where $\sigma > 0$ is the scale, $\phi$ the rotation angle and $(t_x, t_y)$ the translation. It has four parameters $(a, b, t_x, t_y)$, and the equations are *linear* in them. Given detected landmarks $\mathbf{p}_1, \dots, \mathbf{p}_5$ and template positions $\mathbf{q}_1, \dots, \mathbf{q}_5$, we choose the parameters that minimize $\sum_i \|T(\mathbf{p}_i) - \mathbf{q}_i\|^2$: ten equations, four

unknowns, one least-squares solve. Both CosFace and SphereFace describe their crops as obtained "by similarity transformation" from detected landmarks [12], [13].

```
import numpy as np

# A toy 5-point template (left eye, right eye, nose tip, left and right mouth
# corners) in a 112 x 112 crop. The coordinates are our own illustration.
template = np.array([[38., 46], [74, 46], [56, 66], [42, 86], [70, 86]])

def fit_similarity(src, dst):
    """Least-squares a, b, tx, ty with dst ~ [[a, -b], [b, a]] @ src + t."""
    n = len(src)
    A = np.zeros((2 * n, 4))
    A[0::2] = np.c_[src[:, 0], -src[:, 1], np.ones(n), np.zeros(n)]
    A[1::2] = np.c_[src[:, 1],  src[:, 0], np.zeros(n), np.ones(n)]
    a, b, tx, ty = np.linalg.lstsq(A, dst.reshape(-1), rcond=None)[0]
    return np.array([[a, -b, tx], [b, a, ty]])

# "Detected" landmarks: the template rotated by 20 degrees, scaled by 1.8,
# shifted into a large photo, plus 1-pixel detector noise.
rng = np.random.default_rng(0)
ang, sc = np.radians(20), 1.8
R = sc * np.array([[np.cos(ang), -np.sin(ang)], [np.sin(ang), np.cos(ang)]])
detected = template @ R.T + [310, 205] + rng.normal(scale=1.0, size=(5, 2))

M = fit_similarity(detected, template)          # photo -> crop
warped = detected @ M[:, :2].T + M[:, 2]
scale = np.hypot(M[0, 0], M[1, 0])
print(f"scale {scale:.3f} (expect {1 / sc:.3f}), "
      f"rotation {np.degrees(np.arctan2(M[1, 0], M[0, 0])):.1f} deg (expect -20.0)")
print(f"RMS landmark error after alignment: {np.sqrt(((warped - template) **
 ↪  2).sum(1).mean()):.2f} px")
```

```
scale 0.559 (expect 0.556), rotation -19.8 deg (expect -20.0)
RMS landmark error after alignment: 0.56 px
```

The fit recovers the scale and rotation almost exactly, and the remaining error (0.56 pixels) is what the 1-pixel landmark noise leaves after averaging over five points. The matrix `M` is then used to warp the whole image into the crop.

*Online only: Aligning a face: the least-squares similarity transform — https://blog.shyandram.dev/en/learn/deep-learning/face-metric-learning/notes/aligning-a-face-the-least-squares-similarity-transform/*

### The embedding and the template

The embedding network is an ordinary image backbone from DLCV 02 (a ResNet, or a vision transformer) whose last layer outputs a $d$-dimensional vector instead of class scores. FaceNet used $d = 128$ [7]; the ArcFace experiments use $d = 512$ with $112 \times 112$ crops [16]. When a person is enrolled with several images or a video, the embeddings are combined into one **template**; ArcFace

simply uses the feature center (mean) of all images in the template [16]. Since the network's output is normalized before comparison, it is natural to renormalize the mean as well.

### Embeddings on the hypersphere

**Plain version.** If we scale every embedding to length 1, all of them live on the surface of a sphere. Then "how alike are two faces" becomes "how small is the angle between their arrows". Length, which often reflects image quality rather than identity, no longer interferes.

### Why normalize, and why cosine

For unit vectors $\mathbf{u}, \mathbf{v}$ (that is, $\|\mathbf{u}\| = \|\mathbf{v}\| = 1$),

$$\|\mathbf{u} - \mathbf{v}\|^2 = \|\mathbf{u}\|^2 + \|\mathbf{v}\|^2 - 2\,\mathbf{u}^\top \mathbf{v} = 2 - 2\cos\theta,$$

where $\theta$ is the angle between them. So on the sphere, squared Euclidean distance and cosine similarity are the same ranking, and a threshold on one is a threshold on the other. Normalization removes one degree of freedom that a network can otherwise misuse: with an unnormalized softmax, a network can lower its loss on easy samples simply by making their features longer, without making the classes more compact. NormFace analyzed this and showed that training directly on normalized features needs an extra scale factor to converge [11].

High dimension adds a useful fact: **random directions are almost orthogonal**. If $\mathbf{u}$ and $\mathbf{v}$ are independent and uniformly distributed on the unit sphere in $\mathbb{R}^d$, then $\mathbb{E}[\cos\theta] = 0$ and the standard deviation of $\cos\theta$ is $1/\sqrt{d}$. Impostor scores therefore cluster tightly around 0, and the threshold only has to beat the tail of that narrow distribution.

```
import numpy as np

rng = np.random.default_rng(0)
e = rng.normal(size=(2000, 512))
e /= np.linalg.norm(e, axis=1, keepdims=True)       # 2,000 random unit vectors
a, b = e[0], e[1]
cos = a @ b
print(f"cos = {cos:.4f}   |a-b|^2 = {np.sum((a - b) ** 2):.4f}   2-2cos = {2 - 2 * cos:.4f}")

S = e @ e.T
off = S[np.triu_indices(len(e), 1)]                 # all distinct pairs
print(f"random pairs: mean cos {off.mean():.4f}, std {off.std():.4f}, "
      f"1/sqrt(512) = {1 / np.sqrt(512):.4f}, max {off.max():.3f}")
```

```
cos = 0.0472   |a-b|^2 = 1.9057   2-2cos = 1.9057
random pairs: mean cos -0.0000, std 0.0442, 1/sqrt(512) = 0.0442, max 0.214
```

Among almost two million random pairs in 512 dimensions, the largest cosine is only 0.214. A trained network does not produce random embeddings, but the picture holds: different people sit

at angles near 90°, and the job of training is to pull each person's images into a small cap around one direction.

### Pair-based losses: contrastive and triplet

**Plain version.** The most direct way to train an embedding is to show the network pairs or triples of images and say "these two should be close, those two should be far". The difficulty is that almost all such examples are already easy, so the network learns little from them unless we search for the hard ones.

#### Contrastive loss

The contrastive loss, introduced for face verification with a Siamese network by Chopra, Hadsell and LeCun [8] and developed by Hadsell, Chopra and LeCun [9], looks at one pair at a time:

$$\mathcal{L}_{\text{con}}(x_a, x_b) = y\, D^2 + (1-y)\,[\, m - D \,]_+^2, \qquad D = \|f_\theta(x_a) - f_\theta(x_b)\|,$$

where $y = 1$ if the pair shows the same person and $y = 0$ otherwise, $m > 0$ is a margin, and $[z]_+ = \max(z, 0)$. Same-person pairs are pulled together; different-person pairs are pushed apart only until they are $m$ apart. Up to constant factors and the label convention, this squared-hinge form is the one in [9]; [8] introduced the underlying idea of training a Siamese network so that a learned distance separates same-person from different-person pairs.

#### Triplet loss and hard-negative mining

The **triplet loss** of FaceNet [7] compares distances instead of thresholding them. Its definition, and its use in re-identification, are in DLCV 07 and ML 06; in FaceNet's form, with squared distances on unit-length embeddings,

$$\mathcal{L}_{\text{tri}}(a, p, n) = [\, \|f(a) - f(p)\|^2 - \|f(a) - f(n)\|^2 + \alpha \,]_+,$$

where $a$ is an anchor image, $p$ a positive (same person), $n$ a negative (another person), $f = f_\theta$ the embedding, and $\alpha > 0$ the margin. What this chapter adds is the **mining** question. For an anchor–positive pair with squared distance $d_{ap}$, a negative at squared distance $d_{an}$ is

- **easy** if $d_{an} \geq d_{ap} + \alpha$ (zero loss, zero gradient),
- **semi-hard** if $d_{ap} \leq d_{an} < d_{ap} + \alpha$ (farther than the positive, but inside the margin; this is FaceNet's definition [7]),
- **hard** if $d_{an} < d_{ap}$ (closer than the positive).

FaceNet selected semi-hard negatives online within large mini-batches, because the very hardest negatives early in training led to bad local minima [7]. A later and widely used alternative, **batch-hard** mining, takes for every anchor in a batch of $P$ identities × $K$ images the *farthest* positive and the *closest* negative [10].

```
import numpy as np

rng = np.random.default_rng(0)
P, Kimg, alpha = 8, 4, 0.2                  # 8 people x 4 images, margin
centres = rng.normal(size=(P, 64))
y = np.repeat(np.arange(P), Kimg)
e = centres[y] + 1.5 * rng.normal(size=(len(y), 64))
e /= np.linalg.norm(e, axis=1, keepdims=True)
d2 = ((e[:, None] - e[None]) ** 2).sum(-1)  # squared distances

easy = semi = hard = 0
for a in range(len(y)):
    for p in np.flatnonzero((y == y[a]) & (np.arange(len(y)) != a)):
        dn = d2[a, y != y[a]]               # distances to all negatives
        hard += (dn < d2[a, p]).sum()
        semi += ((dn >= d2[a, p]) & (dn < d2[a, p] + alpha)).sum()
        easy += (dn >= d2[a, p] + alpha).sum()
tot = easy + semi + hard
print(f"{tot} triplets: easy {easy / tot:.1%}, semi-hard {semi / tot:.1%}, hard {hard /
 ↪  tot:.1%}")

# batch-hard: for each anchor, farthest positive and closest negative
same = y[:, None] == y[None]
hp = np.where(same, d2, -np.inf).max(1)
hn = np.where(~same, d2, np.inf).min(1)
loss = np.maximum(hp - hn + alpha, 0)
print(f"batch-hard loss {loss.mean():.3f}; anchors with non-zero loss: {(loss > 0).sum()} of
 ↪  {len(y)}")
```

```
2688 triplets: easy 92.8%, semi-hard 5.4%, hard 1.9%
batch-hard loss 0.213; anchors with non-zero loss: 28 of 32
```

In this synthetic batch, 92.8% of all 2,688 valid triplets are easy and contribute nothing. Averaging over all triplets would dilute the few useful ones; batch-hard mining finds a non-zero loss for 28 of the 32 anchors. This is the core weakness of pair-based training at face scale: the number of possible triplets grows with the cube of the dataset size, almost all of them are easy, and the result depends on the sampling strategy and the batch composition.

### Margin-based softmax losses

**Plain version.** Instead of comparing images with each other, give every training identity a learned "anchor direction" on the sphere, and train each image to point at its own identity's direction *by a clear margin*: not just closer than to the other directions, but closer by a fixed angle. One loss term per image then compares it with all identities at once.

### From softmax to normalized softmax

Start from the softmax cross-entropy of DLCV 02. For a feature $\mathbf{x}_i \in \mathbb{R}^d$ of image $i$ with identity label $y_i \in \{1, \dots, C\}$, a last layer with weight columns $W_j \in \mathbb{R}^d$ and biases $b_j$ gives

$$\mathcal{L}_{\text{softmax}} = -\log \frac{e^{W_{y_i}^\top \mathbf{x}_i + b_{y_i}}}{\sum_{j=1}^{C} e^{W_j^\top \mathbf{x}_i + b_j}}.$$

Set $b_j = 0$ and write the logit as $W_j^\top \mathbf{x}_i = \|W_j\| \, \|\mathbf{x}_i\| \cos\theta_j$, where $\theta_j$ is the angle between $\mathbf{x}_i$ and $W_j$. Now normalize both: $\|W_j\| = 1$, and $\|\mathbf{x}_i\|$ fixed to a constant $s$. Then the logits are $s\cos\theta_j$ and the prediction depends on angles only [11], [13], [16]:

$$\mathcal{L}_{\text{norm}} = -\log \frac{e^{s\cos\theta_{y_i}}}{\sum_{j=1}^{C} e^{s\cos\theta_j}}.$$

The **scale** $s > 0$ is necessary because cosines lie in $[-1, 1]$: without it, even a perfect angular arrangement would leave the softmax far from confident, so the loss could not become small. NormFace analyzed this failure to converge [11], and CosFace derived a lower bound for $s$ that grows with the number of classes [13]; both CosFace and ArcFace use $s = 64$ [13], [16].

*Online only: How large must the scale s be? — https://blog.shyandram.dev/en/learn/deep-learning/face-metric-learning/notes/how-large-must-the-scale-s-be/*

Each column $W_j$ is now a unit vector: a **class center**, or **proxy**, for identity $j$ on the sphere. This is the key structural change compared with pair-based losses. Every image is compared with all $C$ class centers in every step, so there is nothing to mine, and the comparison is global rather than limited to one batch. The class centers exist only for training; at test time the last layer is discarded and only the embedding network is kept [16].

### Three ways to add a margin

The normalized softmax is satisfied when $\theta_{y_i} < \theta_j$ for all $j \neq y_i$: the feature is closer in angle to its own center than to any other. That puts the decision boundary exactly halfway between two centers, so samples near the boundary are correct but barely. A **margin** makes the target logit harder to achieve by modifying it from $s\cos\theta_{y_i}$ to $s\,\psi(\theta_{y_i})$ with $\psi(\theta) < \cos\theta$, while the other logits stay $s\cos\theta_j$:

$$\mathcal{L}_{\text{margin}} = -\log \frac{e^{s\,\psi(\theta_{y_i})}}{e^{s\,\psi(\theta_{y_i})} + \sum_{j \neq y_i} e^{s\cos\theta_j}}.$$

The three classic losses differ only in $\psi$:

| Loss | Target logit $\psi(\theta)$ | Margin type | Value in the paper |
|---|---|---|---|
| SphereFace (A-Softmax) [12] | $\cos(m\theta)$, extended to stay monotone (below) | multiplicative angular | $m = 4$ |
| CosFace (LMCL) [13]; also AM-Softmax [14] | $\cos\theta - m$ | additive cosine | $m = 0.35, s = 64$ |
| ArcFace [15], [16] | $\cos(\theta + m)$ | additive angular | $m = 0.5, s = 64$ |

Here $m$ is the margin (an integer for SphereFace, a cosine offset for CosFace and an angle in radians for ArcFace), and $\theta = \theta_{y_i}$ is the angle to the sample's own class center; the last column gives the values reported in the papers. Because $\cos(m\theta)$ is not monotone on $[0, \pi]$, SphereFace uses $\psi(\theta) = (-1)^k \cos(m\theta) - 2k$ for $\theta \in [k\pi/m, (k+1)\pi/m]$, $k \in \{0, \dots, m-1\}$ [12]. The original SphereFace normalized only the weights and needed help from an ordinary softmax early in training [16]. AM-Softmax [14], published at about the same time as CosFace, uses the same additive cosine margin.

```
import numpy as np

def sphereface_psi(theta, m=4):
    """SphereFace's monotone extension of cos(m*theta) to [0, pi]."""
    k = np.floor(theta * m / np.pi)
    return (-1) ** k * np.cos(m * theta) - 2 * k

print(" angle  softmax  SphereFace  CosFace  ArcFace")
for deg in [10, 30, 45, 60, 90]:
    t = np.radians(deg)
    print(f"{deg:5d}   {np.cos(t):6.3f}   {sphereface_psi(t):8.3f}   "
          f"{np.cos(t) - 0.35:6.3f}   {np.cos(t + 0.5):6.3f}")
```

```
 angle  softmax  SphereFace  CosFace  ArcFace
   10    0.985      0.766    0.635    0.781
   30    0.866     -0.500    0.516    0.520
   45    0.707     -1.000    0.357    0.282
   60    0.500     -1.500    0.150    0.024
   90    0.000     -3.000   -0.350   -0.479
```

The table shows the target logit (divided by $s$) at five angles. Every margin lowers the logit of the correct class, so the loss stays large until the angle is small. The penalties grow differently: CosFace subtracts the same 0.35 at every angle; ArcFace's penalty is small near $0°$ (0.204 at $10°$) and larger at wide angles; SphereFace with $m = 4$ is very strict at wide angles.

### The geometry on the circle

**Plain version.** Picture two class centers on a circle. Without a margin, the circle is split exactly halfway between them. With a margin, each class claims less than half, and a strip in between

belongs to nobody: a sample there still has a loss. Training pushes samples out of that strip, toward their own center.

For two classes with centers $W_1, W_2$ an angle $\varphi$ apart, and a feature on the arc between them, a sample of class 1 is "satisfied" when $\psi(\theta_1) > \cos\theta_2$ with $\theta_2 = \varphi - \theta_1$. Solving gives the class-1 boundary:

- normalized softmax: $\theta_1 < \varphi/2$ (no gap);
- SphereFace: $m\,\theta_1 < \theta_2$, so $\theta_1 < \varphi/(m+1)$; the gap between the two classes' boundaries is $\varphi(m-1)/(m+1)$, which **grows with** $\varphi$;
- CosFace: $\cos\theta_1 - m > \cos(\varphi - \theta_1)$; the gap is a fixed amount in *cosine*, so its size in angle depends on $\varphi$;
- ArcFace: $\theta_1 + m < \theta_2$, so $\theta_1 < (\varphi - m)/2$; the gap is exactly $m$ radians for every $\varphi$.

ArcFace's authors call this a margin equal to the **geodesic distance** on the hypersphere, which is where the name comes from [15], [16]. Figure 2 draws all four cases for $\varphi = 80°$; SphereFace is shown with $m = 2$ so that its zone fits on the same picture.

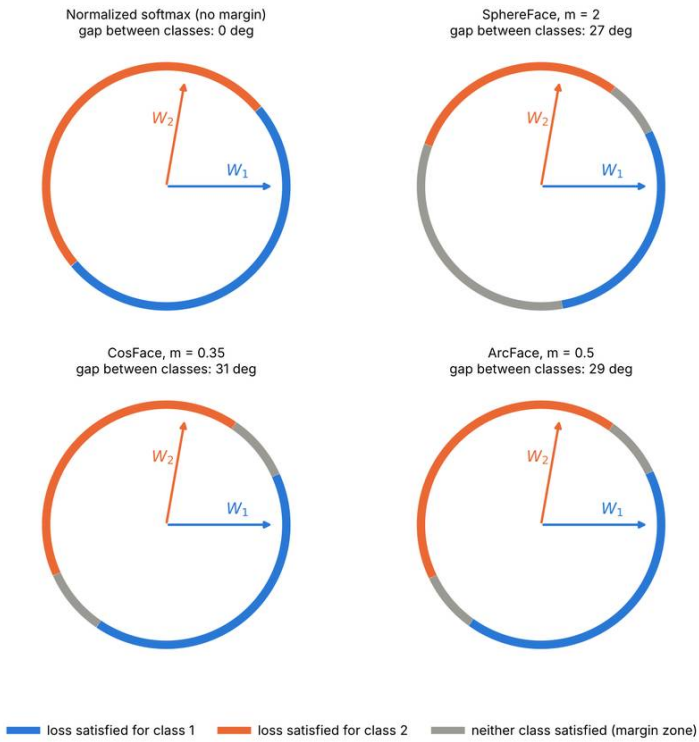



*Figure 2 — Where each loss is satisfied for two classes whose centers are 80° apart. Gray arcs are the margin zones, where a sample of either class still has a loss. The gap between the centers is 0° without a margin, 27° for SphereFace ($m = 2$), 31° for CosFace ($m = 0.35$) and 29° ($= 0.5$ rad) for ArcFace ($m = 0.5$).*

One practical detail follows from the formula: $\cos(\theta + m)$ decreases only while $\theta + m \le \pi$. For a sample whose angle to its own center is larger than $\pi - m$ (a badly misclassified or mislabeled sample), the ArcFace target logit would start to *increase* again as the sample moves away. An im-

plementation has to handle this range explicitly, for example by switching to a monotone penalty there. In our toy code below, all angles stay well below that range.

*Online only: Margin losses side by side: one formula and its edge cases — https://blog.shyandram.dev/en/learn/deep-learning/face-metric-learning/notes/margin-losses-side-by-side-one-formula-and-its-edge-cases/*

### A toy experiment: softmax versus ArcFace

**Plain version.** We train the same small network on synthetic "faces" twice: once with the ordinary softmax and once with ArcFace. Then we look at how tight the training clusters are and, more importantly, how well the network separates *people it never saw*.

The data mimic the face problem. Each synthetic identity is a random point in a 4-dimensional "identity" subspace of a 16-dimensional input; every image adds a large random "nuisance" component (standing in for pose and lighting) in the other 12 dimensions and small noise everywhere. The network is one hidden layer of 64 ReLU units followed by a linear map to a $d = 4$ embedding. We implement the forward pass and the gradients by hand, including the chain rule through the normalization and through $\cos(\theta + m)$, and use plain gradient descent. We train on 100 identities and test verification on 50 identities that never appear in training. TAR at FAR $= 1\%$ (defined in the next section) is the fraction of same-person test pairs accepted when the threshold lets through 1% of different-person pairs.

```
import numpy as np

rng = np.random.default_rng(0)
D, K = 16, 4                                        # input dim, identity dims
U = np.linalg.qr(rng.normal(size=(D, D)))[0]      # random orthonormal basis

def sample(ids, n_per):
    """Identity signal in 4 directions, large 'pose/light' nuisance in 12."""
    y = np.repeat(np.arange(len(ids)), n_per)
    nuis = 1.5 * rng.normal(size=(len(y), D - K))
    x = ids[y] @ U[:, :K].T + nuis @ U[:, K:].T + 0.3 * rng.normal(size=(len(y), D))
    return x, y

unit = lambda v, ax: v / np.linalg.norm(v, axis=ax, keepdims=True)

def train(x, y, C, d=4, margin=None, s=16.0, steps=800, lr=0.2, H=64):
    """One-hidden-layer embedding net + class centres W, plain gradient descent.
    margin=None: ordinary softmax on raw features; otherwise ArcFace logits."""
    r = np.random.default_rng(1)
    W1 = r.normal(size=(D, H)) / 4; b1 = np.zeros(H)
    W2 = r.normal(size=(H, d)) / 8; W = r.normal(size=(d, C))
    Y = np.eye(C)[y]
    for _ in range(steps):
        h = np.maximum(x @ W1 + b1, 0); z = h @ W2          # embeddings
        if margin is None:
            L = z @ W
        else:
```

```
            f, w = unit(z, 1), unit(W, 0)
            th = np.arccos(np.clip(f @ w, -1 + 1e-7, 1 - 1e-7))   # angles to centres
            L = s * np.where(Y > 0, np.cos(th + margin), np.cos(th))
        p = np.exp(L - L.max(1, keepdims=True)); p /= p.sum(1, keepdims=True)
        g = (p - Y) / len(y)                                      # dLoss/dlogits
        if margin is None:
            gz, gW = g @ W.T, z.T @ g
        else:                                                     # chain rule through
         ↪  cos(theta+m)
            gc = g * s * np.where(Y > 0, np.sin(th + margin) / np.sin(th), 1.0)
            gf, gw = gc @ w.T, f.T @ gc
            gz = (gf - (gf * f).sum(1, keepdims=True) * f) / np.linalg.norm(z, axis=1,
             ↪  keepdims=True)
            gW = (gw - (gw * w).sum(0, keepdims=True) * w) / np.linalg.norm(W, axis=0,
             ↪  keepdims=True)
        gh = (gz @ W2.T) * (h > 0)
        W1 -= lr * x.T @ gh; b1 -= lr * gh.sum(0); W2 -= lr * h.T @ gz; W -= lr * gW
    return lambda xx: unit(np.maximum(xx @ W1 + b1, 0) @ W2, 1)

def intra_angle(e, y):            # mean angle (degrees) to own class mean direction
    cen = unit(np.stack([e[y == j].mean(0) for j in np.unique(y)]), 1)
    return np.degrees(np.arccos(np.clip((e * cen[y]).sum(1), -1, 1))).mean()

def tar_at_far(e, y, far=0.01):   # verification on all pairs of unseen people
    i, j = np.triu_indices(len(y), 1)
    sc, same = (e[i] * e[j]).sum(1), y[i] == y[j]
    thr = np.quantile(sc[~same], 1 - far)
    return (sc[same] > thr).mean()

ids = rng.normal(size=(150, K))
xtr, ytr = sample(ids[:100], 10)          # 100 training identities
xte, yte = sample(ids[100:], 10)          # 50 identities never seen in training
for name, m in [("softmax", None), ("norm-softmax", 0.0), ("ArcFace m=0.5", 0.5)]:
    emb = train(xtr, ytr, 100, margin=m)
    print(f"{name:14s} train intra {intra_angle(emb(xtr), ytr):5.1f} deg | "
          f"unseen intra {intra_angle(emb(xte), yte):5.1f} deg | "
          f"unseen TAR@FAR=1% {tar_at_far(emb(xte), yte):.3f}")
```

```
softmax        train intra  27.1 deg | unseen intra  43.9 deg | unseen TAR@FAR=1% 0.082
norm-softmax   train intra  16.1 deg | unseen intra  30.1 deg | unseen TAR@FAR=1% 0.098
ArcFace m=0.5  train intra  11.6 deg | unseen intra  24.4 deg | unseen TAR@FAR=1% 0.248
```

The script runs in about ten seconds on a CPU. In this toy run the mean angle of a training image to its class center drops from 27.1° (softmax) to 16.1° (normalization alone) to 11.6° (ArcFace), and the clusters of *unseen* people tighten too (43.9° to 24.4°). The verification rate on unseen people at FAR = 1% roughly triples, from 0.082 to 0.248. Two cautions. First, the absolute numbers are low because a 4-dimensional embedding of noisy data is a hard problem; only the comparison matters. Second, these are single runs; with two other random seeds the ordering softmax < normalized softmax < ArcFace stayed the same in our runs, but the exact values changed.

Figure 3 shows the same effect in a 2-D embedding with six identities, where it can be seen directly: softmax spreads each class along a ray (it can lower its loss by lengthening features), while

ArcFace packs each class into a short arc.

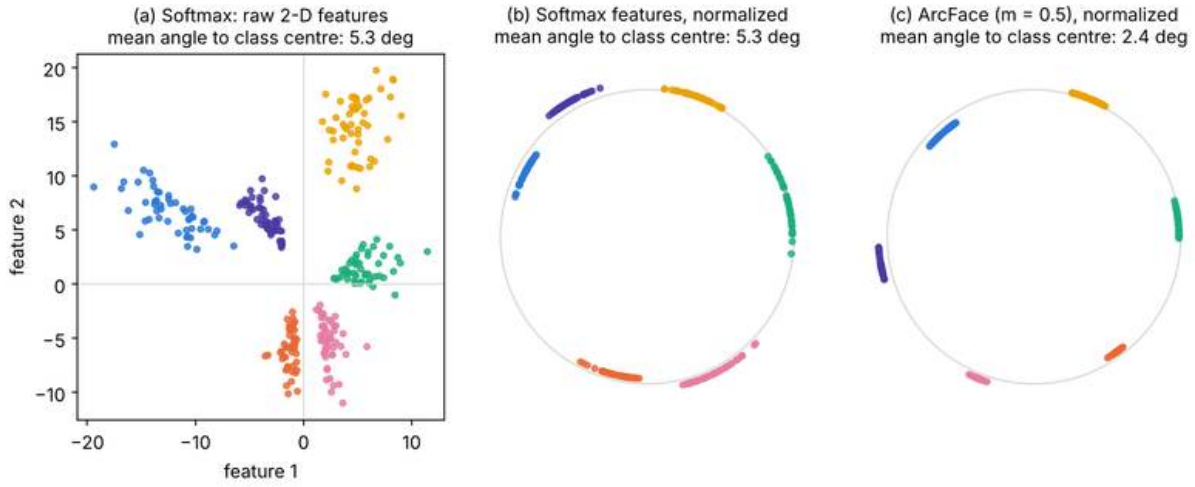



*Figure 3 — Six synthetic identities embedded in 2-D by the toy network. (a) Softmax features spread radially. (b) Normalized, the softmax classes cover wide arcs (mean angle to the class center 5.3°). (c) ArcFace with $m = 0.5$ gives tighter arcs (2.4°). Numbers are from this toy run.*

Margins tighten clusters of the training identities directly. Whether that transfers to new people depends on how many and how varied the training identities are; with very few training identities, a network can separate them perfectly and still learn nothing general. That is one reason why face recognition training sets contain hundreds of thousands to millions of identities (see the datasets section).

### Class centers, sub-centers and adaptive margins

**Plain version.** Once each identity has a center, several refinements follow naturally. They all keep the margin softmax and change one thing about the centers or the margin.

Center loss [17], which predates the angular losses, pulls each feature toward a learned center of its class. Sub-center ArcFace [18], [16] gives each identity several centers, so that noisy or mislabeled images can gather away from the clean ones. MagFace [19] and AdaFace [20] tie the margin to image quality, using the feature magnitude, so that very poor images do not dominate training. Partial FC [21] cuts the cost of millions of centers by updating only a sampled subset of them in each step. The one thing to remember: the class centers are a training device, and these methods tune how strictly and how cheaply images are pulled toward them.

*Online only: Beyond one center per class: sub-centers, quality-aware margins and Partial FC — https://blog.shyandram.dev/en/learn/deep-learning/face-metric-learning/notes/beyond-one-center-per-class-sub-centers-quality-aware-margins-and-partial-fc/*

### Evaluating verification

**Plain version.** Compare many pairs of photos of the same person (**genuine** pairs) and many pairs of different people (**impostor** pairs). A good system gives genuine pairs high scores and impostor pairs low scores. Any threshold then trades two errors: letting impostors in, and turning genuine users away.

### Genuine and impostor scores, FAR and TAR

For a threshold $t$ on the similarity score $s$,

$$\mathrm{FAR}(t) = P(s \geq t \mid \text{impostor}), \qquad \mathrm{FRR}(t) = P(s < t \mid \text{genuine}), \qquad \mathrm{TAR}(t) = 1 - \mathrm{FRR}(t).$$

FAR is the **false accept rate**, FRR the **false reject rate** and TAR the **true accept rate**. NIST's evaluations use the equivalent names **false match rate** (FMR) and **false non-match rate** (FNMR) for one-to-one comparisons [2], [3]. These are exactly the false-positive rate and the false-negative rate of DLCV 02; the "positive" class is "same person".

The two distributions are very unequal in size. With $P$ people and $k$ images each, there are $P\binom{k}{2}$ genuine pairs but about $\binom{Pk}{2} - P\binom{k}{2}$ impostor pairs, which grows with the square of $P$. Because impostor pairs are so many, operating points are stated as **TAR at a fixed FAR**: "TAR@FAR = $10^{-4}$" is the fraction of genuine pairs accepted when the threshold is set so that one impostor pair in ten thousand is accepted. Benchmarks such as IJB-B and IJB-C report TAR@FAR = $10^{-4}$ [16], and NIST's ongoing one-to-one evaluation ranks algorithms by FNMR at FMR = $10^{-6}$ on one of its datasets [3].

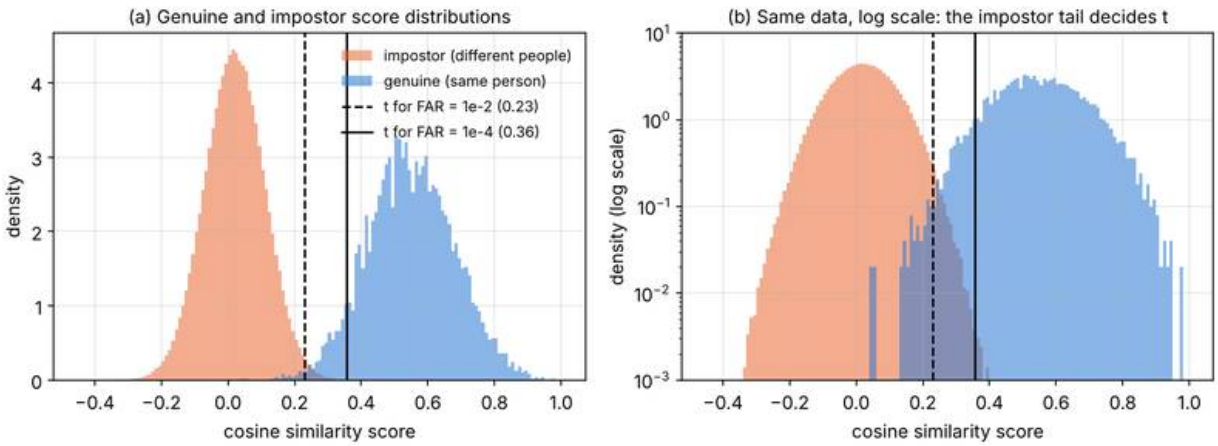



*Figure 4 — Synthetic genuine (blue) and impostor (orange) score distributions. (a) Linear scale. (b) Log scale: the threshold for a small FAR is set by the far tail of the impostor distribution, which is invisible on a linear axis.*

```
import numpy as np
from sklearn.metrics import roc_curve
```

```
rng = np.random.default_rng(0)
genuine = np.clip(rng.normal(0.55, 0.13, 5_000), -1, 1)       # same-person cosines
impostor = np.clip(rng.normal(0.02, 0.09, 500_000), -1, 1)    # different-person cosines
y = np.r_[np.ones(genuine.size), np.zeros(impostor.size)]
fpr, tpr, thr = roc_curve(y, np.r_[genuine, impostor])

for far in [1e-2, 1e-3, 1e-4]:
    i = np.searchsorted(fpr, far, side="right") - 1          # last point with FPR <= far
    print(f"FAR <= {far:.0e}: threshold {thr[i]:.3f}, TAR {tpr[i]:.3f}, "
          f"false accepts {int(round(fpr[i] * impostor.size))}")
i = np.argmin(np.abs(fpr - (1 - tpr)))
print(f"EER ~ {(fpr[i] + 1 - tpr[i]) / 2:.4f} at threshold {thr[i]:.3f}")
```

```
FAR <= 1e-02: threshold 0.231, TAR 0.994, false accepts 4825
FAR <= 1e-03: threshold 0.300, TAR 0.975, false accepts 495
FAR <= 1e-04: threshold 0.356, TAR 0.936, false accepts 49
EER ~ 0.0072 at threshold 0.241
```

The **equal error rate** (EER) is the error at the threshold where FAR equals FRR; it is a convenient single number but describes an operating point that a secure system rarely uses. Notice how the threshold moves: lowering FAR from $10^{-2}$ to $10^{-4}$ raises the threshold from 0.231 to 0.356 and costs 5.8 points of TAR. Notice also how few errors the last line is based on: 49 false accepts. A FAR of $10^{-6}$ cannot be measured at all with 500,000 impostor pairs; you need millions of them, and they must be reasonably independent.

*Online only: Measuring very small error rates — https://blog.shyandram.dev/en/learn/deep-learning/face-metric-learning/notes/measuring-very-small-error-rates/*

### ROC and DET curves

The ROC curve of DLCV 02 plots TAR against FAR. For face verification a **logarithmic FAR axis** is often useful: operating regions such as FAR between $10^{-6}$ and $10^{-2}$ would be squeezed into the left edge on a linear axis. The axis scale is a presentation choice, not part of the definition of ROC; zero FAR cannot be plotted on a logarithmic axis. The **detection error trade-off (DET) curve** [22] plots both error rates, FRR against FAR, with both axes transformed by the inverse of the standard normal cumulative distribution function $\Phi^{-1}$ (the "normal deviate" scale). If genuine and impostor scores are both Gaussian, the DET curve is a straight line, and lower-left is better. Martin et al. introduced it for speaker recognition and explained why it tends to be close to linear [22]; it is now standard for biometric systems.

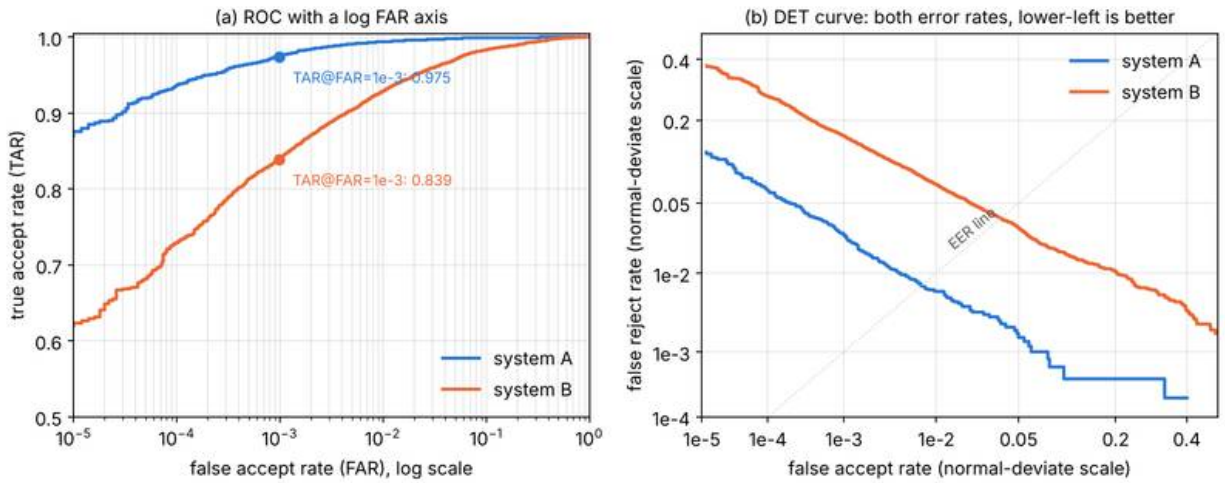



*Figure 5 — (a) ROC with a log FAR axis; dots mark TAR at FAR = $10^{-3}$. (b) DET curves of the same two synthetic systems: both error rates on normal-deviate axes. The EER is where a curve crosses the dotted diagonal.*

### Choosing the threshold

The threshold is part of the system, not of the network, and it must be chosen on data that represent deployment. Three rules follow from the formulas above. (1) Fix the FAR you can tolerate from the application (a phone unlock and a forensic search have very different costs), then choose $t$ as the corresponding quantile of impostor scores on a **held-out** set; tuning $t$ on the test set inflates results. (2) Report TAR and FAR *together* at that threshold, never one alone. (3) Check that the impostor scores of your deployment look like those of your calibration set: a new camera, a new population or a new image quality changes the impostor distribution and therefore the real FAR at a fixed $t$. We return to the last point in the fairness section.

## Evaluating identification

**Plain version.** In a search, the probe is compared with every enrolled person. In a closed-set test we ask: was the right person ranked first (or in the top $k$)? In an open-set test we also ask: when the person is not enrolled at all, did the system stay silent?

### Closed set: rank-1 and the CMC curve

For closed-set identification, sort the gallery by similarity to the probe. **Rank-1 accuracy** is the fraction of probes whose mate is ranked first, and the **cumulative match characteristic** CMC($k$) is the fraction whose mate is within the top $k$. The formula and code are in DLCV 07; a face gallery usually holds one template per person, so mAP is rarely used.

The crucial effect is **gallery size**. Each extra enrolled person is one more chance for an impostor to outscore the true mate. MegaFace was built to measure this: it added up to one million distractor photos to the gallery and reported identification and verification as the number of distractors grew from 10 to 1M [24].

### Open set: FNIR and FPIR

In an open-set search, the system returns a candidate only if its score reaches a threshold $t$. Two error rates describe it [2]:

$$\mathrm{FNIR}(t) = \frac{\#\{\text{mated searches whose mate is not returned with score} \geq t\}}{\#\{\text{mated searches}\}}, \qquad \mathrm{FPIR}(t) = \frac{\#\{\text{non-mated searches returning any candidate with score} \geq t\}}{\#\{\text{non-mated searches}\}}.$$

A **mated** search is one whose probe person is enrolled; a **non-mated** search is one whose probe person is not. FNIR (false negative identification rate) and FPIR (false positive identification rate) play the roles of FRR and FAR for one-to-many search; in this simple version a candidate is returned only at rank 1. If every one of the $N$ gallery comparisons of a non-mated probe were an independent trial with false match rate $\mathrm{FMR}(t)$, then

$$\mathrm{FPIR}(t) \approx 1 - (1 - \mathrm{FMR}(t))^{N} \approx N \cdot \mathrm{FMR}(t) \quad \text{for small } N \cdot \mathrm{FMR}(t),$$

so at a fixed threshold FPIR grows roughly in proportion to the gallery size. This is why one-to-many search needs far lower one-to-one false match rates than verification.

```
import numpy as np

rng = np.random.default_rng(0)
dim, noise = 128, 1.1
unit = lambda v: v / np.linalg.norm(v, axis=-1, keepdims=True)

def identify(n_gallery, n_probe=300, thr=0.36):
    people = rng.normal(size=(n_gallery + n_probe, dim))
    gallery = unit(people[:n_gallery] + noise * rng.normal(size=(n_gallery, dim)))
    mated = rng.choice(n_gallery, n_probe, replace=False)             # enrolled people
    p_in = unit(people[mated] + noise * rng.normal(size=(n_probe, dim)))
    p_out = unit(people[n_gallery:] + noise * rng.normal(size=(n_probe, dim)))  # not
     ↪  enrolled
    S_in, S_out = p_in @ gallery.T, p_out @ gallery.T
    rank1 = (S_in.argmax(1) == mated).mean()
    # open set: a search "returns" the top candidate only if its score >= thr
    fnir = 1 - ((S_in.argmax(1) == mated) & (S_in.max(1) >= thr)).mean()
    fpir = (S_out.max(1) >= thr).mean()
    return rank1, fnir, fpir

for n in [300, 3_000, 30_000]:
    r1, fnir, fpir = identify(n)
    print(f"gallery {n:6d}: rank-1 {r1:.3f} | threshold 0.36: FNIR {fnir:.3f}, FPIR
     ↪  {fpir:.3f}")
```

```
gallery    300: rank-1 0.990 | threshold 0.36: FNIR 0.090, FPIR 0.010
```

```
gallery   3000: rank-1 0.953 | threshold 0.36: FNIR 0.113, FPIR 0.043
gallery  30000: rank-1 0.903 | threshold 0.36: FNIR 0.117, FPIR 0.330
```

With the same synthetic embedding quality, growing the gallery from 300 to 30,000 people lowers rank-1 from 0.990 to 0.903, and at a fixed threshold the share of non-enrolled probes that trigger a false hit rises from 1% to 33%. Figure 6 shows the CMC curves and the FPIR-versus-threshold curves of a similar run. An identification result is meaningless without the gallery size, and an open-set system must be re-thresholded when the gallery grows.

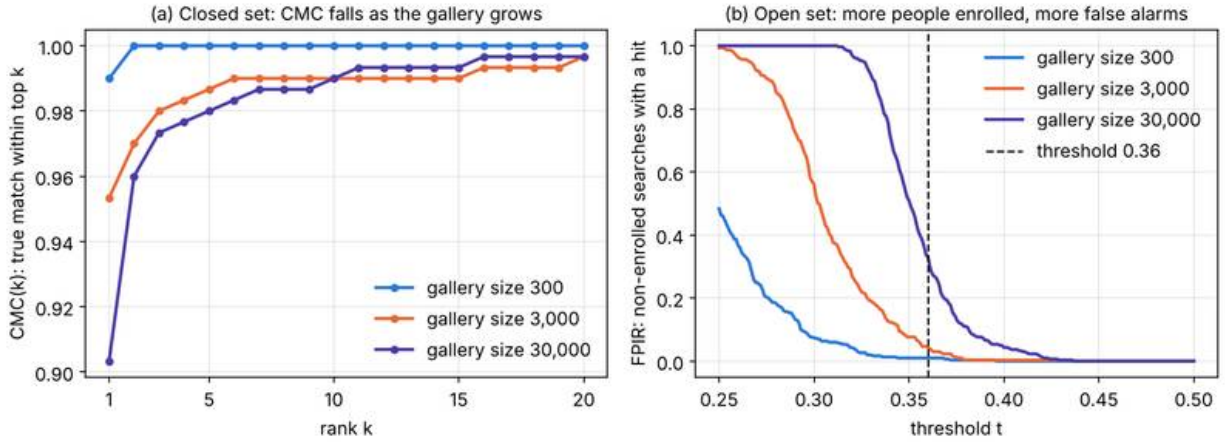



*Figure 6 — A synthetic search experiment. (a) Closed-set CMC drops as the gallery grows. (b) Open-set FPIR at each threshold: at a fixed threshold, more enrolled people means more false alarms for non-enrolled probes.*

## Datasets and benchmarks

**Plain version.** Training sets for faces are huge lists of web photos grouped by person; test sets define pairs or searches with a fixed protocol. Several of the best-known sets were collected from the web without the consent of the people in them, and some have been withdrawn.

| Dataset | Role | What it contains (as described by its source) | Status |
|---|---|---|---|
| LFW [23] | verification test | unconstrained news photos; about 13K photos of 5K people [24] | saturated; a disclaimer added in 2019 warns against concluding commercial suitability from it [30] |
| IJB-B [27] | verification, identification, clustering | 1,845 subjects; 21,798 still images and 55,026 frames from 7,011 videos, with human-labeled boxes and eye/nose locations | standard test set; media under a Creative Commons license [27] |

| Dataset | Role | What it contains (as described by its source) | Status |
|---|---|---|---|
| IJB-C [28] | as IJB-B, larger | 3,531 subjects; 31.3K still images and 117.5K frames from 11,779 videos [16] | standard test set [16] |
| MegaFace [24], [25] | identification at scale | 1M distractor photos of more than 690K individuals; a later training set of 4.7M photos of 672,057 identities | **decommissioned**; data no longer distributed [25] |
| MS-Celeb-1M [26] | training | web images for a benchmark of recognizing one million celebrities linked to a knowledge base | **taken down in April 2019** [30] |
| WebFace260M / WebFace42M [29] | training | 4M identities / 260M faces (noisy) and a cleaned 2M identities / 42M faces | described by its authors as the largest public training set [29] |
| DigiFace-1M [31] | training | synthetic faces rendered with a computer-graphics pipeline | synthetic, not photographs of real people |

**LFW** (Labeled Faces in the Wild) [23] defined unconstrained face verification and its standard protocols. Its accuracy is now saturated: state-of-the-art models exceed 99.8% on it [31], so it no longer separates good systems from very good ones. **IJB-B** and **IJB-C** (IARPA Janus Benchmarks) [27], [28] replaced it as the standard academic test: they include video frames, large pose variation and protocols at operationally relevant points such as TAR@FAR = $10^{-4}$, and they evaluate *templates* (sets of images) rather than single images. **MegaFace** [24] made gallery size the central question. **NIST's evaluations** [2], [3] use large operational datasets that are not public and test submitted algorithms directly, so they are the reference for deployment-scale error rates.

**Withdrawn datasets.** MS-Celeb-1M was taken down by Microsoft in April 2019; Peng, Mathur and Narayanan note that the removal followed critical reports, that many of the "celebrities" were fairly ordinary people, that the images had been aggregated without consent, and that Microsoft gave as its reason that the research challenge was over [30]. MegaFace's own site states that the challenge has concluded and that the data are no longer distributed [25]. The same study found that retracted data often keep circulating through copies, cleaned derivatives and pretrained models, and that licenses did little to restrict such use [30]; the DukeMTMC case in DLCV 07 is similar. In practice: do not use withdrawn datasets or their derivatives, record the provenance and license of every training set, and expect reviewers to ask.

**Synthetic training data** is one response. DigiFace-1M renders faces with a computer-graphics pipeline and motivates this with bias, label noise and the lack of consent in web-crawled data [31]; DCFace generates identities and styles with a diffusion model [32]. Boutros et al. survey this direction and its open challenges [33].

### Fairness and demographic differentials

**Plain version.** A system can have the same overall accuracy as another and still make many more errors for some groups of people than for others. Because a deployed system usually uses *one* threshold for everybody, small differences in score distributions between groups can turn into large differences in error rates.

#### What the measurements show

The most extensive public measurement is NIST's 2019 report on demographic effects [2]. It processed 18.27 million images of 8.49 million people from four U.S. government datasets through 189 mostly commercial algorithms from 99 developers. Its main findings, in the report's own framing:

- **False positives** differed far more than false negatives: across demographic groups, false positive rates often varied by factors of 10 to beyond 100, while false negatives were more algorithm-specific and often varied by factors below 3.
- The pattern **depended on the algorithm and the data**. For example, with high-quality application photos the highest false positive rates were in West and East African and East Asian people for many algorithms, but with a number of algorithms developed in China the East Asian effect was reversed. False positives were also higher for women than for men and elevated for the elderly and children.
- **Image quality** mattered for false negatives: results differed between well-controlled mugshots and lower-quality border-crossing photos.
- Some highly accurate one-to-many algorithms showed **undetectable** false positive differentials, so differentials are not inevitable.
- The report recommends that every demographic claim state which error (false positive or false negative) is meant, at a fixed threshold, for each group, because many earlier reports did not [2].

Academic studies add complementary evidence. Wang et al. built the Racial Faces in-the-Wild (RFW) test set and measured racial bias in commercial APIs and research algorithms [34]. The widely cited Gender Shades study [35] measured error rates of up to 34.7% for darker-skinned women versus at most 0.8% for lighter-skinned men; note that it evaluated commercial **gender classification**, not face recognition, a distinction NIST explicitly asks reporters to make [2]. Drozdowski et al. survey bias estimation and mitigation across biometrics [36].

#### A toy: one threshold, two groups

The mechanism is simple enough to simulate. Assume two groups A and B with identical genuine score distributions, but B's impostor scores are slightly higher: different members of B look a little more alike to the network. In this toy we simply assume the shift; in real systems its causes have to be investigated. Set one global threshold for FMR = $10^{-3}$ on the pooled impostors.

```
import numpy as np

rng = np.random.default_rng(0)
n_imp, n_gen = 400_000, 4_000
# Same genuine distribution; group B's impostor scores sit slightly higher.
imp = {"A": rng.normal(0.00, 0.09, n_imp), "B": rng.normal(0.06, 0.09, n_imp)}
gen = {"A": rng.normal(0.55, 0.13, n_gen), "B": rng.normal(0.55, 0.13, n_gen)}

pooled = np.r_[imp["A"], imp["B"]]
t_global = np.quantile(pooled, 1 - 1e-3)          # FMR = 0.1% on the pooled data
print(f"global threshold {t_global:.3f}")
for g in "AB":
    print(f"  group {g}: FMR {(imp[g] >= t_global).mean():.5f}, "
          f"FNMR {(gen[g] < t_global).mean():.4f}")
for g in "AB":
    t_g = np.quantile(imp[g], 1 - 1e-3)             # per-group threshold, FMR 0.1% each
    print(f"group {g} own threshold {t_g:.3f}: FNMR {(gen[g] < t_g).mean():.4f}")
```

```
global threshold 0.322
  group A: FMR 0.00018, FNMR 0.0403
  group B: FMR 0.00182, FNMR 0.0400
group A own threshold 0.279: FNMR 0.0210
group B own threshold 0.337: FNMR 0.0517
```

A shift of 0.06 in the mean impostor score, two-thirds of one standard deviation, makes group B's false match rate 10 times group A's at the same threshold (0.182% against 0.018%), although the two groups' false non-match rates are equal. Choosing a separate threshold per group equalizes FMR, but then group B's FNMR rises to 5.2% while A's falls to 2.1%: one error rate or the other differs. Which trade-off is acceptable is a policy and legal question, not a purely technical one, and group-specific thresholds raise their own concerns. The technical lessons are firmer: measure both error rates per group at the operating threshold, on data like the deployment data, and remember that the overall FAR can hide a tenfold difference (Figure 7).

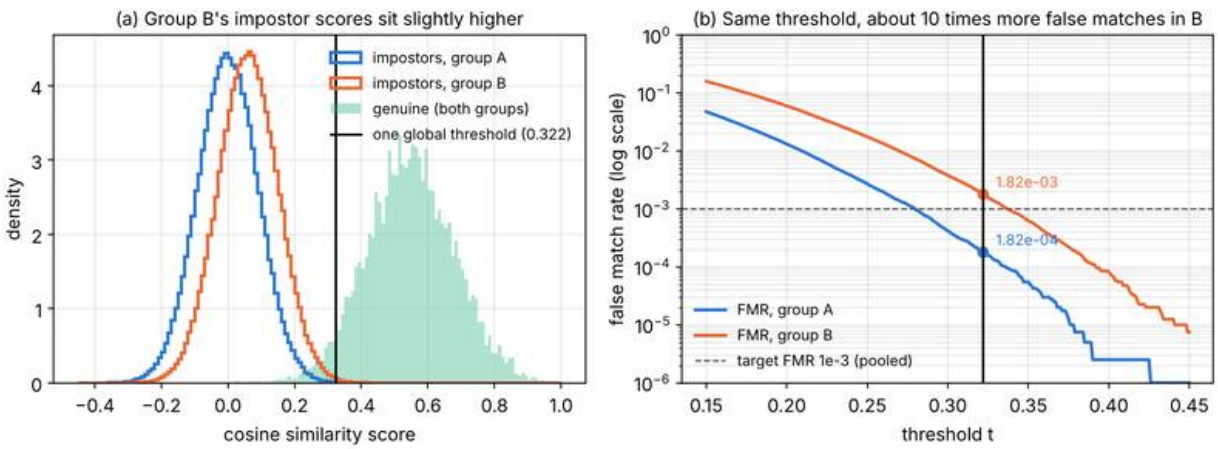


*Figure 7 — A synthetic two-group example. (a) Group B's impostor distribution is shifted by 0.06. (b) At one global threshold, group B's false match rate is about 10 times group A's, although the pooled rate meets the $10^{-3}$ target.*

### Presentation attacks

**Plain version.** A recognition network answers "who is this?", not "is this a real, live face?". A printed photo, a phone screen showing a video, or a mask can be presented to the camera. Detecting such attacks is a separate task.

A **presentation attack** is an attempt to fool the sensor with an artifact; **presentation attack detection** (PAD), also called face anti-spoofing, decides whether the input is *bona fide* (a live person) or an attack. Yu et al. survey deep-learning methods for it [37]: beyond binary "live versus attack" labels, many methods use pixel-wise auxiliary supervision such as pseudo depth maps; the field now emphasizes **domain generalization** (attacks, cameras and lighting not seen in training) and **open-set** attacks of new types; and multi-modal sensors (depth, infrared) or special sensors (light field, flash) help. PAD runs alongside recognition, and the two error types are evaluated separately; a strong recognizer with no PAD is easy to fool with a good photo of the enrolled person. Digital attacks such as morphed passport photos and deepfakes are related but distinct threats.

### Privacy, consent and the law

**Plain version.** A face embedding is a biometric identifier: it can link a person's photos across the internet and cameras, and a person cannot change their face the way they change a password. Many legal systems therefore treat it as sensitive data. This section describes the context; it is not legal advice.

Two points from European law illustrate how regulators view face recognition (rules differ between jurisdictions, and they change):

- Under the EU's General Data Protection Regulation, "biometric data for the purpose of uniquely identifying a natural person" is one of the categories of **sensitive** personal data, which are subject to specific processing conditions (GDPR Article 9) [38].
- The EU AI Act (Regulation (EU) 2024/1689) lists among its **prohibited practices** the untargeted scraping of the internet or CCTV material to create or expand facial recognition databases, and real-time remote biometric identification for law enforcement purposes in publicly accessible spaces; the prohibitions apply from 2 February 2025 [39]. The Regulation itself defines the exact scope and conditions.

For engineers, the practical consequences are concrete. Know where every training image came from and under what terms; the withdrawn datasets above show that "publicly visible" is not the same as "available for any use". Treat stored templates as sensitive data: limit who can query a gallery, log searches, and delete templates that are no longer needed. Prefer designs that keep templates on the user's device for verification when the application allows it. Document the intended

use and the measured error rates per group, as recommended by NIST [2]. Synthetic training data [31], [33] reduces, but does not remove, these concerns, because the deployed system still processes real faces.

### Failure modes and practical advice

**Plain version.** Most failures come from a mismatch between the conditions in training and testing: a face turned away, half covered, much older, blurred, or from a population the system rarely saw. A threshold tuned in one setting silently changes meaning in another.

- **Pose.** Large yaw (profile views) breaks five-point alignment and removes half the face. IJB-B was built with human-labeled face boxes precisely because datasets filtered by a commodity face detector under-represent such poses and occlusions [27], and the ArcFace authors cite pose and age gaps as conditions where plain softmax features degrade [16].
- **Occlusion.** Masks, sunglasses, hands and hair hide parts of the face; detectors may still fire but landmarks drift, and the embedding moves toward the impostor region.
- **Age.** Templates enrolled years earlier match less well; MegaFace evaluated performance with respect to age [24], and NIST found elevated false positives for the elderly and children [2].
- **Image quality.** Low resolution, motion blur, compression and poor lighting raise false non-matches; NIST linked several false-negative differentials to image quality rather than to demographics [2]. Quality-aware methods (MagFace, AdaFace [19], [20]) and quality checks at capture time help.
- **Threshold drift.** A threshold chosen on one dataset gives a different FAR on another camera, population or gallery size (Figures 6 and 7). Recalibrate on deployment-like impostor pairs, and monitor score distributions after deployment, as in ML 11.
- **Label noise and identity overlap.** Web-collected training sets contain mislabeled images and the same person under two names; sub-center ArcFace [18] and dataset cleaning [29] target this. Overlap between training identities and test identities inflates open-set results, so check it before reporting.
- **Evaluation shortcuts.** Reporting only accuracy on LFW, only rank-1 on a small closed gallery, or only one error rate hides exactly the failures above.

### Modern view

#### What the surveys say

Wang and Deng's survey [1] is the standard map of deep face recognition. It sorts the losses into **Euclidean-distance-based** (contrastive, triplet, center loss), **angular/cosine-margin-based** (SphereFace, CosFace, ArcFace) and **softmax variations** (feature and weight normalization), and it traces how Euclidean-distance-based losses dominated the early years and how angular/cosine-margin losses and feature and weight normalization took over from 2017. Drozdowski et al. [36] survey demographic bias across biometrics: how bias is estimated, how it might be mitigated, and the social and technical questions that remain. Yu et al. [37] survey deep face anti-spoofing,

including methods designed for domain generalization and open-set attacks. Boutros et al. [33] survey synthetic data for face recognition, organize its use cases and discuss the challenges that remain.

*Online only: Guide to the Modern view surveys — https://blog.shyandram.dev/en/learn/deep-learning/face-metric-learning/papers/guide-to-the-modern-view-surveys/*

### The state of the art, 2023–2026

- **Margin-based softmax losses on very large datasets remain the standard recipe.** The main changes since ArcFace are adaptive margins tied to image quality (MagFace [19], AdaFace [20]), robustness to label noise (sub-center ArcFace [18]) and efficient training over millions of identities (Partial FC [21], applied with both CNN and ViT backbones).
- **Benchmarks moved to low-FAR, template-based and operational evaluation.** LFW is saturated [31]; IJB-B/C at TAR@FAR = $10^{-4}$ became the academic norm [16], and NIST's FRTE (the evaluation known as FRVT until 2023 [3]) tests submitted algorithms on large operational data at false match rates as low as $10^{-6}$.
- **Synthetic and privacy-aware training data** is an active direction, with graphics-rendered (DigiFace-1M [31]) and diffusion-generated (DCFace [32]) datasets, motivated by consent, bias and label-noise problems of web data [33].
- **Fairness is measured and reported**, not assumed: NIST's demographic analysis [2] and dedicated test sets such as RFW [34] made per-group error rates part of evaluation.

### Open problems

1. **Generalization to unseen conditions**: extreme pose, occlusion, very low resolution and long age gaps.
2. **Demographic differentials**: measuring both error types per group on deployment-like data, and reducing differentials without simply moving errors from one group to another.
3. **Training data without consent problems** that still reaches the accuracy of web-scale real data [33].
4. **Presentation and digital attacks** of types not seen in training [37].
5. **Calibrated, quality-aware scores**, so that a threshold keeps its meaning across cameras and populations.
6. **Governance**: dataset stewardship after release [30], and legal limits that differ between jurisdictions [38], [39].

### Key takeaways

- Embeddings support comparison with identities absent from training; classification labels alone do not.
- State the recognition task and gallery size before selecting metrics or thresholds.
- Detection and alignment errors propagate into embedding comparisons.

- Pair-based losses require informative samples; margin losses shape separation around class centers.
- Report verification TAR at a stated FAR and impostor-pair count; use task-specific identification metrics for search.
- Check errors by group and operating condition, verify dataset availability, and protect biometric templates.

## Exercises

1. **Open or closed?** For each system, say whether it performs verification, closed-set identification or open-set identification, and which error matters most: (a) unlocking a phone, (b) sorting a family's photo album into its five members, (c) searching a watch list of 10,000 people at a stadium entrance.

   **Hint**

   (a) Verification; a false accept is a security failure, a false reject is an inconvenience (the user can retry). (b) Closed-set identification only if every photo shows one of the five; real albums contain guests, so it is open-set in practice. (c) Open-set identification; most people passing are not on the list, so FPIR at the chosen threshold, multiplied by the number of people passing, gives the number of false alarms.

2. **The ArcFace boundary.** Two class centers are $\varphi = 60°$ apart. For a feature on the arc between them, find the angle from $W_1$ at which class 1's loss stops being satisfied for normalized softmax, ArcFace with $m = 0.5$ rad and SphereFace with $m = 4$. Which margin is largest here, and what happens to SphereFace's margin when $\varphi = 120°$?

   **Hint**

   Softmax: 30°. ArcFace: $(\varphi - m)/2 = (60° - 28.6°)/2 \approx 15.7°$, so the gap is 28.6°. SphereFace: $\varphi/(m+1) = 12°$, gap $\varphi(m-1)/(m+1) = 36°$. At $\varphi = 120°$ SphereFace's gap doubles to 72°, while ArcFace's stays 28.6°.

3. **CosFace in the toy.** Change the toy experiment so that the target logit is $s(\cos\theta - m)$ with $m = 0.35$ (CosFace). What is $\partial(\text{target logit})/\partial\cos\theta$ now? Run it and compare the three printed numbers with ArcFace's.

   **Hint**

   Replace `np.cos(th + margin)` with `np.cos(th) - margin` in the logits, and the gradient factor `np.sin(th + margin) / np.sin(th)` with 1 (the derivative of $\cos\theta - m$ with respect to $\cos\theta$ is 1, so the whole factor is just $s$). Expect a result close to ArcFace's; the exact numbers depend on the run.

4. **How many impostor pairs?** You want to claim FAR = $10^{-5}$. (a) With 300,000 impostor pairs and zero false accepts observed, what is the approximate 95% upper bound on the true FAR? (b) How many impostor pairs do you need for the bound to reach $10^{-5}$ with zero errors? (c) Why might your pairs be worth fewer "independent trials" than their count?

   **Hint**

   (a) Rule of three: about $3/300{,}000 = 10^{-5}$. (b) About 300,000, but only if no errors occur; with a few errors you need more. (c) Pairs that reuse the same images or the same people are correlated: one unusual image can produce many false accepts at once.

5. **Gallery growth.** In the identification code, a non-mated probe at threshold 0.36 triggers a false hit with probability 0.010 for a gallery of 300. Use FPIR $\approx 1 - (1 - \text{FMR})^N$ to estimate the per-comparison FMR, then predict FPIR for galleries of 3,000 and 30,000. Compare with the printed values and explain any difference.

   **Hint**

   $1 - (1 - \text{FMR})^{300} = 0.010$ gives FMR $\approx 3.4 \times 10^{-5}$, predicting about 0.096 for 3,000 and 0.64 for 30,000. The printed 0.043 and 0.330 are lower. With only 300 non-mated probes, an FPIR of 0.010 is just 3 false hits, so the FMR estimate is very uncertain; estimate FMR directly from many impostor comparisons instead.

6. **Fair thresholds.** In the two-group code, find the single threshold that equalizes the two groups' FNMR, and one that makes FMR equal. Can one threshold do both? What would you report to a decision maker?

   **Hint**

   Because the genuine distributions are identical, *any* single threshold gives equal FNMR (up to sampling noise: 0.0403 against 0.0400 in the printed run); no single threshold gives equal FMR, because the impostor distributions differ. Report both error rates for both groups at the proposed threshold, and the alternatives (group thresholds, a better model), and leave the choice of trade-off to the people accountable for it.

## References

1. M. Wang and W. Deng, "Deep face recognition: A survey," *Neurocomputing*, vol. 429, pp. 215–244, 2021. doi
2. P. Grother, M. Ngan and K. Hanaoka, "Face Recognition Vendor Test (FRVT) Part 3: Demographic Effects," NIST Interagency Report 8280, National Institute of Standards and Technology, 2019. doi
3. National Institute of Standards and Technology, "Face Recognition Technology Evaluation (FRTE) 1:1 Verification," evaluation web page, accessed 2026. NIST

4. K. Zhang, Z. Zhang, Z. Li and Y. Qiao, "Joint face detection and alignment using multitask cascaded convolutional networks," *IEEE Signal Processing Letters*, vol. 23, no. 10, pp. 1499–1503, 2016. doi
5. J. Deng, J. Guo, E. Ververas, I. Kotsia and S. Zafeiriou, "RetinaFace: Single-shot multi-level face localisation in the wild," in *Proc. IEEE/CVF CVPR*, 2020. CVF
6. Y. Taigman, M. Yang, M. Ranzato and L. Wolf, "DeepFace: Closing the gap to human-level performance in face verification," in *Proc. IEEE CVPR*, 2014. CVF
7. F. Schroff, D. Kalenichenko and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," in *Proc. IEEE CVPR*, 2015. CVF
8. S. Chopra, R. Hadsell and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *Proc. IEEE CVPR*, vol. 1, pp. 539–546, 2005. doi
9. R. Hadsell, S. Chopra and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *Proc. IEEE CVPR*, vol. 2, pp. 1735–1742, 2006. doi
10. A. Hermans, L. Beyer and B. Leibe, "In defense of the triplet loss for person re-identification," arXiv:1703.07737, 2017. arXiv
11. F. Wang, X. Xiang, J. Cheng and A. L. Yuille, "NormFace: L2 hypersphere embedding for face verification," in *Proc. ACM International Conference on Multimedia*, pp. 1041–1049, 2017. doi
12. W. Liu, Y. Wen, Z. Yu, M. Li, B. Raj and L. Song, "SphereFace: Deep hypersphere embedding for face recognition," in *Proc. IEEE CVPR*, 2017. CVF
13. H. Wang, Y. Wang, Z. Zhou, X. Ji, D. Gong, J. Zhou, Z. Li and W. Liu, "CosFace: Large margin cosine loss for deep face recognition," in *Proc. IEEE/CVF CVPR*, 2018. CVF
14. F. Wang, J. Cheng, W. Liu and H. Liu, "Additive margin softmax for face verification," *IEEE Signal Processing Letters*, vol. 25, no. 7, pp. 926–930, 2018. doi
15. J. Deng, J. Guo, N. Xue and S. Zafeiriou, "ArcFace: Additive angular margin loss for deep face recognition," in *Proc. IEEE/CVF CVPR*, 2019. CVF
16. J. Deng, J. Guo, J. Yang, N. Xue, I. Kotsia and S. Zafeiriou, "ArcFace: Additive angular margin loss for deep face recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 10, pp. 5962–5979, 2022. doi
17. Y. Wen, K. Zhang, Z. Li and Y. Qiao, "A discriminative feature learning approach for deep face recognition," in *Proc. ECCV*, pp. 499–515, 2016. doi
18. J. Deng, J. Guo, T. Liu, M. Gong and S. Zafeiriou, "Sub-center ArcFace: Boosting face recognition by large-scale noisy web faces," in *Proc. ECCV*, pp. 741–757, 2020. doi
19. Q. Meng, S. Zhao, Z. Huang and F. Zhou, "MagFace: A universal representation for face recognition and quality assessment," in *Proc. IEEE/CVF CVPR*, 2021. CVF
20. M. Kim, A. K. Jain and X. Liu, "AdaFace: Quality adaptive margin for face recognition," in *Proc. IEEE/CVF CVPR*, 2022. CVF
21. X. An, J. Deng, J. Guo, Z. Feng, X. Zhu, J. Yang and T. Liu, "Killing two birds with one stone: Efficient and robust training of face recognition CNNs by Partial FC," in *Proc. IEEE/CVF CVPR*, 2022. CVF
22. A. Martin, G. Doddington, T. Kamm, M. Ordowski and M. Przybocki, "The DET curve in assessment of detection task performance," in *Proc. Eurospeech*, pp. 1895–1898, 1997. ISCA
23. G. B. Huang, M. Mattar, T. Berg and E. Learned-Miller, "Labeled Faces in the Wild: A database for studying face recognition in unconstrained environments," in *Workshop on Faces in 'Real-Life' Images: Detection, Alignment, and Recognition*, Marseille, 2008. HAL

24. I. Kemelmacher-Shlizerman, S. M. Seitz, D. Miller and E. Brossard, “The MegaFace benchmark: 1 million faces for recognition at scale,” in *Proc. IEEE CVPR*, 2016. CVF
25. University of Washington, “MegaFace and MF2: Million-scale face recognition,” project web page, accessed 2026. project page
26. Y. Guo, L. Zhang, Y. Hu, X. He and J. Gao, “MS-Celeb-1M: A dataset and benchmark for large-scale face recognition,” in *Proc. ECCV*, pp. 87–102, 2016. doi
27. C. Whitelam, E. Taborsky, A. Blanton, B. Maze, J. Adams, T. Miller, N. Kalka, A. K. Jain, J. A. Duncan, K. Allen, J. Cheney and P. Grother, “IARPA Janus Benchmark-B face dataset,” in *Proc. IEEE CVPR Workshops*, 2017. CVF
28. B. Maze, J. Adams, J. A. Duncan, N. Kalka, T. Miller, C. Otto, A. K. Jain, W. T. Niggel, J. Anderson, J. Cheney and P. Grother, “IARPA Janus Benchmark – C: Face dataset and protocol,” in *Proc. International Conference on Biometrics (ICB)*, pp. 158–165, 2018. doi
29. Z. Zhu, G. Huang, J. Deng, Y. Ye, J. Huang, X. Chen, J. Zhu, T. Yang, J. Lu, D. Du and J. Zhou, “WebFace260M: A benchmark unveiling the power of million-scale deep face recognition,” in *Proc. IEEE/CVF CVPR*, 2021. CVF
30. K. Peng, A. Mathur and A. Narayanan, “Mitigating dataset harms requires stewardship: Lessons from 1000 papers,” in *Proc. NeurIPS Datasets and Benchmarks Track*, 2021. proceedings
31. G. Bae, M. de La Gorce, T. Baltrušaitis, C. Hewitt, D. Chen, J. Valentin, R. Cipolla and J. Shen, “DigiFace-1M: 1 million digital face images for face recognition,” in *Proc. IEEE/CVF WACV*, 2023. CVF
32. M. Kim, F. Liu, A. Jain and X. Liu, “DCFace: Synthetic face generation with dual condition diffusion model,” in *Proc. IEEE/CVF CVPR*, 2023. CVF
33. F. Boutros, V. Struc, J. Fierrez and N. Damer, “Synthetic data for face recognition: Current state and future prospects,” *Image and Vision Computing*, vol. 135, art. 104688, 2023. doi
34. M. Wang, W. Deng, J. Hu, X. Tao and Y. Huang, “Racial Faces in the Wild: Reducing racial bias by information maximization adaptation network,” in *Proc. IEEE/CVF ICCV*, pp. 692–702, 2019. doi
35. J. Buolamwini and T. Gebru, “Gender Shades: Intersectional accuracy disparities in commercial gender classification,” in *Proc. Conference on Fairness, Accountability and Transparency*, PMLR 81, pp. 77–91, 2018. PMLR
36. P. Drozdowski, C. Rathgeb, A. Dantcheva, N. Damer and C. Busch, “Demographic bias in biometrics: A survey on an emerging challenge,” *IEEE Transactions on Technology and Society*, vol. 1, no. 2, pp. 89–103, 2020. doi
37. Z. Yu, Y. Qin, X. Li, C. Zhao, Z. Lei and G. Zhao, “Deep learning for face anti-spoofing: A survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, pp. 5609–5631, 2023. doi
38. European Commission, “What personal data is considered sensitive?,” data protection guidance referring to Article 9 of the GDPR, accessed 2026. European Commission
39. European Commission, “AI Act” (Regulation (EU) 2024/1689), policy page, accessed 2026. European Commission
40. M. Turk and A. Pentland, “Eigenfaces for recognition,” *Journal of Cognitive Neuroscience*, vol. 3, no. 1, pp. 71–86, 1991. doi
41. P. N. Belhumeur, J. P. Hespanha and D. J. Kriegman, “Eigenfaces vs. Fisherfaces: Recognition using class specific linear projection,” *IEEE Transactions on Pattern Analysis and Machine Intelli-*

*gence*, vol. 19, no. 7, pp. 711–720, 1997. doi

# DLCV 03 • Object Detection: From R-CNN to DETR

How deep networks find and label every object in an image: box regression, IoU, anchors, NMS and feature pyramids; two-stage, one-stage, anchor-free and transformer detectors; open-vocabulary models; and how detection is measured.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/object-detection/

## What you'll learn

- Define detection outputs and distinguish boxes from pixel masks.
- Explain box regression, IoU, assignment, suppression and feature pyramids.
- Compare proposal-based, dense, anchor-free and set-prediction designs.
- Understand how text prompts change the class vocabulary.
- Read AP protocols and measure the full deployment pipeline.

## The big picture

A classifier answers one question about a whole image: *what is this?* A detector answers a harder one: *what is where?* It must find every object of interest, draw a tight rectangle (a **bounding box**) around each, name it, and say how sure it is. The number of objects is not known in advance: there may be zero, one or three hundred.

> **Explain like I'm five**
>
> Imagine a "Where's Waldo?" page. A **classifier** only tells you "this page has Waldo on it." A **detector** puts a sticker on Waldo, another sticker on every dog and every balloon, and writes on each sticker what it is and how sure it is ("dog, pretty sure"). The tricky parts are the ones you would also find tricky: tiny things far away, things hiding behind other things, and not putting two stickers on the same dog.

Detection is the entry point of most "real" vision systems. Self-driving cars detect pedestrians and vehicles; phones detect faces before focusing; factories detect defects; trackers (see the tracking chapter linked at the end) mostly *link detections over time*. The field also produced many ideas that now appear everywhere in deep learning: multi-task losses, feature pyramids, the focal loss, and set prediction with bipartite matching.

This tutorial assumes you know CNNs and image classification (DLCV 02) and the classical feature ideas of IP 12. Two surveys frame the whole story: Zou et al., *Object Detection in 20 Years* [1], and Liu et al.'s IJCV survey of deep generic detection [2]. We return to both in the Modern view.

**Related chapters.** DLCV 03A follows the YOLO family and real-time DETRs in depth, and DLCV 04 moves from boxes to pixel masks (semantic, instance and panoptic segmentation).

## The detection problem

**Plain version.** The input is an image. The output is a short list of (box, class, score) triples. The training data is a list of human-drawn boxes with class names.

**Precise version.** Given an image $I \in \mathbb{R}^{H\times W\times 3}$, a detector outputs a set

$$\hat{Y} = \{(\hat{b}_i, \hat{c}_i, \hat{s}_i)\}_{i=1}^{N},$$

where $\hat{b}_i \in \mathbb{R}^4$ is a box, $\hat{c}_i \in \{1, \dots, K\}$ is one of $K$ classes, $\hat{s}_i \in [0,1]$ is a confidence score, and $N$ varies from image to image. The ground truth is a set $Y = \{(b_j, c_j)\}_{j=1}^{M}$ of $M$ annotated objects. Two box formats are common: corners $(x_1, y_1, x_2, y_2)$ and center–size $(c_x, c_y, w, h)$. Boxes are **axis-aligned**; rotated boxes are a separate variant used for aerial images and text.

Three properties make detection harder than classification:

1. **Variable-size output.** The network must decide *how many* things to output. Every architecture in this tutorial is, at heart, a different answer to this problem.
2. **Localization and recognition at once.** The model must be invariant to *what* changes about an object's appearance but sensitive to *where* it is, a tension that shapes the design of every head.
3. **Extreme imbalance.** A typical image has a handful of objects but tens of thousands of candidate positions and sizes, nearly all background.

### Detection versus its neighbours

Figure 1 compares the four classic recognition tasks on one synthetic scene. **Classification** gives one label per image. **Detection** gives a box and a label per object. **Semantic segmentation** labels every pixel with a class, so two touching balls merge into one blob. **Instance segmentation** gives each object its own pixel mask, which is detection plus a mask per box; Mask R-CNN [16] is the standard way to get it, as we will see.

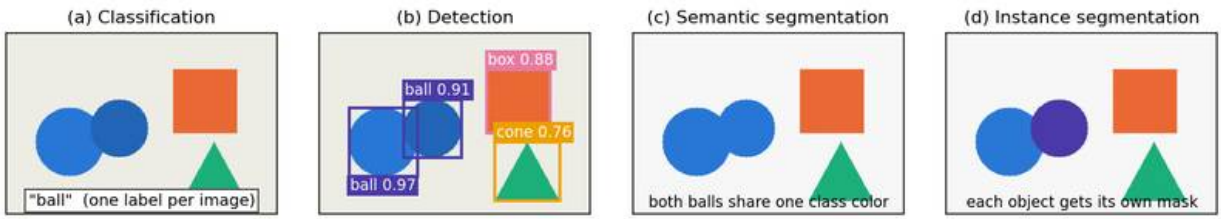



*Figure 1 — The same scene seen by four tasks: (a) classification, (b) detection, (c) semantic segmentation, (d) instance segmentation.*

## Classical detection: sliding windows and hand-made features

**Plain version.** Sliding-window detection applies a classifier across positions and scales, then retains windows predicted to contain an object.

**Sliding windows.** For a window of fixed size $w \times h$ slid with stride $s$ over an image pyramid with scale factor $\lambda$, the number of windows is roughly

$$N_{\text{win}} \approx \sum_{k \geq 0} \frac{(H\lambda^{-k} - h)(W\lambda^{-k} - w)}{s^2},$$

where $k$ indexes the pyramid levels. For a $640 \times 480$ image, a $64 \times 128$ window, $s = 8$ and $\lambda = 1.2$, that is on the order of $10^4$ windows for one aspect ratio, all of which must be classified. This is the template matching of IP 13 scaled up, and it has the same weaknesses: one template shape, many scales, and a lot of wasted work on background.

Three classical detectors show how far hand-made features went. **Viola–Jones** (2001) [5] made face detection real-time: simple rectangle features are computed in constant time, and a cascade of ever stricter classifiers throws away most background windows after a few cheap tests. **HOG + linear SVM** (2005) [6] described each window by histograms of gradient orientations, which made pedestrian detection practical. **Deformable part models** (DPM) [7] added a few movable part filters around a coarse whole-object filter, and led the PASCAL VOC benchmark [3] until 2012.

*Online only: Classical detectors in detail: Viola–Jones, HOG and DPM — https://blog.shyandram.dev/en/learn/deep-learning/object-detection/notes/classical-detectors-in-detail-violajones-hog-and-dpm/*

**Why this era ended.** Classical detectors were *slow* (an exhaustive scan over positions, scales and aspect ratios, each scored by a hand-designed pipeline) and *brittle* (one hand-designed feature type, a few rigid templates, and poor transfer to new categories, viewpoints and clutter). They also had no way to share features across classes. Deep detectors kept the useful ideas, such as scanning in a feature pyramid, cascades, parts and hard-negative mining, and replaced the hand-made features with learned ones.

## Core building blocks

Every detector in the rest of this tutorial is assembled from the same parts. We define each one carefully, with code you can run.

### Box parameterization and regression targets

**Plain version.** Reference-based box regression predicts corrections to a proposal, anchor or reference point rather than starting from unconstrained coordinates.

Given a reference box $P = (P_x, P_y, P_w, P_h)$ in center–size form and its matched ground-truth box $G = (G_x, G_y, G_w, G_h)$, the standard targets introduced with R-CNN [8] are

$$t_x = \frac{G_x - P_x}{P_w}, \qquad t_y = \frac{G_y - P_y}{P_h}, \qquad t_w = \log \frac{G_w}{P_w}, \qquad t_h = \log \frac{G_h}{P_h}.$$

Dividing the shift by the reference size makes the target **scale-invariant** (a 10-pixel error matters more for a 20-pixel box than for a 400-pixel box). The **log** for width and height makes scaling

symmetric (doubling and halving are $+\log 2$ and $-\log 2$) and keeps the decoded size positive: $\hat{G}_w = P_w \exp(\hat{t}_w)$. In practice the targets are often divided by fixed standard deviations, such as $(0.1, 0.1, 0.2, 0.2)$, so that all four have similar magnitude.

```
import numpy as np

def to_cxcywh(b):
    return np.stack([(b[:, 0] + b[:, 2]) / 2, (b[:, 1] + b[:, 3]) / 2,
                     b[:, 2] - b[:, 0], b[:, 3] - b[:, 1]], axis=1)

def to_xyxy(c):
    return np.stack([c[:, 0] - c[:, 2] / 2, c[:, 1] - c[:, 3] / 2,
                     c[:, 0] + c[:, 2] / 2, c[:, 1] + c[:, 3] / 2], axis=1)

def encode(anchors, gt, std=(0.1, 0.1, 0.2, 0.2)):
    """Regression targets (tx, ty, tw, th) of gt relative to anchors (both xyxy)."""
    P, G = to_cxcywh(anchors), to_cxcywh(gt)
    t = np.stack([(G[:, 0] - P[:, 0]) / P[:, 2],
                  (G[:, 1] - P[:, 1]) / P[:, 3],
                  np.log(G[:, 2] / P[:, 2]),
                  np.log(G[:, 3] / P[:, 3])], axis=1)
    return t / np.array(std)          # normalize so all four have similar scale

def decode(anchors, t, std=(0.1, 0.1, 0.2, 0.2)):
    P = to_cxcywh(anchors)
    t = t * np.array(std)
    c = np.stack([P[:, 0] + t[:, 0] * P[:, 2],
                  P[:, 1] + t[:, 1] * P[:, 3],
                  P[:, 2] * np.exp(t[:, 2]),
                  P[:, 3] * np.exp(t[:, 3])], axis=1)
    return to_xyxy(c)

anchors = np.array([[0, 0, 32, 32], [100, 100, 228, 164]], float)
gt      = np.array([[4, 2, 40, 30], [90, 110, 250, 170]], float)
t = encode(anchors, gt)
print(np.round(t, 3))
print(np.allclose(decode(anchors, t), gt))   # round trip
```

```
[[ 1.875  0.     0.589 -0.668]
 [ 0.469  1.25   1.116 -0.323]]
True
```

These targets are usually trained with the **smooth L1** (Huber) loss of Fast R-CNN [9],

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5\,x^2 & \text{if } |x| < 1, \\ |x| - 0.5 & \text{otherwise,} \end{cases}$$

applied to each coordinate difference $x = \hat{t} - t$. It behaves like L2 for small errors (smooth gradients) and like L1 for large ones (robust to outliers early in training). Its weakness is that the four coordinates are treated as independent numbers, while the evaluation metric, IoU, treats the box as a whole. That mismatch motivated IoU-based losses.

### IoU and its variants

**Plain version.** IoU asks: of all the area covered by either box, what fraction is covered by both? It is 1 for a perfect match and 0 for boxes that do not touch.

For two boxes $A$ and $B$,

$$\mathrm{IoU}(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|},$$

where $|\cdot|$ is area. IoU is scale-invariant, and it is the yardstick for evaluation (a detection counts as correct if its IoU with an unmatched ground-truth box exceeds a threshold), for label assignment and for NMS. Figure 2 shows how demanding the thresholds are: shifting a square by only about 18% of its side along both axes already drops IoU to 0.5.

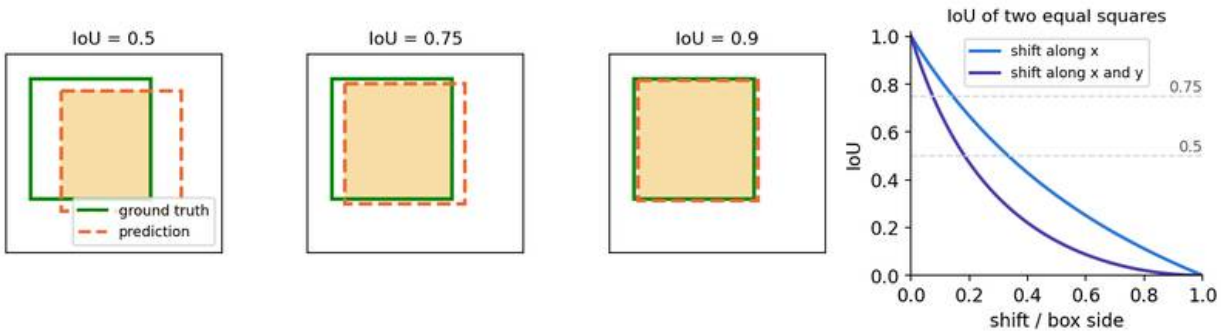



*Figure 2 — What IoU thresholds look like. Left: a prediction at IoU 0.5, 0.75 and 0.9 with the ground truth. Right: IoU of two equal squares as one is shifted along x, or along x and y.*

As a loss, $1 - \mathrm{IoU}$ has a flaw: when two boxes do not overlap, IoU is 0 no matter how far apart they are, so the gradient is zero. Three fixes are widely used:

- **GIoU** [11] adds a penalty based on the smallest box $C$ that encloses both: $\mathrm{GIoU} = \mathrm{IoU} - \frac{|C \setminus (A \cup B)|}{|C|}$, which lies in $(-1, 1]$ and keeps decreasing as non-overlapping boxes drift apart.
- **DIoU** [12] penalizes the distance between centers directly: $\mathrm{DIoU} = \mathrm{IoU} - \frac{\rho^2(\mathbf{b}, \mathbf{b}^{gt})}{c^2}$, where $\rho$ is the Euclidean distance between the box centers $\mathbf{b}$ and $\mathbf{b}^{gt}$, and $c$ is the diagonal length of $C$. It converges faster than GIoU because it pulls the centers together even when one box contains the other, where the GIoU penalty vanishes.
- **CIoU** [12] adds an aspect-ratio term: $\mathrm{CIoU} = \mathrm{DIoU} - \alpha v$, with $v = \frac{4}{\pi^2}\left(\arctan\frac{w^{gt}}{h^{gt}} - \arctan\frac{w}{h}\right)^2$ measuring aspect-ratio mismatch and $\alpha = \frac{v}{(1-\mathrm{IoU})+v}$ a trade-off weight.

The corresponding losses are $1 - \mathrm{GIoU}$, $1 - \mathrm{DIoU}$ and $1 - \mathrm{CIoU}$.

```
import numpy as np

def iou_family(a, b, eps=1e-9):
```

```
    """a, b: (N, 4) boxes as (x1, y1, x2, y2). Returns IoU, GIoU, DIoU, CIoU."""
    ix1, iy1 = np.maximum(a[:, 0], b[:, 0]), np.maximum(a[:, 1], b[:, 1])
    ix2, iy2 = np.minimum(a[:, 2], b[:, 2]), np.minimum(a[:, 3], b[:, 3])
    inter = np.clip(ix2 - ix1, 0, None) * np.clip(iy2 - iy1, 0, None)
    wa, ha = a[:, 2] - a[:, 0], a[:, 3] - a[:, 1]
    wb, hb = b[:, 2] - b[:, 0], b[:, 3] - b[:, 1]
    union = wa * ha + wb * hb - inter
    iou = inter / (union + eps)
    # smallest enclosing box C
    cx1, cy1 = np.minimum(a[:, 0], b[:, 0]), np.minimum(a[:, 1], b[:, 1])
    cx2, cy2 = np.maximum(a[:, 2], b[:, 2]), np.maximum(a[:, 3], b[:, 3])
    area_c = (cx2 - cx1) * (cy2 - cy1)
    giou = iou - (area_c - union) / (area_c + eps)
    # center distance over enclosing-box diagonal
    rho2 = ((a[:, 0] + a[:, 2]) - (b[:, 0] + b[:, 2])) ** 2 / 4 \
         + ((a[:, 1] + a[:, 3]) - (b[:, 1] + b[:, 3])) ** 2 / 4
    c2 = (cx2 - cx1) ** 2 + (cy2 - cy1) ** 2
    diou = iou - rho2 / (c2 + eps)
    v = 4 / np.pi**2 * (np.arctan(wb / hb) - np.arctan(wa / ha)) ** 2
    alpha = v / (1 - iou + v + eps)
    ciou = diou - alpha * v
    return iou, giou, diou, ciou

gt = np.array([[10, 10, 50, 50]] * 3, float)
pred = np.array([[20, 20, 60, 60],    # overlapping, shifted
                 [60, 10, 100, 50],   # disjoint, nearby
                 [200, 10, 240, 50]], float)  # disjoint, far away
for name, v in zip(["IoU", "GIoU", "DIoU", "CIoU"], iou_family(pred, gt)):
    print(f"{name:5s}", np.round(v, 3))
```

```
IoU   [0.391 0.    0.   ]
GIoU  [ 0.311 -0.111 -0.652]
DIoU  [ 0.351 -0.258 -0.662]
CIoU  [ 0.351 -0.258 -0.662]
```

The two disjoint predictions both have IoU 0, but GIoU and DIoU correctly say that the far one is worse. CIoU equals DIoU here because all boxes are squares, so there is no aspect-ratio mismatch.

### Anchors and label assignment

**Plain version.** An anchor-based dense detector places reference boxes on a feature grid, then predicts object scores and box corrections at each location.

At a feature map with stride $s$ (each cell covers $s \times s$ input pixels), anchors are centered at $((i+\frac{1}{2})s, (j+\frac{1}{2})s)$ with a few scales $\sigma$ and aspect ratios $r = h/w$, giving widths $w = \sigma/\sqrt{r}$ and heights $h = \sigma\sqrt{r}$ so that the area stays $\sigma^2$. Faster R-CNN uses 3 scales × 3 ratios = 9 anchors per location [10] (Figure 3, left).

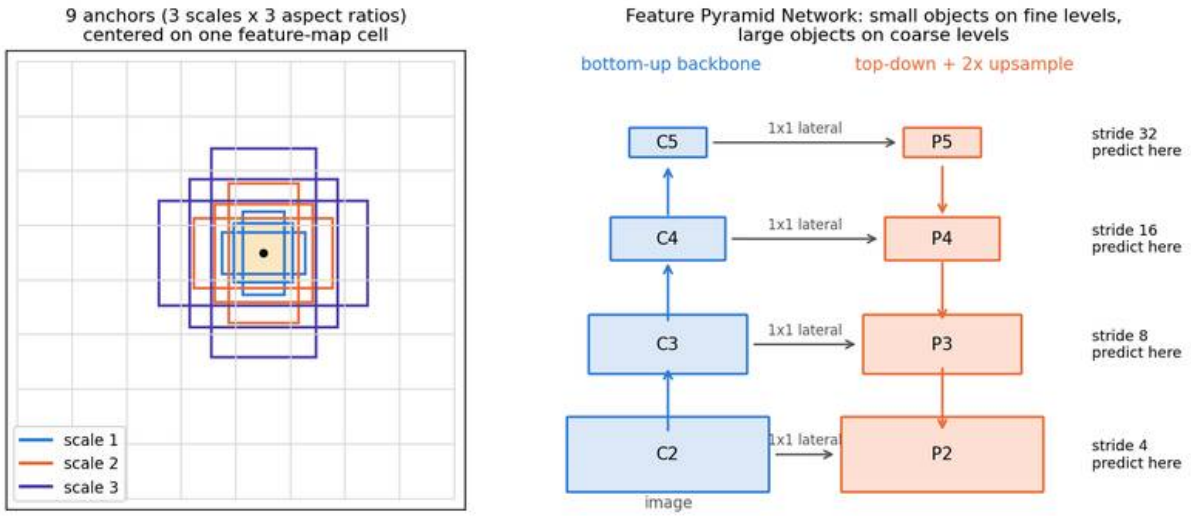



*Figure 3 — Left: nine anchors (3 scales × 3 aspect ratios) at one feature-map cell. Right: a Feature Pyramid Network; each level predicts objects of a matching size range.*

**Label assignment** decides which anchors are trained as which object. The classic rule is IoU-based: in the region proposal network of Faster R-CNN an anchor is *positive* if its IoU with some ground-truth box is at least 0.7 (or if it is the best anchor for that box), *negative* if its IoU with every box is below 0.3, and ignored otherwise [10]; RetinaNet uses 0.5 and 0.4 [15]. Each positive anchor regresses toward its matched box with the targets above. Assignment turned out to matter as much as architecture: anchor-free detectors (FCOS [29]) assign by *location* and object size instead, and DETR [30] replaces all fixed rules by a learned one-to-one matching.

Anchors have real costs: they introduce hyperparameters (scales, ratios, thresholds) that must be tuned per dataset, and they multiply the number of predictions. YOLO9000 [19] chose anchor shapes by running k-means on the training boxes, using $1 - \text{IoU}$ as the distance.

*Online only: Anchor shapes by k-means: clustering with an IoU distance — https://blog.shyandram.dev/en/learn/deep-learning/object-detection/notes/anchor-shapes-by-k-means-clustering-with-an-iou-distance/*

### Non-maximum suppression (NMS) and Soft-NMS

**Plain version.** A dense detector fires many times on the same object (neighboring anchors all see it). NMS keeps the most confident box and deletes the others that overlap it too much.

Greedy NMS, per class: sort the boxes by score; repeatedly take the highest-scoring remaining box $M$, keep it, and remove every box $b_i$ with $\text{IoU}(M, b_i) > N_t$ (typically $N_t = 0.5$ to $0.7$). The weakness is crowded scenes: two real people standing close may overlap by more than $N_t$, and the lower-scoring one is deleted, losing recall.

**Soft-NMS** [13] replaces deletion with a score decay. With the Gaussian variant,

$$s_i \leftarrow s_i \exp\!\left(-\frac{\mathrm{IoU}(M, b_i)^2}{\sigma}\right),$$

where $s_i$ is the score of box $b_i$ and $\sigma$ controls how fast overlapping boxes are penalized. A heavy overlap still pushes the score down a lot, but a genuine neighbor survives with a reduced score instead of disappearing.

```
import numpy as np

def iou_one_to_many(box, boxes):
    x1 = np.maximum(box[0], boxes[:, 0]); y1 = np.maximum(box[1], boxes[:, 1])
    x2 = np.minimum(box[2], boxes[:, 2]); y2 = np.minimum(box[3], boxes[:, 3])
    inter = np.clip(x2 - x1, 0, None) * np.clip(y2 - y1, 0, None)
    area = lambda b: (b[..., 2] - b[..., 0]) * (b[..., 3] - b[..., 1])
    return inter / (area(box) + area(boxes) - inter)

def nms(boxes, scores, thr=0.5):
    order = np.argsort(-scores)
    keep = []
    while order.size:
        i = order[0]
        keep.append(i)
        ious = iou_one_to_many(boxes[i], boxes[order[1:]])
        order = order[1:][ious <= thr]          # drop everything that overlaps too much
    return np.array(keep)

def soft_nms(boxes, scores, sigma=0.5, score_thr=0.001):
    scores = scores.copy()
    idx = np.arange(len(scores))
    keep = []
    while idx.size:
        j = np.argmax(scores[idx]); i = idx[j]
        keep.append(i)
        idx = np.delete(idx, j)
        ious = iou_one_to_many(boxes[i], boxes[idx])
        scores[idx] *= np.exp(-ious**2 / sigma)  # decay instead of delete
        idx = idx[scores[idx] > score_thr]
    return np.array(keep), scores

# two people standing close together + duplicates
boxes = np.array([[10, 10, 60, 110], [14, 12, 62, 112], [8, 6, 58, 104],
                  [40, 15, 90, 115], [43, 18, 92, 118]], float)
scores = np.array([0.95, 0.90, 0.80, 0.85, 0.70])
print("greedy NMS keeps:", nms(boxes, scores, 0.5))
keep, s = soft_nms(boxes, scores)
print("Soft-NMS order:", keep, "rescored:", np.round(s[keep], 3))
```

```
greedy NMS keeps: [0 3]
Soft-NMS order: [0 3 2 4 1] rescored: [0.95  0.761 0.183 0.145 0.058]
```

Greedy NMS keeps one box per person. Soft-NMS keeps everything but ranks the duplicates far

below the two real detections; a final score threshold or the AP computation then treats them as low-confidence guesses. Figure 4 shows the same effect on a synthetic scene.

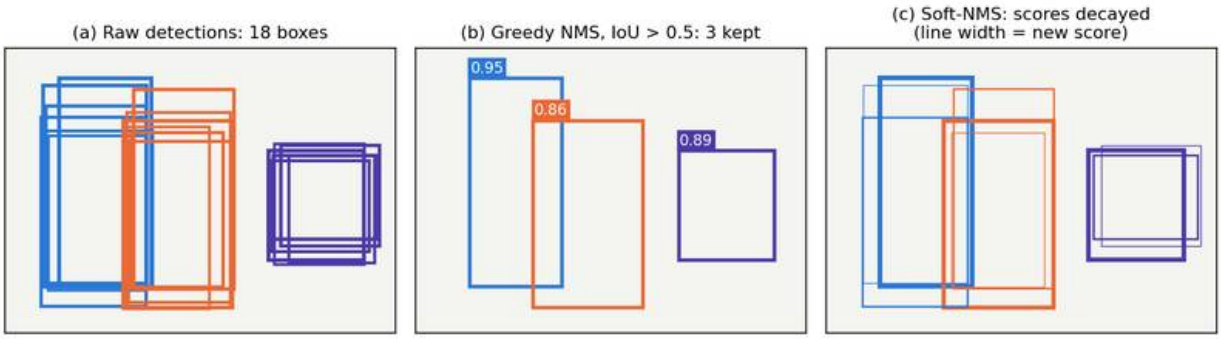



*Figure 4 — NMS on synthetic detections. (a) 18 raw boxes around three objects. (b) Greedy NMS at IoU 0.5 keeps one box per object. (c) Soft-NMS keeps all boxes but decays the scores of overlapping ones (thinner line = lower score).*

NMS has three further problems: it has a hand-tuned threshold; it is a sequential, data-dependent step that is awkward to run on accelerators; and it is not part of training, so the network is never taught to avoid duplicates. Removing it is one of the main motivations behind DETR and YOLOv10 [24].

### Feature pyramids (FPN)

**Plain version.** Small objects need high-resolution feature maps; large objects need features that have seen a lot of context. A feature pyramid gives you both, at every scale.

A CNN backbone naturally produces a pyramid $\{C_2, C_3, C_4, C_5\}$ with strides $4, 8, 16, 32$, but its shallow levels are semantically weak. The **Feature Pyramid Network** [14] adds a top-down path: starting from the coarsest level, it upsamples by 2 and adds a $1 \times 1$-projected lateral connection from the backbone,

$$P_l = \mathrm{Conv}_{3\times3}\big(\mathrm{Conv}_{1\times1}(C_l) + \mathrm{Up}_{2\times}(P_{l+1})\big),$$

so every output level $P_l$ has the same channel width and strong semantics. In a two-stage detector with FPN, a region of interest of size $w \times h$ is pooled from level $l = \lfloor l_0 + \log_2(\sqrt{wh}/224) \rfloor$ (with $l_0 = 4$ in [14]), so a region twice as large moves one level up; dense detectors use a similar size-to-level rule. Later necks (extra bottom-up paths, weighted fusion) refine the same idea; nearly every detector after 2017 uses one (Figure 3, right).

### Focal loss and class imbalance

**Plain version.** If 99.9% of your candidates are background, a normal loss is dominated by millions of easy "nothing here" examples. The focal loss turns down the volume on examples the model already gets right.

Let $p \in [0,1]$ be the predicted probability of the foreground class and $y \in \{0,1\}$ the label. Define $p_t = p$ if $y = 1$ and $p_t = 1 - p$ otherwise. The **focal loss** [15] is

$$\mathrm{FL}(p_t) = -\alpha_t (1 - p_t)^{\gamma} \log(p_t),$$

where $\gamma \geq 0$ is the focusing parameter and $\alpha_t$ a class-balancing weight ($\alpha$ for foreground, $1-\alpha$ for background). With $\gamma = 0$ this is weighted cross-entropy. With the paper's default $\gamma = 2$, $\alpha = 0.25$, an easy negative with $p_t = 0.99$ is down-weighted by $(0.01)^2 = 10^{-4}$ relative to cross-entropy, while hard examples keep almost their full weight. The loss is normalized by the number of positive anchors. Two-stage detectors avoid the problem differently (the proposal stage removes most background, then a fixed foreground:background sampling ratio), and SSD [26] uses **hard negative mining** (keeping only the highest-loss negatives at about 3:1). The focal loss made a *dense* one-stage detector as accurate as two-stage ones for the first time.

## Two-stage detectors: the R-CNN family

**Plain version.** First propose a few thousand places where *something* might be; then look closely at each one and decide *what* it is and exactly where its edges are.

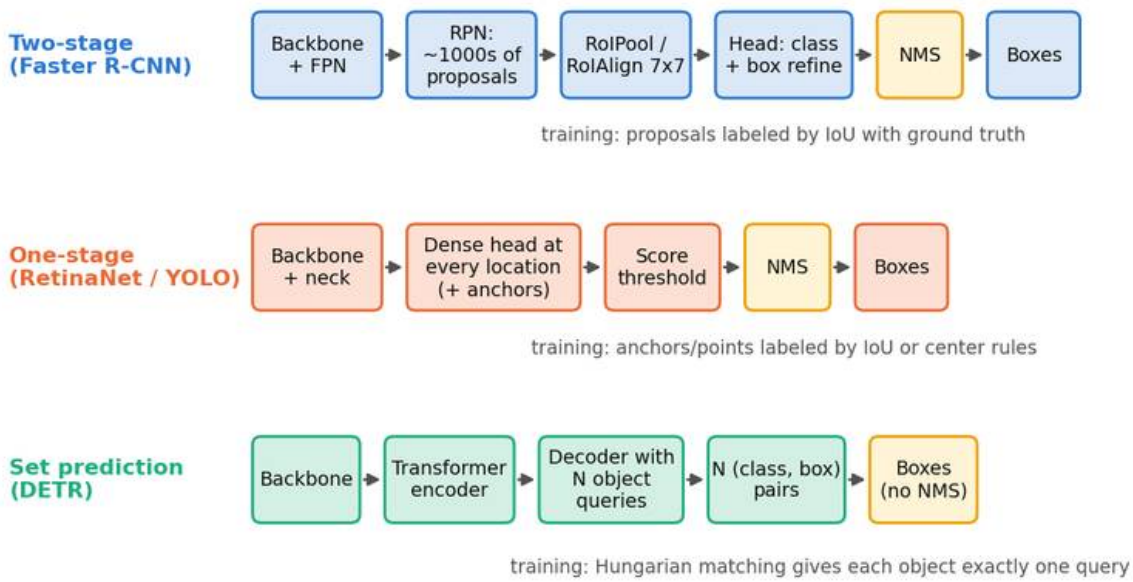



*Figure 5 — Three detector pipelines. NMS (yellow) is a post-processing step for two-stage and one-stage detectors; DETR-style set prediction removes it by learning one-to-one assignments.*

**R-CNN (2014)** [8] ran a bottom-up grouping algorithm to get about 2,000 class-agnostic region proposals per image, warped each one to a fixed size, passed it through a classification-pretrained CNN, scored the features with per-class linear SVMs and refined the box with the regression targets above. It improved mean AP on PASCAL VOC 2012 by more than 30% relative to the previous

best result, and it established the recipe "pretrain on classification, fine-tune on detection" that is still standard. It was very slow, because the CNN ran separately on every proposal.

**Fast R-CNN (2015)** [9] runs the CNN *once* on the whole image and then crops each proposal from the shared feature map with **RoI pooling**: the proposal's projection onto the feature map is divided into a fixed $H \times W$ grid (e.g. $7 \times 7$) and each bin is max-pooled, so every proposal yields the same-size feature regardless of its shape. A single network is trained with a multi-task loss,

$$L = L_{\text{cls}}(p, u) + \lambda\,[u \geq 1]\,L_{\text{loc}}(t^u, v),$$

where $p$ is the softmax over $K + 1$ classes, $u$ the true class (0 for background), $[u \geq 1]$ is 1 for foreground proposals only, $t^u$ the predicted box offsets for class $u$, $v$ the target offsets and $L_{\text{loc}}$ the smooth L1 loss.

**Faster R-CNN (2015)** [10] made the proposals themselves learned. A small **Region Proposal Network** (RPN) slides over the shared feature map and, for each of the 9 anchors per location, predicts an objectness score and box offsets. The top proposals (after NMS) feed the Fast R-CNN head. Because both stages share the backbone, proposals become almost free, and the whole detector is one network trained end to end.

**Mask R-CNN (2017)** [16] adds a small fully convolutional branch that predicts a binary mask per class inside each RoI, giving instance segmentation. Its key detail is **RoIAlign**: RoI pooling rounds the proposal boundaries and bin edges to the feature grid, which shifts features by up to half a stride. RoIAlign keeps real-valued coordinates and samples each bin with bilinear interpolation. This sounds minor, but pixel-accurate masks need it, and box accuracy at high IoU improves too.

*Online only: RoI pooling versus RoIAlign: sampling without rounding — https://blog.shyandram.dev/en/learn/deep-learning/object-detection/notes/roi-pooling-versus-roialign-sampling-without-rounding/*

**Cascade R-CNN (2018)** [17] addresses a subtle problem. A head trained with positives at IoU $\geq 0.5$ learns to produce boxes that are "good at 0.5" but not at 0.75. Simply raising the threshold leaves too few positives and overfits. Cascade R-CNN chains several heads trained at increasing IoU thresholds (0.5, 0.6, 0.7), each refining the boxes of the previous one, so each stage receives a training distribution of higher quality. It was a large gain on the strict COCO metrics.

Two-stage detectors are accurate, flexible (the RoI head can predict masks, keypoints or attributes) and still the default for many instance-segmentation tasks, but the per-RoI head and NMS cost time.

## One-stage detectors: YOLO, SSD and RetinaNet

**Plain version.** Skip the proposal step. Make one dense prediction at every location of the feature map in a single forward pass, then clean it up with NMS.

**YOLO v1 (2016)** [18] divided the image into an $S \times S$ grid ($S = 7$). Each cell predicted $B = 2$ boxes, each with a confidence, plus one set of $C$ class probabilities, so the output was a single $S \times S \times (5B + C)$ tensor ($7 \times 7 \times 30$ for PASCAL VOC's 20 classes). The cell containing an object's center was responsible for it. It ran in real time and, because it saw the whole image, made fewer background

errors than Fast R-CNN, but it localized less precisely and struggled with small objects and groups, since each cell could only predict a fixed number of boxes and one class.

The **YOLO family** then grew through many versions [19]–[25]. The core idea stayed the same, one dense pass over a grid followed by NMS, while each version improved one part: anchors and predictions at several scales, stronger backbones and training tricks, better label assignment, attention, and finally training that no longer needs NMS (YOLOv10 [24]).

*Online only: The YOLO family, version by version — https://blog.shyandram.dev/en/learn/deep-learning/object-detection/notes/the-yolo-family-version-by-version/*

**SSD (2016)** [26] attached small convolutional predictors to *several* feature maps of different resolutions, each with its own set of default boxes (anchors) of several aspect ratios, which handled scale far better than YOLO v1. It trained with hard negative mining and heavy data augmentation.

**RetinaNet (2017)** [15] combined an FPN backbone, dense anchors (9 per location) and two small subnetworks for classification and box regression, trained with the focal loss. The paper's analysis showed that class imbalance, not the architecture, had been the main reason one-stage detectors lagged behind two-stage ones.

## Anchor-free detectors: points instead of boxes

**Plain version.** Anchors are guesses about what objects look like. Anchor-free detectors drop them and predict boxes from *points*: corners, centers, or every pixel inside the object.

**CornerNet (2018)** [27] detects an object as a pair of keypoints: a heatmap for top-left corners, a heatmap for bottom-right corners, and an **embedding** per corner. Corners from the same object are trained to have similar embeddings, so pairing is done by embedding distance. Because a corner often lies outside the object, the paper introduced **corner pooling**, which looks along rows and columns to collect evidence for a corner.

**CenterNet, "Objects as Points" (2019)** [28] represents each object by its center. The network outputs a heatmap $\hat{Y} \in [0,1]^{\frac{W}{R} \times \frac{H}{R} \times K}$ (output stride $R$, one channel per class), plus a size $(w, h)$ and a sub-pixel offset at each location. Ground-truth centers are splatted with a Gaussian, and the heatmap is trained with a pixel-wise focal loss. At inference, peaks are simply local maxima, found with a $3 \times 3$ max-pooling comparison, so **no IoU-based NMS** is needed. The same design extends to 3D boxes and human pose by regressing more quantities at the center.

**FCOS (2019)** [29] treats detection like segmentation. Every feature-map location that falls inside a ground-truth box is a positive sample and regresses the four distances to the box sides,

$$l^* = x - x_1, \qquad t^* = y - y_1, \qquad r^* = x_2 - x, \qquad b^* = y_2 - y,$$

where $(x, y)$ is the location in image coordinates and $(x_1, y_1, x_2, y_2)$ the box. Each FPN level only handles objects whose largest regression distance falls in its own range, which resolves most ambiguity when boxes overlap. Locations far from the object's center produce poor boxes, so FCOS adds a **centerness** branch,

$$\text{centerness} = \sqrt{\frac{\min(l^*, r^*)}{\max(l^*, r^*)} \cdot \frac{\min(t^*, b^*)}{\max(t^*, b^*)}},$$

that is 1 at the center and decays toward the edges; it multiplies the class score at test time so off-center boxes rank lower before NMS. FCOS removed every anchor hyperparameter and matched or beat anchor-based RetinaNet.

## Set prediction with transformers: DETR and its descendants

**Plain version.** Instead of predicting thousands of candidates and deleting duplicates, predict a small *set* of answers directly, and train so that each real object is claimed by exactly one prediction.

**DETR (2020)** [30] uses a CNN backbone, a transformer encoder over the feature map, and a transformer decoder that takes $N$ learned **object queries** (100 in the paper) and outputs $N$ (class, box) predictions in parallel. There are no anchors, no proposals and no NMS. Since $N$ exceeds the number of objects, a special "no object" class $\emptyset$ pads the ground truth to size $N$.

The key is training with **bipartite matching**. Let $y = \{y_j\}$ be the padded ground truth and $\hat{y} = \{\hat{y}_i\}_{i=1}^{N}$ the predictions. Find the permutation $\hat{\sigma}$ of $N$ elements with lowest total matching cost,

$$\hat{\sigma} = \arg\min_{\sigma \in \mathfrak{S}_N} \sum_{j=1}^{N} \mathcal{L}_{\text{match}}(y_j, \hat{y}_{\sigma(j)}), \qquad \mathcal{L}_{\text{match}} = -\mathbb{1}_{\{c_j \neq \emptyset\}}\, \hat{p}_{\sigma(j)}(c_j) + \mathbb{1}_{\{c_j \neq \emptyset\}}\, \mathcal{L}_{\text{box}}(b_j, \hat{b}_{\sigma(j)}),$$

where $\mathfrak{S}_N$ is the set of permutations, $c_j$ and $b_j$ are the class and box of target $j$, $\hat{p}_{\sigma(j)}(c_j)$ is the predicted probability of class $c_j$ for the matched query, and

$\mathcal{L}_{\text{box}} = \lambda_{L1}\|b_j - \hat{b}_{\sigma(j)}\|_1 + \lambda_{\text{iou}}(1 - \text{GIoU}(b_j, \hat{b}_{\sigma(j)}))$. This assignment problem is solved exactly by the **Hungarian algorithm** [31] in polynomial time. The training loss (the **set loss**) then uses the matched pairs:

$$\mathcal{L}_{\text{Hungarian}} = \sum_{j=1}^{N} \Big[ -\log \hat{p}_{\hat{\sigma}(j)}(c_j) + \mathbb{1}_{\{c_j \neq \emptyset\}}\, \mathcal{L}_{\text{box}}(b_j, \hat{b}_{\hat{\sigma}(j)}) \Big],$$

with the log-probability term down-weighted by a factor of 10 for $c_j = \emptyset$ to balance the many "no object" slots. Because each object is matched to exactly one query, a duplicate prediction is *penalized* as a false "no object", and the decoder's self-attention learns to suppress duplicates, which is why NMS is unnecessary.

The matching step is short with SciPy. The cost below combines the three terms with DETR's weights; `linear_sum_assignment` accepts the rectangular $N \times M$ matrix directly (DETR computes the L1 term on normalized $(c_x, c_y, w, h)$; corner format is used here for brevity).

```
import numpy as np
from scipy.optimize import linear_sum_assignment

def giou_matrix(a, b):
```

```
    x1 = np.maximum(a[:, None, 0], b[None, :, 0]); y1 = np.maximum(a[:, None, 1], b[None, :,
     ↪  1])
    x2 = np.minimum(a[:, None, 2], b[None, :, 2]); y2 = np.minimum(a[:, None, 3], b[None, :,
     ↪  3])
    inter = np.clip(x2 - x1, 0, None) * np.clip(y2 - y1, 0, None)
    area = lambda b: (b[:, 2] - b[:, 0]) * (b[:, 3] - b[:, 1])
    union = area(a)[:, None] + area(b)[None, :] - inter
    cx1 = np.minimum(a[:, None, 0], b[None, :, 0]); cy1 = np.minimum(a[:, None, 1], b[None,
     ↪  :, 1])
    cx2 = np.maximum(a[:, None, 2], b[None, :, 2]); cy2 = np.maximum(a[:, None, 3], b[None,
     ↪  :, 3])
    c = (cx2 - cx1) * (cy2 - cy1)
    return inter / union - (c - union) / c

rng = np.random.default_rng(1)
N, C = 6, 3                                      # 6 queries, 3 real classes (+ "no object")
logits = rng.normal(size=(N, C + 1))
prob = np.exp(logits) / np.exp(logits).sum(1, keepdims=True)
pred_boxes = np.array([[.10, .10, .40, .50], [.55, .20, .90, .60], [.12, .08, .42, .48],
                       [.30, .60, .60, .95], [.00, .00, .20, .20], [.50, .50, .70, .70]])
gt_boxes  = np.array([[.10, .10, .40, .50], [.32, .62, .62, .93], [.56, .22, .88, .62]])
gt_labels = np.array([0, 2, 1])

w_cls, w_l1, w_giou = 1.0, 5.0, 2.0              # weights used by DETR
cost = (-w_cls * prob[:, gt_labels]                          # (N, M) class term
        + w_l1 * np.abs(pred_boxes[:, None] - gt_boxes[None]).sum(-1)
        - w_giou * giou_matrix(pred_boxes, gt_boxes))
rows, cols = linear_sum_assignment(cost)         # works on rectangular N x M matrices
print("query -> gt:", dict(zip(rows.tolist(), cols.tolist())))
print("unmatched queries (trained as 'no object'):", sorted(set(range(N)) -
 ↪  set(rows.tolist())))
```

```
query -> gt: {0: 0, 1: 2, 3: 1}
unmatched queries (trained as 'no object'): [2, 4, 5]
```

Query 2 is a near-duplicate of query 0, yet it is left unmatched and will be trained toward "no object". That is the mechanism that replaces NMS.

**The price.** DETR needed very long training schedules (hundreds of epochs, compared with roughly a dozen for Faster R-CNN) and was weak on small objects, because global attention over a high-resolution feature map is expensive, so it used only a single coarse level. The descendants fixed these problems one by one.

Three descendants show the direction. **Deformable DETR** [32] lets each query look at only a few sampled points across several feature levels, which makes multi-scale features affordable and shortens training a lot. **DINO** [33] (not the self-supervised method of the same name) speeds up convergence further with denoising training and better query initialization. **RT-DETR** [34] makes the design fast enough for real-time use, still without NMS.

*Online only: DETR descendants in detail — https://blog.shyandram.dev/en/learn/deep-learning/object-detection/notes/detr-descendants-in-detail/*

Today the DETR line and the YOLO line are converging: YOLOv10 borrows one-to-one assignment to drop NMS, and real-time DETRs borrow efficient convolutional necks.

## Open-vocabulary and grounding detectors

**Plain version.** A closed-set detector can only find the 80 or 365 classes it was trained on. An open-vocabulary detector takes the class list *as text at test time*, so you can ask for "red fire hydrant" or "forklift" without retraining.

The trick is to replace the last classification layer, a fixed matrix of $K$ class weight vectors, with **text embeddings**. For a region feature $\mathbf{z}_i$ and the text embedding $\mathbf{e}_k$ of a prompt such as "a photo of a forklift", the class score is a scaled cosine similarity,

$$s_{ik} = \tau \cdot \frac{\mathbf{z}_i^\top \mathbf{e}_k}{\|\mathbf{z}_i\| \, \|\mathbf{e}_k\|},$$

where $\tau$ is a learned temperature. Any phrase you can embed becomes a class. The vision side must learn to align regions with language, which requires image–text data far larger than any box-annotated dataset.

- **OWL-ViT (2022)** [35] starts from a transformer image encoder and a text encoder pretrained contrastively on image–text pairs, removes the final pooling, and attaches light box and class heads to every output token. The class head is the text embedding above. It supports zero-shot (text query) and one-shot (image-exemplar query) detection, and the paper shows consistent gains with model scale.
- **Grounding DINO (2024)** [36] fuses language deeply into a DINO detector: a feature enhancer with cross-attention between image and text, language-guided query selection, and a cross-modality decoder. It is pretrained on detection, grounding (phrase-to-region) and caption data, and reports 52.5 AP on COCO zero-shot transfer (with no COCO training data) and a 26.1 mean AP on a benchmark collection of diverse real-world detection datasets.
- **YOLO-World (2024)** [37] brings open vocabulary to real-time detectors with a vision–language path aggregation network (RepVL-PAN) and region–text contrastive pretraining. Its *prompt-then-detect* mode encodes the user's vocabulary once offline, so inference costs about the same as a closed-set YOLO. It reports 35.4 AP on LVIS at 52.0 FPS on a V100 GPU.

**Promptable foundation models** go one step further. SAM 3 [38] (2025) accepts a short noun phrase or image exemplars as a *concept prompt* and returns masks with identities for **all** matching instances in images and videos; the authors built a data engine producing a dataset with 4M unique concept labels and report doubling the accuracy of existing systems on their promptable concept segmentation benchmark. In such systems, detection becomes one output of a general "find this concept" model rather than a separately trained network.

## Small objects and dense scenes

**Plain version.** Tiny objects have very few pixels, and stride-32 features may squeeze them into less than one cell. Crowds are hard because objects hide each other and NMS confuses neighbors with duplicates.

**Small objects.** COCO defines "small" as area below $32^2$ pixels [40]. A roughly square object near that area limit spans about $2 \times 2$ stride-16 feature-cell widths; small area alone does not bound both side lengths for a thin object. Downsampling can remove the detail needed to localize it. IoU also becomes harsh: a 2-pixel error on a $10 \times 10$ box can drop IoU below 0.7. Common remedies are high-resolution feature levels ($P_2$ with stride 4), feature pyramids, image tiling or slicing at inference, larger input resolutions, copy-paste and scale augmentation, and assignment rules that do not starve small boxes of positive samples. The survey and benchmark of Cheng et al. [39] reviews these strategies and introduces two large small-object benchmarks: SODA-D (24,828 traffic images, 278,433 instances, 9 categories) and SODA-A (2,513 high-resolution aerial images, 872,069 instances, 9 classes).

**Dense and occluded scenes.** In crowds, true objects overlap heavily, so a fixed NMS threshold trades recall against duplicates. Soft-NMS [13] helps; set-prediction detectors avoid the threshold altogether by learning which predictions are duplicates. Occlusion also makes box annotation ambiguous (full extent or visible part?), and evaluation then depends on the protocol. These issues return in tracking, where objects disappear and reappear.

## Evaluating detectors

**Plain version.** Sort all detections by confidence. Walk down the list: each detection is either a hit (it matches an unclaimed real object well enough) or a false alarm. Track how precision and recall change; the area under that curve is the Average Precision.

### Precision, recall and AP

For one class and one IoU threshold $\theta$, sort all detections in the test set by descending score. A detection is a **true positive** (TP) if its IoU with a not-yet-matched ground-truth box of the same class is at least $\theta$; otherwise it is a **false positive** (FP). Each ground-truth box can be matched only once, which is how duplicates are punished. After the top $k$ detections,

$$\text{precision}(k) = \frac{\text{TP}(k)}{k}, \qquad \text{recall}(k) = \frac{\text{TP}(k)}{M},$$

where $M$ is the number of ground-truth objects of that class. To make the curve monotone, precision is **interpolated** with its upper envelope, $p_{\text{interp}}(r) = \max_{r' \geq r} p(r')$. Then:

- **VOC 11-point AP** (PASCAL VOC up to 2009 [3]): $\text{AP} = \frac{1}{11} \sum_{r \in \{0, 0.1, \ldots, 1\}} p_{\text{interp}}(r)$.
- **VOC all-point AP** (VOC 2010 onwards [41]): the exact area under the interpolated curve, $\text{AP} = \sum_n (r_{n+1} - r_n)\, p_{\text{interp}}(r_{n+1})$ over the distinct recall values.

- **COCO AP** [40]: the mean of $p_{\text{interp}}$ at 101 recall points $0, 0.01, \ldots, 1$.

Figure 6 shows a PR curve, its envelope and the AP area.

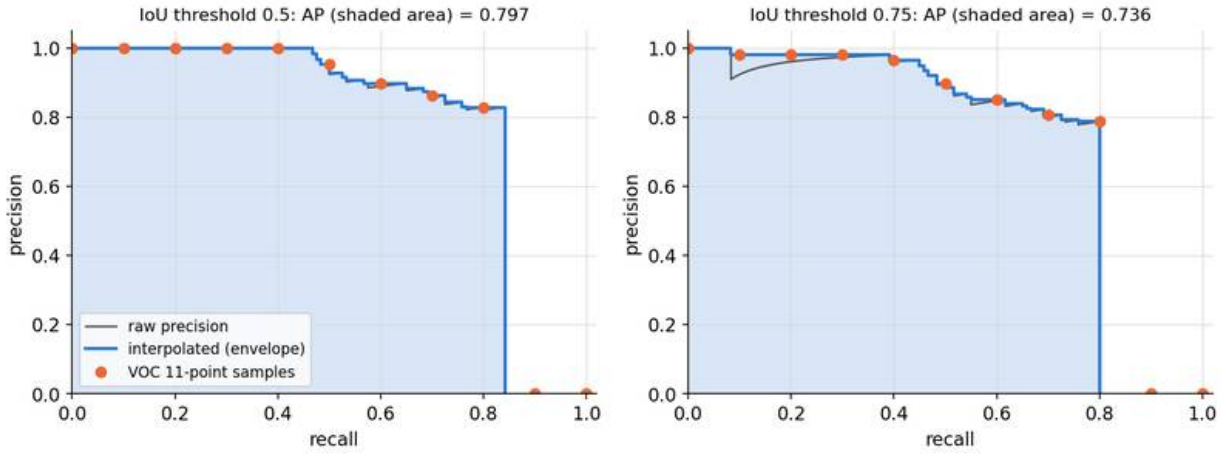



*Figure 6 — Precision–recall on synthetic detections at two IoU thresholds. The shaded area under the interpolated envelope is the all-point AP; orange dots are the VOC 11-point samples. Precision drops to 0 where recall ends, because some objects were never detected.*

### The COCO protocol

COCO's primary metric, written **AP** or **mAP@[.5:.95]**, averages AP over 10 IoU thresholds $\theta \in \{0.50, 0.55, \ldots, 0.95\}$ and over all 80 categories [40]:

$$\text{AP}_{\text{COCO}} = \frac{1}{10\,K} \sum_{\theta} \sum_{k=1}^{K} \text{AP}_k(\theta).$$

It rewards precise localization, not just finding the object. Companion metrics are $\text{AP}_{50}$ (the VOC-style metric), $\text{AP}_{75}$ (strict), and $\text{AP}_S$, $\text{AP}_M$, $\text{AP}_L$ for small (area $< 32^2$), medium ($32^2$ to $96^2$) and large ($> 96^2$) objects. In the default category-aware COCO evaluator, the 100-detection cutoff is applied per image **and category**; a detector may impose its own stricter total-output cap. Average recall (AR) uses maximum-detection settings of 1, 10 and 100 under the same category-wise evaluation [40]. Note that COCO calls its category-averaged metric "AP"; "mAP" in papers usually means the same thing.

The code below computes everything from scratch on synthetic predictions (jittered ground truth, about 15% misses and some random false positives).

```
import numpy as np

def box_iou(a, b):
    """Pairwise IoU between (N,4) and (M,4) xyxy boxes -> (N, M)."""
    x1 = np.maximum(a[:, None, 0], b[None, :, 0]); y1 = np.maximum(a[:, None, 1], b[None, :,
     ↪  1])
```

```
    x2 = np.minimum(a[:, None, 2], b[None, :, 2]); y2 = np.minimum(a[:, None, 3], b[None, :,
     ↪  3])
    inter = np.clip(x2 - x1, 0, None) * np.clip(y2 - y1, 0, None)
    area = lambda b: (b[:, 2] - b[:, 0]) * (b[:, 3] - b[:, 1])
    return inter / (area(a)[:, None] + area(b)[None, :] - inter)

def match(dets, gts, thr):
    """dets: list over images of (boxes, scores); gts: list of gt boxes.
    Returns scores, TP flags (sorted by score), and number of GT."""
    recs = []
    for img, ((db, ds), gb) in enumerate(zip(dets, gts)):
        for k in range(len(ds)):
            recs.append((ds[k], img, k))
    recs.sort(key=lambda r: -r[0])                      # global ranking by confidence
    used = [np.zeros(len(g), bool) for g in gts]
    tp = np.zeros(len(recs))
    for n, (s, img, k) in enumerate(recs):
        gb = gts[img]
        if len(gb) == 0:
            continue
        ious = box_iou(dets[img][0][k:k + 1], gb)[0]
        ious[used[img]] = -1                            # each GT can be matched once
        j = ious.argmax()
        if ious[j] >= thr:
            tp[n] = 1; used[img][j] = True
    return tp, sum(len(g) for g in gts)

def pr_curve(tp, n_gt):
    ctp, cfp = np.cumsum(tp), np.cumsum(1 - tp)
    return ctp / n_gt, ctp / (ctp + cfp)                # recall, precision

def ap(rec, prec, mode="all"):
    r = np.concatenate([[0], rec, [1]]); p = np.concatenate([[0], prec, [0]])
    p = np.maximum.accumulate(p[::-1])[::-1]            # monotone envelope
    if mode == "all":                                   # VOC2010+: exact area
        i = np.where(r[1:] != r[:-1])[0]
        return np.sum((r[i + 1] - r[i]) * p[i + 1])
    pts = np.linspace(0, 1, 11 if mode == "voc11" else 101)
    return np.mean([p[r >= t].max() if np.any(r >= t) else 0 for t in pts])

# synthetic data: 50 images, 1-4 objects each; detector = jittered GT + false positives
rng = np.random.default_rng(0)
gts, dets = [], []
for _ in range(50):
    n = rng.integers(1, 5)
    xy = rng.uniform(0, 200, (n, 2)); wh = rng.uniform(20, 80, (n, 2))
    g = np.hstack([xy, xy + wh]); gts.append(g)
    keep = rng.random(n) > 0.15                         # miss ~15% of objects
    jit = rng.normal(0, 0.06, (n, 4)) * np.hstack([wh, wh])
    db = (g + jit)[keep]; ds = rng.uniform(0.4, 1.0, keep.sum())
    m = rng.integers(0, 3)                              # 0-2 false positives
    fxy = rng.uniform(0, 200, (m, 2)); fwh = rng.uniform(20, 80, (m, 2))
    db = np.vstack([db, np.hstack([fxy, fxy + fwh])]); ds = np.concatenate([ds,
     ↪  rng.uniform(0, 0.7, m)])
    dets.append((db, ds))
```

```
for thr in (0.5, 0.75):
    tp, n_gt = match(dets, gts, thr)
    rec, prec = pr_curve(tp, n_gt)
    print(f"IoU {thr}: VOC11 {ap(rec, prec, 'voc11'):.3f}  all-point {ap(rec, prec):.3f}"
          f"  COCO-101 {ap(rec, prec, 'coco'):.3f}")
aps = [ap(*pr_curve(*match(dets, gts, t)), "coco") for t in np.arange(0.5, 0.96, 0.05)]
print("AP@[.5:.95] =", round(float(np.mean(aps)), 3))
```

```
IoU 0.5: VOC11 0.776  all-point 0.797  COCO-101 0.797
IoU 0.75: VOC11 0.749  all-point 0.736  COCO-101 0.738
AP@[.5:.95] = 0.532
```

Three lessons are visible. The three interpolation rules give similar but *not identical* numbers, so AP values computed with different protocols must never be compared. AP@[.5:.95] is much lower than AP50 because jittered boxes fail the strict thresholds. And this is a single class; real mAP averages over classes, so rare classes count as much as frequent ones.

### What AP does not tell you

AP summarizes ranking quality over the whole test set, so it does not choose an operating threshold, and it does not separate *types* of errors: localization errors, confusion with similar classes, duplicates and background false positives all look the same. When diagnosing a model, break errors down by type and by object size, and look at the images.

## Datasets and benchmarks

**Plain version.** Detection progress has been driven by a few public datasets; each one changed what "good" meant.

| Dataset | Classes | Scale (from the papers) | Annotations | Why it matters |
|---|---|---|---|---|
| PASCAL VOC [3] | 20 | Small by today's standards | Boxes (+ some masks) | The 2005–2012 benchmark; AP50 |
| MS COCO [4] | 91 types labeled; 80 used for detection | 328k images, 2.5M labeled instances | Boxes + instance masks | Cluttered everyday scenes, many small objects; AP@[.5:.95] |
| Objects365 [42] | 365 | Large-scale, high-quality boxes | Boxes | Common pretraining set for strong detectors |
| Open Images V4 [43] | 600 boxable classes | 9.2M images; 15.4M boxes | Boxes, image labels, relationships | Very large; 19.8k concepts at image level |
| LVIS [44] | over 1000 | ~2M masks in 164k images | Instance masks | Long-tailed vocabulary; rare-class evaluation |

Two trends stand out. Category counts grew (20, then 80, then hundreds and over a thousand), and with them came **long-tail** problems: in LVIS many categories have only a handful of training examples, so a loss or sampler that ignores frequency fails on rare classes. And the biggest models are now pretrained on Objects365 or on image–text data before fine-tuning on COCO, which makes "trained on COCO only" and "with extra data" separate leaderboards.

## Speed, accuracy and deployment

**Plain version.** In practice you pick a point on a curve: more accuracy usually means more computation. What counts is the latency on *your* hardware, end to end.

- **FLOPs are not latency.** FLOPs count arithmetic, but runtime also depends on memory traffic, operator support and parallelism. Depth-wise convolutions and attention can be cheap in FLOPs but slow on some accelerators.
- **Measure end to end.** Report latency including preprocessing, the network *and* post-processing. NMS cost grows with the number of candidate boxes (worst case quadratic in their number), is data-dependent, and often runs on the CPU. RT-DETR's paper [34] analyses exactly this and is one reason NMS-free designs such as RT-DETR and YOLOv10 [24] are attractive for deployment.
- **Input resolution** is the strongest knob: halving the side length cuts compute by about 4 times but hurts small objects most.
- **Quantization** (e.g. 8-bit integer weights and activations) and **re-parameterization** (training a multi-branch block, then folding it into a single convolution, as used by YOLOv7 [22]) reduce latency with small accuracy loss. Box-regression outputs and the score head are often the most sensitive to quantization, so calibrate with real data and check AP75, not just AP50.
- **Batch size 1** is the usual edge setting; throughput numbers measured with large batches do not transfer.

### A tiny detector you can train

To see box regression and an IoU loss work end to end, the PyTorch script below trains a small CNN to localize one rectangle or ellipse in noisy $32 \times 32$ images. It is a single-object detector: the network directly regresses $(c_x, c_y, w, h)$ in $[0, 1]$ through a sigmoid, trained with L1 plus a GIoU loss, as DETR does for its matched queries.

```
import time, torch, torch.nn as nn
torch.manual_seed(0)
S = 32
torch.set_num_threads(1)  # avoid CPU oversubscription in the small teaching example

def make_batch(n):
    """One white rectangle or ellipse per 32x32 noisy image; target = (cx, cy, w, h) in [0,
     ↪  1]."""
    img = 0.1 * torch.randn(n, 1, S, S)
```

```
    wh = torch.rand(n, 2) * 0.4 + 0.15
    c = wh / 2 + torch.rand(n, 2) * (1 - wh)
    yy, xx = torch.meshgrid(torch.arange(S), torch.arange(S), indexing="ij")
    xx, yy = (xx[None] + 0.5) / S, (yy[None] + 0.5) / S
    dx = (xx - c[:, 0, None, None]) / (wh[:, 0, None, None] / 2)
    dy = (yy - c[:, 1, None, None]) / (wh[:, 1, None, None] / 2)
    rect = (dx.abs() <= 1) & (dy.abs() <= 1)
    ell = dx**2 + dy**2 <= 1
    shape = torch.where((torch.rand(n) < 0.5)[:, None, None], rect, ell)
    return img + shape[:, None].float(), torch.cat([c, wh], 1)

def xyxy(b):
    return torch.cat([b[:, :2] - b[:, 2:] / 2, b[:, :2] + b[:, 2:] / 2], 1)

def giou(p, g):
    p, g = xyxy(p), xyxy(g)
    lt, rb = torch.max(p[:, :2], g[:, :2]), torch.min(p[:, 2:], g[:, 2:])
    inter = (rb - lt).clamp(min=0).prod(1)
    area = lambda b: (b[:, 2:] - b[:, :2]).prod(1)
    union = area(p) + area(g) - inter
    c = (torch.max(p[:, 2:], g[:, 2:]) - torch.min(p[:, :2], g[:, :2])).prod(1)
    return inter / union, inter / union - (c - union) / c

net = nn.Sequential(
    nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),   # 16
    nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),  # 8
    nn.Conv2d(32, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),  # 4
    nn.Flatten(), nn.Linear(32 * 4 * 4, 128), nn.ReLU(),
    nn.Linear(128, 4), nn.Sigmoid())                               # (cx, cy, w, h) in [0, 1]
opt = torch.optim.Adam(net.parameters(), 2e-3)

t0 = time.time()
for step in range(500):
    x, y = make_batch(64)
    pred = net(x)
    iou, g = giou(pred, y)
    loss = (pred - y).abs().sum(1).mean() + (1 - g).mean()        # L1 + GIoU loss
    opt.zero_grad(); loss.backward(); opt.step()

with torch.no_grad():
    x, y = make_batch(1000)
    iou, _ = giou(net(x), y)
print(f"train time {time.time() - t0:.1f}s | test mean IoU {iou.mean():.3f}"
      f" | IoU>=0.5: {(iou >= 0.5).float().mean():.1%} | IoU>=0.75: {(iou >=
      ↪ 0.75).float().mean():.1%}")
```

```
train time 25.3s | test mean IoU 0.882 | IoU>=0.5: 99.9% | IoU>=0.75: 97.3%
```

(One CPU run is shown; timing and final scores can vary with the hardware and numerical libraries.) Because there is exactly one object, there is no assignment and no NMS. Exercise 6 asks you to extend this to several objects, which is where every hard problem in this tutorial reappears.

## Modern view

Figure 7 places the methods of this tutorial on a timeline, using the year of each cited publication.

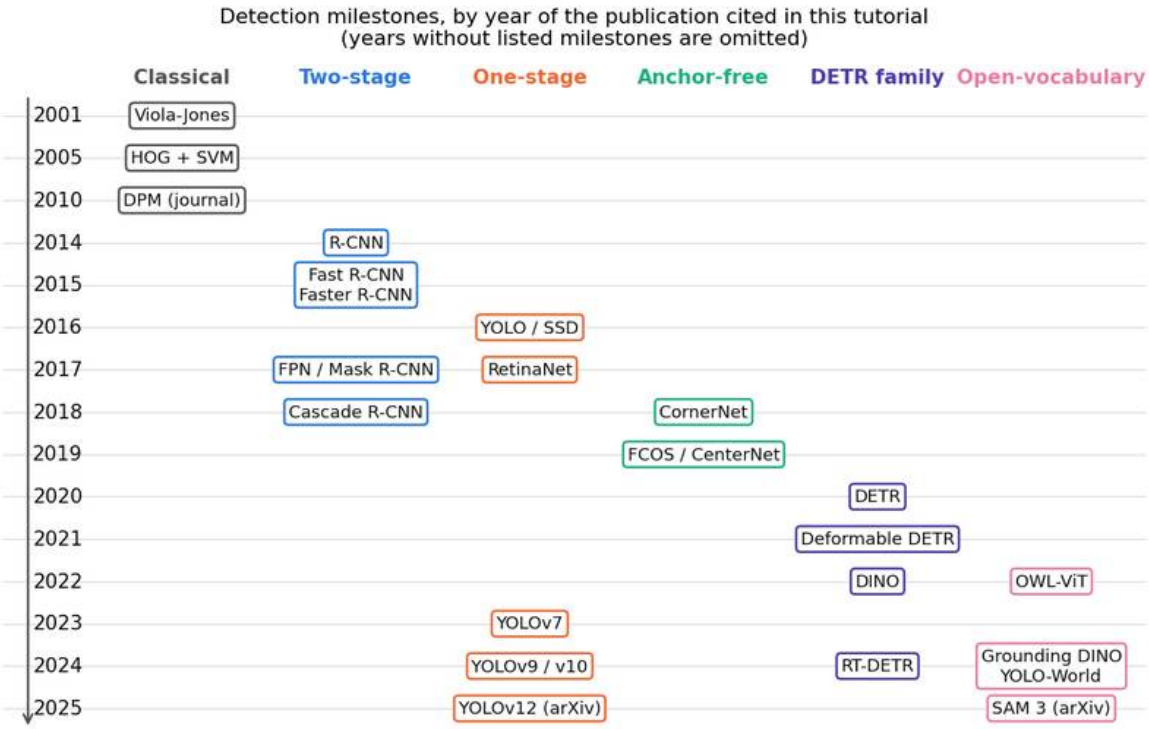



*Figure 7 — Milestones by year of the publication cited in this tutorial, grouped by family. Years without a listed milestone are omitted.*

### What the surveys say

Five surveys frame the field. Zou et al. [1] give the broadest history, from Viola–Jones to deep detectors, and show that most "new" ideas have older roots: multi-scale scanning became feature pyramids, and hard-negative mining became the focal loss. Liu et al. [2] are the best single reference for the CNN era and stress that training strategies, data and evaluation mattered as much as architectures. Shehzadi et al. [45] review recent DETR improvements, Zhu and Chen [46] organize open-vocabulary methods and warn that vocabulary overlap between pretraining and test data complicates evaluation, and Cheng et al. [39] show that small objects remain far harder than large ones.

*Online only: Guide to the Modern view surveys — https://blog.shyandram.dev/en/learn/deep-learning/object-detection/papers/guide-to-the-modern-view-surveys/*

### The state of the art, 2023–2026

- **Real-time closed-set detection** is now a contest between the YOLO line (YOLOv9 [23], YOLOv10 [24], YOLOv12 [25]) and real-time DETRs (RT-DETR [34]). The two lines borrow from each other: one-to-one assignment and NMS-free heads in YOLO, convolutional necks and hybrid encoders in DETRs. Attention is now affordable at real-time budgets.
- **High-accuracy closed-set detection** is dominated by DETR-style models such as DINO [33] with large backbones and large-scale pretraining (Objects365 [42] and beyond). Gains on COCO have become small, and the benchmark is close to saturated for the largest models, which is pushing attention to harder benchmarks: LVIS [44], open vocabulary and cross-domain evaluation.
- **Open-vocabulary and grounding detection** (OWL-ViT [35], Grounding DINO [36], YOLO-World [37]) is moving into practice: in many applications you now *prompt* a detector rather than train one, and then optionally fine-tune it.
- **Promptable foundation models** (SAM 3 [38]) unify detection, segmentation and tracking under concept prompts.

### Open problems

1. **Localization quality versus confidence.** Classification score and box quality are trained separately and are often poorly correlated, so the "best" box can be suppressed by a more confident but worse one.
2. **Small, crowded and occluded objects** remain far from solved [39].
3. **Long-tail and open-world recognition.** Rare classes and truly unseen objects, and knowing *when* the model does not know.
4. **Evaluation.** COCO AP hides error types, is saturated at the top, and open-vocabulary evaluation depends on prompts and on what the pretraining data already contained [46].
5. **Efficiency on real hardware**, including NMS-free designs, quantization-friendly heads and fair latency reporting.

## Key takeaways

- Detection predicts a variable-size set of boxes, classes and scores.
- Box parameterization defines localization targets; assignment determines which prediction learns each object.
- NMS suppresses duplicate predictions; crowded scenes make that decision harder.
- Feature pyramids address scale, while focal loss changes the contribution of easy and hard examples.
- Separate proposal refinement, dense prediction and one-to-one set matching when comparing architectures.
- Open-vocabulary detection changes the class interface; it still needs localization and vocabulary-aware evaluation.
- Report the AP convention, IoU thresholds, object scales and complete timing protocol.

## Exercises

1. **Encode by hand.** An anchor has center $(100, 100)$ and size $64 \times 32$. A ground-truth box has center $(110, 96)$ and size $80 \times 40$. Compute $(t_x, t_y, t_w, t_h)$ without normalization, then decode them back.

   **Hint**

   $t_x = 10/64 = 0.156$, $t_y = -4/32 = -0.125$, $t_w = \log(80/64) = 0.223$, $t_h = \log(40/32) = 0.223$. Decoding: $c_x = 100 + 0.156 \cdot 64 = 110$, $w = 64e^{0.223} = 80$, and similarly for $y$ and $h$.

2. **IoU versus shift.** Show that two equal $a \times a$ squares shifted by $d$ along one axis have IoU $= (a - d)/(a + d)$. What shift gives IoU 0.5 and 0.75? Repeat for an equal shift along both axes (Figure 2, right).

   **Hint**

   Intersection $a(a - d)$, union $2a^2 - a(a - d) = a(a + d)$. IoU 0.5 at $d = a/3$; IoU 0.75 at $d = a/7$. For a diagonal shift, IoU $= (a - d)^2/(2a^2 - (a - d)^2)$, so IoU 0.5 at $d = a(1 - \sqrt{2/3}) \approx 0.18a$.

3. **When greedy NMS fails.** Modify the NMS example so that two different people overlap with IoU 0.6, and run greedy NMS at thresholds 0.5 and 0.7, and Soft-NMS. Which recovers both people, and what new problem does a high threshold create?

   **Hint**

   At 0.5 the lower-scored person is deleted. At 0.7 both survive, but duplicates of the same person with IoU between 0.5 and 0.7 also survive, adding false positives. Soft-NMS keeps the second person with a reduced score ($e^{-0.36/0.5} \approx 0.49$ times its original score), which still counts toward recall in AP.

4. **Focal loss numbers.** With $\gamma = 2$ and $\alpha = 0.25$, compute the focal loss and the plain cross-entropy for a background anchor with $p = 0.01$ (easy) and $p = 0.6$ (hard). By what factor does each change?

   **Hint**

   For background, $p_t = 1 - p$ and $\alpha_t = 0.75$. Easy: CE $= -\log 0.99 = 0.01005$; FL $= 0.75 \cdot 0.01^2 \cdot 0.01005 \approx 7.5{\times}10^{-7}$, a factor of about $1.3{\times}10^4$ smaller. Hard: CE $= -\log 0.4 = 0.916$; FL $= 0.75 \cdot 0.6^2 \cdot 0.916 \approx 0.247$, only about 3.7 times smaller.

5. **Protocols disagree.** Using the AP code, find (by changing the random seed or the jitter) a case where VOC 11-point AP is *higher* than all-point AP, and one where it is lower. Explain why neither is always larger.

**Hint**

The 11-point rule samples the envelope at fixed recalls, so it overestimates when the envelope is high just to the right of a sample point and drops right after it, and underestimates in the opposite case. The output above already shows both: at IoU 0.5 the 11-point value is lower, at 0.75 it is higher.

6. **From one object to many.** Extend the tiny PyTorch detector to images with up to three shapes. Option A: predict an $8 \times 8$ grid of (objectness, box) like YOLO v1, assigning each object to the cell containing its center. Option B: predict 5 (class, box) slots and train with Hungarian matching like DETR. Compare the failure modes.

**Hint**

Option A fails when two centers fall in the same cell and needs NMS to remove duplicates from neighboring cells. Option B needs no NMS, but trains more slowly, and early in training the matching is unstable (the same object may be assigned to different slots from step to step). Use `scipy.optimize.linear_sum_assignment` on a detached cost matrix.

Detections are the raw material of tracking. The next step, linking boxes across video frames and keeping identities, is covered in DLCV 07 ·Object Tracking.

## References


1. Z. Zou, K. Chen, Z. Shi, Y. Guo and J. Ye, "Object detection in 20 years: A survey," *Proceedings of the IEEE*, vol. 111, no. 3, 2023. doi
2. L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu and M. Pietikäinen, "Deep learning for generic object detection: A survey," *International Journal of Computer Vision*, 2020. doi
3. M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn and A. Zisserman, "The PASCAL Visual Object Classes (VOC) challenge," *International Journal of Computer Vision*, vol. 88, 2010. doi
4. T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick et al., "Microsoft COCO: Common objects in context," in *Proc. ECCV*, 2014. doi
5. P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proc. IEEE CVPR*, 2001. doi
6. N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. IEEE CVPR*, 2005. doi
7. P. F. Felzenszwalb, R. B. Girshick, D. McAllester and D. Ramanan, "Object detection with discriminatively trained part-based models," *IEEE TPAMI*, 2010. doi
8. R. Girshick, J. Donahue, T. Darrell and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE CVPR*, 2014. arXiv
9. R. Girshick, "Fast R-CNN," in *Proc. IEEE ICCV*, 2015. arXiv
10. S. Ren, K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *IEEE TPAMI*, 2017 (conference version 2015). doi
11. H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid and S. Savarese, "Generalized intersection over union: A metric and a loss for bounding box regression," in *Proc. IEEE/CVF CVPR*, 2019. arXiv

12. Z. Zheng, P. Wang, W. Liu, J. Li, R. Ye and D. Ren, “Distance-IoU loss: Faster and better learning for bounding box regression,” in *Proc. AAAI*, 2020. arXiv
13. N. Bodla, B. Singh, R. Chellappa and L. S. Davis, “Soft-NMS — Improving object detection with one line of code,” in *Proc. IEEE ICCV*, 2017. arXiv
14. T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan and S. Belongie, “Feature pyramid networks for object detection,” in *Proc. IEEE CVPR*, 2017. doi
15. T.-Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár, “Focal loss for dense object detection,” in *Proc. IEEE ICCV*, 2017. doi
16. K. He, G. Gkioxari, P. Dollár and R. Girshick, “Mask R-CNN,” in *Proc. IEEE ICCV*, 2017. doi
17. Z. Cai and N. Vasconcelos, “Cascade R-CNN: Delving into high quality object detection,” in *Proc. IEEE/CVF CVPR*, 2018. doi
18. J. Redmon, S. Divvala, R. Girshick and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proc. IEEE CVPR*, 2016. doi
19. J. Redmon and A. Farhadi, “YOLO9000: Better, faster, stronger,” *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7263–7271, 2017. CVF ·arXiv
20. J. Redmon and A. Farhadi, “YOLOv3: An incremental improvement,” tech. report, arXiv:1804.02767, 2018. arXiv
21. A. Bochkovskiy, C.-Y. Wang and H.-Y. M. Liao, “YOLOv4: Optimal speed and accuracy of object detection,” arXiv:2004.10934, 2020. arXiv
22. C.-Y. Wang, A. Bochkovskiy and H.-Y. M. Liao, “YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors,” in *Proc. IEEE/CVF CVPR*, 2023. doi
23. C.-Y. Wang, I-H. Yeh and H.-Y. M. Liao, “YOLOv9: Learning what you want to learn using programmable gradient information,” in *Proc. ECCV*, 2024. doi
24. A. Wang, H. Chen, L. Liu, K. Chen, Z. Lin et al., “YOLOv10: Real-time end-to-end object detection,” in *Proc. NeurIPS*, 2024. arXiv
25. Y. Tian, Q. Ye and D. Doermann, “YOLOv12: Attention-centric real-time object detectors,” arXiv:2502.12524, 2025. arXiv
26. W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu and A. C. Berg, “SSD: Single shot multibox detector,” in *Proc. ECCV*, 2016. arXiv
27. H. Law and J. Deng, “CornerNet: Detecting objects as paired keypoints,” in *Proc. ECCV*, 2018. doi
28. X. Zhou, D. Wang and P. Krähenbühl, “Objects as points,” arXiv:1904.07850, 2019. arXiv
29. Z. Tian, C. Shen, H. Chen and T. He, “FCOS: Fully convolutional one-stage object detection,” in *Proc. IEEE/CVF ICCV*, 2019. arXiv
30. N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov and S. Zagoruyko, “End-to-end object detection with transformers,” in *Proc. ECCV*, 2020. doi
31. H. W. Kuhn, “The Hungarian method for the assignment problem,” *Naval Research Logistics Quarterly*, vol. 2, 1955. doi
32. X. Zhu, W. Su, L. Lu, B. Li, X. Wang and J. Dai, “Deformable DETR: Deformable transformers for end-to-end object detection,” in *Proc. ICLR*, 2021. arXiv
33. H. Zhang, F. Li, S. Liu, L. Zhang, H. Su, J. Zhu, L. M. Ni and H.-Y. Shum, “DINO: DETR with improved denoising anchor boxes for end-to-end object detection,” arXiv:2203.03605, 2022. arXiv
34. Y. Zhao, W. Lv, S. Xu, J. Wei, G. Wang et al., “DETRs beat YOLOs on real-time object detection,”

in *Proc. IEEE/CVF CVPR*, 2024. doi
35. M. Minderer, A. Gritsenko, A. Stone, M. Neumann, D. Weissenborn et al., "Simple open-vocabulary object detection with vision transformers," in *Proc. ECCV*, 2022. arXiv
36. S. Liu, Z. Zeng, T. Ren, F. Li, H. Zhang et al., "Grounding DINO: Marrying DINO with grounded pre-training for open-set object detection," in *Proc. ECCV*, 2024. doi
37. T. Cheng, L. Song, Y. Ge, W. Liu, X. Wang et al., "YOLO-World: Real-time open-vocabulary object detection," in *Proc. IEEE/CVF CVPR*, 2024. doi
38. N. Carion, L. Gustafson, Y.-T. Hu, S. Debnath, R. Hu et al., "SAM 3: Segment anything with concepts," arXiv:2511.16719, 2025. arXiv
39. G. Cheng, X. Yuan, X. Yao, K. Yan, Q. Zeng et al., "Towards large-scale small object detection: Survey and benchmarks," *IEEE TPAMI*, vol. 45, no. 11, 2023. arXiv
40. COCO Consortium, "COCO detection evaluation," official evaluation description (12 metrics, 101-point interpolation). link ·official evaluator
41. M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn and A. Zisserman, "The PASCAL Visual Object Classes challenge: A retrospective," *International Journal of Computer Vision*, vol. 111, 2015. doi
42. S. Shao, Z. Li, T. Zhang, C. Peng et al., "Objects365: A large-scale, high-quality dataset for object detection," in *Proc. IEEE/CVF ICCV*, 2019. doi
43. A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin et al., "The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale," *International Journal of Computer Vision*, 2020. arXiv
44. A. Gupta, P. Dollár and R. Girshick, "LVIS: A dataset for large vocabulary instance segmentation," in *Proc. IEEE/CVF CVPR*, 2019. doi
45. T. Shehzadi, K. A. Hashmi, M. Liwicki, D. Stricker and M. Z. Afzal, "Object detection with transformers: A review," *Sensors*, vol. 25, no. 19, 2025. doi
46. C. Zhu and L. Chen, "A survey on open-vocabulary detection and segmentation: Past, present, and future," *IEEE TPAMI*, 2024. doi

# DLCV 03A • One-Stage Detection, Read Through YOLOv4

A detailed introduction to YOLOv4 as the representative one-stage detector: its anatomy, every training and inference component it tests, its experiments, its limits, and paper-by-paper introductions to the YOLO versions, one-stage detectors and real-time DETRs before and after it.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/

## What you'll learn

- Trace YOLOv4 from input through backbone, neck, head and box decoding.
- Separate training-only changes from changes that affect inference cost.
- Read ablations for isolated effects, interactions and untested components.
- Compare latency under stated hardware, precision, input size and timing boundaries.
- Use the paper guides to trace later YOLO and real-time DETR design changes.

## The big picture

YOLOv4 is a paper about a recipe. Its authors combined established ideas with modifications of their own, collected published tricks for training and building detectors, tested which of them help a fast detector on one ordinary GPU, and assembled the winners into one model. That makes it a useful worked example: backbone, feature fusion, decoding, assignment, augmentation and post-processing are visible in one system. Later detectors change several of these decisions, giving us concrete alternatives to compare.

> **Explain like I'm five**
>
> Imagine a cook who wants the best fast lunch that any home kitchen can make. Cookbooks list hundreds of tips. Some tips change only how you shop and prepare (they cost nothing at the table); others add a small step while cooking (they cost a few seconds per plate). The cook tries the tips one by one, keeps the ones that make the lunch tastier without making it slower, and writes down the final recipe. YOLOv4 is that recipe book for a detector: the "free" tips change only training, the "special" tips add a little work at test time.

Why it matters: one-stage detectors, including the YOLO family, are practical options for cameras, robots and industrial inspection. YOLOv4 sits in the middle of that line. Read closely, it explains the parts that its predecessors introduced (grids, anchors, multi-scale heads) and the parts that its successors replaced (anchors, NMS, the coupled head). This chapter therefore introduces YOLOv4 in detail in the main text and gives every predecessor and successor its own introduction in the Paper guides.

This chapter assumes DLCV 03. Its Core building blocks define box regression targets, IoU and its variants, anchors and label assignment, non-maximum suppression (NMS), feature pyramids and the focal loss; its sections on one-stage detectors, anchor-free detectors and DETR introduce the families we build on. We use that material and link to it rather than repeat it.

### The family at a glance

**Plain version.** YOLO names a family developed by several teams. Identify the specific version, paper and implementation before comparing results.

### A timeline

Figure 1 places the versions by the year they first appeared in public, grouped by who released them, and marks whether each has a peer-reviewed paper. Three facts are easy to miss. First, **version numbers do not form one lineage**: after YOLOv3, its original authors stopped, and later numbers were taken by different groups. YOLOv4 came from a new team, two of whose authors later wrote YOLOv7 and YOLOv9 [1], [2]. Second, several widely used versions (YOLOv5, YOLOv8, YOLO11) were released as **software without a paper**; their documentation says so [3], [4], [5]. Third, the name is unregulated: in August 2026 a preprint titled "YOLOv14" was posted to arXiv and withdrawn by its authors about two weeks later [6], so a new number by itself says nothing about quality.

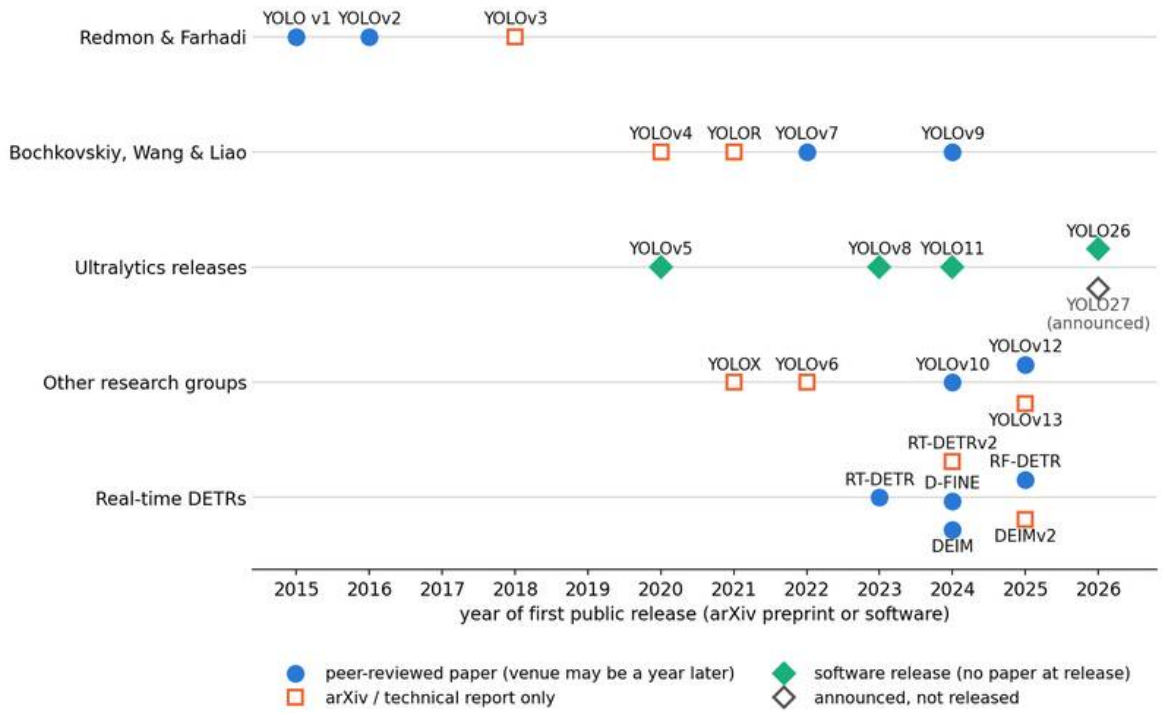



*Figure 1 — The YOLO family and real-time DETRs by year of first public release. Filled circles have a peer-reviewed paper (often published a year after the preprint), open squares are arXiv preprints or technical reports only, filled diamonds are software releases without a paper at release time, and the open diamond is announced but not released.*

### The papers introduced in this chapter

The table lists every paper that has its own introduction in the Paper guides, in three groups: the one-stage detectors YOLOv4 builds on, the one-stage detectors that came after it, and the real-time DETRs, which are not one-stage detectors but now compete with them. The Ultralytics releases without a paper are summarized in the Supplementary notes.

| Paper | Year, venue | The idea in one line | Related part of YOLOv4 |
|---|---|---|---|
| *Predecessors* | | | |
| YOLO v1 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolo-v1-detection-as-a-single-regression/) [7] | 2016, CVPR | One network regresses all boxes from an $S \times S$ grid | Anatomy |
| YOLOv2 / YOLO9000 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov2-and-yolo9000-anchors-priors-and-joint-training/) [8] | 2017, CVPR | k-means anchors, bounded centre offsets, multi-scale training | Head |
| YOLOv3 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov3-three-scales-and-darknet-53/) [9] | 2018, tech report | Three output scales, Darknet-53, logistic class scores | Head |
| SSD (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/ssd-multi-scale-default-boxes/) [10] | 2016, ECCV | Default boxes on several feature maps | Anatomy |
| RetinaNet (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/retinanet-and-the-focal-loss/) [11] | 2017, ICCV | Focal loss for the foreground–background imbalance | Box loss |
| FCOS (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/fcos-per-pixel-anchor-free-detection/) [12] | 2019, ICCV | Anchor-free per-pixel boxes with centre-ness | Label assignment |

| Paper | Year, venue | The idea in one line | Related part of YOLOv4 |
|---|---|---|---|
| CenterNet (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/centernet-objects-as-points/) [13] | 2019, arXiv | Objects as heatmap peaks, no IoU-based NMS | Post-processing |
| *Successors* | | | |
| YOLOR (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolor-explicit-features-and-learned-implicit-representations/) [14] | 2021, arXiv | Learned input-independent tensors align features and refine predictions | Backbone |
| YOLOX (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolox-anchor-free-yolo-with-simota/) [15] | 2021, arXiv | Anchor-free decoupled head, SimOTA assignment | Head |
| YOLOv6 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov6-a-re-parameterized-industrial-detector/) [16] | 2022, tech report | Re-parameterizable backbone, task-aligned assignment, quantization | Backbone |
| YOLOv7 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov7-a-trainable-bag-of-freebies/) [1] | 2023, CVPR | E-ELAN, planned re-parameterization, lead-guided labels | Complete recipe |
| YOLOv9 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov9-programmable-gradient-information/) [2] | 2024, ECCV | Auxiliary reversible branch for reliable gradients; GELAN | Backbone |
| YOLOv10 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov10-nms-free-training-with-dual-assignments/) [17] | 2024, NeurIPS | Dual one-to-many / one-to-one heads, no NMS | Post-processing |

| Paper | Year, venue | The idea in one line | Related part of YOLOv4 |
|---|---|---|---|
| YOLOv12 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov12-an-attention-centric-yolo/) [18] | 2025, NeurIPS | Area attention at real-time cost | Attention |
| YOLOv13 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov13-hypergraph-correlation/) [19] | 2025, arXiv | Hypergraph-based feature correlation | Neck |
| YOLO26 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolo26-a-dfl-free-end-to-end-yolo/) [20] | 2026, arXiv | NMS-free by default, no DFL, new optimizer and small-object assignment | Label assignment |
| *Real-time DETRs* | | | |
| RT-DETR (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rt-detr-a-real-time-detr/) [21] | 2024, CVPR | Hybrid encoder, uncertainty-minimal query selection | Post-processing |
| RT-DETRv2 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rt-detrv2-a-bag-of-freebies-for-rt-detr/) [22] | 2024, tech report | Per-scale sampling points, discrete sampling for deployment | Complete recipe |
| D-FINE (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/d-fine-boxes-as-refined-distributions/) [23] | 2025, ICLR | Boxes as distributions refined layer by layer | Box loss |
| DEIM (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/deim-dense-one-to-one-matching/) [24] | 2025, CVPR | More positives through Mosaic-style dense one-to-one matching | Mosaic |

| Paper | Year, venue | The idea in one line | Related part of YOLOv4 |
|---|---|---|---|
| RF-DETR (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rf-detr-architecture-search-on-a-dinov2-backbone/) [25] | 2026, ICLR | Weight-sharing architecture search on a DINOv2 backbone | Backbone |
| DEIMv2 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/deimv2-dinov3-features-for-real-time-detrs/) [26] | 2025, arXiv | DINOv3 features adapted to multi-scale detection | Backbone |

**A manageable first reading**

Start with **Challenge**, the detector anatomy, head decoding and **Results**. Then return to assignment, losses and training components one at a time; each section starts with its purpose before the equations. Use the ablations to check which effects were actually measured. The paper guides are separate optional pages: choose one predecessor and one successor after you can explain YOLOv4's full input-to-box path. You do not need to read every model to complete the exercises.

## Challenge

**Plain version.** YOLOv4 tests published techniques to combine accurate detection, real-time inference and training on a single conventional GPU.

### A fast detector that one ordinary GPU can train

The paper (Bochkovskiy, Wang and Liao, arXiv, April 2020) [27] starts from a practical complaint: the most accurate networks of the time did not run in real time and needed many GPUs and large mini-batches to train. Its stated goal is a detector that is fast "in production systems", meaning fast in measured frames per second on a GPU rather than small in a theoretical operation count, and that anyone can train and test on a single 1080 Ti or 2080 Ti card [27].

Two predecessors frame the problem. YOLOv3 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov3-three-scales-and-darknet-53/) [9] was fast but, by 2020, clearly less accurate than the best detectors on the strict COCO metric: at 608 × 608 it reached 33.0% AP in the YOLOv4 comparison table [27]. EfficientDet [28] was accurate and efficient in floating-point operations, but the paper's measurements show that low operation counts do not translate into high GPU frame rates (see Backbone). YOLOv4 wants the accuracy of the second at the speed of the first.

### What "real-time" means and how YOLOv4 times it

A detector is real-time when the time it needs per frame is shorter than the time between frames. **Latency** is the time from receiving one frame to having its final boxes; **throughput** is how many frames are finished per second when many are processed together. For one frame,

$$T_{\text{e2e}} = T_{\text{pre}} + T_{\text{net}} + T_{\text{post}},$$

where $T_{\text{pre}}$ is pre-processing (decoding, resizing and normalizing the image, copying it to the accelerator), $T_{\text{net}}$ is the forward pass of the network, and $T_{\text{post}}$ is post-processing (decoding boxes, the confidence threshold, NMS). A camera at $f$ frames per second leaves $1/f$ seconds per frame: about 33.3 ms at 30 FPS. With batches of $B$ frames, throughput is $B/T_{\text{batch}}(B)$, where $T_{\text{batch}}(B)$ is the time for one batch; it grows with $B$, but every frame then waits for its whole batch, so a live camera cares about latency at **batch size 1**. Converting latency to FPS as $1000/T$ (with $T$ in milliseconds) is only meaningful at batch 1.

YOLOv4's speed tables follow this convention. Their captions state that the results are compared "with batch=1 without using tensorRT", that is, without NVIDIA's optimizing inference engine [29], and the tables are split by GPU architecture (Maxwell, Pascal and Volta) because different papers timed their models on different cards [27]. They mark "real-time" as 30 FPS or more. The paper does not state the numeric precision or whether NMS is inside the timed interval; the YOLOX report later timed its comparisons "with FP16-precision and batch=1 on a V100 to align the settings of YOLOv4" [15]. Keep these conditions in mind when you read the numbers in Results.

### Why the paper tests "freebies" and "specials"

Many published "features" claim to improve CNNs, but often only for one model, one task or small datasets. The paper therefore sorts candidate improvements into two bags and tests them [27]:

- **Bag of freebies (BoF):** methods that change only the training strategy or only increase the training cost, so inference stays exactly as fast. Data augmentation, regularization, label smoothing, box-regression losses and assignment rules belong here.
- **Bag of specials (BoS):** plug-in modules and post-processing steps that add a small inference cost but can improve accuracy noticeably. Receptive-field blocks, attention modules, feature-fusion paths, activation functions and NMS variants belong here.

The challenge, then, is not to invent one module but to find the combination that is best for a detector that must stay fast and must train on one GPU. The rest of this introduction follows the paper's own order: the architecture it selects, the freebies and specials it adds, the experiments that justify the choices, and what remains open.

### Method

**Plain version.** YOLOv4 retains the YOLOv3 head, adds CSPDarknet53 and an SPP/PANet neck, and combines training and inference refinements.

### The detector anatomy: input, backbone, neck and head

**Plain version.** Almost every detector has three parts. A **backbone** turns the image into feature maps at several resolutions. A **neck** mixes information between those resolutions. A **head** looks at every cell of every map and predicts boxes and classes.

The paper's Figure 2 and Section 2.1 give this vocabulary, which later papers adopted [27]:

- **Input:** an image, image patches or an image pyramid. YOLOv4 uses a single image resized to a square network resolution, typically 416, 512 or 608 pixels per side.
- **Backbone:** a network pre-trained on ImageNet [30] classification. GPU detectors use backbones such as VGG [31] or ResNet [32] variants; CPU detectors use mobile backbones.
- **Neck:** layers between backbone and head that collect feature maps from different stages, usually with several top-down and bottom-up paths, such as FPN [33] and PAN [34]; "additional blocks" such as SPP [35] enlarge the receptive field.
- **Head:** the predictor. A **dense prediction** head (one-stage detector) predicts at every location of the feature maps, with anchors (YOLO, SSD, RetinaNet) or without (CenterNet, FCOS). A **sparse prediction** head (two-stage detector, the R-CNN family) predicts only for a set of proposals.

YOLOv4's choice, in the paper's words reduced to a list, is: backbone CSPDarknet53 [36], neck SPP [35] and PAN [34], head YOLOv3 [9]. The head is **anchor-based** and **one-stage**: at three output strides it predicts, for each cell and each of three anchors, four box numbers, one objectness score and one score per class.

For a square input of side $L$ pixels and output strides 8, 16 and 32, the three output grids have $L/8$, $L/16$ and $L/32$ cells per side, and the number of predictions per image is

$$N_{\text{pred}} = A\left[\left(\tfrac{L}{8}\right)^2 + \left(\tfrac{L}{16}\right)^2 + \left(\tfrac{L}{32}\right)^2\right],$$

where $A = 3$ is the number of anchors per cell. At $L = 608$ the grids are $76^2 + 38^2 + 19^2 = 7{,}581$ cells, so $N_{\text{pred}} = 22{,}743$; at $L = 416$ they give 10,647 predictions. Each prediction is a vector of $4+1+K$ numbers for $K$ classes; with COCO's 80 classes each output layer has $3\times 85 = 255$ channels, the value in the released configuration [37]. All of these predictions must be filtered down to a few boxes by a confidence threshold and NMS, which is why post-processing cost matters for real-time use.

### Backbone: CSPDarknet53

**Plain version.** The backbone must see far enough to cover large objects and their context, have enough capacity for many objects at once, and run fast on a GPU. The paper compared three candidates and picked CSPDarknet53: YOLOv3's Darknet-53 with "cross-stage partial" connections that send only half of each stage's channels through the expensive blocks.

**Why this backbone.** Section 3.1 argues that a detector, unlike a classifier, needs a higher input resolution (for small objects), more layers (so the receptive field covers the larger input) and more parameters (to detect many objects of different sizes in one image). Table 1 of the paper compares three backbones at 512 × 512, with frame rates measured on an RTX 2070 [27]:

| Backbone | Receptive field | Parameters | BFLOPs | FPS (RTX 2070) |
|---|---|---|---|---|
| CSPResNeXt50 | 425 × 425 | 20.6 M | 31 | 62 |
| CSPDarknet53 | 725 × 725 | 27.6 M | 52 | 66 |
| EfficientNet-B3 [38] | 1311 × 1311 | 12.0 M | 11 | 26 |

Two lessons sit in this small table. First, operation counts do not predict GPU speed: EfficientNet-B3 needs about a fifth of CSPDarknet53's operations but runs at less than half its frame rate. Second, the paper reports that CSPResNeXt50 is the better ImageNet classifier while CSPDarknet53 is the better COCO detector backbone, so "best classifier" and "best detector backbone" differ; the ablations in Backbones and pre-trained weights test this directly. CSPDarknet53 has 29 convolutional layers of size 3 × 3 against 16 for CSPResNeXt50, hence its larger receptive field.

**What cross-stage partial connections do.** CSPNet [36] illustrates its design using DenseNet-style stages [39] and also applies it to other backbones, including ResNeXt [40]. In the DenseNet example, in which each layer receives the concatenation of all earlier outputs. The authors attribute part of the computational redundancy to duplicated gradient information. This is a design explanation, not a proof that all layers learn identical features. CSPNet splits the work. Write the base feature map of a stage as two channel groups,

$$x_0 = [\, x'_0,\ x''_0 \,],$$

where $[\cdot,\cdot]$ is concatenation along channels. Only $x''_0$ goes through the stage's $k$ blocks, producing $x_1, \dots, x_k$; their output passes through a transition layer (a 1 × 1 convolution) to give $x_T$; then $x_T$ is concatenated with the untouched part $x'_0$ and passed through a second transition layer:

$$x_T = w_T * [\, x''_0, x_1, \dots, x_k \,], \qquad x_U = w_U * [\, x'_0, x_T \,],$$

where $*$ is convolution and $w_T, w_U$ are the transition weights. The CSPNet paper gives three purposes: splitting the paths can reduce duplicated gradient information, processing fewer channels can reduce computation, and the design can reduce memory traffic [36]. It also compares fusion orders and finds that the "fusion last" order, in which the block output is transformed before it is merged with $x'_0$, truncates the duplicated gradient flow and costs little accuracy [36]. Applied to Darknet-53, each residual stage becomes such a split–process–merge stage.

**Activation.** The backbone uses Mish (see Activation: Mish); in the released configuration, Mish appears in the backbone and the leaky rectifier in the neck and head [37].

### Neck, part 1: the SPP block

**Plain version.** Objects can be much larger than what one cell "sees". The SPP block lets each cell of the coarsest feature map look at a large neighbourhood cheaply, by max-pooling over windows of several sizes and stacking the results as extra channels.

Spatial pyramid pooling was introduced to turn feature maps of any size into a fixed-length vector for classification [35]. That version cannot be used inside a fully convolutional detector. YOLOv3's SPP variant, which YOLOv4 adopts, keeps the spatial size: it applies max pooling with kernel sizes $k \in \{1, 5, 9, 13\}$ and stride 1, with padding so that the output has the same height and width, and concatenates the four results along channels [27]. For an input of shape $C \times H \times W$,

$$\mathrm{SPP}(F) = [\, F,\ \mathrm{MaxPool}_5(F),\ \mathrm{MaxPool}_9(F),\ \mathrm{MaxPool}_{13}(F) \,] \in \mathbb{R}^{4C \times H \times W},$$

where $\mathrm{MaxPool}_k$ is $k \times k$ max pooling with stride 1 and "same" padding, and $k = 1$ is the identity. The block is placed on the stride-32 map, where $H = W = 19$ for a 608 input, so a $13 \times 13$ window covers most of the map. The paper quotes YOLOv3-608 with SPP gaining 2.7% AP50 on COCO at 0.5% extra computation, and states that SPP "significantly increases the receptive field" while causing almost no reduction in speed [27]. The released configuration contains exactly the three max-pooling layers of sizes 5, 9 and 13 [37].

```
import numpy as np
from scipy.ndimage import maximum_filter

def spp(x, kernels=(5, 9, 13)):
    """YOLO-style SPP. x: (C, H, W). Stride-1 max pooling with 'same' padding keeps H and W;
    the input and the pooled copies are concatenated along the channel axis."""
    pooled = [maximum_filter(x, size=(1, k, k), mode="constant", cval=-np.inf) for k in
     ↪  kernels]
    return np.concatenate([x] + pooled, axis=0)

rng = np.random.default_rng(0)
x = rng.normal(size=(512, 19, 19))          # e.g. the stride-32 map of a 608 x 608 input
print("SPP:", x.shape, "->", spp(x).shape)

one = np.zeros((1, 19, 19)); one[0, 9, 9] = 1.0   # a single strong activation
y = spp(one)
print("cells reached by one activation per group:", [int((g > 0).sum()) for g in y])
```

```
SPP: (512, 19, 19) -> (2048, 19, 19)
cells reached by one activation per group: [1, 25, 81, 169]
```

The channel count quadruples while the grid stays $19 \times 19$, and one strong activation now influences up to $13 \times 13 = 169$ cells of the next layer. A $1 \times 1$ convolution after the block reduces the channels again.

### Neck, part 2: PANet path aggregation

**Plain version.** Deep feature maps know *what* is in the image; shallow ones know *where* edges are. A feature pyramid sends meaning downward to the fine maps; PANet adds a second path that sends precise location information back up to the coarse maps. YOLOv4 uses this two-way neck instead of YOLOv3's one-way one, and merges the paths by concatenation instead of addition.

FPN [33] (explained in DLCV 03) builds levels $P_3, P_4, P_5$ by upsampling coarse features and adding them to finer ones (a top-down path). PANet [34] observed that the path from low-level features to the top of the pyramid is still very long, and added a **bottom-up path augmentation**. Starting from $N_2 = P_2$ in the original paper, each new level is

$$N_{i+1} = \mathrm{conv}_{3\times3}\Big(\mathrm{conv}_{3\times3}^{\text{stride }2}(N_i) + P_{i+1}\Big),$$

where $\mathrm{conv}_{3\times3}^{\text{stride }2}$ halves the resolution of $N_i$, $+$ is element-wise addition with the FPN level $P_{i+1}$ of the same resolution, and the outer $3\times3$ convolution produces the new map [34]. In YOLOv4 the pyramid runs over strides 8, 16 and 32, and the three outputs $N_3, N_4, N_5$ feed the three heads. Section 3.1 says the authors chose PANet "instead of the FPN used in YOLOv3" [27].

**YOLOv4's modification.** Figure 6 of the paper replaces the shortcut (addition) in PAN with **concatenation**: $[\,\mathrm{down}(N_i),\ P_{i+1}\,]$ along channels, followed by convolutions [27]. Concatenation keeps both inputs separate and lets the next convolution learn how to weight them, at the price of more channels. The paper lists this among the changes made "to make the designed detector more suitable for training on single GPU", but it does not report a separate ablation of addition versus concatenation.

### Head: the YOLOv3 head and box decoding

**Plain version.** At each cell, for each of three template boxes (anchors), the head outputs four numbers that move and stretch the template, one "is there an object?" score and one score per class. YOLOv4 keeps this head and fixes one numerical problem: in YOLOv3, a box centre close to a cell border was very hard to predict.

**Decoding.** The head follows YOLOv2 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov2-and-yolo9000-anchors-priors-and-joint-training/) [8] and YOLOv3 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov3-three-scales-and-darknet-53/) [9]. For a cell whose top-left corner is at $(c_x, c_y)$ in grid units and an anchor of width $p_w$ and height $p_h$, the network outputs $(t_x, t_y, t_w, t_h)$ and the box is

$$b_x = \sigma(t_x) + c_x, \qquad b_y = \sigma(t_y) + c_y, \qquad b_w = p_w\, e^{t_w}, \qquad b_h = p_h\, e^{t_h},$$

where $\sigma(t) = 1/(1+e^{-t})$ is the logistic sigmoid, $(b_x, b_y)$ is the box centre in grid units (multiply by the stride for pixels) and $(b_w, b_h)$ its size. An objectness output $t_o$ gives $\sigma(t_o)$, and each class has its own sigmoid score, so a box can carry several labels; YOLOv3 trains the class scores with binary cross-entropy [9]. Each output stride has three anchors; the released YOLOv4 configuration uses the nine anchors (in pixels) $12\times16$, $19\times36$, $40\times28$ at stride 8, $36\times75$, $76\times55$, $72\times146$ at stride 16 and $142\times110$, $192\times243$, $459\times401$ at stride 32 [37].

**The grid-sensitivity problem.** Because $\sigma(t_x) \in (0,1)$ never reaches 0 or 1, a centre that lies exactly on a cell border needs $t_x \to \pm\infty$, and centres near borders need very large $|t_x|$. Section 4.3 of the paper calls this "grid sensitivity": the network must output extreme values to place a centre near $c_x$ or $c_x + 1$ [27]. The fix is to multiply the sigmoid by a factor greater than 1. The paper states only that; the released code writes it as

$$b_x = s\,\sigma(t_x) - \frac{s-1}{2} + c_x, \qquad b_y = s\,\sigma(t_y) - \frac{s-1}{2} + c_y,$$

where $s \geq 1$ is the scale factor (`scale_x_y` in the configuration) and the shift $-(s-1)/2$ keeps the range centred on the cell, so the centre can now cover $(c_x - (s-1)/2,\ c_x + 1 + (s-1)/2)$, slightly beyond the cell [37]. The released configuration uses $s = 1.2$, 1.1 and 1.05 at strides 8, 16 and 32 [37]. The snippet decodes one prediction and shows how much less extreme $t_x$ has to be.

```
import numpy as np

sigmoid = lambda t: 1 / (1 + np.exp(-t))
logit = lambda p: np.log(p / (1 - p))

def decode(t, cell, anchor, stride, s=1.0):
    """One YOLOv3/YOLOv4 prediction t = (tx, ty, tw, th) -> box centre and size in pixels.
    s = 1 is the YOLOv3 formula; s > 1 is YOLOv4's grid-sensitivity fix."""
    tx, ty, tw, th = t
    bx = s * sigmoid(tx) - (s - 1) / 2 + cell[0]        # centre in grid units
    by = s * sigmoid(ty) - (s - 1) / 2 + cell[1]
    return bx * stride, by * stride, anchor[0] * np.exp(tw), anchor[1] * np.exp(th)

# cell (10, 7) of the stride-8 grid, anchor 36 x 75 pixels, raw outputs all zero
print("box (cx, cy, w, h):", np.round(decode((0, 0, 0, 0), (10, 7), (36, 75), 8), 1))

# which raw output tx puts the centre at a given position inside its cell?
for pos in (0.5, 0.9, 0.99, 1.0):
    row = []
    for s in (1.0, 1.2):
        p = (pos + (s - 1) / 2) / s                     # the sigmoid value that is needed
        row.append(f"s={s}: " + (f"{logit(p):5.2f}" if p < 1 else "  inf"))
    print(f"centre at {pos:4.2f} of the cell -> tx = " + " | ".join(row))
```

```
box (cx, cy, w, h): [84. 60. 36. 75.]
centre at 0.50 of the cell -> tx = s=1.0:  0.00 | s=1.2:  0.00
centre at 0.90 of the cell -> tx = s=1.0:  2.20 | s=1.2:  1.61
centre at 0.99 of the cell -> tx = s=1.0:  4.60 | s=1.2:  2.29
centre at 1.00 of the cell -> tx = s=1.0:   inf | s=1.2:  2.40
```

With $s = 1.2$ a centre on the border needs $t_x = 2.40$ instead of infinity, so the targets the network must learn stay in a comfortable range. In the ablation table this is the column "S" (see Detector bag of freebies).

**Training targets.** The head is trained with three terms: a box loss (CIoU in YOLOv4, see Box loss), a binary cross-entropy objectness loss, and binary cross-entropy class losses on the positive

predictions. In YOLOv3, a prediction whose box already overlaps a ground-truth box well but is not assigned to it is ignored in the objectness loss rather than pushed to "background"; YOLOv3 used an IoU of 0.5 for this rule [9], and the released YOLOv4 configuration sets it to 0.7 (`ignore_thresh`) [37]. Which predictions count as positives is the subject of the next section.

### Label assignment: several anchors per object

**Plain version.** During training, every object must be "given" to some predictions, which are then taught to fire on it. YOLOv3 gave each object to exactly one anchor, the best-shaped one. YOLOv4 keeps this best match and can add anchors whose shapes are similar enough, so an object can train several predictions.

In YOLOv3, an object is assigned to the single anchor whose shape has the highest IoU with the object's box, at the cell containing the object's centre [9]. YOLOv4's bag of freebies includes "using multiple anchors for a single ground truth". The released code assigns the best anchor first, then adds anchors with

$$\mathrm{IoU}(\text{truth}, \text{anchor}) > \tau$$

as positives, where the IoU compares only widths and heights (both boxes placed on the same centre) and $\tau$ is the "IoU threshold" hyper-parameter [27]. The genetic search (below) selected $\tau = 0.213$, which is also the value in the released configuration (`iou_thresh`) [37]. Because the nine anchors are split across three strides, one object can now be learned on two output levels. The best-anchor fallback still provides a match when no anchor passes the threshold. The snippet demonstrates shape matching; a full implementation must also handle targets competing for the same prediction.

```
import numpy as np

# the nine anchors (width, height in pixels) of yolov4.cfg; three per output stride
anchors = np.array([[12, 16], [19, 36], [40, 28], [36, 75], [76, 55], [72, 146],
                    [142, 110], [192, 243], [459, 401]], float)
stride = np.repeat([8, 16, 32], 3)

def shape_iou(wh, anchors):
    """IoU of boxes that share a centre: only width and height matter."""
    inter = np.minimum(wh[0], anchors[:, 0]) * np.minimum(wh[1], anchors[:, 1])
    return inter / (wh[0] * wh[1] + anchors.prod(1) - inter)

for wh in [(30, 60), (100, 90), (60, 120)]:
    iou = shape_iou(np.array(wh, float), anchors)
    pos = np.union1d([iou.argmax()], np.flatnonzero(iou > 0.213))
    print(f"object {wh[0]:3d}x{wh[1]:<3d} | best anchor {iou.argmax()} (IoU {iou.max():.2f})
    ↪  | "
          f"IoU > 0.213: anchors {pos.tolist()} on strides
           ↪  {sorted(set(stride[pos].tolist()))}")
```

```
object  30x60  | best anchor 3 (IoU 0.67) | IoU > 0.213: anchors [1, 2, 3, 4] on strides [8,
 ↪  16]
```

```
object 100x90  | best anchor 6 (IoU 0.58) | IoU > 0.213: anchors [3, 4, 5, 6] on strides [16,
 ↪  32]
object  60x120 | best anchor 5 (IoU 0.68) | IoU > 0.213: anchors [3, 4, 5, 6] on strides [16,
 ↪  32]
```

Each object now trains four predictions instead of one. More positives give more training signal, especially for objects whose shape falls between two anchors, but they also teach several predictions to fire on the same object, which makes NMS indispensable. In the ablation table this is the column "IT". Later detectors replaced fixed shape rules with assignment based on the network's own predictions (YOLOX (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolox-anchor-free-yolo-with-simota/), YOLOv6 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov6-a-re-parameterized-industrial-detector/)) and eventually with one-to-one assignment that removes NMS (YOLOv10 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov10-nms-free-training-with-dual-assignments/)). DLCV 03 explains the IoU-based rules of two-stage detectors and of RetinaNet for comparison.

### Activation: Mish

**Plain version.** An activation function decides how a neuron responds to its input. YOLOv3 used the leaky rectifier, which has a sharp corner at zero. Mish is a smooth curve that behaves like the identity for large inputs and lets small negative values through, which the paper found to help both the classifier and the detector.

Mish is defined as [41]

$$\mathrm{Mish}(x) = x \tanh(\mathrm{softplus}(x)) = x \tanh(\ln(1 + e^x)),$$

where $\mathrm{softplus}(x) = \ln(1 + e^x)$. For large positive $x$, $\tanh(\mathrm{softplus}(x)) \to 1$ and $\mathrm{Mish}(x) \approx x$; for large negative $x$, it tends to 0; in between it dips slightly below zero, so it is not monotonic. The leaky rectifier of YOLO v1 to v3 is $\phi(x) = x$ for $x > 0$ and $0.1x$ otherwise [7]. The paper's activation candidates were ReLU, leaky ReLU, parametric ReLU, ReLU6, SELU, Swish [42] and Mish; it removed parametric ReLU and SELU as hard to train and ReLU6 as designed for quantized networks, and notes that Swish and Mish are both continuously differentiable [27].

```
import numpy as np

mish = lambda x: x * np.tanh(np.log1p(np.exp(x)))   # x * tanh(softplus(x))
leaky = lambda x: np.where(x > 0, x, 0.1 * x)         # the leaky rectifier of YOLO v1-v3

for v in (-3.0, -1.0, 0.0, 1.0, 3.0):
    print(f"x = {v:4.1f} | Mish {mish(v):7.4f} | leaky {float(leaky(v)):5.2f}")
x = np.linspace(-6, 6, 120001)
i = mish(x).argmin()
print(f"Mish minimum {mish(x[i]):.4f} at x = {x[i]:.3f}")
h = 1e-6
print(f"slope at 0: Mish {(mish(h) - mish(-h)) / (2 * h):.4f}; leaky jumps from 0.1 to 1")
```

```
x = -3.0 | Mish -0.1456 | leaky -0.30
x = -1.0 | Mish -0.3034 | leaky -0.10
x =  0.0 | Mish  0.0000 | leaky  0.00
x =  1.0 | Mish  0.8651 | leaky  1.00
x =  3.0 | Mish  2.9865 | leaky  3.00
Mish minimum -0.3088 at x = -1.192
slope at 0: Mish 0.6000; leaky jumps from 0.1 to 1
```

Mish is smooth everywhere (its slope at 0 is 0.6, while the leaky rectifier's slope jumps from 0.1 to 1), its smallest value is about $-0.31$, and for inputs above about 3 it is almost the identity. In the paper's classifier experiments Mish raised CSPResNeXt-50's top-1 accuracy from 77.9% to 78.9% on its own, and the detector built on the Mish-trained CSPDarknet53 was the best one (see Ablations) [27]. The price is a slightly more expensive function than a rectifier.

### Box loss: from IoU to CIoU

**Plain version.** The box loss tells the network how wrong a predicted box is. Measuring errors in the four coordinates separately ignores that they describe one box. IoU-based losses measure the overlap instead, and the CIoU loss adds two more hints: how far apart the centres are and whether the shape is right.

The paper's Section 2.2 retraces the history [27]. Squared error on unnormalized box coordinates depends on their units and does not directly measure overlap. **UnitBox** uses $-\log(\text{IoU})$ for overlapping boxes [43]; the commonly used overlap loss $1-\text{IoU}$ remains finite at zero overlap. The latter considers the box as a whole and is invariant to a common rescaling, but is constant with zero gradient while boxes remain disjoint. **GIoU** [44] adds a penalty based on the smallest box enclosing both. **DIoU** and **CIoU** [45] add a centre-distance term and, for CIoU, an aspect-ratio term. The CIoU loss for a predicted box $B$ and ground truth $B^{gt}$ is

$$\mathcal{L}_{\text{CIoU}} = 1 - \text{IoU} + \frac{\rho^2(\mathbf{b}, \mathbf{b}^{gt})}{c^2} + \alpha v, \qquad v = \frac{4}{\pi^2}\Big(\arctan\frac{w^{gt}}{h^{gt}} - \arctan\frac{w}{h}\Big)^2, \qquad \alpha = \frac{v}{(1-\text{IoU}) + v},$$

where $\mathbf{b}$ and $\mathbf{b}^{gt}$ are the two box centres, $\rho$ is the Euclidean distance between them, $c$ is the diagonal length of the smallest box enclosing both, $(w, h)$ and $(w^{gt}, h^{gt})$ are the widths and heights, $v$ measures the aspect-ratio mismatch and $\alpha$ is a weight that gives the aspect term more influence once the boxes overlap well [45]. Dropping the last term gives the DIoU loss, and dropping the last two gives the IoU loss. DLCV 03 computes all four measures for disjoint and overlapping boxes; the snippet below instead splits the CIoU loss into its three parts for one tall ground-truth box.

```
import numpy as np

def iou_terms(a, b):
    """a, b: boxes (x1, y1, x2, y2). Returns IoU, the DIoU penalty rho^2/c^2 and the CIoU
     ↪  term alpha*v."""
```

```
    iw = max(0.0, min(a[2], b[2]) - max(a[0], b[0])); ih = max(0.0, min(a[3], b[3]) -
     ↪  max(a[1], b[1]))
    inter = iw * ih
    wa, ha, wb, hb = a[2] - a[0], a[3] - a[1], b[2] - b[0], b[3] - b[1]
    iou = inter / (wa * ha + wb * hb - inter)
    cw = max(a[2], b[2]) - min(a[0], b[0]); ch = max(a[3], b[3]) - min(a[1], b[1])
    rho2 = ((a[0] + a[2] - b[0] - b[2]) ** 2 + (a[1] + a[3] - b[1] - b[3]) ** 2) / 4
    v = 4 / np.pi ** 2 * (np.arctan(wb / hb) - np.arctan(wa / ha)) ** 2
    alpha = v / ((1 - iou) + v) if v > 0 else 0.0
    return iou, rho2 / (cw ** 2 + ch ** 2), alpha * v

gt = (100, 100, 200, 300)                                  # 100 x 200 pixels, tall
preds = {"centred, lying flat":       (50, 150, 250, 250),
         "shifted, right shape":      (125, 125, 225, 325),
         "far away":                  (300, 100, 400, 300)}
for name, p in preds.items():
    iou, d, av = iou_terms(p, gt)
    print(f"{name:26s} IoU {iou:.3f} | L_IoU {1 - iou:.3f} | L_DIoU {1 - iou + d:.3f} |
     ↪  L_CIoU {1 - iou + d + av:.3f}")
```

```
centred, lying flat        IoU 0.333 | L_IoU 0.667 | L_DIoU 0.667 | L_CIoU 0.700
shifted, right shape       IoU 0.488 | L_IoU 0.512 | L_DIoU 0.530 | L_CIoU 0.530
far away                   IoU 0.000 | L_IoU 1.000 | L_DIoU 1.308 | L_CIoU 1.308
```

The centred but wrongly shaped box gets no DIoU penalty (the centres coincide) but a CIoU penalty for its shape; the shifted box of the right shape gets a distance penalty only; the far-away box has IoU 0, so the IoU loss says nothing about how far it is, while the distance term does. YOLOv4 adopts CIoU, arguing that it gives better convergence speed and accuracy [27]; the released configuration sets `iou_loss=ciou` with a loss weight of 0.07 (`iou_normalizer`, the "loss normalizer" found by the genetic search) [37].

### Post-processing: NMS and DIoU-NMS

**Plain version.** With several positives per object, several predictions fire on each object, and non-maximum suppression keeps the best one and deletes the others that overlap it too much. Its cost depends on the scene. DIoU-NMS changes the deletion test so that two boxes whose centres are far apart are less likely to be treated as duplicates, which helps with neighbouring objects.

**Greedy NMS and its cost.** Greedy NMS (defined in DLCV 03) takes the highest-scoring box, computes its IoU with every remaining box, deletes those above a threshold, and repeats. If $n$ boxes enter and few are deleted, the number of IoU computations approaches

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2},$$

where $k$ counts the boxes still remaining after each kept box. If most boxes are duplicates of a few objects, they are deleted early and the work is much smaller. So NMS time depends on the number

of candidates, on the confidence threshold that decides how many enter, and on the scene, unlike the forward pass, which takes about the same time for every image. The snippet times a NumPy greedy NMS on synthetic candidates; counts of IoU computations are deterministic, while the times depend on your machine.

```
import numpy as np

def nms_count(boxes, scores, thr=0.5):
    """Greedy NMS that also counts how many box-pair IoUs it computes."""
    order = np.argsort(-scores)
    keep, n_iou = [], 0
    while order.size:
        i, rest = order[0], order[1:]
        keep.append(i)
        n_iou += rest.size
        x1 = np.maximum(boxes[i, 0], boxes[rest, 0]); y1 = np.maximum(boxes[i, 1],
         ↪  boxes[rest, 1])
        x2 = np.minimum(boxes[i, 2], boxes[rest, 2]); y2 = np.minimum(boxes[i, 3],
         ↪  boxes[rest, 3])
        inter = np.clip(x2 - x1, 0, None) * np.clip(y2 - y1, 0, None)
        area = lambda b: (b[..., 2] - b[..., 0]) * (b[..., 3] - b[..., 1])
        iou = inter / (area(boxes[i]) + area(boxes[rest]) - inter)
        order = rest[iou <= thr]
    return len(keep), n_iou

def random_boxes(n, n_objects, rng, size=640):
    """n candidate boxes jittered around n_objects true objects."""
    centres = rng.uniform(40, size - 40, (n_objects, 2))
    owner = rng.integers(0, n_objects, n)
    c = centres[owner] + rng.normal(0, 6, (n, 2))
    wh = rng.uniform(30, 60, (n, 2))
    return np.hstack([c - wh / 2, c + wh / 2]), rng.random(n)

rng = np.random.default_rng(0)
for n_objects in (5, 500):                       # few objects vs a crowded scene
    for n in (100, 1000, 8400):
        b, s = random_boxes(n, n_objects, rng)
        kept, n_iou = nms_count(b, s)
        print(f"objects {n_objects:3d} | boxes {n:4d} | kept {kept:4d} | IoUs {n_iou:9,d}")
```

```
objects   5 | boxes  100 | kept   21 | IoUs       770
objects   5 | boxes 1000 | kept   43 | IoUs     7,809
objects   5 | boxes 8400 | kept   86 | IoUs    90,699
objects 500 | boxes  100 | kept   94 | IoUs     4,621
objects 500 | boxes 1000 | kept  513 | IoUs   209,490
objects 500 | boxes 8400 | kept 1259 | IoUs 3,039,292
```

The printed counts are deterministic; Figure 2 separately measures wall-clock time, which varies by machine. With the same 8,400 candidates, the crowded scene needs about 34 times as many IoU computations as the sparse one. Optimized implementations can run on a GPU and cap

the number of boxes, so these counts are a toy analysis rather than a runtime prediction: post-processing time grows with the number of candidates and with crowding. Figure 2 repeats the experiment over a range of candidate counts. This is the cost that the NMS-free successors remove (YOLOv10 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov10-nms-free-training-with-dual-assignments/), RT-DETR (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rt-detr-a-real-time-detr/)); the RT-DETR paper measured it directly for real detectors.

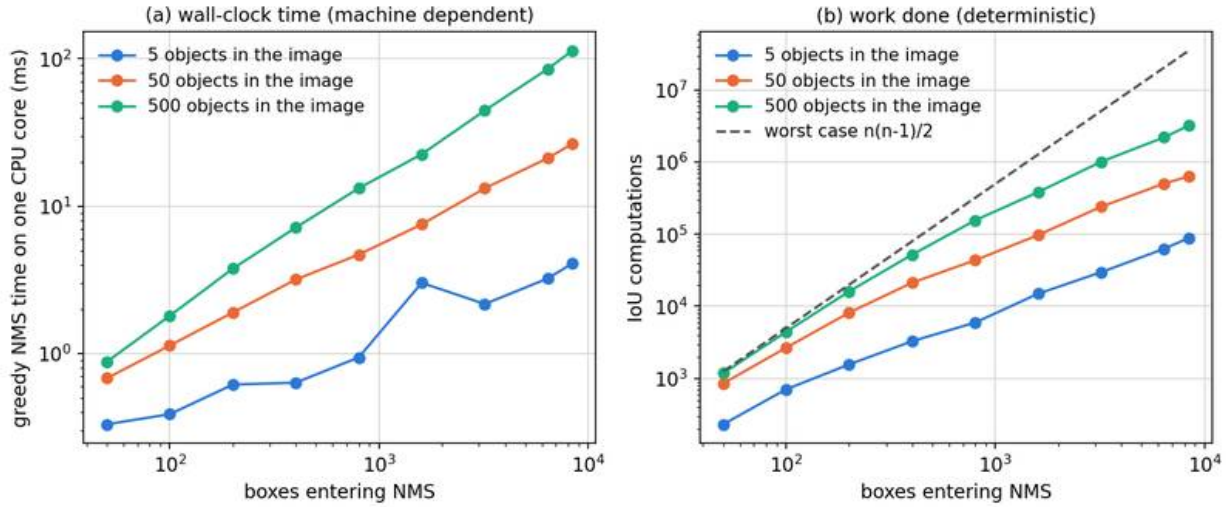



*Figure 2 — Cost of greedy NMS on synthetic candidates (toy run in NumPy). (a) Wall-clock time on one CPU core, which depends on the machine. (b) Number of IoU computations, which is deterministic; the dashed line is the worst case $n(n-1)/2$.*

**DIoU-NMS.** Greedy NMS deletes a box $B_i$ when its IoU with the kept box $M$ exceeds the threshold, so two real people standing close together can be merged into one. The DIoU paper proposed to subtract the centre-distance penalty before the test [45]:

$$s_i \leftarrow \begin{cases} s_i, & \mathrm{IoU}(M, B_i) - \dfrac{\rho^2(\mathbf{b}_M, \mathbf{b}_i)}{c^2} < \varepsilon, \\ 0, & \text{otherwise}, \end{cases}$$

where $s_i$ is the score of $B_i$ (0 means deleted), $\rho$ is the distance between the centres $\mathbf{b}_M$ and $\mathbf{b}_i$, $c$ is the diagonal of the box enclosing both, and $\varepsilon$ is the NMS threshold. Two boxes with distant centres "probably locate different objects" and survive more easily [45]. YOLOv4 lists DIoU-NMS in its bag of specials for the detector [27]. The snippet repeats the box geometry so it can be run on its own.

```
import numpy as np

def iou_terms(a, b):
    """a, b: boxes (x1, y1, x2, y2). Returns IoU, the DIoU penalty rho^2/c^2 and the CIoU
    ↪  term alpha*v."""
```

```
    iw = max(0.0, min(a[2], b[2]) - max(a[0], b[0])); ih = max(0.0, min(a[3], b[3]) -
     ↪  max(a[1], b[1]))
    inter = iw * ih
    wa, ha, wb, hb = a[2] - a[0], a[3] - a[1], b[2] - b[0], b[3] - b[1]
    iou = inter / (wa * ha + wb * hb - inter)
    cw = max(a[2], b[2]) - min(a[0], b[0]); ch = max(a[3], b[3]) - min(a[1], b[1])
    rho2 = ((a[0] + a[2] - b[0] - b[2]) ** 2 + (a[1] + a[3] - b[1] - b[3]) ** 2) / 4
    v = 4 / np.pi ** 2 * (np.arctan(wb / hb) - np.arctan(wa / ha)) ** 2
    alpha = v / ((1 - iou) + v) if v > 0 else 0.0
    return iou, rho2 / (cw ** 2 + ch ** 2), alpha * v


def nms(boxes, scores, thr=0.5, diou=False):
    order, keep = list(np.argsort(-scores)), []
    while order:
        m = order.pop(0); keep.append(int(m))
        crit = lambda j: iou_terms(boxes[m], boxes[j])[0] - (iou_terms(boxes[m], boxes[j])[1]
         ↪  if diou else 0)
        order = [j for j in order if crit(j) < thr]
    return keep

# two people side by side (centres 32 px apart) and one duplicate of the first person
boxes = np.array([[100, 100, 200, 200], [132, 100, 232, 200], [103, 100, 203, 200]], float)
scores = np.array([0.9, 0.8, 0.7])
print("greedy NMS keeps:", nms(boxes, scores), "| DIoU-NMS keeps:", nms(boxes, scores,
 ↪  diou=True))
```

```
greedy NMS keeps: [0] | DIoU-NMS keeps: [0, 1]
```

The second person overlaps the first with IoU 0.515, just above the threshold, so greedy NMS deletes it; the distance penalty lowers the criterion to 0.478 and DIoU-NMS keeps it, while the near-duplicate (centres 3 pixels apart) is still removed. The effect is confined to a narrow band of overlaps, and cannot resolve all crowded-scene ambiguity. The YOLOv4 ablations do not isolate its contribution.

### Data augmentation: Mosaic and CutMix

**Plain version.** Mosaic glues four training images into one, so every training image shows objects in four different contexts and at smaller sizes. It is the augmentation most associated with YOLOv4 and the one most copied by later detectors.

Section 2.2 of the paper surveys pixel-wise augmentations (photometric changes, random scaling, cropping, flipping, rotation), occlusion-style augmentations that erase rectangles, and augmentations that combine images: MixUp [46] blends two images with weights $\lambda$ and $1-\lambda$ and blends their labels the same way, and CutMix [47] pastes a rectangle from one image into another and mixes the labels in proportion to the pasted area [27]. YOLOv4 uses CutMix for the backbone's classifier training and introduces **Mosaic**, which "mixes 4 training images" [27]. The

paper's acknowledgements credit Glenn Jocher for the idea of Mosaic, along with the genetic hyper-parameter search and the grid-sensitivity fix [27].

The paper gives two reasons for Mosaic: objects are seen "outside their normal context", and batch normalization computes its statistics from four different images per training image, which "significantly reduces the need for a large mini-batch size" [27]. That second reason links Mosaic to the single-GPU goal.

The paper does not specify the procedure in more detail than its Figure 3, and implementations differ in how they scale and crop. A simple version, written out below, chooses a random split point on the output canvas, places the four images so that each touches the split point with one corner, cuts away whatever falls outside each image's quadrant, and clips or drops the boxes accordingly.

```
import numpy as np

def mosaic(images, boxes, size=416, rng=None, min_keep=0.25):
    """Place four images around a random split point (xc, yc) of a size x size canvas.
    images: four (H, W, 3) arrays; boxes: four (n, 4) arrays of (x1, y1, x2, y2) in pixels.
    Each image touches the split point with one corner; whatever falls outside its quadrant
     ↪ is cut.
    A box is kept if at least `min_keep` of its area survives the cut."""
    xc, yc = (int(v) for v in rng.uniform(0.3, 0.7, 2) * size)
    canvas = np.full((size, size, 3), 114, np.uint8)
    out, n_cut, n_drop = [], 0, 0
    for k, (img, bx) in enumerate(zip(images, boxes)):
        h, w = img.shape[:2]
        ox = xc - w if k in (0, 2) else xc          # where the image's top-left corner lands
        oy = yc - h if k in (0, 1) else yc
        rx0, rx1 = (0, xc) if k in (0, 2) else (xc, size)   # the quadrant it may fill
        ry0, ry1 = (0, yc) if k in (0, 1) else (yc, size)
        x0, x1, y0, y1 = max(rx0, ox), min(rx1, ox + w), max(ry0, oy), min(ry1, oy + h)
        canvas[y0:y1, x0:x1] = img[y0 - oy:y1 - oy, x0 - ox:x1 - ox]
        shifted = bx + [ox, oy, ox, oy]
        clipped = np.clip(shifted, [x0, y0, x0, y0], [x1, y1, x1, y1])
        area = lambda b: (b[:, 2] - b[:, 0]) * (b[:, 3] - b[:, 1])
        keep = area(clipped) >= min_keep * area(shifted)
        n_cut += int((keep & (area(clipped) < area(shifted))).sum()); n_drop +=
         ↪ int((~keep).sum())
        out.append(clipped[keep])
    return canvas, np.concatenate(out), (xc, yc), n_cut, n_drop

rng = np.random.default_rng(0)
imgs, bxs = [], []
for k in range(4):                                  # four synthetic 256 x 256 "photos"
    img = np.full((256, 256, 3), 60 * k + 40, np.uint8)
    b = []
    for _ in range(3):                              # three rectangles = three objects each
        x, y = rng.integers(0, 200, 2); w, h = rng.integers(30, 90, 2)
        x2, y2 = min(x + w, 256), min(y + h, 256)
        img[y:y2, x:x2] = rng.integers(0, 256, 3); b.append([x, y, x2, y2])
    imgs.append(img); bxs.append(np.array(b, float))
```

```
canvas, boxes, split, n_cut, n_drop = mosaic(imgs, bxs, rng=rng)
print("canvas", canvas.shape, "| split point", split)
print(f"objects in: {sum(len(b) for b in bxs)} | kept: {len(boxes)} (of which cut: {n_cut}) |
 ↪  dropped: {n_drop}")
```

```
canvas (416, 416, 3) | split point (223, 181)
objects in: 12 | kept: 9 (of which cut: 3) | dropped: 3
```

Figure 3 draws this example. Three of the twelve objects survive only partly and three are cut away almost completely; the rule for dropping a box (here, less than a quarter of its area left) is an implementation choice that the paper does not discuss. Mosaic later became the standard YOLO augmentation, and DEIM (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/deim-dense-one-to-one-matching/) uses it for a different purpose: to put more objects in each image so that one-to-one matching receives more positives.

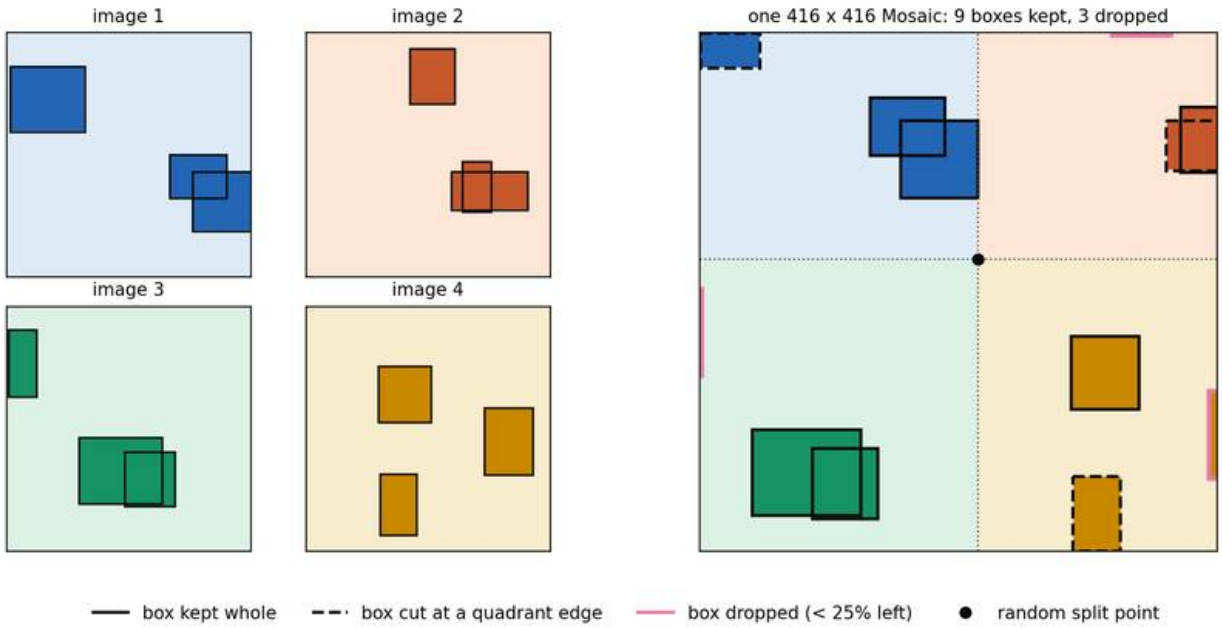



*Figure 3 — Mosaic on four synthetic images (the snippet's procedure, our implementation). Each image touches the random split point with one corner and is cut at its quadrant's edges; boxes are clipped, and boxes with less than 25% of their area left are dropped.*

### Self-adversarial training

**Plain version.** Self-adversarial training (SAT) first lets the network change the training image, not its weights, so that the image fools the network into seeing no object; then it trains the network on that altered image in the normal way. It is a data augmentation generated by the detector itself.

The paper introduces SAT as "a new data augmentation technique that operates in 2 forward backward stages" [27]. In the first stage the network "alters the original image instead of the network weights", executing "an adversarial attack on itself" to create "the deception that there is no desired object"; in the second stage it is "trained to detect an object on this modified image in the normal way" [27]. The paper describes the stages in words only. One way to write them, in our notation rather than the paper's, is

$$x' = x + \eta \nabla_x \mathcal{L}(f_\theta(x), y), \qquad \theta \leftarrow \theta - \lambda \nabla_\theta \mathcal{L}(f_\theta(x'), y),$$

where $x$ is the training image, $y$ its boxes and labels, $f_\theta$ the detector with weights $\theta$, $\mathcal{L}$ the detection loss, $\eta$ the size of the image change (the weights are frozen during this step) and $\lambda$ the learning rate. Stage 1 moves the image to increase the chosen objective; increasing a generic detection loss can also disturb localization rather than merely hide an object. This illustrative objective is not an exact SAT specification from the paper; stage 2 is an ordinary training step on the altered image. SAT requires additional training computation and adds no inference stage; the paper does not separately measure its training-time overhead. The paper lists SAT among the detector's bag of freebies, but its ablation tables contain no SAT column, so its contribution is not measured separately (see Limits of the evidence).

### Regularization: DropBlock and class label smoothing

**Plain version.** Regularizers make training harder on purpose so that the network does not memorize the training set. DropBlock switches off whole square patches of a feature map instead of scattered single units; label smoothing tells the network never to be completely sure about a class.

**DropBlock.** Dropout [48] zeroes random units independently. In a convolutional feature map, neighbouring units are strongly correlated, so a zeroed unit can be reconstructed from its neighbours and dropout regularizes weakly. DropBlock [49] zeroes contiguous $b \times b$ blocks. Following its Algorithm 1: sample seed positions $M_{ij} \sim \text{Bernoulli}(\gamma)$ only where a whole block fits inside the map, zero a $b \times b$ square around each seed, multiply the activations by the mask, and rescale them by $\text{count}(M)/\text{count_ones}(M)$ to compensate approximately for the fraction of surviving activations [49]. To drop roughly a fraction $1 - p_{\text{keep}}$ of the units on a $f \times f$ map, the seed probability is

$$\gamma = \frac{1 - p_{\text{keep}}}{b^2} \cdot \frac{f^2}{(f - b + 1)^2},$$

where $p_{\text{keep}}$ is the keep probability as in dropout, $b$ the block size, $f$ the feature-map size and $(f - b + 1)^2$ the number of valid seed positions; the formula is approximate because blocks can overlap [49]. The DropBlock paper also found that a fixed $p_{\text{keep}}$ hurts early training and gradually decreases it from 1 to the target value [49].

```
import numpy as np

def dropblock_mask(feat, block, keep_prob, rng):
    """One DropBlock mask for a feat x feat map (Ghiasi et al., Algorithm 1)."""
```

```
    gamma = (1 - keep_prob) / block ** 2 * feat ** 2 / (feat - block + 1) ** 2
    half = block // 2
    seeds = np.zeros((feat, feat), bool)                        # only where a whole block
     ↪  fits
    seeds[half:feat - half, half:feat - half] = rng.random((feat - block + 1,) * 2) < gamma
    mask = np.ones((feat, feat))
    for i, j in zip(*np.nonzero(seeds)):
        mask[i - half:i + half + 1, j - half:j + half + 1] = 0.0
    return mask, gamma

rng = np.random.default_rng(0)
feat, block, keep = 38, 7, 0.9
masks = [dropblock_mask(feat, block, keep, rng)[0] for _ in range(2000)]
gamma = dropblock_mask(feat, block, keep, rng)[1]
dropped = 1 - np.mean([m.mean() for m in masks])
print(f"gamma = {gamma:.5f} | target drop rate {1 - keep:.2f} | measured {dropped:.3f}")
m = masks[0]
print("first mask: dropped units", int((m == 0).sum()), "| rescale factor", round(m.size /
 ↪  m.sum(), 3))
```

```
gamma = 0.00288 | target drop rate 0.10 | measured 0.094
first mask: dropped units 171 | rescale factor 1.134
```

On a $38 \times 38$ map with $7 \times 7$ blocks, a seed probability below 0.3% already removes about 9.4% of the units, a little less than the 10% target because overlapping blocks are counted once. YOLOv4 chose DropBlock "without hesitation" because its authors' comparison with other methods was favourable [27], and lists it in both the backbone's and the detector's bag of freebies; the detector ablation table has no DropBlock column.

**Class label smoothing.** With hard one-hot targets, cross-entropy keeps pushing the correct logit up, which encourages over-confidence. Label smoothing [50] replaces the target distribution by

$$q'(k) = (1 - \epsilon)\,\delta_{k,y} + \frac{\epsilon}{K},$$

where $k$ indexes the $K$ classes, $y$ is the true class, $\delta_{k,y}$ is 1 if $k = y$ and 0 otherwise, and $\epsilon$ is the smoothing amount [50]. YOLOv4's detector uses independent sigmoid scores, and its ablation table describes LS as "class label smoothing for sigmoid activation" [27]; for one two-way sigmoid output the same formula with $K = 2$ moves the targets from 1 and 0 to $1 - \epsilon/2$ and $\epsilon/2$. The paper does not state the value of $\epsilon$. Label smoothing helped the classifier (CSPResNeXt-50: 77.9% → 78.1% top-1) but lowered the detector's AP when used alone (38.0% → 33.0%; see Detector bag of freebies) [27].

### Normalization: cross mini-batch normalization

**Plain version.** Batch normalization needs statistics computed over many images. On one GPU, a detector only fits a few images at a time. Cross mini-batch normalization (CmBN) collects the statistics over the several small mini-batches that together make up one training batch.

**Batch normalization** [51] normalizes each channel with the mean $\mu_{\mathcal{B}}$ and variance $\sigma^2_{\mathcal{B}}$ of the current mini-batch $\mathcal{B}$: $\hat{z} = \gamma\,(z - \mu_{\mathcal{B}})/\sqrt{\sigma^2_{\mathcal{B}} + \epsilon} + \beta$, where $z$ is an activation, $\gamma, \beta$ are learned scale and shift and $\epsilon$ is a small constant. With only 4 or 8 images per mini-batch these statistics are noisy. **Cross-iteration batch normalization (CBN)** [52] averages statistics over several recent training iterations; because the weights change between iterations, it compensates the old statistics with a Taylor-polynomial approximation of how they would look under the current weights [52]. Synchronizing statistics across GPUs does not solve the single-GPU constraint.

YOLOv4 trains with a batch of 64 images split into mini-batches of 8 or 4 (the configuration's `subdivisions`), and the weights are updated once per batch [27], [37]. **CmBN** "collects statistics only between mini-batches within a single batch" [27]. In our reading of the paper's Figure 4, the statistics used for the $t$-th mini-batch of a batch accumulate those of mini-batches $1, \dots, t$ of the same batch, for example

$$\mu^{(t)} = \frac{1}{t}\sum_{\tau=1}^{t} \mu_{\mathcal{B}_\tau}, \qquad \sigma^{2\,(t)} = \frac{1}{t}\sum_{\tau=1}^{t} \left(\sigma^2_{\mathcal{B}_\tau} + \mu^2_{\mathcal{B}_\tau}\right) - \left(\mu^{(t)}\right)^2,$$

where $\mathcal{B}_\tau$ is the $\tau$-th equally sized mini-batch and the variances are population moments. Averaging variances alone would omit differences between mini-batch means. Since weights do not change inside this batch, no cross-update compensation is needed. This illustrates correct moment aggregation; it is not an equation supplied by YOLOv4 or a complete specification of its released implementation. In the ablation (column CBN) CmBN alone gave 38.4% AP against the baseline's 38.0% [27].

### Attention and fusion: the modified SAM and weighted residuals

**Plain version.** An attention module lets the network re-weight its features, emphasizing informative positions. YOLOv4 takes the spatial attention module of CBAM and simplifies it to an element-wise gate. It also lists a learned-weight way of adding feature maps, taken from EfficientDet.

**SAM.** The spatial attention module of CBAM [53] pools the feature map $F \in \mathbb{R}^{C\times H\times W}$ along the channel axis by average and by max, concatenates the two $1 \times H \times W$ maps, and convolves them:

$$M_s(F) = \sigma\big(f^{7\times 7}([\,\mathrm{AvgPool}_c(F);\ \mathrm{MaxPool}_c(F)\,])\big), \qquad F' = M_s(F) \otimes F,$$

where $f^{7\times 7}$ is a $7 \times 7$ convolution, $\sigma$ the sigmoid and $\otimes$ multiplication broadcast over channels, so one weight per position scales all channels [53]. The paper notes that SAM costs about 0.1% extra computation on ImageNet classifiers and "does not affect the speed of inference on the GPU at all", whereas it estimates that the representative channel-attention module adds about 10% to GPU inference time [27]. YOLOv4 modifies SAM "from spatial-wise attention to point-wise attention" (its Figure 5): in that figure the two pooling steps are replaced by a convolution applied to $F$ itself, so the gate $\sigma(\mathrm{conv}(F))$ has the same shape $C \times H \times W$ as $F$ and scales every element separately [27]. In the specials ablation, adding SAM to CSPResNeXt50-PANet-SPP raised AP from 42.4% to 42.7% [27].

**Multi-input weighted residual connections (MiWRC).** EfficientDet's feature network adds several inputs with learned non-negative weights, using "fast normalized fusion" [28]:

$$O = \sum_i \frac{w_i}{\varepsilon + \sum_j w_j} I_i,$$

where $I_i$ are the input feature maps, $w_i \geq 0$ (enforced by a ReLU) are learned weights and $\varepsilon = 0.0001$ avoids division by zero [28]. YOLOv4 lists MiWRC in the bag of specials for the backbone but reports no experiment that isolates it.

### Training schedule: cosine annealing, genetic search and random shapes

**Plain version.** Several freebies concern how training is run rather than what the network computes: how the learning rate changes, how the many hyper-parameters are chosen, and which input sizes the network sees.

**Cosine annealing.** Instead of dropping the learning rate in steps, SGDR [54] lowers it along half a cosine:

$$\eta_t = \eta_{\min} + \tfrac{1}{2}(\eta_{\max} - \eta_{\min})\big(1 + \cos\left(\pi\, T_{\text{cur}}/T_i\right)\big),$$

where $\eta_{\max}$ and $\eta_{\min}$ bound the learning rate, $T_{\text{cur}}$ counts the epochs since the last restart and $T_i$ is the length of the current cycle, so $\eta_t$ falls smoothly from $\eta_{\max}$ to $\eta_{\min}$ [54]. YOLOv4's column CA ("altering the learning rate during sinusoid training") gave 38.7% AP alone against 38.0% for the step schedule [27].

**Genetic hyper-parameter search (GA).** A genetic algorithm keeps a population of hyper-parameter settings, trains briefly with each, keeps the best (the "fittest"), and creates new settings by mutating and combining them. The paper uses it "for selecting the optimal hyperparameters during network training on the first 10% of time periods" [27]. The search ran YOLOv3-SPP with the GIoU loss for 300 epochs on a 5,000-image "min-val" set and selected a learning rate of 0.00261, momentum 0.949, the assignment IoU threshold 0.213 and the box-loss normalizer 0.07 [27]. Alone, GA gave 38.9% AP against 38.0% [27].

**Random training shapes and dynamic mini-batch size.** Like YOLOv2 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov2-and-yolo9000-anchors-priors-and-joint-training/)'s multi-scale training [8], YOLOv4 changes the input resolution during training ("random training shapes") so that one model works at several resolutions [27]. "Dynamic mini-batch size" (DM) automatically increases the mini-batch size when the random shape is small, so that the GPU is used fully; alone it lowered AP to 35.3% [27].

**Optimized anchors (OA).** The anchors are re-estimated for the 512 × 512 training resolution, in the spirit of YOLOv2's k-means "dimension clusters" (explained in the DLCV 03 Supplementary notes) [27].

### The complete recipe

**Plain version.** Putting the pieces together, YOLOv4 is CSPDarknet53 + SPP + PAN + YOLOv3 head, trained with a long list of freebies and fitted with a few cheap specials.

Section 3.4 of the paper lists the final model as follows [27]:

| Part | What YOLOv4 uses | Kind |
|---|---|---|
| Backbone | CSPDarknet53 | architecture |
| Neck | SPP, PAN (modified: concatenation) | architecture |
| Head | YOLOv3 (anchor-based, 3 scales × 3 anchors) | architecture |
| Backbone, freebies | CutMix and Mosaic, DropBlock, class label smoothing | training only |
| Backbone, specials | Mish, cross-stage partial connections, MiWRC | small inference cost |
| Detector, freebies | CIoU loss, CmBN, DropBlock, Mosaic, self-adversarial training, grid-sensitivity elimination, multiple anchors per ground truth, cosine annealing, genetically searched hyper-parameters, random training shapes | training only |
| Detector, specials | Mish, SPP block, modified SAM, modified PAN, DIoU-NMS | small inference cost |

Later detectors kept the overall shape of this recipe (a CSP-style backbone, an FPN–PAN neck, Mosaic, IoU-based losses, many positives per object) and replaced specific items; Problems maps those changes to the components above.

### Results

**Plain version.** On COCO, YOLOv4 reached 43.5% AP at 608 × 608. Its 416-pixel model exceeded 30 FPS on all three tested GPU generations; the 608-pixel model reached only 23 FPS on Maxwell. The useful result is a measured speed–accuracy trade-off, not real-time operation for every resolution and GPU.

### Training and evaluation setup

All detector experiments train on COCO [55] and report accuracy on test-dev 2017 with the COCO AP metrics [56] (AP averaged over IoU thresholds 0.50 to 0.95, plus AP50, AP75 and AP for small, medium and large objects) [27]. The classifier experiments train on ImageNet [30] and report ILSVRC 2012 validation accuracy [27].

The detector's default schedule is 500,500 training steps with a step-decay learning rate starting at 0.01 and multiplied by 0.1 at steps 400,000 and 450,000; momentum 0.9 and weight decay 0.0005;

one GPU, multi-scale training, a batch size of 64 and a mini-batch size of 8 or 4 depending on memory [27]. The classifier's default is 8,000,000 steps, batch 128 with mini-batch 32, a polynomial-decay learning rate starting at 0.1 with 1,000 warm-up steps, momentum 0.9 and weight decay 0.005; the BoF classifier experiments train 50% longer [27]. Everything runs on a single 1080 Ti or 2080 Ti [27].

### The main comparison

Tables 8, 9 and 10 of the paper compare YOLOv4 with other detectors on COCO test-dev, one table per GPU architecture: Maxwell (GTX Titan X or Tesla M40), Pascal (Titan X Pascal, Titan Xp, GTX 1080 Ti or Tesla P100) and Volta (Titan Volta or Tesla V100), all "with batch=1 without using tensorRT" [27]. YOLOv4's own rows are identical in accuracy across the three tables, because only the speed is re-measured:

| Input | AP | AP50 | AP75 | APS | APM | APL | FPS Maxwell | FPS Pascal | FPS Volta |
|---|---|---|---|---|---|---|---|---|---|
| 416 | 41.2 | 62.8 | 44.3 | 20.4 | 44.4 | 56.0 | 38 | 54 | 96 |
| 512 | 43.0 | 64.9 | 46.5 | 24.3 | 46.1 | 55.2 | 31 | 43 | 83 |
| 608 | 43.5 | 65.7 | 47.3 | 26.7 | 46.7 | 53.3 | 23 | 33 | 62 |

How to read it against the other rows:

- **Against YOLOv3** (Table 8, Maxwell): YOLOv3-608 reaches 33.0% AP and YOLOv3-SPP-608 36.2% AP, both at 20 FPS, so YOLOv4-608 is about 10 AP points more accurate at a slightly higher frame rate on the same GPU class [27].
- **Against EfficientDet** (Table 10, Volta): EfficientDet-D0 to D3 reach 33.8%, 39.6%, 43.0% and 45.8% AP at 62.5, 50.0, 41.7 and 23.8 FPS. YOLOv4-512 matches D2's 43.0% AP at 83 FPS instead of 41.7; D3 is more accurate (45.8%) but runs at 23.8 FPS, below the 30 FPS line [27]. This is the basis of the paper's claim that YOLOv4 runs about twice as fast as EfficientDet at comparable accuracy.
- **Against the slower, more accurate detectors**: several two-stage and keypoint-based detectors in Table 9 reach higher AP (the highest, 46.8%, runs at 1.3 FPS), but none of them is real-time [27]. YOLOv4 does not claim the highest AP overall, only the best accuracy among detectors at 30 FPS or more.

Two details deserve attention. First, the large-object AP *falls* as the input grows (56.0 → 55.2 → 53.3), while small-object AP rises from 20.4 to 26.7; a larger input helps small objects but makes large objects relatively larger than the receptive field and the anchors were tuned for. Second, the abstract quotes "~65 FPS on Tesla V100" for the 43.5% AP model, while Table 10 lists 62 FPS for it [27]; the table is the more specific source.

### Speed: a measurement checklist

When you measure, or read, a latency number for any detector, check the following; each item is a condition YOLOv4's tables fix, leave open, or vary.

1. **Hardware, engine and precision** are stated and the same for every model being compared. YOLOv4 states the GPU family and "without TensorRT", but not the precision.
2. **Batch size 1** for latency; throughput numbers with large batches do not transfer to a live camera.
3. **End to end**: pre-processing, the network and post-processing including NMS, or a clear statement of what is excluded. YOLOv4 does not state whether NMS is timed; its successors' end-to-end benchmarks add NMS explicitly (see RT-DETR (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rt-detr-a-real-time-detr/) and YOLOv10 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov10-nms-free-training-with-dual-assignments/)), averaged over the COCO validation images because NMS time depends on the image.
4. **Same device, not only the same architecture family**: YOLOv4's tables group cards of one GPU generation (for example a Titan X Maxwell and a Tesla M40), and the speeds of other methods come from their own papers, so small FPS differences between rows are not measurements on identical hardware.
5. **Warm-up and averaging**: the first iterations of any engine are slow (memory allocation, kernel selection), so discard them and average over many images.
6. **Input size and confidence threshold** used during timing; for NMS-based detectors, the threshold used for computing AP (often very low) also changes the post-processing time.

## Ablations

**Plain version.** The paper's experiments are a long series of "try one change, keep it if it helps" tables: first for the backbone as a classifier, then for the detector's freebies and specials, then for the backbone choice and the mini-batch size. Reading them carefully shows which components really matter and which were kept on weaker evidence.

### Classifier ablations

Tables 2 and 3 train the two candidate backbones as ImageNet classifiers with different freebies and activations (top-1 / top-5 accuracy) [27].

| CSPResNeXt-50 setting | Top-1 | Top-5 |
|---|---|---|
| baseline | 77.9 | 94.0 |
| MixUp | 77.2 | 94.0 |
| CutMix | 78.0 | 94.3 |
| Mosaic | 78.1 | 94.5 |
| bilateral blurring | 77.5 | 93.8 |
| label smoothing | 78.1 | 94.4 |

| CSPResNeXt-50 setting | Top-1 | Top-5 |
|---|---|---|
| Swish | 64.5 | 86.0 |
| Mish | 78.9 | 94.5 |
| CutMix + Mosaic + label smoothing | 78.5 | 94.8 |
| CutMix + Mosaic + label smoothing + Mish | 79.8 | 95.2 |

For CSPDarknet-53 the baseline is 77.2 / 93.6, the three freebies give 77.8 / 94.4 and adding Mish gives 78.7 / 94.8 [27]. What this proves: CutMix, Mosaic, label smoothing and Mish each help the classifier, and they combine; MixUp and blurring do not. The Swish row (64.5%) stands out; the paper does not discuss it, and a drop that large looks more like a failed training run than a property of the activation, so it should not be read as evidence against Swish in general.

### Detector bag of freebies

Table 4 tests detector freebies on CSPResNeXt50-PANet-SPP at 512 × 512 [27]. The columns are S (grid-sensitivity elimination), M (Mosaic), IT (several anchors per ground truth above the IoU threshold), GA (genetic hyper-parameters), LS (class label smoothing), CBN (CmBN), CA (cosine annealing), DM (dynamic mini-batch size), OA (optimized anchors) and the box loss. Reading the rows in groups:

| Rows | Setting | AP | AP50 | AP75 |
|---|---|---|---|---|
| baseline | MSE loss, no freebie | 38.0 | 60.0 | 40.8 |
| one freebie, MSE loss | S | 37.7 | 59.9 | 40.5 |
| | M | 39.1 | 61.8 | 42.0 |
| | IT | 36.9 | 59.7 | 39.4 |
| | GA | 38.9 | 61.7 | 41.9 |
| | LS | 33.0 | 55.4 | 35.4 |
| | CBN | 38.4 | 60.7 | 41.3 |
| | CA | 38.7 | 60.7 | 41.9 |
| | DM | 35.3 | 57.2 | 38.0 |
| S + loss | GIoU / DIoU / CIoU | 39.4 / 39.1 / 39.6 | 59.4 / 58.8 / 59.2 | 42.5 / 42.1 / 42.6 |
| combinations | S + M + IT + GA, CIoU | 41.5 | 64.0 | 44.8 |
| | M + GA + OA, CIoU | 36.1 | 56.5 | 38.4 |
| | S + M + IT + GA + OA, MSE | 40.3 | 64.0 | 43.1 |
| | S + M + IT + GA + OA, GIoU | 42.4 | 64.4 | 45.9 |
| | S + M + IT + GA + OA, CIoU | 42.4 | 64.4 | 45.9 |

What the table proves:

- **Mosaic is the strongest single freebie** (+1.1 AP alone), followed by GA, cosine annealing and CmBN, which each add a few tenths.

- **IoU-type losses beat MSE**, and among them CIoU is best by a small margin when combined with S (39.6 against 39.4 for GIoU); in the best combination GIoU and CIoU tie at 42.4.
- **Some freebies hurt in isolation**: S alone (−0.3), IT alone (−1.1), DM (−2.7) and LS (−5.0). S and IT pay off only in combination: S + M + IT + GA with CIoU reaches 41.5, and the best setting, 42.4 AP, includes both. A plausible reading is that more positives (IT) and wider centre ranges (S) need the stronger training of the combined recipe; the paper does not analyse it.
- **The best rows do not contain LS, CBN, CA or DM**, although Section 3.4 lists label smoothing, CmBN and cosine annealing in YOLOv4's detector freebies. The table never shows a row with all of the final freebies together, so their joint effect on the detector is not measured here.

### Detector bag of specials

Table 5 adds modules to CSPResNeXt50-PANet-SPP at 512 × 512 [27]: the base with SPP and PAN reaches 42.4% AP; adding SAM gives 42.7% (AP50 64.6, AP75 46.3); the other combinations it tests (a receptive-field block, a Gaussian box-uncertainty head on top of SAM, and an adaptive feature-fusion block with the receptive-field block) all lower AP, to between 41.1% and 41.8% [27]. What it proves: SPP, PAN and SAM are the useful specials for this detector; adding more modules is not free. DIoU-NMS, Mish in the detector and MiWRC have no row.

### Backbones and pre-trained weights

Table 6 trains the detector with different classifier weights for initialization, all at 512 × 512 with otherwise similar settings [27]:

| Detector backbone (initialized from) | AP | AP50 | AP75 |
|---|---|---|---|
| CSPResNeXt50 (default classifier) | 42.4 | 64.4 | 45.9 |
| CSPResNeXt50 (classifier with BoF) | 42.3 | 64.3 | 45.7 |
| CSPResNeXt50 (classifier with BoF + Mish) | 42.3 | 64.2 | 45.8 |
| CSPDarknet53 (classifier with BoF) | 42.4 | 64.5 | 46.0 |
| CSPDarknet53 (classifier with BoF + Mish) | 43.0 | 64.9 | 46.5 |

What it proves: for CSPResNeXt50, a better classifier (79.8% instead of 77.9% top-1) does *not* give a better detector; for CSPDarknet53, the better classifier does. This is the experimental basis for the backbone choice, and the general lesson that ImageNet accuracy is a poor proxy for detection quality.

### Mini-batch size

Table 7 trains with mini-batches of 4 and 8 images [27]. Without the freebies and specials, CSPResNeXt50-PANet-SPP at 608 reaches 37.1% AP with mini-batch 4 and 38.4% with mini-batch 8; with them, CSPDarknet53-PANet-SPP at 512 reaches 41.6% and 41.7%. What it proves, within this setting: after the full recipe the mini-batch size "has almost no effect", which supports the claim that the detector can be trained well on one conventional GPU. Mosaic's four-images-per-sample statistics and CmBN are the components designed for this; the table does not separate their contributions.

## Conclusion

YOLOv4 shows that a classic one-stage, anchor-based detector, given a stronger backbone and neck and a carefully tested set of training tricks, can match much slower detectors in accuracy while running in real time, and can be trained on a single consumer GPU. Its experiments establish that Mosaic, IoU-type box losses, genetic hyper-parameter search, more positives per object, SPP, PAN and point-wise SAM help such a detector, that some popular tricks do not, and that the best classifier backbone is not automatically the best detector backbone. The paper's lasting contribution is less any single module than the vocabulary (backbone, neck, head; freebies and specials) and the practice of testing training tricks on the detector itself.

## Problems

**Plain version.** YOLOv4 kept the parts that later detectors found most limiting: anchors, a shared head for classes and boxes, many positives per object and therefore NMS. Its evidence also has gaps: several listed components are never tested on the detector, the ablations are single runs on the test set, and the paper was never peer reviewed.

### Limits of the method

- **Anchors are dataset-specific.** The nine anchors are clustered for COCO (and re-optimized for one input size); a new dataset or resolution needs new anchors, and an object whose shape matches no anchor is learned poorly. YOLOX (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolox-anchor-free-yolo-with-simota/) argues exactly this when it drops anchors.
- **A coupled head.** One set of convolutions predicts classes and boxes together. YOLOX (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolox-anchor-free-yolo-with-simota/) shows that separating them speeds up convergence.
- **Fixed-rule assignment.** Positives are chosen by anchor shape and object centre, not by how well the network currently predicts; later work assigns positives from the network's own predictions (SimOTA in YOLOX, task-aligned assignment in YOLOv6 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov6-a-re-parameterized-industrial-detector/)).

- **NMS dependency.** Several positives per object make NMS necessary, with its threshold to tune, its cost that grows with the scene (Figure 2) and its failure on neighbouring objects; DIoU-NMS only narrows that failure. The NMS-free line starts with YOLOv10 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov10-nms-free-training-with-dual-assignments/).
- **Large objects at high resolution.** APL drops from 56.0 at 416 to 53.3 at 608 in the paper's own results [27].

### Limits of the evidence

- **Not peer reviewed.** YOLOv4 exists as an arXiv preprint [27]; it was never published at a reviewed venue (unlike YOLOv7 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov7-a-trainable-bag-of-freebies/), by two of the same authors).
- **Ablations on test-dev, single runs.** The detector ablations report test-dev results, a split intended for final evaluation, and give single numbers without variance; differences of a few tenths of AP may be within run-to-run variation.
- **Untested components.** Self-adversarial training, DropBlock in the detector, DIoU-NMS, Mi-WRC and the PAN modification have no ablation row, and no row combines all the final freebies. The final recipe therefore includes components whose effect on this detector is asserted rather than measured.
- **Speed conditions.** Numeric precision and the treatment of NMS are not stated, and competitor speeds come from other papers on the same GPU generation, not the same card. The abstract's "~65 FPS" and Table 10's 62 FPS disagree slightly.

### The paper and the released configuration

The configuration file `cfg/yolov4.cfg` in the authors' repository (as accessed for this chapter) differs from the paper's list in places [37]: it uses a step learning-rate schedule (`policy=steps`, steps 400,000 and 450,000) rather than cosine annealing, greedy NMS (`nms_kind=greedynms`) rather than DIoU-NMS, and contains no SAM or DropBlock layer, while it does use Mosaic, the CIoU loss, the grid-sensitivity factors and the 0.213 assignment threshold. Its learning rate is 0.0013 rather than the 0.00261 found by the genetic search. A repository file can change after publication, so this does not show what was used for the paper's tables; it shows that "YOLOv4" in practice may not include every item of Section 3.4, which matters when you compare reproductions.

### Export, reduced precision and re-parameterization

Training frameworks are not deployment engines. A trained detector is usually **exported** to an exchange format such as ONNX [57] and compiled by an inference engine for the target device, for example TensorRT for NVIDIA GPUs [29]. Three things change on the way:

- **Precision.** FP16 uses half the storage per represented value compared with FP32; the speed and accuracy changes depend on the graph and runtime. INT8 quantization represents selected weights and activations with 8-bit integers and scale parameters [58]. Calibration or quantization-aware training is needed. Validate all output stages and check AP at strict IoU thresholds after quantizing, not only AP50. YOLOv6 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov6-a-re-parameterized-industrial-detector/) reports that re-parameterized models are harder to quantize and changes its training to fix this [16].
- **Graph simplification.** Batch normalization is folded into the preceding convolution, and re-parameterized multi-branch blocks (used from YOLOv6 and YOLOv7 on) are folded into single convolutions; the folding is derived in the Supplementary notes.
- **Unsupported operators and post-processing.** Every operator must exist in the target engine. Mish, for example, is a less common operator than a rectifier. RT-DETRv2 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rt-detrv2-a-bag-of-freebies-for-rt-detr/) adds a discrete sampling operator for this reason [22], and YOLO26 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolo26-a-dfl-free-end-to-end-yolo/) cites simpler export as one reason for removing its distribution-based box head [20]. Removing NMS can simplify one part of export, but decoding, score selection and coordinate restoration still have to be implemented and timed.

*Online only: Folding a re-parameterized block into one convolution — https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/notes/folding-a-re-parameterized-block-into-one-convolution/*

### What later one-stage detectors changed

Each change below replaces or extends one YOLOv4 component; the linked paper guides explain the method in full.

| YOLOv4 component | What came later | Paper |
|---|---|---|
| Anchor-based coupled head | Anchor-free decoupled head, one prediction per location | YOLOX (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolox-anchor-free-yolo-with-simota/) |
| Fixed-rule positives (IoU of shapes > 0.213) | Prediction-aware assignment: SimOTA, then task-aligned assignment | YOLOX (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolox-anchor-free-yolo-with-simota/), YOLOv6 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov6-a-re-parameterized-industrial-detector/) |

| YOLOv4 component | What came later | Paper |
|---|---|---|
| CSPDarknet53 blocks | Re-parameterizable blocks folded into plain convolutions for inference | YOLOv6 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov6-a-re-parameterized-industrial-detector/), YOLOv7 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov7-a-trainable-bag-of-freebies/) |
| Training only the final head | Auxiliary heads and branches that supervise the network during training only | YOLOv7 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov7-a-trainable-bag-of-freebies/), YOLOv9 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov9-programmable-gradient-information/) |
| Several positives + NMS (DIoU-NMS) | A one-to-one head trained alongside a one-to-many head; no NMS | YOLOv10 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov10-nms-free-training-with-dual-assignments/), YOLO26 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolo26-a-dfl-free-end-to-end-yolo/) |
| SAM, a cheap spatial gate | Real self-attention made affordable (area attention); hypergraph correlation | YOLOv12 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov12-an-attention-centric-yolo/), YOLOv13 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov13-hypergraph-correlation/) |
| CIoU on four box numbers | Distribution-based box sides (DFL), later removed again | YOLOv6 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov6-a-re-parameterized-industrial-detector/), YOLO26 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolo26-a-dfl-free-end-to-end-yolo/) |
| SGD with step or cosine schedule | MuSGD, a hybrid of SGD and the Muon optimizer | YOLO26 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolo26-a-dfl-free-end-to-end-yolo/) |

What stayed: the CSP idea in backbones, an FPN-plus-PAN neck, Mosaic, IoU-type box losses and the habit of measuring speed on real hardware.

### Beyond one-stage: real-time DETRs

**Plain version.** DETR showed that a transformer can output the final set of boxes directly, with one prediction per object and no NMS, but it was slow to train and slow to run. Real-time DETRs keep the direct-set idea and redesign the expensive parts; they are not one-stage dense detectors, but since 2023 they compete with YOLOs at the same speed.

DETR [59] and its descendants are introduced in DLCV 03. The real-time line that started with RT-DETR differs from YOLOv4 in exactly the component that YOLOv4 left untouched: it trains with one-to-one (Hungarian [60]) matching, so each object has a single prediction and no NMS is needed. Each paper has its own introduction in the Paper guides:

| Paper | What it changes |
|---|---|
| RT-DETR (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rt-detr-a-real-time-detr/) | A hybrid encoder (attention on the coarsest level only, convolutional fusion across levels), uncertainty-minimal query selection, adjustable decoder depth, and an end-to-end speed benchmark that times NMS for YOLOs |
| RT-DETRv2 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rt-detrv2-a-bag-of-freebies-for-rt-detr/) | Per-scale sampling points, an optional discrete sampling operator for deployment, dynamic augmentation and scale-specific learning rates |
| D-FINE (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/d-fine-boxes-as-refined-distributions/) | Box edges as probability distributions refined decoder layer by layer, with self-distillation from the last layer |
| DEIM (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/deim-dense-one-to-one-matching/) | More positives under one-to-one matching through Mosaic and MixUp, and a matchability-aware classification loss |
| RF-DETR (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rf-detr-architecture-search-on-a-dinov2-backbone/) | A DINOv2 backbone and weight-sharing architecture search that yields many speed–accuracy trade-offs without retraining |
| DEIMv2 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/deimv2-dinov3-features-for-real-time-detrs/) | DINOv3 features turned into multi-scale features by a light adapter, plus pruned convolutional backbones for the smallest models |

## Modern view

### What the surveys say

**A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS** (Terven, Córdova-Esparza and Romero-González, *Machine Learning and Knowledge Extraction*, 2023) [61] walks through 16 versions in order, starting from the metrics and post-processing, then the architecture and training tricks of each. Its discussion organizes the history by four axes: anchors (absent in YOLO v1, added in YOLOv2, removed again from YOLOX on), framework (Darknet, then PyTorch), backbone (Darknet, then cross-stage partial connections, re-parameterization and architecture search), and the constant speed–accuracy trade-off. It expects COCO to be replaced by harder benchmarks and YOLO models to expand into tracking and multimodal tasks.

**YOLO Evolution: A Comprehensive Benchmark and Architectural Review of YOLOv12,**

**YOLO11, and Their Previous Versions** (Jegham, Koh, Abdelatti and Hendawi, arXiv, 2024, revised 2025) [62] is an experimental comparison rather than a history: it evaluates versions from YOLOv3 to YOLOv12 on datasets chosen for varying object sizes, aspect ratios and small single-class objects, and reports precision, recall, mAP, processing time, GFLOPs and model size. Its takeaway is that no version wins everywhere: in its experiments YOLO11 gave the best overall balance, YOLOv10 was fast but weaker on overlapping objects, and YOLOv12's added complexity did not pay off. Treat these as one group's results on its datasets, not general rankings.

**Ultralytics YOLO Evolution: An Overview of YOLO27, YOLO26, YOLO11, YOLOv8, and YOLOv5** (Sapkota and Karkee, arXiv, 2025, revised 2026) [63] reviews only the Ultralytics releases, covering architecture, benchmarking, deployment (export and quantization) and applications, and lists open challenges: dense scenes, combining CNNs with transformers, open-vocabulary perception, domain generalization and hardware-aware optimization. It is a third-party review, not an Ultralytics publication.

**Object Detection with Transformers: A Review** (Shehzadi et al., *Sensors*, 2025) [64] covers the DETR side: improvements to DETR grouped by backbone, query design and attention mechanism, which is the context for the real-time DETRs.

### The state of the art, 2023–2026

- **NMS-free is becoming the default.** YOLOv10 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov10-nms-free-training-with-dual-assignments/) [17] introduced it to YOLOs in 2024; YOLO26 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolo26-a-dfl-free-end-to-end-yolo/) [20] ships it as the default deployment path; the real-time DETRs discussed here omit NMS at inference. The announced YOLO27 describes query-based decoding without NMS for its larger models [65].
- **Attention is affordable.** YOLOv12 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov12-an-attention-centric-yolo/) [18] and YOLOv13 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/yolov13-hypergraph-correlation/) [19] put attention-like modules inside YOLOs, where YOLOv4 could only afford a cheap spatial gate; RT-DETR (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rt-detr-a-real-time-detr/) [21] restricts attention to the coarsest level.
- **Pretrained vision transformers as backbones.** RF-DETR (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rf-detr-architecture-search-on-a-dinov2-backbone/) [25] and DEIMv2 (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/deimv2-dinov3-features-for-real-time-detrs/) [26] start from self-supervised backbones (DINOv2 [66], DINOv3 [67]), whose benefits must be checked separately for accuracy, transfer and latency.
- **Training efficiency.** DEIM (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/deim-dense-one-to-one-matching/) [24] and YOLO26 (with a new optimizer and a progressive loss) target shorter training rather than only higher AP, a direct descendant of YOLOv4's single-GPU goal.

### Open problems

1. **Fair speed comparison.** There is no shared, maintained end-to-end benchmark across devices; most tables still mix conditions, and RF-DETR (https://blog.shyandram.dev/en/learn/deep-learning/real-time-detection/papers/rf-detr-architecture-search-on-a-dinov2-backbone/) shows that even GPU power throttling changes reported latencies.
2. **Small and crowded objects** remain the main accuracy gap at real-time input sizes; YOLO26's small-object assignment and DEIMv2's unchanged small-object AP both illustrate it.
3. **Quantization-friendly heads.** Score and box outputs lose accuracy at INT8, and attention layers complicate low-precision deployment.
4. **Peer review and reproducibility.** YOLOv4 itself and several widely deployed successors have no peer-reviewed paper, released code can differ from the paper, and version numbers are claimed freely; claims need independent checks.
5. **Beyond closed sets.** Bringing open-vocabulary detection to the same latency as closed-set YOLOs without losing accuracy.

### Key takeaways

- Real-time performance is an end-to-end latency budget on a stated device, not an isolated FPS value.
- YOLOv4 combines CSPDarknet53, SPP and a modified PANet with the YOLOv3 head.
- Distinguish training-only techniques from inference components when interpreting cost.
- Ablations support some components more directly than others; isolated gains and combined gains can differ.
- Classifier accuracy and operation counts do not determine detector accuracy or measured speed.
- Later detectors revisit heads, assignment, matching and feature fusion; compare complete training and inference settings.

### Exercises

1. **Budget arithmetic.** A camera runs at 25 FPS. Pre-processing takes 3 ms, the network 18 ms, and NMS between 1 ms (empty street) and 9 ms (crowd). Does the system keep up in both cases, and how much time is left for tracking? Which single change would help most in crowds?

   **Hint**

   The budget is 40 ms. The total is 22 ms in the empty street and 30 ms in the crowd, leaving 18 ms and 10 ms. The NMS term is the only one that depends on the scene, so an NMS-free head (or a higher confidence threshold, at a cost in recall) helps most in crowds.

2. **Count YOLOv4's predictions.** How many predictions does YOLOv4 make for a 512 × 512 input, and how many for a 1024 × 1024 input? If NMS cost grew with the square of the number

of candidates entering it, how much more expensive would NMS be at 1024 than at 512 when the same fraction of candidates survives the confidence threshold?

**Hint**

At 512 the grids are $64^2 + 32^2 + 16^2 = 5{,}376$ cells, times 3 anchors = 16,128 predictions. At 1024 the count is four times larger, 64,512. A fourfold increase in candidates gives a sixteenfold increase in the worst-case number of IoU computations; real NMS sits between linear and quadratic growth, as the snippet's crowded and sparse scenes show.

3. **Grid sensitivity by hand.** With $s = 1.1$ (YOLOv4's stride-16 setting), which range of centre offsets inside a cell can the head produce, and which $t_x$ places the centre exactly on the cell's right border? Modify the decoding snippet to check your answer.

**Hint**

The offset $s\,\sigma(t_x) - (s-1)/2$ ranges over $(-0.05, 1.05)$. The right border (offset 1) needs $\sigma(t_x) = 1.05/1.1 \approx 0.9545$, so $t_x = \ln(0.9545/0.0455) \approx 3.04$.

4. **DropBlock budget.** On a $19 \times 19$ map with block size 7, what seed probability $\gamma$ targets a drop rate of 10%? Use the snippet to measure the actual drop rate and explain why it differs more from 10% than on the $38 \times 38$ map.

**Hint**

$\gamma = 0.1/49 \cdot 361/169 \approx 0.00436$. On a small map, a larger share of seeds produce blocks that overlap each other, and only $13 \times 13$ seed positions are valid, so overlap and the approximation in the formula matter more; expect a measured rate noticeably below 10%.

5. **Design a better ablation.** Table 4 tests most freebies one at a time against a weak baseline and in a few combinations. Design an experiment that measures the contribution of each of YOLOv4's final detector freebies in the presence of all the others. How many training runs does it need, and what would you report besides the mean AP?

**Hint**

A leave-one-out ablation: train the full recipe once, then once without each component (about a dozen runs for the detector's freebies), with several seeds per setting so that you can report a standard deviation. Evaluate on a validation split (val2017), not test-dev, and report AP, AP50, AP75 and the size-specific APs, plus training time, since some freebies (self-adversarial training) cost training time.

6. **Two speed tables.** Paper A reports 4.1 ms on an A100 GPU, FP16, batch 32, forward pass only. Paper B reports 6.3 ms on a T4 GPU, FP16, batch 1, end to end with NMS. List every reason why you cannot conclude that A is faster, and design the experiment that would let you compare them.

**Hint**

Different GPUs, different batch sizes (A's per-image time at batch 32 hides latency), different timed stages (A excludes pre- and post-processing). Run both exported models on one device with one engine and precision, at batch 1, at the same input size, timing end to end over the same images after warm-up, with the confidence threshold you will deploy.

Detections from a real-time detector are the input to tracking, covered in DLCV 07 ·Object Tracking.

## References


1. C.-Y. Wang, A. Bochkovskiy and H.-Y. M. Liao, "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," in *Proc. IEEE/CVF CVPR*, pp. 7464–7475, 2023. doi
2. C.-Y. Wang, I-H. Yeh and H.-Y. M. Liao, "YOLOv9: Learning what you want to learn using programmable gradient information," in *Proc. ECCV*, 2024. doi
3. G. Jocher, "Ultralytics YOLOv5," software (first released 2020; v7.0 released 22 November 2022). release ·documentation
4. G. Jocher, A. Chaurasia and J. Qiu, "Ultralytics YOLOv8," software, version 8.0.0, 2023 (no paper). docs ·code
5. G. Jocher and J. Qiu, "Ultralytics YOLO11," software, version 11.0.0, 2024 (no paper). docs · code
6. J. Lu, J. Jia, J. Yawl and C. Zhang, "YOLOv14: Adaptive real-time object detection for diverse imaging conditions," arXiv:2608.04720, 2026 (withdrawn by the authors, 20 August 2026). arXiv
7. J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE CVPR*, pp. 779–788, 2016. doi ·arXiv
8. J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proc. IEEE CVPR*, pp. 6517–6525, 2017. doi ·arXiv
9. J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," tech. report, arXiv:1804.02767, 2018. arXiv
10. W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Fu, A. C. Berg, "SSD: Single Shot MultiBox Detector," *Lecture Notes in Computer Science / Computer Vision – ECCV 2016*, 2016. doi · manuscript
11. T.-Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár, "Focal loss for dense object detection," in *Proc. IEEE ICCV*, pp. 2999–3007, 2017. doi
12. Z. Tian, C. Shen, H. Chen, T. He, "FCOS: Fully Convolutional One-Stage Object Detection," *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019. doi ·manuscript
13. X. Zhou, D. Wang, P. Krähenbühl, "Objects as Points," *arXiv preprint*, 2019. source
14. C. Wang, I. Yeh, H. M. Liao, "You Only Learn One Representation: Unified Network for Multiple Tasks," *arXiv preprint*, 2021. source
15. Z. Ge, S. Liu, F. Wang, Z. Li and J. Sun, "YOLOX: Exceeding YOLO series in 2021," arXiv:2107.08430, 2021. arXiv

16. C. Li, L. Li, H. Jiang, K. Weng, Y. Geng et al., "YOLOv6: A single-stage object detection framework for industrial applications," tech. report, arXiv:2209.02976, 2022. arXiv
17. A. Wang, H. Chen, L. Liu, K. Chen, Z. Lin, J. Han and G. Ding, "YOLOv10: Real-time end-to-end object detection," in *Proc. NeurIPS*, 2024. arXiv
18. Y. Tian, Q. Ye and D. Doermann, "YOLOv12: Attention-centric real-time object detectors," in *Advances in Neural Information Processing Systems 38* (*NeurIPS*), 2025. proceedings ·arXiv
19. M. Lei, S. Li, Y. Wu, H. Hu, Y. Zhou et al., "YOLOv13: Real-time object detection with hypergraph-enhanced adaptive visual perception," arXiv:2506.17733, 2025. arXiv
20. G. Jocher, J. Qiu, M. Liu, S. Lyu, F. C. Akyon and M. E. Kalfaoglu, "Ultralytics YOLO26: Unified real-time end-to-end vision models," arXiv:2606.03748, 2026 (software released January 2026). arXiv ·docs
21. Y. Zhao, W. Lv, S. Xu, J. Wei, G. Wang, Q. Dang, Y. Liu, J. Chen, "DETRs Beat YOLOs on Real-time Object Detection," *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024. source
22. W. Lv, Y. Zhao, Q. Chang, K. Huang, G. Wang and Y. Liu, "RT-DETRv2: Improved baseline with bag-of-freebies for real-time detection transformer," arXiv:2407.17140, 2024. arXiv
23. Y. Peng, H. Li, P. Wu, Y. Zhang, X. Sun and F. Wu, "D-FINE: Redefine regression task in DETRs as fine-grained distribution refinement," in *Proc. ICLR*, 2025. arXiv ·code
24. S. Huang, Z. Lu, X. Cun, Y. Yu, X. Zhou, X. Shen, "DEIM: DETR with Improved Matching for Fast Convergence," *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025. source
25. I. Robinson, P. Robicheaux, M. Popov, D. Ramanan and N. Peri, "RF-DETR: Neural architecture search for real-time detection transformers," in *Proc. ICLR*, 2026. arXiv
26. S. Huang, Y. Hou, L. Liu, X. Yu and X. Shen, "Real-time object detection meets DINOv3," arXiv:2509.20787, 2025 (revised 2026). arXiv
27. A. Bochkovskiy, C.-Y. Wang and H.-Y. M. Liao, "YOLOv4: Optimal speed and accuracy of object detection," arXiv:2004.10934, 2020. arXiv
28. M. Tan, R. Pang, Q. V. Le, "EfficientDet: Scalable and Efficient Object Detection," *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2020. doi ·manuscript
29. NVIDIA, "TensorRT documentation," official documentation. link
30. J. Deng, W. Dong, R. Socher, L. Li, Kai Li, Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009. source
31. K. Simonyan, A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv preprint*, 2014. source
32. K. He, X. Zhang, S. Ren, J. Sun, "Deep Residual Learning for Image Recognition," *arXiv preprint*, 2015. source
33. T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan and S. Belongie, "Feature pyramid networks for object detection," in *Proc. IEEE CVPR*, 2017. doi
34. S. Liu, L. Qi, H. Qin, J. Shi and J. Jia, "Path aggregation network for instance segmentation," in *Proc. IEEE/CVF CVPR*, pp. 8759–8768, 2018. doi
35. K. He, X. Zhang, S. Ren, J. Sun, "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition," *Lecture Notes in Computer Science / Computer Vision – ECCV 2014*, 2014. doi ·manuscript
36. C.-Y. Wang, H.-Y. M. Liao, I-H. Yeh, Y.-H. Wu, P.-Y. Chen and J.-W. Hsieh, "CSPNet: A new

backbone that can enhance learning capability of CNN," *CVPR Workshops*, 2020. arXiv ·doi
37. A. Bochkovskiy and contributors, "Darknet: YOLOv4 configuration and detection layer," official implementation, accessed 2 November 2026. configuration ·assignment and decoding
38. M. Tan, Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," *ICML*, 2019. source ·proceedings
39. G. Huang, Z. Liu, L. Van Der Maaten, K. Q. Weinberger, "Densely Connected Convolutional Networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. doi ·manuscript
40. S. Xie, R. Girshick, P. Dollar, Z. Tu, K. He, "Aggregated Residual Transformations for Deep Neural Networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. doi ·manuscript
41. D. Misra, "Mish: A Self Regularized Non-Monotonic Activation Function," *BMVC*, 2020. source
42. P. Ramachandran, B. Zoph, Q. V. Le, "Searching for Activation Functions," *arXiv preprint*, 2017. source
43. J. Yu, Y. Jiang, Z. Wang, Z. Cao, T. Huang, "UnitBox: An Advanced Object Detection Network," *ACM Multimedia*, 2016. source
44. H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, S. Savarese, "Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression," *arXiv preprint*, 2019. source
45. Z. Zheng, P. Wang, W. Liu, J. Li, R. Ye and D. Ren, "Distance-IoU loss: Faster and better learning for bounding box regression," in *Proc. AAAI*, pp. 12993–13000, 2020. doi
46. H. Zhang, M. Cisse, Y. N. Dauphin and D. Lopez-Paz, "mixup: Beyond empirical risk minimization," in *Proc. ICLR*, 2018. arXiv
47. S. Yun, D. Han, S. Chun, S. J. Oh, Y. Yoo, J. Choe, "CutMix: Regularization Strategy to Train Strong Classifiers With Localizable Features," *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019. doi ·manuscript
48. Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, 2014. source
49. G. Ghiasi, T. Lin, Q. V. Le, "DropBlock: A regularization method for convolutional networks," *NeurIPS*, 2018. source ·proceedings
50. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. doi ·manuscript
51. S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. ICML*, PMLR vol. 37, pp. 448–456, 2015. link
52. Z. Yao, Y. Cao, S. Zheng, G. Huang, S. Lin, "Cross-Iteration Batch Normalization," *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021. doi ·manuscript
53. S. Woo, J. Park, J. Lee, I. S. Kweon, "CBAM: Convolutional Block Attention Module," *Lecture Notes in Computer Science / Computer Vision – ECCV 2018*, 2018. doi ·manuscript
54. I. Loshchilov, F. Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts," *ICLR*, 2017. source
55. T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Proc. ECCV*, pp. 740–755, 2014. doi

56. COCO Consortium, “COCO detection evaluation,” official evaluation description. link
57. ONNX, “Open Neural Network Exchange,” official project page. link
58. B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Proc. IEEE/CVF CVPR*, pp. 2704–2713, 2018. doi
59. N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov and S. Zagoruyko, “End-to-end object detection with transformers,” in *Proc. ECCV*, 2020. doi
60. H. W. Kuhn, “The Hungarian method for the assignment problem,” *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955. doi
61. J. Terven, D.-M. Córdova-Esparza and J.-A. Romero-González, “A comprehensive review of YOLO architectures in computer vision: From YOLOv1 to YOLOv8 and YOLO-NAS,” *Machine Learning and Knowledge Extraction*, vol. 5, no. 4, pp. 1680–1716, 2023. doi ·arXiv
62. N. Jegham, C. Y. Koh, M. Abdelatti and A. Hendawi, “YOLO evolution: A comprehensive benchmark and architectural review of YOLOv12, YOLO11, and their previous versions,” arXiv:2411.00201, 2024 (revised 2025). arXiv
63. R. Sapkota and M. Karkee, “Ultralytics YOLO evolution: An overview of YOLO27, YOLO26, YOLO11, YOLOv8, and YOLOv5 object detectors for computer vision and pattern recognition,” arXiv:2510.09653, 2025 (revised 2026). arXiv
64. T. Shehzadi, K. A. Hashmi, M. Liwicki, D. Stricker and M. Z. Afzal, “Object detection with transformers: A review,” *Sensors*, vol. 25, no. 19, 2025. doi
65. Ultralytics, “YOLO27 (coming soon),” official documentation page, accessed 2 November 2026. docs
66. M. Oquab, T. Darcet, T. Moutakanni, H. Vo, M. Szafraniec et al., “DINOv2: Learning robust visual features without supervision,” arXiv:2304.07193, 2023. arXiv
67. O. Siméoni, H. V. Vo, M. Seitzer, F. Baldassarre, M. Oquab et al., “DINOv3,” arXiv:2508.10104, 2025. arXiv
68. X. Ding, X. Zhang, N. Ma, J. Han, G. Ding and J. Sun, “RepVGG: Making VGG-style ConvNets great again,” in *Proc. IEEE/CVF CVPR*, pp. 13728–13737, 2021. doi
69. R. Girshick, “Fast R-CNN,” *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015. doi ·manuscript
70. S. Ren, K. He, R. Girshick, J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” *arXiv preprint*, 2015. source
71. M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, A. Zisserman, “The Pascal Visual Object Classes (VOC) Challenge,” *International Journal of Computer Vision*, 2009. source
72. A. Shrivastava, A. Gupta, R. Girshick, “Training Region-Based Object Detectors with Online Hard Example Mining,” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. doi ·manuscript
73. H. Zhang, Y. Wang, F. Dayoub and N. Sünderhauf, “VarifocalNet: An IoU-aware dense object detector,” in *Proc. IEEE/CVF CVPR*, pp. 8510–8519, 2021. doi
74. YOLOX contributors, “YOLOX detection head,” official implementation, accessed 2 November 2026. code
75. K. Duan, S. Bai, L. Xie, H. Qi, Q. Huang, Q. Tian, “CenterNet: Keypoint Triplets for Object Detection,” *arXiv preprint*, 2019. source
76. Z. Ge, S. Liu, Z. Li, O. Yoshie and J. Sun, “OTA: Optimal transport assignment for object detec-

tion," in *Proc. IEEE/CVF CVPR*, pp. 303–312, 2021. doi
77. C. Li, L. Li, Y. Geng, et al., "YOLOv6 v3.0: A Full-Scale Reloading," *arXiv preprint*, 2023. source
78. C. Feng, Y. Zhong, Y. Gao, M. R. Scott, W. Huang, "TOOD: Task-Aligned One-Stage Object Detection," *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021. source
79. Z. Gevorgyan, "SIoU Loss: More Powerful Learning for Bounding Box Regression," *arXiv preprint*, 2022. source
80. X. Li, W. Wang, L. Wu, S. Chen, X. Hu, J. Li, J. Tang and J. Yang, "Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection," in *Advances in Neural Information Processing Systems 33 (NeurIPS)*, 2020. proceedings
81. X. Ding, H. Chen, X. Zhang, K. Huang, J. Han, G. Ding, "Re-parameterizing Your Optimizers rather than Architectures," *ICLR*, 2023. source
82. S. Zhang, C. Chi, Y. Yao, Z. Lei, S. Z. Li, "Bridging the Gap Between Anchor-Based and Anchor-Free Detection via Adaptive Training Sample Selection," *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. doi ·manuscript
83. C. Wang, H. M. Liao, I. Yeh, "Designing Network Design Strategies Through Gradient Path Analysis," *arXiv preprint*, 2022. source
84. T. Dao, D. Fu, S. Ermon, A. Rudra, C. Ré, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," *Advances in Neural Information Processing Systems 35*, 2022. doi ·manuscript
85. D. P. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization," *ICLR*, 2015. source
86. K. Jordan, "Muon: An optimizer for hidden layers in neural networks," technical blog, 2024. source
87. S. Shao, Z. Li, T. Zhang, C. Peng, G. Yu, X. Zhang, J. Li, J. Sun, "Objects365: A Large-Scale, High-Quality Dataset for Object Detection," *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019. source

# DLCV 03B • 3D Object Detection: LiDAR, Cameras and Bird's-Eye View

How machines find cars, people and cyclists in 3D: point clouds and coordinate frames, LiDAR detectors from PointNet to CenterPoint, camera-only bird's-eye-view methods, sensor fusion, rotated 3D boxes and IoU, and how KITTI, nuScenes and Waymo measure the results.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/3d-object-detection/

## What you'll learn

- Define 3D boxes, coordinate frames and sensor calibration.
- Compare point, voxel, pillar and hybrid representations.
- Understand camera-only depth inference and multi-sensor fusion.
- Encode orientation and distinguish BEV overlap from full 3D overlap.
- Interpret benchmark metrics and test range, weather, calibration and deployment cost.

## The big picture

A 2D detector draws rectangles on an image. A **3D detector** places boxes in the real world: where each object is in metres, how big it is, which way it faces and, for moving things, how fast it goes. A self-driving car does not brake for a rectangle of pixels; it brakes because a pedestrian is 14 metres ahead and walking into its lane.

> **Explain like I'm five**
>
> Close your eyes in a dark room and wave a torch that can only tell you how far away the wall is in each direction. After a while you have a cloud of dots that outline the furniture. A 3D detector looks at such a cloud of dots and says: "there is a sofa here, 2 metres long, turned a bit to the left". Dots on things far away are few and far between, so faraway sofas are hard. If you only have a photo instead of dots, you see the sofa clearly but have to guess how far away it is.

**Why it matters.** 3D detection is the core of the perception stack in autonomous driving, where it feeds tracking, motion prediction and planning [1]. Robots need it to grasp and avoid objects, and augmented reality needs it to anchor virtual content to real things. The task also forced new ideas that 2D vision never needed: networks for unordered point sets, sparse convolution, the bird's-eye-view representation, and ways to "lift" flat images into 3D.

This chapter assumes the 2D detection vocabulary of DLCV 03 ·Object Detection: box regression targets, IoU, anchors and label assignment, NMS and the focal loss (its core building blocks), centre-based detection (anchor-free detectors), DETR-style set prediction (set prediction with transformers) and average precision (evaluating detectors). We reuse those ideas and only explain what changes in 3D. Two surveys frame the field: Mao et al. [1] and Qian et al. [2]; we return to them in the Modern view.

### The 3D detection problem

**Plain version.** A 3D detector turns sensor observations into boxes, classes and scores. The upright-box setting models yaw while excluding roll and pitch.

**Precise version.** A 3D detector outputs a set

$$\hat{Y} = \{(\hat{\mathbf{b}}_i, \hat{c}_i, \hat{s}_i, \hat{\mathbf{v}}_i)\}_{i=1}^{N}, \qquad \hat{\mathbf{b}}_i = (x, y, z, l, w, h, \theta),$$

where $(x, y, z)$ is the box centre in metres, $(l, w, h)$ are its length, width and height, $\theta$ is the **yaw** (heading angle about the vertical axis), $\hat{c}_i$ is one of $K$ classes, $\hat{s}_i \in [0, 1]$ is a confidence, $\hat{\mathbf{v}}_i = (v_x, v_y)$ is an optional ground-plane velocity in metres per second, and $N$ varies from scene to scene. This 7-number box is called a **7-DoF** (degrees of freedom) box; the Waymo Open Dataset, for example, labels every object this way, with length, width and height measured along the box's own $x$, $y$ and $z$ axes [3]. Roll and pitch are assumed to be zero, which is reasonable for road users on a roughly flat road.

Compared with 2D detection, three things change:

1. **The box lives in metric space.** Sizes do not shrink with distance, so a car is about 4.5 m long whether it is 5 m or 80 m away. This makes size priors (anchors, class-average sizes) far more useful than in images.
2. **Orientation matters.** A 2D box is axis-aligned. A 3D box must turn with the object, and a wrong yaw can ruin the overlap even when the centre is perfect.
3. **The data are sparse and uneven.** A LiDAR sees surfaces, not volumes, mostly the sides that face it, and far objects get only a few points.

### Sensors and coordinate frames

**Plain version.** Each sensor measures in its own coordinates. To combine them, we move everything into one shared frame attached to the car, using the known position and orientation of each sensor, its **calibration**.

Three sensors dominate (Figure 1, left):

- **LiDAR** sends laser pulses and times their echoes. A spinning unit returns a **point cloud**: tens of thousands of points $(x, y, z)$ per sweep, each with an **intensity** (reflectance). Depth is measured directly and accurately, but there is no colour, and the points are sparse.
- **Cameras** give dense colour and texture at low cost but no depth: a pixel tells you the *direction* of a ray, not how far along the ray the object is.
- **Radar** returns a handful of points per object with range and, through the Doppler effect, radial velocity. It works in fog and rain, but its angular resolution is poor.

The nuScenes car, for example, carries 6 cameras, 5 radars and 1 LiDAR covering 360 degrees [4]; the Waymo car carries five LiDARs and five cameras [3].

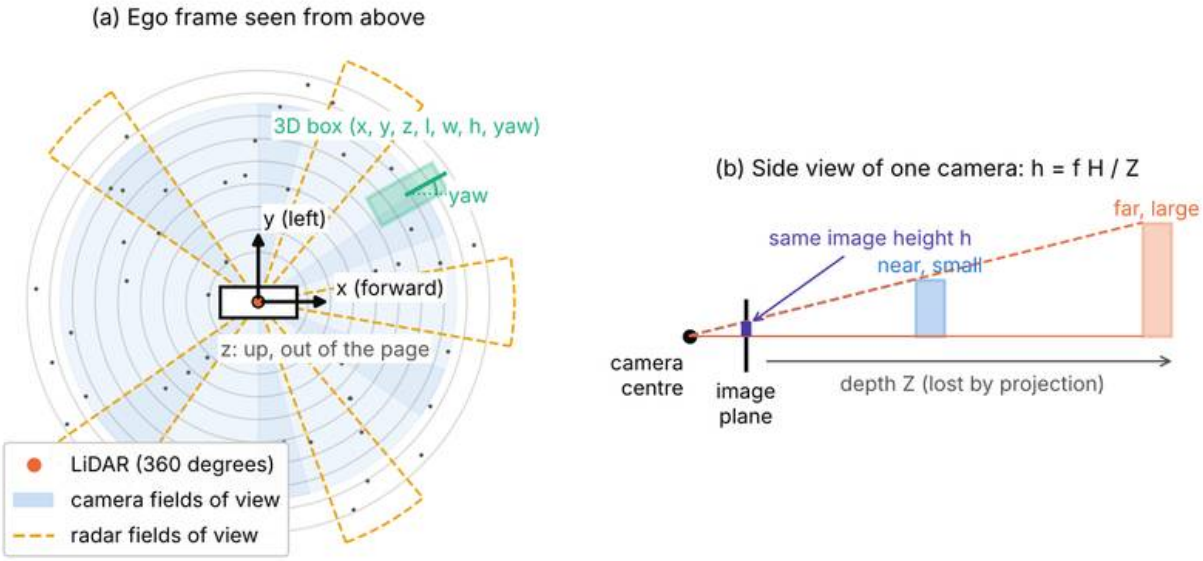



*Figure 1 — (a) The ego (vehicle) frame seen from above: x forward, y left, z up, with the fields of view of a LiDAR, six cameras and several radars, and one 7-DoF box. (b) Why a single camera loses depth: a near small object and a far large object project to the same image height.*

A convention used by the Waymo dataset, and by this chapter, is a right-handed **vehicle frame** with $x$ forward, $y$ to the left and $z$ up; each sensor has its own frame, and a 4×4 matrix called the **extrinsics** maps sensor coordinates into the vehicle frame [3]. In formulas, a point $\mathbf{p}_s$ measured by a sensor becomes

$$\mathbf{p}_v = R\,\mathbf{p}_s + \mathbf{t},$$

where $R$ is a 3×3 rotation matrix and $\mathbf{t}$ a 3-vector translation, together the sensor's extrinsics. A camera additionally has **intrinsics** $K$, a 3×3 matrix holding the focal lengths and the image centre in pixels. A point $\mathbf{p}_c = (X, Y, Z)$ in camera coordinates (with $Z$ pointing out of the lens) projects to the pixel

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z} K \begin{bmatrix} X \\ Y \\ Z \end{bmatrix},$$

where $(u, v)$ are pixel coordinates. The division by $Z$ is the heart of the matter: every point on the ray $\lambda(X, Y, Z)$, $\lambda > 0$, lands on the same pixel, so the depth is lost (Figure 1, right). Calibration matters in practice: if $R$ is off by half a degree, a point 60 m away moves by about half a metre, which is enough to paint a pedestrian's pixels onto the wall behind them.

### The bird's-eye view

**Plain version.** BEV represents road scenes from above, making object size and position easier to compare in metric space. Vertical overlap remains possible.

In the BEV we drop $z$ and keep a 2D grid over $(x, y)$, for example 0.16 m cells. Height becomes a feature channel instead of an axis. Because objects keep their metric size and do not occlude each other in this view, an ordinary 2D convolutional detector can run on the BEV grid, and the planner downstream also thinks in this map. The price is that height detail is compressed, which matters little for cars on a road and more for indoor scenes with tables and shelves.

### Point clouds: what the network actually sees

**Plain version.** Point clouds are unordered and spatially sparse. Distant objects usually receive fewer returns, so the representation must handle uneven sampling.

A LiDAR sweep is a set $P = \{\mathbf{p}_j\}_{j=1}^{M}$ with $\mathbf{p}_j = (x_j, y_j, z_j, r_j)$, where $r_j$ is the intensity and $M$ is the number of points, often tens of thousands. Three properties shape every network in this chapter:

1. **Unordered.** Any permutation of the list describes the same scene, so a network $f$ that consumes the points must be **permutation invariant**: $f(\mathbf{p}_1, \dots, \mathbf{p}_M) = f(\mathbf{p}_{\pi(1)}, \dots, \mathbf{p}_{\pi(M)})$ for every permutation $\pi$ of the indices.
2. **Sparse.** Points lie on surfaces. Divide the space around a car into 16 cm columns and more than 9 in 10 of them are empty (we measure this below).
3. **Range-dependent density.** A spinning LiDAR fires rays at fixed angular steps $\Delta\phi$ (horizontal) and $\Delta\epsilon$ (vertical). At range $r$, neighbouring rays are about $r\,\Delta\phi$ and $r\,\Delta\epsilon$ apart, so an object of fixed face area $A$ receives about

$$n(r) \approx \frac{A}{r^2\,\Delta\phi\,\Delta\epsilon}$$

points, where $n(r)$ is the expected number of points and the angles are in radians. Doubling the distance cuts the points by about four.

The snippet below builds a toy spinning LiDAR (32 beams, one ray every 0.2 degrees) and counts the rays that hit a car-sized box placed at increasing range. It uses the **slab method**: a ray hits a box if the intervals where it lies between each pair of parallel faces overlap.

```
import numpy as np

# A toy spinning LiDAR mounted 1.8 m above the ground: 32 beams between -16 and +10 degrees
# elevation, one ray every 0.2 degrees of azimuth (1800 rays per beam per turn).
elev = np.deg2rad(np.linspace(-16, 10, 32))
azim = np.deg2rad(np.arange(0, 360, 0.2))
E, A = np.meshgrid(elev, azim, indexing="ij")
dirs = np.stack([np.cos(E) * np.cos(A), np.cos(E) * np.sin(A), np.sin(E)], -1).reshape(-1, 3)
origin = np.array([0.0, 0.0, 1.8])

def hits_on_box(center, size):
    """Count rays whose first hit is an axis-aligned box (slab method)."""
    lo, hi = center - size / 2, center + size / 2
    with np.errstate(divide="ignore", invalid="ignore"):
        t1, t2 = (lo - origin) / dirs, (hi - origin) / dirs
    t_near = np.nanmax(np.minimum(t1, t2), axis=1)
```

```
    t_far = np.nanmin(np.maximum(t1, t2), axis=1)
    return int(np.sum((t_far >= t_near) & (t_far > 0)))

car = np.array([4.5, 1.8, 1.5])                      # length, width, height in metres
ranges = np.array([5, 10, 20, 40, 80])
counts = np.array([hits_on_box(np.array([r, 0, car[2] / 2]), car) for r in ranges])
for r, c in zip(ranges, counts):
    print(f"car at {r:2d} m: {c:5d} points")
slope = np.polyfit(np.log(ranges[1:]), np.log(counts[1:]), 1)[0]
print(f"log-log slope (10-80 m): {slope:.2f}")
```

```
car at  5 m:  2817 points
car at 10 m:   924 points
car at 20 m:   172 points
car at 40 m:    39 points
car at 80 m:     7 points
log-log slope (10-80 m): -2.33
```

The count drops from thousands of points at 5 m to 7 points at 80 m. The fitted slope is a bit steeper than $-2$ because only a handful of the 32 beams cross a 1.5 m tall object far away, so the vertical sampling becomes coarse; real sensors with more beams follow $1/r^2$ more closely. Figure 2 shows the same effect on a whole scene: identical cars, very different evidence.

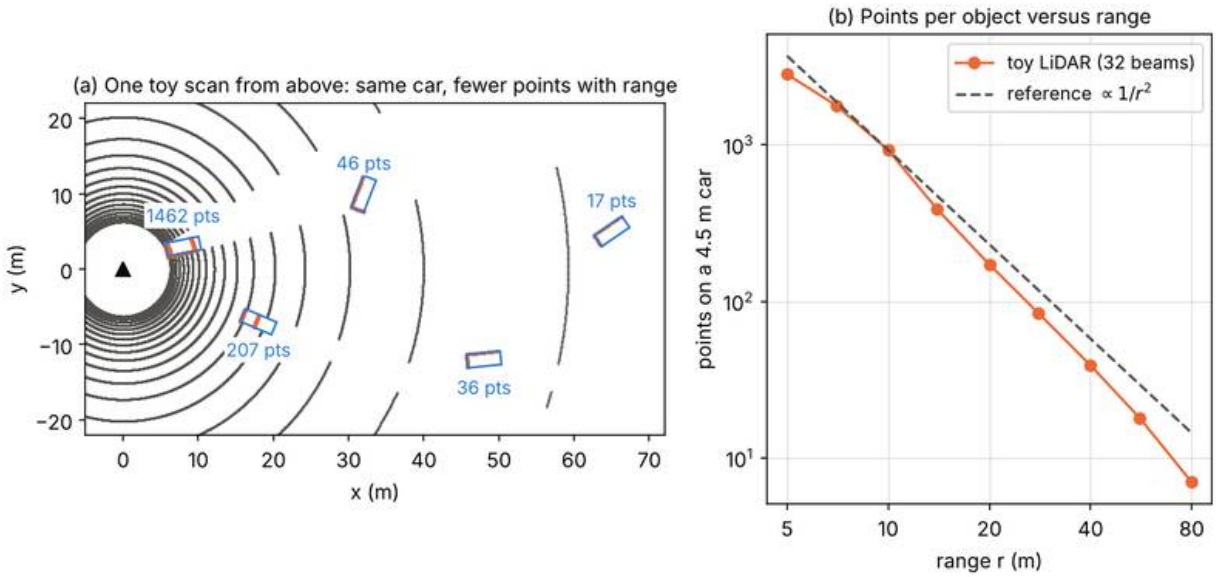



*Figure 2 — (a) A toy ray-cast LiDAR scan seen from above: five identical cars, labelled with the number of points that hit each. (b) Points on a 4.5 m car versus range, with a $1/r^2$ reference line.*

### A classical baseline: ground removal, clustering and box fitting

**Plain version.** Before deep learning, a common LiDAR pipeline was: remove the road, group the remaining points into blobs, and draw a tight rectangle around each blob. It needs no training data, and it tells you exactly which problems the networks later had to solve.

The three steps are:

1. **Ground removal with RANSAC** [5]. Fit a plane $\mathbf{n}^\top \mathbf{p} = d$ (unit normal $\mathbf{n}$, offset $d$) by repeatedly picking three random points, building the plane through them, and counting the **inliers**, points with $|\mathbf{n}^\top \mathbf{p} - d| < \tau$ for a tolerance $\tau$. The plane with the most inliers is the ground. (RANSAC is explained in more detail in the Supplementary notes of IP 12 ·Feature Extraction.)
2. **Clustering with DBSCAN** [6]. Group the remaining points in the BEV: two points are neighbours if they are closer than $\varepsilon$, a point with at least $m$ neighbours is a core point, and clusters grow from core points; isolated points are labelled noise. DBSCAN needs no number of clusters, which suits a scene with an unknown number of objects. (See PR 07 ·Clustering, Mixtures and EM for clustering in general.)
3. **Oriented box fitting.** For each cluster, try yaw angles $\theta$, rotate the points into the box frame, take the tight axis-aligned rectangle, and score it. Searching over angles with a fitting criterion is the idea of search-based L-shape fitting [7]. We use a simple criterion: the mean distance from each point to the nearest rectangle edge. A LiDAR usually sees an "L" (two sides of a car), and the right yaw puts those points on two edges.

```
import numpy as np
from sklearn.cluster import DBSCAN

rng = np.random.default_rng(0)

def box_surface(cx, cy, l, w, h, yaw, n):
    """n points on the sides of a yawed box that face the sensor at the origin."""
    c, s = np.cos(yaw), np.sin(yaw)
    pts = []
    for nx, ny, half, along in [(1, 0, l / 2, w), (-1, 0, l / 2, w), (0, 1, w / 2, l), (0,
     ↪ -1, w / 2, l)]:
        normal = np.array([c * nx - s * ny, s * nx + c * ny])
        mid = np.array([cx, cy]) + half * normal
        if normal @ mid >= 0:                    # face points away from the sensor: hidden
            continue
        t = rng.uniform(-along / 2, along / 2, n)
        tangent = np.array([-normal[1], normal[0]])
        xy = mid + t[:, None] * tangent
        pts.append(np.column_stack([xy, rng.uniform(0, h, n)]))
    return np.vstack(pts)

# ground plane z = 0 with slight noise, sampled uniformly in range and angle (density ~ 1/r)
r, a = rng.uniform(2, 60, 20000), rng.uniform(-np.pi / 2, np.pi / 2, 20000)
ground = np.column_stack([r * np.cos(a), r * np.sin(a), rng.normal(0, 0.03, r.size)])
objects = [(12, -4, 4.5, 1.8, 1.5, 0.3), (25, 6, 4.2, 1.7, 1.4, -1.0), (40, -10, 0.8, 0.6,
 ↪ 1.7, 0.0)]
cloud = [ground] + [box_surface(*o, n=int(40000 / np.hypot(o[0], o[1]) ** 2) + 5) for o in
 ↪ objects]
```

```
pts = np.vstack(cloud) + rng.normal(0, 0.02, (sum(len(c) for c in cloud), 3))

# 1) RANSAC plane fit: pick 3 points, count inliers within 0.1 m, keep the best plane
best = None
for _ in range(100):
    p = pts[rng.choice(len(pts), 3, replace=False)]
    n = np.cross(p[1] - p[0], p[2] - p[0])
    if np.linalg.norm(n) < 1e-9:
        continue
    n /= np.linalg.norm(n)
    inl = np.abs((pts - p[0]) @ n) < 0.1
    if best is None or inl.sum() > best.sum():
        best = inl
above = pts[~best]
print(f"{len(pts)} points, {best.sum()} ground inliers, {len(above)} left")

# 2) DBSCAN in the bird's-eye view
labels = DBSCAN(eps=0.7, min_samples=5).fit_predict(above[:, :2])

# 3) oriented box per cluster: search the yaw; score = mean distance of points to the nearest
 ↪  edge
for k in range(labels.max() + 1):
    q = above[labels == k]
    best_score, best_fit = np.inf, None
    for yaw in np.deg2rad(np.arange(0, 180, 1)):
        R = np.array([[np.cos(yaw), -np.sin(yaw)], [np.sin(yaw), np.cos(yaw)]])
        uv = q[:, :2] @ R                              # coordinates along the box axes
        lo, hi = uv.min(0), uv.max(0)
        edge_dist = np.minimum(np.minimum(uv - lo, hi - uv)[:, 0], np.minimum(uv - lo, hi -
         ↪  uv)[:, 1])
        if edge_dist.mean() < best_score and hi[0] - lo[0] >= hi[1] - lo[1]:   # x axis =
         ↪  long side
            best_score, best_fit = edge_dist.mean(), (R @ ((lo + hi) / 2), hi - lo,
             ↪  np.rad2deg(yaw))
    ctr, size, yaw = best_fit
    print(f"cluster {k}: {len(q):4d} pts, centre ({ctr[0]:5.1f}, {ctr[1]:5.1f}), "
          f"size {size[0]:.1f} x {size[1]:.1f} m, yaw {yaw:4.0f} deg, height {q[:,
           ↪  2].max():.1f} m")
```

```
20694 points, 19944 ground inliers, 750 left
cluster 0:  471 pts, centre ( 12.0,  -4.0), size 4.6 x 1.9 m, yaw   17 deg, height 1.5 m
cluster 1:  124 pts, centre ( 25.0,   6.1), size 4.1 x 1.9 m, yaw  123 deg, height 1.4 m
cluster 2:   53 pts, centre ( 40.0,  -9.9), size 0.8 x 0.6 m, yaw    0 deg, height 1.7 m
```

On this synthetic scene (true objects: a car at $(12, -4)$ with yaw 17°, a car at $(25, 6)$ with yaw $-57°$, which is 123° modulo 180°, and a $0.8 \times 0.6$ m pedestrian at $(40, -10)$) the pipeline recovers centres, sizes and yaws well. Figure 3 shows the result. A tempting simpler criterion, the smallest rectangle area, fails here: the hull of an L-shaped point set is roughly a triangle, and a rectangle along the triangle's long diagonal encloses it with the same area, so the area criterion can pick a yaw about 20° off.

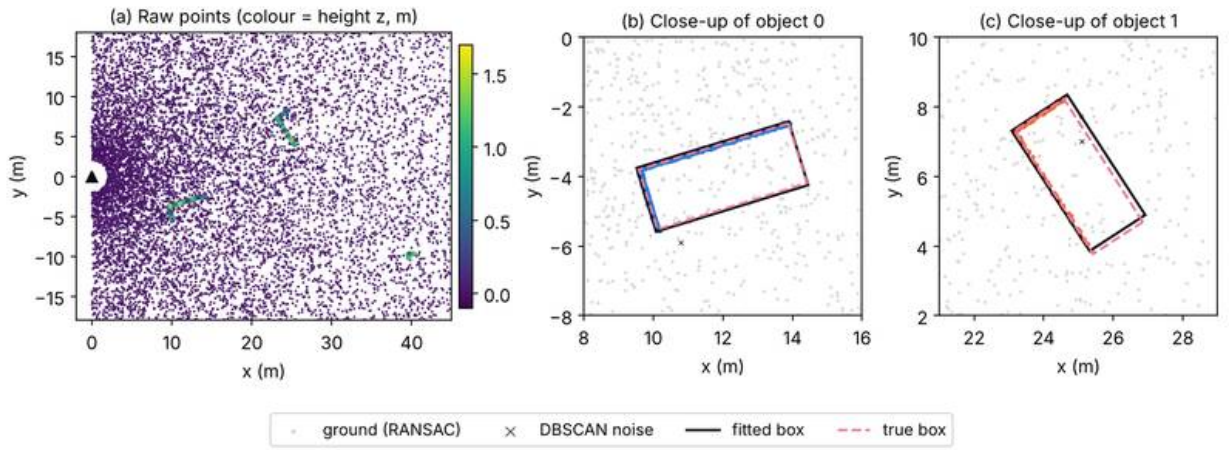



*Figure 3 — The classical pipeline on a synthetic scene. (a) Raw points coloured by height. (b, c) After RANSAC ground removal and DBSCAN clustering: object points (colour), the fitted box (black) and the true box (dashed). The LiDAR sees only the two sides facing it.*

The baseline also shows its limits, and each one is a reason for learning:

- **No semantics.** A cluster is "something". Telling a pedestrian from a pole, or a car from a bush, needs learned features.
- **Partial views.** The box is fitted to visible points only. A far car seen from the back gives a short line, so its length is unknown. Learned detectors use class-level size priors to complete what is not seen.
- **Merging and splitting.** Two people walking together merge into one cluster; a truck seen through a gap splits into two.
- **Heading.** The fit gives yaw only modulo 180°: it cannot tell the front of a car from its back.

### 3D boxes: encoding, IoU and NMS

These are the 3D versions of the building blocks of DLCV 03. We explain only what is new.

### Box encoding and the yaw problem

**Plain version.** As in 2D, a network predicts small corrections to a reference box (an anchor or a point). The new part is the angle: angles wrap around, and a box turned by 180° covers exactly the same space, so a naive angle loss punishes the network for answers that are geometrically correct.

VoxelNet [8], SECOND [9] and PointPillars [10] use the same residuals between a ground-truth box (superscript $g$) and an anchor (superscript $a$):

$$\Delta x = \frac{x^g - x^a}{d^a}, \quad \Delta y = \frac{y^g - y^a}{d^a}, \quad \Delta z = \frac{z^g - z^a}{h^a}, \quad \Delta l = \log \frac{l^g}{l^a}, \quad \Delta w = \log \frac{w^g}{w^a}, \quad \Delta h = \log \frac{h^g}{h^a},$$

where $d^a = \sqrt{(l^a)^2 + (w^a)^2}$ is the diagonal of the anchor's base, which normalizes horizontal offsets in both directions by one length. These are the 2D targets of DLCV 03 with one more dimension.

For the angle, VoxelNet regressed the raw difference $\theta^g - \theta^a$. SECOND [9] pointed out that this creates a conflict: a prediction off by exactly $\pi$ gives the same box but a large loss. Its **sine-error loss**

$$L_\theta = \text{SmoothL1}(\sin(\theta^p - \theta^t))$$

(with $\theta^p$ the predicted and $\theta^t$ the target angle, and SmoothL1 the smooth L1 loss of DLCV 03: quadratic for errors below 1, linear above) is zero both for a perfect prediction and for one flipped by $\pi$. Because it cannot tell front from back, SECOND adds a small **direction classifier** with two bins that decides which way the box points [9]; PointPillars uses the same pair of losses [10]. CenterPoint instead regresses $(\sin\theta, \cos\theta)$ and recovers the angle with $\theta = \text{atan2}(\sin\theta, \cos\theta)$, which is continuous and unambiguous [11].

*Online only: Yaw encodings compared: raw angle, sine error and (sin, cos) — https://blog.shyandram.dev/en/learn/deep-learning/3d-object-detection/notes/yaw-encodings-compared-raw-angle-sine-error-and-sin-cos/*

### BEV IoU and 3D IoU of rotated boxes

**Plain version.** IoU still means "overlap divided by total area", but the boxes are now rotated, so the overlap is a polygon, not a rectangle. In 3D, multiply that polygon's area by the overlap of the two height intervals.

For two upright boxes $A$ and $B$, let $A_{\text{bev}} \cap B_{\text{bev}}$ be the intersection of their BEV rectangles (a convex polygon with up to 8 corners) and let

$$o_z = \max\Big(0,\ \min\left(z_A + \tfrac{h_A}{2}, z_B + \tfrac{h_B}{2}\right) - \max\left(z_A - \tfrac{h_A}{2}, z_B - \tfrac{h_B}{2}\right)\Big)$$

be the overlap of their vertical extents, where $z$ is a box's centre height and $h$ its height. Then

$$\text{IoU}_{\text{bev}} = \frac{|A_{\text{bev}} \cap B_{\text{bev}}|}{l_A w_A + l_B w_B - |A_{\text{bev}} \cap B_{\text{bev}}|}, \qquad \text{IoU}_{\text{3D}} = \frac{|A_{\text{bev}} \cap B_{\text{bev}}|\, o_z}{l_A w_A h_A + l_B w_B h_B - |A_{\text{bev}} \cap B_{\text{bev}}|\, o_z},$$

where $|\cdot|$ is area and $l, w, h$ are the box sizes. The intersection polygon of two convex polygons can be computed with **Sutherland–Hodgman clipping** [12]: cut the first polygon by each edge line of the second in turn, keeping only the part on the inner side. The polygon's area then follows from the **shoelace formula** $\frac{1}{2}|\sum_k (x_k y_{k+1} - x_{k+1} y_k)|$ over its corners $(x_k, y_k)$, with indices taken cyclically.

```
import numpy as np

def corners(box):
    """BEV corners (counter-clockwise) of box = (x, y, z, l, w, h, yaw); z is the box
    ↪ centre."""
    x, y, _, l, w, _, yaw = box
```

```
    c, s = np.cos(yaw), np.sin(yaw)
    local = np.array([[l, w], [-l, w], [-l, -w], [l, -w]]) / 2
    return local @ np.array([[c, s], [-s, c]]) + [x, y]

def clip(subject, clipper):
    """Sutherland-Hodgman: keep the part of polygon `subject` inside convex CCW polygon
     ↪  `clipper`."""
    out = list(subject)
    for i in range(len(clipper)):
        a, b = clipper[i], clipper[(i + 1) % len(clipper)]
        inside = lambda p: (b[0] - a[0]) * (p[1] - a[1]) - (b[1] - a[1]) * (p[0] - a[0]) >= 0
        pts, out = out, []
        for j in range(len(pts)):
            p, q = pts[j], pts[(j + 1) % len(pts)]
            if inside(q):
                if not inside(p):
                    out.append(cross_point(p, q, a, b))
                out.append(q)
            elif inside(p):
                out.append(cross_point(p, q, a, b))
        if not out:
            break
    return np.array(out)

def cross_point(p, q, a, b):
    """Intersection of segment p-q with the infinite line through a and b."""
    d1, d2 = q - p, b - a
    t = ((a[0] - p[0]) * d2[1] - (a[1] - p[1]) * d2[0]) / (d1[0] * d2[1] - d1[1] * d2[0])
    return p + t * d1

def area(poly):                                  # shoelace formula
    if len(poly) < 3:
        return 0.0
    x, y = poly[:, 0], poly[:, 1]
    return 0.5 * abs(x @ np.roll(y, -1) - y @ np.roll(x, -1))

def iou_bev_3d(a, b):
    inter = area(clip(corners(a), corners(b)))
    bev = inter / (a[3] * a[4] + b[3] * b[4] - inter)
    dz = max(0.0, min(a[2] + a[5] / 2, b[2] + b[5] / 2) - max(a[2] - a[5] / 2, b[2] - b[5] /
     ↪  2))
    inter3 = inter * dz
    return bev, inter3 / (a[3] * a[4] * a[5] + b[3] * b[4] * b[5] - inter3)

car = (0, 0, 0, 4, 2, 1.5, 0)
cases = {"identical":                 (car, car),
         "rotated by 180 deg":        (car, (0, 0, 0, 4, 2, 1.5, np.pi)),
         "rotated by 90 deg":         (car, (0, 0, 0, 4, 2, 1.5, np.pi / 2)),
         "unit square vs 45 deg":     ((0, 0, 0, 1, 1, 1, 0), (0, 0, 0, 1, 1, 1, np.pi / 4)),
         "shifted 1 m along length":  (car, (1, 0, 0, 4, 2, 1.5, 0)),
         "same BEV, raised 0.5 m":    (car, (0, 0, 0.5, 4, 2, 1.5, 0)),
         "far apart":                 (car, (10, 0, 0, 4, 2, 1.5, 0.3))}
for name, (a, b) in cases.items():
    bev, iou3 = iou_bev_3d(np.array(a, float), np.array(b, float))
    print(f"{name:26s} BEV IoU {bev:.4f}   3D IoU {iou3:.4f}")
```

```
identical                  BEV IoU 1.0000   3D IoU 1.0000
rotated by 180 deg         BEV IoU 1.0000   3D IoU 1.0000
rotated by 90 deg          BEV IoU 0.3333   3D IoU 0.3333
unit square vs 45 deg      BEV IoU 0.7071   3D IoU 0.7071
shifted 1 m along length   BEV IoU 0.6000   3D IoU 0.6000
same BEV, raised 0.5 m     BEV IoU 1.0000   3D IoU 0.5000
far apart                  BEV IoU 0.0000   3D IoU 0.0000
```

Every value matches a hand calculation: a $4 \times 2$ box against itself rotated by $90°$ overlaps in a $2 \times 2$ square, so IoU $= 4/(8 + 8 - 4) = 1/3$; a unit square against itself rotated by $45°$ overlaps in a regular octagon of area $2(\sqrt{2} - 1)$, which gives $\sqrt{2}/2 \approx 0.7071$; a box raised by a third of its height keeps the full BEV overlap but only half of the 3D one. Note the second line: a box turned by $180°$ has IoU 1. IoU is blind to the heading direction, which is why the Waymo metric adds a heading term (see below).

Figure 4 (right) shows how harsh yaw errors are for elongated objects. With a perfect centre and size, a car-sized box falls below IoU 0.7 at a yaw error of about $17°$, and a bus-sized box at about $8.5°$, while a square pedestrian box never drops below 0.7.

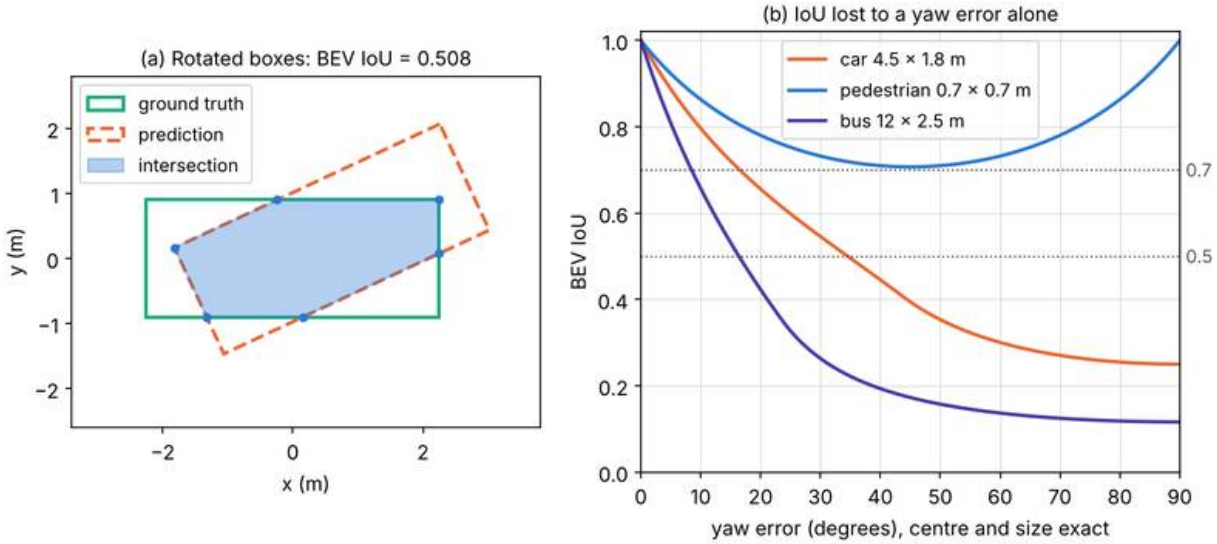



*Figure 4 — (a) Two rotated boxes; their intersection is a polygon found by clipping. (b) BEV IoU lost to a yaw error alone, for three object shapes with exact centre and size.*

**Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

### Rotated NMS and its alternatives

**Plain version.** Removing duplicate boxes works as in 2D, but "overlap" must now be measured between rotated boxes, which is slower. Two shortcuts are common: approximate the overlap with axis-aligned boxes, or avoid NMS altogether with centre peaks or set prediction.

Greedy NMS in 3D sorts boxes by score and removes those whose BEV IoU with a kept box exceeds a threshold, using the rotated IoU above. Because objects on a road rarely overlap in the BEV, the threshold can be low. PointPillars found that NMS on axis-aligned BEV boxes with threshold 0.5 gave similar results to rotated NMS and was much faster [10]. CenterPoint reads objects off the local maxima of a centre heatmap, like CenterNet [13], which removes most duplicates by construction; the paper still applies NMS to the peaks before its second stage [11]. Camera methods in the DETR family [14], such as DETR3D, predict a set of boxes and need no NMS at all [15].

## LiDAR-based detectors

**Plain version.** LiDAR detectors differ mainly in how they turn an unordered point set into something a network can process: keep the points (point-based), drop them into a 3D grid (voxels), drop them into vertical columns (pillars), or mix the two. After that, most use a 2D detection head on the BEV.

### Point-based: PointNet, PointNet++ and PointRCNN

**Plain version.** Process every point with the same small network, then summarize all points with a maximum. A maximum does not care about the order of its inputs, so the result is permutation invariant.

**PointNet** [16] computes

$$f(\mathbf{p}_1, \dots, \mathbf{p}_M) = \gamma\Big( \max_{j=1,\dots,M} h(\mathbf{p}_j) \Big),$$

where $h$ is a shared multilayer perceptron applied to each point, max is taken element-wise over the points (a symmetric function, so the order does not matter), and $\gamma$ is a second network that maps the pooled feature to the output. It sees each point alone and then the whole set at once, so it misses local structure. **PointNet++** [17] fixes this with **set abstraction** layers: choose a subset of centre points by **farthest point sampling** (each new centre is the point farthest from those already chosen), gather the neighbours within a radius of each centre (a **ball query**), and run a small PointNet on each neighbourhood. Stacking these layers gives a hierarchy of local features, like a CNN's growing receptive fields. Both are mainly backbones: they produce features, and a detection head is built on top.

**PointRCNN** [18] is a two-stage detector that works directly on points. Stage 1 segments the scene into foreground and background points with a PointNet++ backbone and proposes a 3D box from each foreground point, bottom-up. Stage 2 pools the points inside each proposal, transforms them

into the proposal's own **canonical coordinates** (centred and rotated so the box is axis-aligned), and refines the box. Point-based detectors are accurate on small objects, but sampling and neighbour search on large clouds are slow.

### Voxel-based: VoxelNet and SECOND

**Plain version.** Divide space into small 3D cells (voxels), summarize the points in each cell with a tiny PointNet, and then run convolutions on the resulting 3D grid. Because almost all voxels are empty, compute only where there are points.

**VoxelNet** [8] groups points by voxel and augments each point with its offset from the voxel's centroid, giving a 7-D input $(x, y, z, r, x - v_x, y - v_y, z - v_z)$, where $(v_x, v_y, v_z)$ is the mean of the points in the voxel. A stack of **voxel feature encoding** (VFE) layers, each a shared fully connected layer followed by element-wise max pooling within the voxel, turns each voxel into one feature vector. 3D convolutions then process the grid, and a region proposal network in the style of Faster R-CNN [19] predicts boxes on the BEV feature map. It was end-to-end and strong, but dense 3D convolutions made it slow.

**SECOND** [9] replaced the dense 3D convolutions by **sparse convolutions**, which store and compute only the occupied voxels. It uses mostly **submanifold** sparse convolutions [20], whose outputs are active only where the inputs are active, so the set of occupied sites does not grow layer after layer. SECOND also introduced the sine-error angle loss and ground-truth sampling augmentation (both described in this chapter), and sparse voxel backbones of this kind were adopted by later detectors such as PV-RCNN [21].

*Online only: Sparse and submanifold convolution — https://blog.shyandram.dev/en/learn/deep-learning/3d-object-detection/notes/sparse-and-submanifold-convolution/*

### Pillar-based: PointPillars

**Plain version.** Make the cells infinitely tall. Each vertical column (a **pillar**) is summarized by a tiny PointNet, and the result is a flat "pseudo-image" that an ordinary 2D CNN can process quickly.

**PointPillars** [10] has three stages: a pillar feature network, a 2D convolutional backbone and an SSD-style detection head [22]. Each point in a pillar is decorated to $D = 9$ dimensions: $(x, y, z, r)$, its offsets $(x_c, y_c, z_c)$ from the mean of the points in the pillar, and its offsets $(x_p, y_p)$ from the pillar's centre in the $x$–$y$ plane. The non-empty pillars are packed into a dense tensor of shape $(D, P, N)$, with at most $P$ pillars per sample and $N$ points per pillar (12,000 and 100 in the paper, at a 0.16 m grid), zero-padded or randomly subsampled. A linear layer, batch normalization and ReLU are applied to every point, a max over the $N$ points gives one $C$-dimensional feature per pillar, and the features are **scattered** back to their grid positions to form a $C \times H \times W$ pseudo-image. The paper reports 62 Hz for its full pipeline, and a faster variant at 105 Hz [10].

*Online only: PointPillars: a reading guide — https://blog.shyandram.dev/en/learn/deep-learning/3d-object-detection/papers/pointpillars-a-reading-guide/*

The snippet performs the input side of this pipeline on a toy cloud: binning, the 9-D decoration, the dense $(P, N, D)$ tensor and the scatter (the PointNet weights are random here; in a real model they are learned).

```
import numpy as np

rng = np.random.default_rng(0)
# toy cloud: ground sampled uniformly in range and angle (so density falls as 1/r) + two
 ↪  "cars"
r, a = rng.uniform(2, 70, 10000), rng.uniform(-0.5, 0.5, 10000)
ground = np.column_stack([r * np.cos(a), r * np.sin(a), rng.normal(-1.7, 0.03, r.size)])
cars = [c + rng.uniform([-2.2, -0.9, -1.7], [2.2, 0.9, -0.2], (n, 3))
        for c, n in [(np.array([10.0, 2.0, 0.0]), 3000), (np.array([35.0, -6.0, 0.0]), 60)]]
xyz = np.vstack([ground] + cars)
pts = np.column_stack([xyz, rng.uniform(0, 1, len(xyz))])          # x, y, z, reflectance

# 1) bin into a bird's-eye-view grid of 0.16 m pillars over x in [0, 70.4), y in [-40, 40)
vx, x0, y0, W, H = 0.16, 0.0, -40.0, 440, 500
ix = np.floor((pts[:, 0] - x0) / vx).astype(int)
iy = np.floor((pts[:, 1] - y0) / vx).astype(int)
ok = (ix >= 0) & (ix < W) & (iy >= 0) & (iy < H)
pts, ix, iy = pts[ok], ix[ok], iy[ok]
pillar_id, inv, counts = np.unique(iy * W + ix, return_inverse=True, return_counts=True)
print(f"{len(pts)} points -> {len(pillar_id)} non-empty pillars out of {W * H} "
      f"({100 * (1 - len(pillar_id) / (W * H)):.1f}% empty); max {counts.max()} points in one
       ↪  pillar")

# 2) per-point 9-D features: x, y, z, r, offsets to the pillar mean, offsets to the pillar
 ↪  centre
mean = np.zeros((len(pillar_id), 3))
np.add.at(mean, inv, pts[:, :3])
mean /= counts[:, None]
centre = np.column_stack([(ix + 0.5) * vx + x0, (iy + 0.5) * vx + y0])
feat = np.column_stack([pts, pts[:, :3] - mean[inv], pts[:, :2] - centre])

# 3) dense tensor (P pillars, N points, D=9): pad with zeros, keep at most N points per
 ↪  pillar
P, N = len(pillar_id), 16          # PointPillars caps P at 12000 and N at 100, sampling at
 ↪  random
order = np.argsort(inv, kind="stable")
slot = np.arange(len(inv)) - np.searchsorted(inv[order], inv[order])   # rank inside its
 ↪  pillar
keep = slot < N
dense = np.zeros((P, N, 9))
dense[inv[order][keep], slot[keep]] = feat[order][keep]
print("dense pillar tensor:", dense.shape, "| points dropped by the cap N:",
 ↪  int((~keep).sum()))

# 4) a one-layer 'PointNet' (random weights here): linear + ReLU per point, max over points,
 ↪  scatter
Wt = rng.normal(0, 0.3, (9, 64))
pillar_feat = np.maximum(dense @ Wt, 0).max(axis=1)                    # (P, 64)
canvas = np.zeros((64, H * W))
canvas[:, pillar_id] = pillar_feat.T
```

```
print("pseudo-image:", canvas.reshape(64, H, W).shape)
```

```
13060 points -> 9481 non-empty pillars out of 220000 (95.7% empty); max 24 points in one
↪  pillar
dense pillar tensor: (9481, 16, 9) | points dropped by the cap N: 23
pseudo-image: (64, 500, 440)
```

More than 95% of the 220,000 cells are empty in this toy, close to the roughly 97% sparsity the PointPillars paper reports for a real 64-beam sweep at the same grid size [10]. This is why pillarization and scattering, not the convolutions, decide how much memory the input step needs. Figure 5 shows the grid on a ray-cast scan.

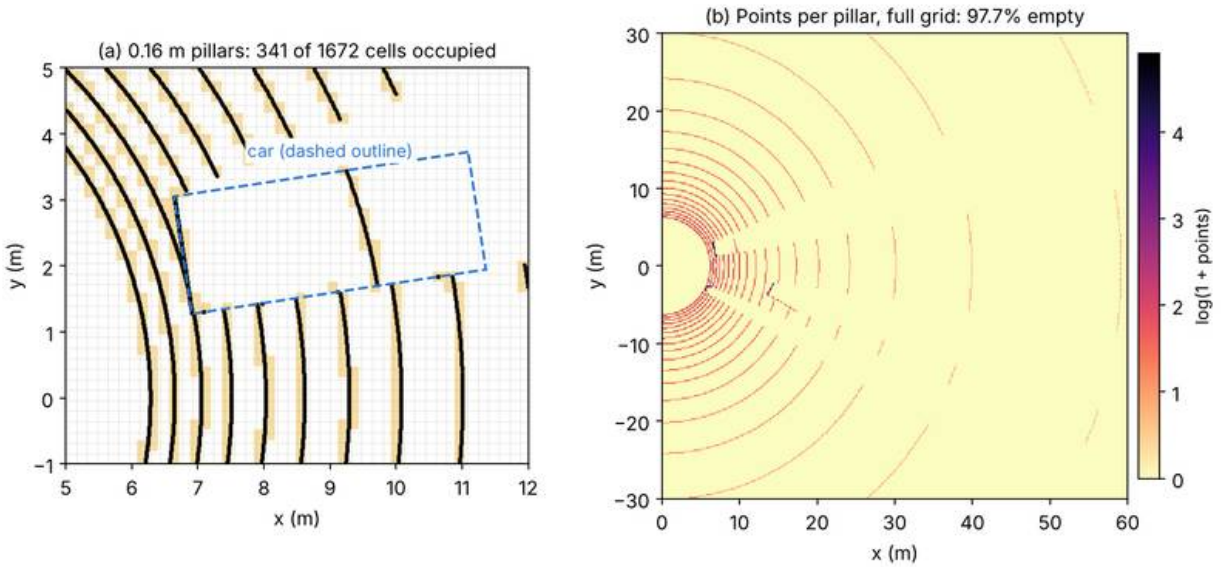



*Figure 5 — Pillarization of a toy ray-cast scan. (a) Occupied 0.16 m pillars (shaded) in a close-up. (b) Points per pillar over the whole grid: 97.7% of the pillars are empty, and occupied ones thin out with range.*

### Centre-based heads: CenterPoint

**Plain version.** Instead of anchors at several orientations, find object *centres* as peaks in a BEV heatmap and read every other property from the feature at the peak. A centre has no orientation, so the rotation problem of anchors disappears.

**CenterPoint** [11] keeps a standard voxel or pillar backbone and replaces the anchor head by a centre head in the spirit of CenterNet [13]. The heatmap has one channel per class and is trained with a focal loss [23] against Gaussian peaks rendered at the projected box centres; because objects are small in the BEV, the Gaussian radius is enlarged to at least 2 cells. At each centre, regression heads predict a sub-voxel offset, the height above ground, the 3D size, the yaw as $(\sin\theta, \cos\theta)$ and a 2D velocity. The velocity comes from feeding two consecutive sweeps, and it turns tracking into

greedy closest-point matching of centres between frames. A light second stage samples features at the centres of the box faces and predicts a refined box and a confidence trained toward the box's 3D IoU with the ground truth. Centre-based heads spread beyond LiDAR: the camera detector BEVDet, for example, uses CenterPoint's head [24].

### Point–voxel hybrids: PV-RCNN

**Plain version.** Voxels are fast and give good proposals; points keep exact positions. Use voxels to propose and points to refine.

**PV-RCNN** [21] runs a sparse 3D voxel CNN to get features and proposals, then summarizes the multi-scale voxel features into a small set of **keypoints** (voxel set abstraction). For each proposal, **RoI-grid pooling** gathers keypoint features at a regular grid of points inside the box, with several radii, and a refinement head predicts the final box and confidence. Two-stage hybrids of this kind were among the most accurate LiDAR detectors before transformer backbones.

### Transformers and sequence models on points

**Plain version.** A car covers only a tiny part of a 150 m scene, so shrinking the feature map, as 2D networks do, throws away detail. Attention lets features interact over long distances without shrinking, if it is restricted to the occupied voxels.

**SST** [25] observed that downsampling brings little benefit in LiDAR detection and kept a single stride throughout, using attention among nearby occupied voxels to regain the receptive field that downsampling would have provided; the paper reports that this helps most on small objects such as pedestrians. **DSVT** [26] reorganized this attention so that it runs fully in parallel on a GPU and is easy to deploy. **LION** [27] and **Voxel Mamba** [28] replace attention by linear-time sequence models (a linear recurrent network and a state space model) that read the voxels as long sequences. The one thing to remember: these backbones buy long-range context at a cost that grows with the number of occupied voxels, not with the size of the scene.

*Online only: Attention on sparse voxels: windows, sets and sequences — https://blog.shyandram.dev/en/learn/deep-learning/3d-object-detection/notes/attention-on-sparse-voxels-windows-sets-and-sequences/*

## Camera-only 3D detection

**Plain version.** Cameras are cheap and see colour, texture and signs, but a photo is flat. Every camera detector must answer the same question: how far away is each thing it sees? This section first shows why that is hard, then the two main ways to build a 3D map from images: push pixels out into 3D, or pull image features into 3D positions.

### Why one camera is not enough: depth ambiguity

**Plain version.** A photo of a toy car close up and a real car far away can look identical. To get depth from one image, a network must rely on clues such as the object's typical size, where it touches the road and perspective, and small errors in those clues become large errors in metres far away.

For a pinhole camera with focal length $f$ (in pixels), an object of real height $H$ at depth $Z$ appears $h = fH/Z$ pixels tall, so $Z = fH/h$. Differentiating gives

$$\left|\frac{\partial Z}{\partial h}\right| = \frac{fH}{h^2} = \frac{Z^2}{fH},$$

so a one-pixel measurement error costs a depth error that grows with the square of the distance, and an error in the assumed size $H$ scales the depth by the same factor.

```
import numpy as np

f, H = 1000.0, 1.5                      # focal length in pixels, true car height in metres
for Z in [10, 20, 40, 80]:
    h = f * H / Z                       # image height of the car in pixels
    z_pix = f * H / (h - 1)             # depth if we mis-measure the height by one pixel
    z_size = f * (1.1 * H) / h          # depth if we assume a car 10% taller than it is
    print(f"Z = {Z:2d} m: car is {h:6.2f} px tall | 1 px error -> {z_pix - Z:5.2f} m | "
          f"10% size error -> {z_size - Z:4.1f} m")
```

```
Z = 10 m: car is 150.00 px tall | 1 px error ->  0.07 m | 10% size error ->  1.0 m
Z = 20 m: car is  75.00 px tall | 1 px error ->  0.27 m | 10% size error ->  2.0 m
Z = 40 m: car is  37.50 px tall | 1 px error ->  1.10 m | 10% size error ->  4.0 m
Z = 80 m: car is  18.75 px tall | 1 px error ->  4.51 m | 10% size error ->  8.0 m
```

At 80 m a single pixel already moves the car by about 4.5 m, far outside a 3D IoU of 0.7. This is why monocular 3D detectors trail LiDAR so far on far objects. Monocular methods such as **FCOS3D** [29] adapt a 2D one-stage detector (FCOS [30]) to regress depth, size and orientation at the projected 3D centre. **Pseudo-LiDAR** [31] made an influential point: much of the gap between image-based and LiDAR-based detectors came from the *representation*, not only from depth quality; turning an estimated depth map into a 3D point cloud and running a LiDAR detector on it worked far better than processing the depth map as an image.

### Lifting images into the bird's-eye view

**Plain version.** Each pixel does not know its depth, so give it a probability distribution over depths, place a copy of its feature at every possible depth weighted by that probability, and then drop all those 3D features onto the BEV grid. The result can be fused across cameras and processed like a LiDAR BEV map.

**Lift-Splat-Shoot** (LSS) [32] "lifts" each image into a frustum of features. At each pixel, the network predicts a context vector $\mathbf{c} \in \mathbb{R}^C$ and a distribution $\boldsymbol{\alpha}$ over a set $\mathcal{D}$ of discrete depths ($\alpha_d \geq 0$, $\sum_{d \in \mathcal{D}} \alpha_d = 1$). The feature placed at depth $d$ along the pixel's ray is

$$\mathbf{c}_d = \alpha_d \, \mathbf{c},$$

an outer product of depth probabilities and context. A confident depth puts the feature at one place; an uncertain depth spreads it along the ray. The "splat" step transforms all frustum points into the vehicle frame with the camera extrinsics and intrinsics and sum-pools them into BEV pillars, using a cumulative-sum trick to make the pooling fast [32]. The paper evaluated BEV segmentation and planning; **BEVDet** [24] turned the same view transform into a 3D detector with a BEV encoder and a centre-based head, plus data augmentation applied in both image and BEV space. **BEVDepth** [33] found that the depth distribution learned only from the detection loss was poor, and supervised it explicitly with depth from LiDAR points projected into the images, together with a depth network aware of the camera parameters.

*Online only: Lift-Splat-Shoot: a reading guide — https://blog.shyandram.dev/en/learn/deep-learning/3d-object-detection/papers/lift-splat-shoot-a-reading-guide/*

### Querying images from 3D: DETR3D, PETR and BEVFormer

**Plain version.** Go the other way round. Start from 3D guesses, either object queries or a grid of BEV cells, project them into the images with the known camera matrices, and fetch the image features they land on. No depth map is needed.

**DETR3D** [15] decodes each object query to a 3D reference point, projects it into all cameras, samples the image features there by bilinear interpolation and uses them to refine the query; training uses DETR's set-to-set loss [14], so no NMS is needed. **PETR** [34] instead encodes 3D position into the image features themselves: the camera frustum is discretized into points at several depths, transformed to 3D with the camera parameters, and passed through a small network to produce a **3D position embedding** that is added to the image features; ordinary DETR-style queries then attend to these position-aware features. **BEVFormer** [35] keeps a grid of learnable **BEV queries**, one per BEV cell. Spatial cross-attention lets each BEV query sample features in the image regions where its location projects, using deformable attention [36], and temporal self-attention lets it look at the BEV features of the previous frame, which helps with velocity and with objects that are briefly hidden. Its BEV map supports detection and map segmentation heads.

Time turned out to be a strong source of depth for cameras: motion between frames is a form of stereo. **StreamPETR** [37] propagates object queries from frame to frame online and reports camera-only results comparable to LiDAR-based methods on nuScenes, and **Far3D** [38] targets long-range camera detection with 3D queries generated from 2D detections.

### Fusing LiDAR, cameras and radar

**Plain version.** LiDAR is good at "where", cameras at "what". Fusion combines them, and the main design question is *where* in the network to combine: at the raw points, at the proposals, in a

shared BEV map, or through attention.

We group methods by where fusion happens; Wang et al. [39] give a finer taxonomy.

- **Point-level (early) fusion. PointPainting** [40] projects each LiDAR point into an image semantic segmentation and appends the class scores to the point; any LiDAR detector can then run on the "painted" cloud.
- **Proposal-level fusion. MV3D** [41] generates 3D proposals from a BEV map of the point cloud and fuses region features from the BEV, the LiDAR front view and the image. **Frustum PointNets** [42] go from 2D to 3D: a 2D detector finds the object in the image, and a PointNet processes only the points inside the frustum behind that 2D box.
- **BEV-level fusion.** Two methods named **BEVFusion** turn camera features into a BEV map (with an LSS-style view transform) and combine them with the LiDAR BEV map. One, from MIT [43], fuses with concatenation and a convolutional encoder, supports detection and map segmentation, and speeds up the camera-to-BEV pooling by more than 40 times. The other [44] keeps the camera stream independent of the LiDAR, so the detector still works when LiDAR fails, and reports large gains under simulated LiDAR malfunctions.
- **Query-level fusion. TransFusion** [45] first predicts object queries from LiDAR, then lets each query attend to image features with a transformer decoder. This "soft" association by attention, rather than a hard projection of points to pixels, makes it more robust to poor image conditions and calibration errors. **UniTR** [46] goes further and processes camera and LiDAR tokens with one shared transformer backbone.

**Radar.** Radar adds velocity and robustness to weather. **CenterFusion** [47] detects object centres in the image, associates radar returns with each detection through a frustum around it, and uses the radar features to improve depth, rotation and velocity, raising the nuScenes detection score of a camera detector substantially.

### Training: assignment and augmentation

**Plain version.** Training a 3D detector resembles 2D training, with two twists: assignment uses rotated BEV overlaps or centre heatmaps, and data augmentation can cut objects out of one scene and paste them into another, which is impossible to do convincingly with photos.

**Assignment.** Anchor-based heads place a few anchors per BEV cell (one size per class, usually at yaw 0° and 90°) and assign them by BEV IoU, like the 2D rules of DLCV 03. VoxelNet's car anchors, for instance, are positive above BEV IoU 0.6 and negative below 0.45 [8]. Because 3D scenes contain many more anchors than objects, these heads use the focal loss [23]. Centre-based heads replace IoU assignment by Gaussian heatmap targets [11].

**Augmentation.** LiDAR scenes are few (KITTI has 7,481 training scans [48]) and objects per scene are fewer still. Standard augmentations are [8], [9], [10]:

- **Ground-truth sampling.** Build a database of all labelled boxes and the points inside them, then paste randomly chosen objects into each training scene [9]. PointPillars samples 15 cars, 0 pedestrians and 8 cyclists per scene on KITTI [10]. VoxelNet already used a collision test to

reject physically impossible results when perturbing boxes [8], and the same check is needed when pasting.

- **Per-object perturbation.** Rotate and translate each box with its points; VoxelNet draws rotations from $[-\pi/10, \pi/10]$ and translations from a zero-mean Gaussian with standard deviation 1 m [8].
- **Global transforms.** Rotate the whole scene about the vertical axis (VoxelNet draws the angle from $[-\pi/4, \pi/4]$, simulating the car making a turn), scale it by a factor from $[0.95, 1.05]$, and flip it left–right [8], [10].

These are cheap because a point cloud is just coordinates: rotating points and boxes together keeps the labels exact. The catch is consistency across sensors: an object pasted into a LiDAR scan is not in the camera images, so fusion models need extra care.

*Online only: Ground-truth sampling in detail — https://blog.shyandram.dev/en/learn/deep-learning/3d-object-detection/notes/ground-truth-sampling-in-detail/*

### Datasets and metrics

**Plain version.** Three public benchmarks shaped the field. KITTI is small and front-facing, with a strict 3D IoU metric. nuScenes and Waymo are large and 360-degree; nuScenes matches boxes by centre distance and adds error terms, and Waymo weights each correct box by how well its heading matches.

| Dataset | Sensors (from the papers) | Scale (from the papers) | Main detection metric |
|---|---|---|---|
| KITTI 3D [49], [48] | Two stereo camera pairs (grey and colour), a Velodyne HDL-64E LiDAR | 7,481 training and 7,518 test images with point clouds; 80,256 labelled objects | 3D AP at IoU 0.7 (car) / 0.5 (pedestrian, cyclist), easy / moderate / hard |
| nuScenes [4], [50] | 6 cameras, 5 radars, 1 LiDAR, 360° | 1,000 scenes of 20 s; 23 classes and 8 attributes (10 classes for detection) | mAP with centre-distance matching; NDS |
| Waymo Open [3], [51] | 5 LiDARs, 5 cameras | 1,150 scenes of 20 s; about 12M labelled 3D LiDAR objects | mAPH at LEVEL_1 / LEVEL_2 |

### KITTI: 3D AP with difficulty levels

On the KITTI 3D benchmark a detection is correct if its 3D IoU with a ground-truth box is at least 0.7 for cars and 0.5 for pedestrians and cyclists [48]. Objects are split into three cumulative difficulty levels by their image appearance [48]:

- **Easy:** 2D box at least 40 px tall, fully visible, at most 15% truncated.
- **Moderate:** at least 25 px tall, at most partly occluded, at most 30% truncated.
- **Hard:** at least 25 px tall, at most "difficult to see", at most 50% truncated.

Methods are ranked by moderate AP. Since 8 October 2019 the benchmark computes AP from 40 recall positions instead of the original 11, following the analysis of Simonelli et al. [48], [52];

papers write this as $\mathrm{AP}|_{R_{40}}$, and numbers computed with the two rules must not be compared. The labels cover only objects visible in the front camera, so KITTI LiDAR detectors usually keep only the points that project into the image [10].

### nuScenes: centre-distance mAP and NDS

**Matching by distance.** nuScenes does not use IoU to decide whether a detection is correct. A prediction matches the ground-truth object with the smallest 2D centre distance on the ground plane, if that distance is below a threshold [50]. AP is computed per class for thresholds $\{0.5, 1, 2, 4\}$ m, integrating the precision–recall curve only where recall and precision exceed 0.1, and

$$\mathrm{mAP} = \frac{1}{|\mathbb{C}|\,|\mathbb{D}|} \sum_{c \in \mathbb{C}} \sum_{d \in \mathbb{D}} \mathrm{AP}_{c,d},$$

where $\mathbb{C}$ is the set of 10 detection classes and $\mathbb{D} = \{0.5, 1, 2, 4\}$ the set of distance thresholds in metres [50]. Each class is evaluated only within its own range from the ego vehicle (30 m for barriers and traffic cones, 40 m for bicycles, motorcycles and pedestrians, 50 m for the vehicle classes), and at most 500 boxes per sample may be submitted [50].

Why distance? IoU punishes the same centre error far more on small objects than on large ones (Figure 6). A 0.25 m error, for example, takes a 0.7 m pedestrian box to about IoU 0.47, below 0.5, while a car shifted by the same amount along its length keeps an IoU near 0.9.

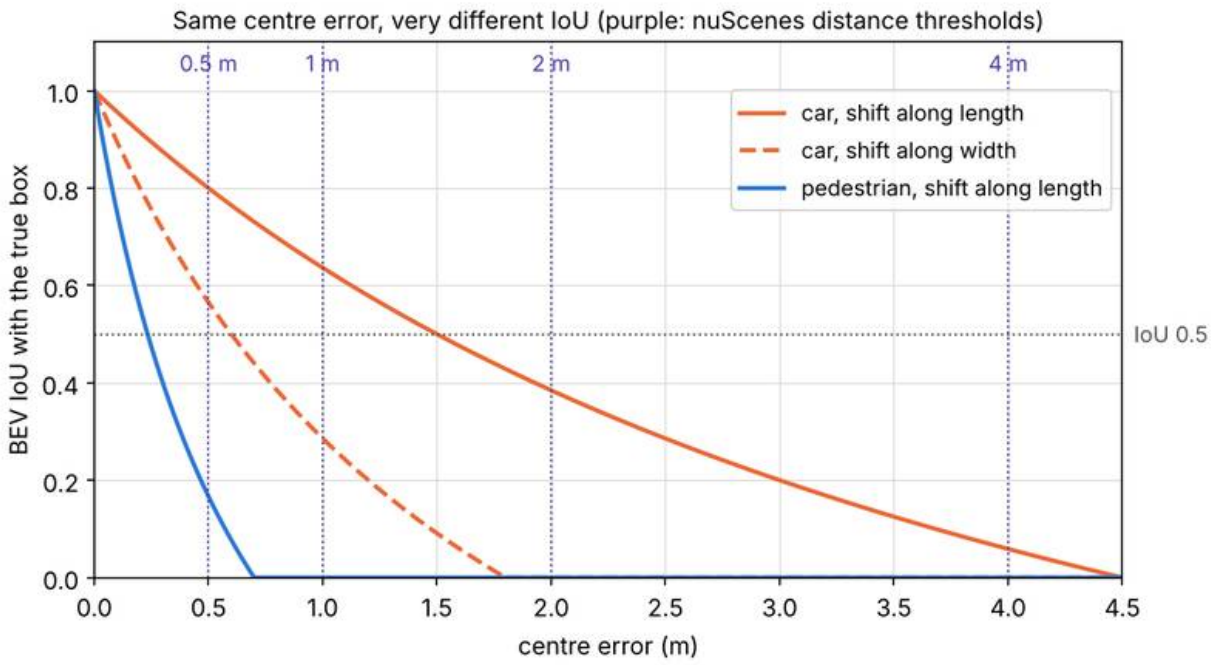



*Figure 6 — BEV IoU against centre error for a car ($4.5 \times 1.8$ m) and a pedestrian ($0.7 \times 0.7$ m), with sizes and yaw exact. The purple lines are the nuScenes matching thresholds.*

The snippet makes the same point with AP. Two classes get identical centre errors (Gaussian, standard deviation 0.25 m per axis) and the same number of false positives; only their sizes differ. (We compute plain 101-point interpolated AP and skip nuScenes' 0.1 clipping, which does not change the comparison.)

```
import numpy as np

rng = np.random.default_rng(0)
size = {"car": (4.5, 1.8), "pedestrian": (0.7, 0.7)}     # BEV length x width in metres

def bev_iou(a, b):
    """Axis-aligned BEV IoU of boxes (x, y, l, w); yaw is 0 in this toy."""
    ox = max(0, min(a[0] + a[2] / 2, b[0] + b[2] / 2) - max(a[0] - a[2] / 2, b[0] - b[2] /
     ↪  2))
    oy = max(0, min(a[1] + a[3] / 2, b[1] + b[3] / 2) - max(a[1] - a[3] / 2, b[1] - b[3] /
     ↪  2))
    return ox * oy / (a[2] * a[3] + b[2] * b[3] - ox * oy)

def average_precision(gts, preds, is_match):
    """Greedy matching in score order; AP = mean interpolated precision at 101 recall
     ↪  points."""
    used, tp = np.zeros(len(gts), bool), []
    for p in sorted(preds, key=lambda p: -p[-1]):
        ok = [j for j in range(len(gts)) if not used[j] and is_match(p, gts[j])]
        if ok:
            j = min(ok, key=lambda j: np.hypot(p[0] - gts[j][0], p[1] - gts[j][1]))
            used[j] = True
        tp.append(bool(ok))
    tp = np.array(tp)
    recall = np.cumsum(tp) / len(gts)
    precision = np.cumsum(tp) / np.arange(1, len(tp) + 1)
    envelope = np.maximum.accumulate(precision[::-1])[::-1]
    return np.mean([envelope[recall >= r].max() if np.any(recall >= r) else 0
                    for r in np.linspace(0, 1, 101)])

for cls, (l, w) in size.items():
    gts = [(x, y, l, w) for x, y in rng.uniform(-50, 50, (100, 2))]
    preds = [(g[0] + dx, g[1] + dy, l, w, s) for g, (dx, dy), s in
             zip(gts, rng.normal(0, 0.25, (100, 2)), rng.uniform(0.3, 1, 100))]
    preds += [(x, y, l, w, s) for (x, y), s in zip(rng.uniform(-50, 50, (20, 2)),
     ↪  rng.uniform(0, 0.6, 20))]
    ap_iou = average_precision(gts, preds, lambda p, g: bev_iou(p, g) >= 0.5)
    ap_d = [average_precision(gts, preds, lambda p, g, t=t: np.hypot(p[0] - g[0], p[1] -
     ↪  g[1]) <= t)
            for t in (0.5, 1, 2, 4)]
    print(f"{cls:10s} AP(BEV IoU>=0.5) = {ap_iou:.3f} | AP(centre dist <= 0.5/1/2/4 m) = "
          f"{' / '.join(f'{a:.3f}' for a in ap_d)} | mean = {np.mean(ap_d):.3f}")
```

```
car        AP(BEV IoU>=0.5) = 0.951 | AP(centre dist <= 0.5/1/2/4 m) = 0.789 / 0.988 / 0.988
 ↪  / 0.990 | mean = 0.939
pedestrian AP(BEV IoU>=0.5) = 0.126 | AP(centre dist <= 0.5/1/2/4 m) = 0.791 / 0.979 / 0.979
 ↪  / 0.979 | mean = 0.932
```

In this toy run, IoU matching rates the pedestrian detector as nearly useless (AP 0.126) and the car detector as excellent (0.951), although both made identical centre errors. Centre-distance matching gives both about the same score. Distance matching does not ignore box quality; it moves it into separate error terms.

**True-positive errors and NDS.** For matched detections at the 2 m threshold, nuScenes measures five **TP errors**: translation error ATE (centre distance in metres), scale error ASE ($1 - \text{IoU}$ after aligning centre and orientation), orientation error AOE (smallest yaw difference in radians), velocity error AVE (m/s) and attribute error AAE ($1-$ attribute accuracy) [50]. Their class means are combined with mAP into the **nuScenes detection score**:

$$\text{NDS} = \frac{1}{10}\Big[\,5\,\text{mAP} + \sum_{e\in\mathbb{E}} (1 - \min(1, e))\Big],$$

where $\mathbb{E} = \{\text{mATE}, \text{mASE}, \text{mAOE}, \text{mAVE}, \text{mAAE}\}$ is the set of mean TP errors [50]. Half of NDS comes from mAP, the other half from box quality, orientation, velocity and attributes, which is why methods that estimate velocity well (with multiple sweeps or frames) gain a lot of NDS.

*Online only: Computing NDS step by step — https://blog.shyandram.dev/en/learn/deep-learning/3d-object-detection/notes/computing-nds-step-by-step/*

### Waymo: AP weighted by heading

The Waymo Open Dataset uses 3D IoU matching with thresholds 0.7 for vehicles and 0.5 for pedestrians and cyclists, and its primary metric is **APH**, average precision weighted by heading [51]. Each true positive counts not as 1 but as a heading accuracy

$$a_h = 1 - \frac{\min\left(|\tilde{\theta} - \theta|,\ 2\pi - |\tilde{\theta} - \theta|\right)}{\pi} \in [0, 1],$$

where $\tilde{\theta}$ and $\theta$ are the predicted and ground-truth headings in $[-\pi, \pi]$ [3], [53]. A correct box facing exactly backwards ($|\tilde{\theta} - \theta| = \pi$) therefore counts as 0, which closes the loophole that IoU leaves open. Ground-truth boxes are split into two difficulty levels: **LEVEL_2** for boxes with 1 to 5 LiDAR points (or marked as LEVEL_2 in the released labels), **LEVEL_1** for the rest (boxes with no LiDAR points are not evaluated), and LEVEL_2 metrics include both [51]. Its 2020 3D detection challenge ranks entries by **mAPH at LEVEL_2**, the mean over vehicles, pedestrians and cyclists, and break results down by range: $[0, 35)$, $[35, 50)$ and beyond 50 m [51].

### Efficiency and deployment

**Plain version.** A car's computer must finish perception well before the next sweep arrives, on embedded hardware, every time. The representation decides most of the cost: how many cells, how many points per cell, and whether the operators run fast on the target chip.

- **Grid size and range set the budget.** Halving the cell size quadruples the BEV cells. PointPillars traded accuracy for speed exactly this way, from 62 Hz at its default grid to 105 Hz with coarser pillars [10].

- **Sparse operators need support.** Sparse convolution and point sampling are efficient in principle, but they rely on custom kernels; pillar networks run on dense 2D convolutions that every inference engine supports. DSVT made deployment an explicit design goal, building its pooling without custom CUDA operators and reporting real-time inference after deployment [26].
- **Camera view transforms are expensive.** Lifting every pixel to many depths creates huge intermediate point sets; BEVFusion's optimized BEV pooling reduced that step's latency by more than 40 times [43].
- **Benchmark speed is not vehicle speed.** PointPillars' paper notes that KITTI uses only the points inside the front camera's view, about 10% of the sweep, and that timings on desktop GPUs do not transfer to embedded computers [10]. Measure end to end, on the target hardware, with full 360-degree input.
- **Post-processing and temporal input count too.** Rotated NMS runs over many boxes; multi-sweep inputs multiply the points. nuScenes limits the past sensor data a detection method may use to about 0.5 s [50].

### Failure modes

**Plain version.** 3D detectors fail where the evidence is thin (far, small or hidden objects), where the sensors are disturbed (weather, dirt, miscalibration), and where the test data differ from the training data (another city, another sensor).

- **Far and sparse objects.** Points fall roughly with $1/r^2$ (Figure 2), so far objects have a few points or none. Benchmarks acknowledge this: Waymo assigns boxes with 1–5 points or an explicit difficulty flag to LEVEL_2, and its LEVEL_2 evaluation includes both difficulty levels [51]. nuScenes evaluates each class only within 30–50 m [50]. Camera depth errors grow with $Z^2$, which hurts the same objects.
- **Occlusion and truncation.** The sensor sees only surfaces facing it, and a parked car hides the pedestrian behind it. KITTI's difficulty levels are defined by occlusion and truncation [48]; temporal models help by remembering briefly hidden objects [35].
- **Confusable classes.** PointPillars reports that pedestrians and cyclists are often confused with each other, and pedestrians with poles and tree trunks [10].
- **Weather.** Fog, rain and snow scatter laser pulses, creating spurious near returns and losing far ones. Hahner et al. simulate physically based fog on clear-weather LiDAR scans and show that training with it improves detection in real fog [54].
- **Calibration and synchronization errors.** Fusion that hard-projects points to pixels breaks when extrinsics drift; TransFusion's soft association was designed for this [45], and LSS showed that training with noisy extrinsics or dropped cameras makes a camera BEV model more robust to them [32].
- **Common corruptions.** A benchmark of 27 LiDAR and camera corruptions applied to KITTI, nuScenes and Waymo, testing 24 detectors, found that motion-related corruptions hurt all models most, that LiDAR–camera fusion models were more robust, and that camera-only models were extremely vulnerable to image corruptions [55].
- **Domain shift between datasets.** A detector trained on one dataset can drop sharply on another. Wang et al. found that the main hurdle between German and US driving datasets was simply

the different average car size, and that correcting for it closes much of the gap [56]. The Waymo paper itself reports a pronounced gap between data from different cities [3]. Different LiDAR models (beam count, mounting height) shift the point distribution too.

### Modern view

Figure 7 places the methods of this chapter on a timeline, by the year of each cited publication.

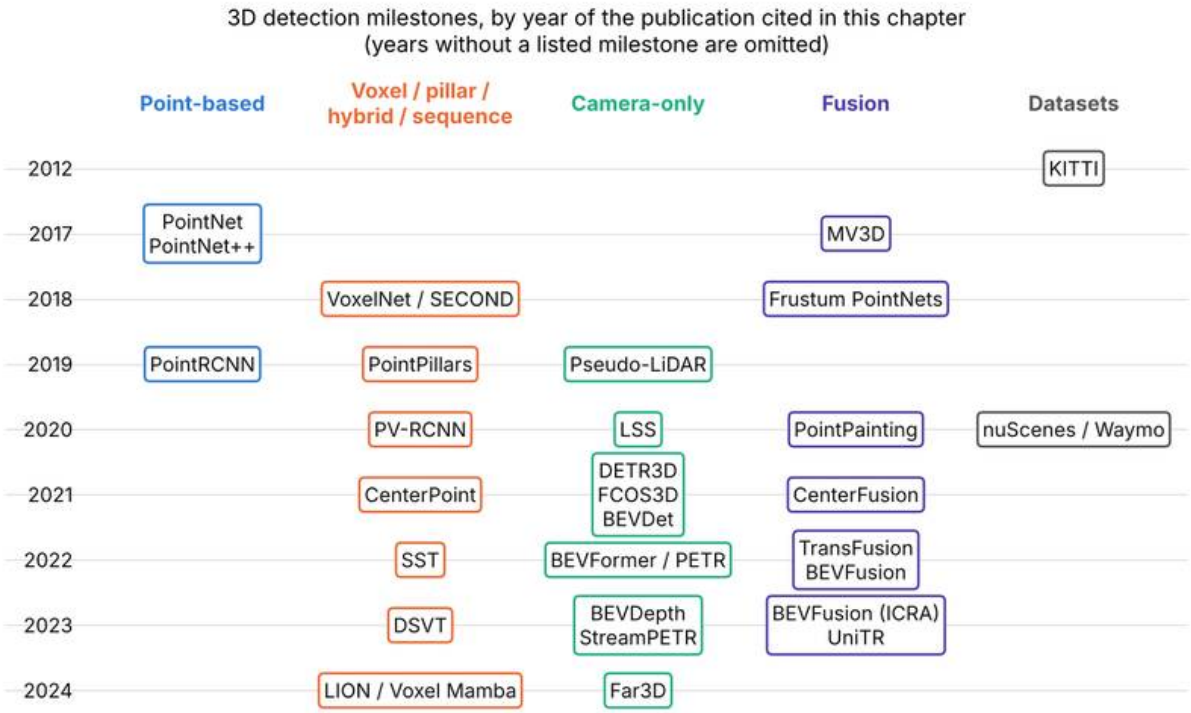



*Figure 7 — Milestones by year of the publication cited in this chapter, grouped by family. Years without a listed milestone are omitted.*

### What the surveys say

Six surveys frame the field, each with its own taxonomy. Mao et al. [1] give the most complete map of 3D detection for driving: LiDAR methods by data representation (points, grids, point–voxel, range images), camera methods (monocular, stereo, multi-view), fusion, transformers, temporal and label-efficient detection, with an analysis of accuracy and speed trends. Qian et al. [2] cover sensors, datasets and metrics and include a case study with runtime, error and robustness analyses. Guo et al. [57] review deep learning on point clouds as a whole, of which detection is one task. For cameras, Ma et al. [58] and Li et al. [59] survey bird's-eye-view perception; the latter frames the core problems as the view transformation, BEV ground truth, multi-sensor pipelines and generalization across sensor setups, and adds a practical "recipe" of tricks. Wang et

al. [39] organize fusion methods by fusion stage, inputs and granularity. Two shared takeaways: the BEV has become the common meeting place of sensors and tasks, and accuracy gains on the standard benchmarks have outpaced gains in robustness and deployability.

*Online only: Guide to the Modern view surveys — https://blog.shyandram.dev/en/learn/deep-learning/3d-object-detection/papers/guide-to-the-modern-view-surveys/*

### The state of the art, 2023–2026

- **LiDAR backbones** have moved from sparse convolutions toward sparse transformers and linear-time sequence models over serialized voxels (DSVT [26], LION [27], Voxel Mamba [28]). Point Transformer V3 [60] argues that scale matters more than intricate design, and improves further with joint training on several datasets.
- **Camera-only detection** has narrowed the gap to LiDAR, mainly through temporal modelling and sparse queries (StreamPETR [37]) and better depth supervision (BEVDepth [33]); long-range camera detection is an active direction (Far3D [38]).
- **Fusion** in a shared BEV or token space (BEVFusion [43], [44], UniTR [46]) is the standard recipe for the strongest systems, and robustness to missing or degraded sensors is now a design goal rather than an afterthought.
- **End-to-end driving.** Detection is increasingly one stage of a single network trained for planning, as in UniAD [61], where perception, prediction and planning tasks communicate through unified query interfaces.

### Open problems

1. **Far, small and rare objects.** Sparse evidence at range, and classes that are rare in training data (animals, debris, unusual vehicles), remain weak points; closed-set datasets do not cover the long tail [1].
2. **Robustness.** Weather, sensor faults, calibration drift and motion corruptions still cause large drops [55], [54].
3. **Generalization across sensors and places.** Models trained on one LiDAR model, camera rig or city often fail on another [56], [3].
4. **Metrics that match driving.** AP treats a missed pedestrian at 10 m and one at 45 m alike; nuScenes and Waymo add heading, velocity and range breakdowns, but, as Mao et al. note, maximizing AP alone may not be optimal for the driving system as a whole [1].
5. **Labels.** 3D boxes are expensive to annotate, which motivates semi-supervised, self-supervised and auto-labelling methods [1].

### Key takeaways

- State box conventions, units and coordinate transforms before interpreting detections.
- Point-cloud sparsity and range-dependent sampling shape the useful representations.
- BEV simplifies metric-space reasoning, but vertical structure and calibration still matter.

- Camera-only detection must infer depth; temporal and additional sensor evidence can help but introduce alignment requirements.
- Orientation errors can be hidden by overlap metrics; check yaw conventions and heading-sensitive measures.
- KITTI, nuScenes and Waymo use different matching and scoring rules; report the protocol and measure the deployed pipeline.

## Exercises

1. **Points at range.** A LiDAR has a horizontal step of 0.1° and a vertical step of 0.4°. Estimate how many points fall on the 1.8 m × 1.5 m back of a car at 20 m and at 60 m, using $n(r) \approx A/(r^2 \Delta\phi \Delta\epsilon)$. How many beams cross the car vertically at 60 m?

   **Hint**

   Convert to radians: $\Delta\phi = 1.75{\times}10^{-3}$, $\Delta\epsilon = 6.98{\times}10^{-3}$. With $A = 2.7\ \text{m}^2$, $n(20) \approx 2.7/(400{\cdot}1.22{\times}10^{-5}) \approx 550$ and $n(60) \approx 62$. Vertically, rays are $r\,\Delta\epsilon \approx 0.42$ m apart at 60 m, so only about 3–4 beams cross a 1.5 m tall object; the rest of the points come from the fine horizontal step.

2. **IoU by hand.** Two 4 m × 2 m × 1.5 m boxes share a centre; one is rotated by 90° about the vertical axis. Compute the BEV IoU and the 3D IoU. Then raise the second box by 0.75 m and recompute the 3D IoU. Check your answers with the IoU snippet.

   **Hint**

   BEV intersection is a 2 m × 2 m square: $\text{IoU}_{\text{bev}} = 4/(8 + 8 - 4) = 1/3$. With equal heights the 3D IoU is also $6/(12 + 12 - 6) = 1/3$. Raised by 0.75 m, the height overlap is 0.75 m, so the intersection is 3 m³ and $\text{IoU}_{\text{3D}} = 3/(24 - 3) \approx 0.143$.

3. **Break the classical pipeline.** In the classical-baseline snippet, move the pedestrian to $(13.5, -2.2)$, next to the first car (in our run its points come within about 0.15 m of the car's points), and run it again. What does DBSCAN do? Change `eps` to fix it, and explain what new failure a small `eps` causes for the far objects.

   **Hint**

   In our run, every `eps` from 0.7 down to 0.15 merges the pedestrian and the car into one cluster with a wrong box (5.1 × 1.8 m at yaw 28° instead of 4.5 × 1.8 m at 17°). At `eps=0.12` the pedestrian separates, but the car at 25 m already breaks into several fragments; at `eps=0.1` even the near car splits into its two visible sides. Far objects have fewer points, so the gaps between their points are larger (on a real scanner they grow like $r\,\Delta\phi$). A fixed `eps` cannot suit all ranges, a direct consequence of the $1/r^2$ density.

4. **Matching protocols.** Using the matching snippet, find the standard deviation of the centre error at which the car's AP under IoU ≥ 0.5 matches the pedestrian's AP at standard deviation 0.25 m. What does this say about comparing per-class APs computed with IoU matching?

**Hint**

In our runs the car's IoU-based AP falls to 0.117 at a standard deviation of 0.75 m, about three times the pedestrian's 0.25 m (roughly the ratio of the car's width to the pedestrian's), in line with Figure 6. Per-class IoU-based APs therefore mix detection quality with object size, so a low pedestrian AP does not mean the pedestrian detector localizes worse in metres.

5. **Heading-weighted AP.** A detector finds all 10 cars in a scene with no false positives and perfect IoU, but 3 of them face backwards and 2 are off by 30°. What are its AP and APH (assume all detections have the same score)? Which nuScenes TP error would reveal the same problem?

**Hint**

AP is 1, since every box matches. Heading accuracies: $5 \times 1 + 3 \times 0 + 2 \times (1 - 30/180) = 6.67$, so APH $\approx 0.667$. In nuScenes the orientation error AOE (in radians) would rise, lowering NDS through the term $1 - \min(1, \text{mAOE})$.

6. **Pillar budget.** In the pillarization snippet, change the grid to 0.32 m and to 0.08 m (adjust `W` and `H` so the grid still covers 70.4 m × 80 m). Report the number of non-empty pillars, the share of empty cells and the number of points dropped by the cap $N$. Relate the trend to the speed–accuracy trade-off reported by PointPillars.

**Hint**

In our runs, 0.32 m gives 7,567 non-empty pillars (86.2% empty) and drops 1,734 points at $N = 16$; 0.08 m gives 10,908 pillars (98.8% empty) and drops none. Coarser pillars mean fewer, fuller pillars: a smaller pseudo-image and more points dropped by the cap, so faster but less precise. Finer pillars mean more pillars with one or two points each, a larger pseudo-image and more empty cells. PointPillars reports higher speed with coarser grids and a loss mainly for pedestrians and cyclists.

Detections are the input to tracking, which links them over time; see DLCV 07 ·Object Tracking.

## References


1. J. Mao, S. Shi, X. Wang and H. Li, "3D object detection for autonomous driving: A comprehensive survey," *International Journal of Computer Vision*, vol. 131, no. 8, 2023. doi
2. R. Qian, X. Lai and X. Li, "3D object detection for autonomous driving: A survey," *Pattern Recognition*, vol. 130, 2022. doi
3. P. Sun, H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik et al., "Scalability in perception for autonomous driving: Waymo Open Dataset," in *Proc. IEEE/CVF CVPR*, 2020. CVF
4. H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong et al., "nuScenes: A multimodal dataset for autonomous driving," in *Proc. IEEE/CVF CVPR*, 2020. CVF
5. M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, 1981. doi

6. M. Ester, H.-P. Kriegel, J. Sander and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. KDD*, 1996. AAAI
7. X. Zhang, W. Xu, C. Dong and J. M. Dolan, "Efficient L-shape fitting for vehicle detection using laser scanners," in *Proc. IEEE Intelligent Vehicles Symposium (IV)*, 2017. doi
8. Y. Zhou and O. Tuzel, "VoxelNet: End-to-end learning for point cloud based 3D object detection," in *Proc. IEEE CVPR*, 2018. CVF
9. Y. Yan, Y. Mao and B. Li, "SECOND: Sparsely embedded convolutional detection," *Sensors*, vol. 18, no. 10, 3337, 2018. doi
10. A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang and O. Beijbom, "PointPillars: Fast encoders for object detection from point clouds," in *Proc. IEEE/CVF CVPR*, 2019. CVF
11. T. Yin, X. Zhou and P. Krähenbühl, "Center-based 3D object detection and tracking," in *Proc. IEEE/CVF CVPR*, 2021. CVF
12. I. E. Sutherland and G. W. Hodgman, "Reentrant polygon clipping," *Communications of the ACM*, vol. 17, no. 1, 1974. doi
13. X. Zhou, D. Wang and P. Krähenbühl, "Objects as points," arXiv:1904.07850, 2019. arXiv
14. N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov and S. Zagoruyko, "End-to-end object detection with transformers," in *Proc. ECCV*, 2020. doi
15. Y. Wang, V. Guizilini, T. Zhang, Y. Wang, H. Zhao and J. Solomon, "DETR3D: 3D object detection from multi-view images via 3D-to-2D queries," in *Proc. Conference on Robot Learning (CoRL)*, 2021. arXiv
16. C. R. Qi, H. Su, K. Mo and L. J. Guibas, "PointNet: Deep learning on point sets for 3D classification and segmentation," in *Proc. IEEE CVPR*, 2017. arXiv
17. C. R. Qi, L. Yi, H. Su and L. J. Guibas, "PointNet++: Deep hierarchical feature learning on point sets in a metric space," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. NeurIPS
18. S. Shi, X. Wang and H. Li, "PointRCNN: 3D object proposal generation and detection from point cloud," in *Proc. IEEE/CVF CVPR*, 2019. arXiv
19. S. Ren, K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *IEEE TPAMI*, vol. 39, no. 6, 2017. doi
20. B. Graham, M. Engelcke and L. van der Maaten, "3D semantic segmentation with submanifold sparse convolutional networks," in *Proc. IEEE CVPR*, 2018. CVF
21. S. Shi, C. Guo, L. Jiang, Z. Wang, J. Shi, X. Wang and H. Li, "PV-RCNN: Point-voxel feature set abstraction for 3D object detection," in *Proc. IEEE/CVF CVPR*, 2020. arXiv
22. W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu and A. C. Berg, "SSD: Single shot multibox detector," in *Proc. ECCV*, 2016. arXiv
23. T.-Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár, "Focal loss for dense object detection," in *Proc. IEEE ICCV*, 2017. doi
24. J. Huang, G. Huang, Z. Zhu, Y. Ye and D. Du, "BEVDet: High-performance multi-camera 3D object detection in bird-eye-view," arXiv:2112.11790, 2021. arXiv
25. L. Fan, Z. Pang, T. Zhang, Y.-X. Wang, H. Zhao, F. Wang, N. Wang and Z. Zhang, "Embracing single stride 3D object detector with sparse transformer," in *Proc. IEEE/CVF CVPR*, 2022. CVF
26. H. Wang, C. Shi, S. Shi, M. Lei, S. Wang, D. He, B. Schiele and L. Wang, "DSVT: Dynamic sparse voxel transformer with rotated sets," in *Proc. IEEE/CVF CVPR*, 2023. CVF
27. Z. Liu, J. Hou, X. Wang, X. Ye, J. Wang, H. Zhao and X. Bai, "LION: Linear group RNN for 3D

object detection in point clouds," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2024. NeurIPS
28. G. Zhang, L. Fan, C. He, Z. Lei, Z. Zhang and L. Zhang, "Voxel Mamba: Group-free state space models for point cloud based 3D object detection," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2024. NeurIPS
29. T. Wang, X. Zhu, J. Pang and D. Lin, "FCOS3D: Fully convolutional one-stage monocular 3D object detection," in *Proc. IEEE/CVF ICCV Workshops*, 2021. arXiv
30. Z. Tian, C. Shen, H. Chen and T. He, "FCOS: Fully convolutional one-stage object detection," in *Proc. IEEE/CVF ICCV*, 2019. arXiv
31. Y. Wang, W.-L. Chao, D. Garg, B. Hariharan, M. Campbell and K. Q. Weinberger, "Pseudo-LiDAR from visual depth estimation: Bridging the gap in 3D object detection for autonomous driving," in *Proc. IEEE/CVF CVPR*, 2019. arXiv
32. J. Philion and S. Fidler, "Lift, splat, shoot: Encoding images from arbitrary camera rigs by implicitly unprojecting to 3D," in *Proc. ECCV*, 2020. arXiv
33. Y. Li, Z. Ge, G. Yu, J. Yang, Z. Wang, Y. Shi, J. Sun and Z. Li, "BEVDepth: Acquisition of reliable depth for multi-view 3D object detection," in *Proc. AAAI*, 2023. arXiv
34. Y. Liu, T. Wang, X. Zhang and J. Sun, "PETR: Position embedding transformation for multi-view 3D object detection," in *Proc. ECCV*, 2022. arXiv
35. Z. Li, W. Wang, H. Li, E. Xie, C. Sima, T. Lu, Y. Qiao and J. Dai, "BEVFormer: Learning bird's-eye-view representation from multi-camera images via spatiotemporal transformers," in *Proc. ECCV*, 2022. arXiv
36. X. Zhu, W. Su, L. Lu, B. Li, X. Wang and J. Dai, "Deformable DETR: Deformable transformers for end-to-end object detection," in *Proc. ICLR*, 2021. arXiv
37. S. Wang, Y. Liu, T. Wang, Y. Li and X. Zhang, "Exploring object-centric temporal modeling for efficient multi-view 3D object detection," in *Proc. IEEE/CVF ICCV*, 2023. CVF
38. X. Jiang, S. Li, Y. Liu, S. Wang, F. Jia, T. Wang, L. Han and X. Zhang, "Far3D: Expanding the horizon for surround-view 3D object detection," in *Proc. AAAI*, 2024. arXiv
39. Y. Wang, Q. Mao, H. Zhu, J. Deng, Y. Zhang, J. Ji, H. Li and Y. Zhang, "Multi-modal 3D object detection in autonomous driving: A survey," *International Journal of Computer Vision*, vol. 131, no. 8, 2023. doi
40. S. Vora, A. H. Lang, B. Helou and O. Beijbom, "PointPainting: Sequential fusion for 3D object detection," in *Proc. IEEE/CVF CVPR*, 2020. arXiv
41. X. Chen, H. Ma, J. Wan, B. Li and T. Xia, "Multi-view 3D object detection network for autonomous driving," in *Proc. IEEE CVPR*, 2017. arXiv
42. C. R. Qi, W. Liu, C. Wu, H. Su and L. J. Guibas, "Frustum PointNets for 3D object detection from RGB-D data," in *Proc. IEEE CVPR*, 2018. CVF
43. Z. Liu, H. Tang, A. Amini, X. Yang, H. Mao, D. Rus and S. Han, "BEVFusion: Multi-task multi-sensor fusion with unified bird's-eye view representation," in *Proc. IEEE ICRA*, 2023. arXiv
44. T. Liang, H. Xie, K. Yu, Z. Xia, Z. Lin, Y. Wang, T. Tang, B. Wang et al., "BEVFusion: A simple and robust LiDAR-camera fusion framework," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2022. arXiv
45. X. Bai, Z. Hu, X. Zhu, Q. Huang, Y. Chen, H. Fu and C.-L. Tai, "TransFusion: Robust LiDAR-camera fusion for 3D object detection with transformers," in *Proc. IEEE/CVF CVPR*, 2022. arXiv

46. H. Wang, H. Tang, S. Shi, A. Li, Z. Li, B. Schiele and L. Wang, “UniTR: A unified and efficient multi-modal transformer for bird’s-eye-view representation,” in *Proc. IEEE/CVF ICCV*, 2023. arXiv
47. R. Nabati and H. Qi, “CenterFusion: Center-based radar and camera fusion for 3D object detection,” in *Proc. IEEE/CVF WACV*, 2021. doi
48. A. Geiger, P. Lenz, C. Stiller and R. Urtasun, “3D object detection evaluation 2017,” The KITTI Vision Benchmark Suite, official benchmark page (data sizes, IoU thresholds, difficulty levels, 40 recall positions). link
49. A. Geiger, P. Lenz and R. Urtasun, “Are we ready for autonomous driving? The KITTI vision benchmark suite,” in *Proc. IEEE CVPR*, 2012. doi
50. nuScenes devkit, “nuScenes detection task,” official evaluation documentation (classes, detection ranges, mAP, TP metrics, NDS, submission rules). link
51. Waymo Open Dataset, “3D detection challenge (2020),” official challenge page (APH, IoU thresholds, LEVEL_1/LEVEL_2, range breakdown). link
52. A. Simonelli, S. Rota Bulò, L. Porzi, M. López-Antequera and P. Kontschieder, “Disentangling monocular 3D object detection,” in *Proc. IEEE/CVF ICCV*, 2019. doi
53. Waymo Open Dataset, `metrics_utils.cc`, official metric implementation (function `ComputeHeadingAccuracy`). code
54. M. Hahner, C. Sakaridis, D. Dai and L. Van Gool, “Fog simulation on real LiDAR point clouds for 3D object detection in adverse weather,” in *Proc. IEEE/CVF ICCV*, 2021. arXiv
55. Y. Dong, C. Kang, J. Zhang, Z. Zhu, Y. Wang, X. Yang, H. Su, X. Wei et al., “Benchmarking robustness of 3D object detection to common corruptions in autonomous driving,” in *Proc. IEEE/CVF CVPR*, 2023. arXiv
56. Y. Wang, X. Chen, Y. You, L. E. Li, B. Hariharan, M. Campbell, K. Q. Weinberger and W.-L. Chao, “Train in Germany, test in the USA: Making 3D object detectors generalize,” in *Proc. IEEE/CVF CVPR*, 2020. arXiv
57. Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu and M. Bennamoun, “Deep learning for 3D point clouds: A survey,” *IEEE TPAMI*, vol. 43, no. 12, 2021. doi
58. Y. Ma, T. Wang, X. Bai, H. Yang, Y. Hou, Y. Wang, Y. Qiao, R. Yang and X. Zhu, “Vision-centric BEV perception: A survey,” *IEEE TPAMI*, vol. 46, no. 12, 2024. doi
59. H. Li, C. Sima, J. Dai, W. Wang, L. Lu, H. Wang, J. Zeng, Z. Li et al., “Delving into the devils of bird’s-eye-view perception: A review, evaluation and recipe,” *IEEE TPAMI*, vol. 46, no. 4, 2024. doi
60. X. Wu, L. Jiang, P.-S. Wang, Z. Liu, X. Liu, Y. Qiao, W. Ouyang, T. He and H. Zhao, “Point Transformer V3: Simpler, faster, stronger,” in *Proc. IEEE/CVF CVPR*, 2024. arXiv
61. Y. Hu, J. Yang, L. Chen, K. Li, C. Sima, X. Zhu, S. Chai, S. Du et al., “Planning-oriented autonomous driving,” in *Proc. IEEE/CVF CVPR*, 2023. arXiv

# DLCV 04 • Image Segmentation: Semantic, Instance and Panoptic

How deep networks label every pixel: the three segmentation tasks, fully convolutional and encoder–decoder networks, atrous convolution and multi-scale context, losses and metrics (mIoU, mask AP, panoptic quality), mask classification with MaskFormer and Mask2Former, and promptable and open-vocabulary segmentation.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/image-segmentation/

## What you'll learn

- Distinguish semantic, instance and panoptic output requirements.
- Track resolution through fully convolutional networks, decoders and skip connections.
- Choose losses for class imbalance and boundary-sensitive tasks.
- Compute and interpret mIoU, mask AP and panoptic quality.
- Compare mask classification, promptable segmentation and open-vocabulary interfaces.

## The big picture

A detector draws a box around each object (DLCV 03). A box is a rough answer: it contains background pixels, and it says nothing about the object's exact outline. **Segmentation** gives the exact answer. It decides, for every pixel, what that pixel belongs to.

> **Explain like I'm five**
>
> Give a child a colouring book page of a street and some crayons. **Semantic segmentation** is colouring by kind: every car blue, every person orange, the road grey, the sky light blue. Two cars that touch become one blue blob. **Instance segmentation** is putting a numbered sticker on each car and each person and colouring exactly inside that object's outline; the road and sky are left blank. **Panoptic segmentation** does both: every bit of the page gets a colour, and each car and person also gets its own number.

Why it matters. A self-driving car needs to know which pixels are drivable road, not just that "a road is somewhere in this box". A radiologist measuring a tumour needs its exact area. A photo editor that removes a person needs that person's outline. Robots that grasp objects, satellite images of fields and forests, and medical scans all need per-pixel answers.

This chapter is the deep-learning sequel of two classical chapters. IP 10 and IP 11 segment an image with thresholds, region growing, graph cuts, watersheds and active contours, using no training data; this chapter learns the segmentation from labeled examples. DLCV 03 introduced

boxes, IoU, anchors, feature pyramids, the focal loss and Mask R-CNN; we reuse those ideas and link back instead of repeating them. The promptable Segment Anything models are introduced in DLCV 01 and ML 10; here we only place them among the other tasks. Three surveys frame the field: Minaee et al. for the CNN era [1], Li et al. for transformer-based segmentation [2], and Zhou et al. for segmentation with foundation models [3]. We return to them in the Modern view.

## Three tasks, one label map

**Plain version.** Semantic segmentation assigns classes to pixels; instance segmentation separates objects; panoptic segmentation combines classes with object identities where applicable.

**Precise version.** Let $\Omega = \{1, \dots, H\} \times \{1, \dots, W\}$ be the set of pixel positions of an $H \times W$ image and let $\{1, \dots, K\}$ be the class labels.

- **Semantic segmentation** predicts a label map $y : \Omega \to \{1, \dots, K\}$, one class $y(\mathbf{x})$ per pixel $\mathbf{x}$. It does not distinguish two objects of the same class.
- **Instance segmentation** predicts a set $\{(m_j, c_j, s_j)\}_{j=1}^{N}$, where $m_j \in \{0,1\}^{H \times W}$ is a binary mask, $c_j$ its class, $s_j \in [0,1]$ a confidence score and $N$ varies per image. Masks may overlap, and only countable object classes are segmented. It is detection with a mask instead of a box.
- **Panoptic segmentation** [4] maps each pixel $i$ to a pair $(l_i, z_i)$, where $l_i$ is the class and $z_i$ an instance id. Pixels with the same pair form one **segment**. Segments cannot overlap, since each pixel carries one pair, and no confidence scores are required.

The panoptic paper names the two kinds of classes after an older distinction in vision. **Things** are countable objects with a well-defined shape, such as people, cars and animals. **Stuff** is amorphous regions of similar texture or material, such as sky, road and grass [4]. For a stuff class the instance id is ignored: all the sky in an image is one segment. Which classes are stuff and which are things is a design choice of each dataset. Pixels that are ambiguous or outside every class get a special **void** label and are not evaluated.

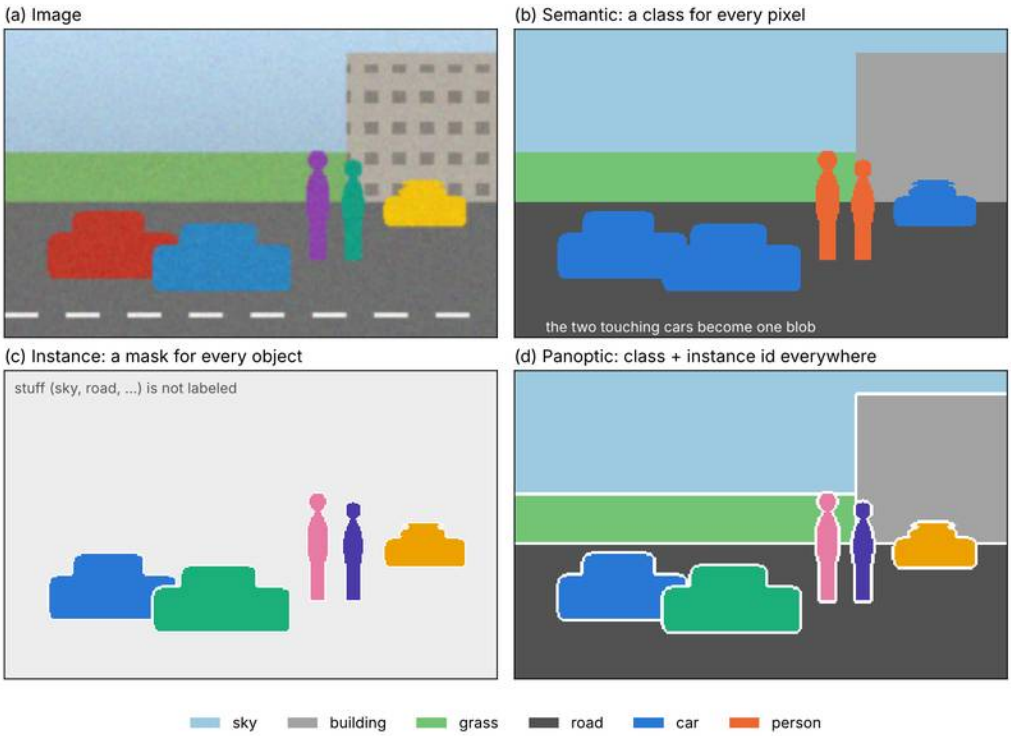



*Figure 1 — The three tasks on one synthetic scene. (b) Semantic: the two touching cars become one region. (c) Instance: one mask per object, stuff is not labeled. (d) Panoptic: every pixel gets a class, and every thing also gets its own identity.*

| | Semantic | Instance | Panoptic |
|---|---|---|---|
| Labels | class per pixel | mask + class + score per object | (class, instance id) per pixel |
| Classes covered | stuff and things | things only | stuff and things |
| Overlaps allowed | no | yes | no |
| Confidence scores | no | yes | no |
| Standard metric | mean IoU | mask AP | panoptic quality (PQ) |

The table explains why the three tasks long had separate methods: their outputs have different shapes, and each metric rewards something different. Most of this chapter follows that history: per-pixel classifiers for semantic segmentation first, then detector-based instance segmentation, then the panoptic task that forced the two to meet, and finally one architecture that handles all three.

## From a classifier to a per-pixel classifier

**Plain version.** Keep spatial features instead of globally pooling them. Predict a grid of class scores, then upsample it to the image size.

### Fully convolutional networks

Long, Shelhamer and Darrell's **fully convolutional network** (**FCN**) [5] replaced the fully connected layers of a classification network with convolutions. A convolution works on an input of any size, so the network now maps an $H \times W$ image to a score map

$$Z \in \mathbb{R}^{K \times \frac{H}{s} \times \frac{W}{s}},$$

where $K$ is the number of classes and $s$ is the **output stride**, the ratio between the input resolution and the score-map resolution. A classification backbone pools and strides repeatedly; for the networks FCN started from, $s = 32$, so a $512 \times 512$ image yields a $16 \times 16$ grid of class scores. Each score vector $Z[:, u, v]$ summarizes a large window of the image, which is good for recognizing *what* is there, but one cell covers $32 \times 32$ pixels, which is bad for saying exactly *where* a boundary runs.

FCN trains the whole network end to end with a per-pixel cross-entropy loss after upsampling $Z$ to the input size, starting from weights pretrained for classification. To recover detail, it fuses predictions from finer layers: the stride-32 scores are upsampled by 2 and added to a score map computed from the stride-16 layer, and again for stride 8. The paper calls the three versions FCN-32s, FCN-16s and FCN-8s, and shows that each finer fusion gives more detailed outlines [5]. Its summary of the idea is a good one-liner for the whole field: combine "what" from a deep, coarse layer with "where" from a shallow, fine layer. DLCV 01 introduces the same idea in its section on encoder–decoders.

### Upsample the scores, not the labels

**Plain version.** If you decide the class at low resolution and then enlarge the decision, every boundary becomes a staircase of $s \times s$ blocks. If you first enlarge the *scores* smoothly and then decide per pixel, the boundaries fall between grid points.

Bilinear interpolation computes the score at a fine position from the four nearest coarse cells. For a fine pixel that sits at fractional coarse coordinates $(u + a, v + b)$ with $u, v$ integers and $a, b \in [0, 1)$,

$$\tilde{Z}_k = (1-a)(1-b)\, Z_k[u, v] + a(1-b)\, Z_k[u+1, v] + (1-a)\, b\, Z_k[u, v+1] + a\, b\, Z_k[u+1, v+1],$$

where $Z_k$ is the score map of class $k$ and $\tilde{Z}_k$ the interpolated score. The predicted label is $\hat{y} = \arg\max_k \tilde{Z}_k$. FCN implements upsampling as a strided transposed convolution initialized to bilinear interpolation, and lets the network learn it [5]; transposed convolution is explained in ML 03. Most later models simply use fixed bilinear upsampling for the last step.

The experiment below isolates the effect of output stride. It builds an **ideal** coarse score map, the fraction of each class inside each $s \times s$ cell, an idealized output for a stride-$s$ head, and compares the two orders of "enlarge" and "decide".

```
import numpy as np
from scipy import ndimage as ndi
```

```
# Synthetic ground truth: 0 background, 1 large disk, 2 thin bar, 3 small square
H = W = 256
yy, xx = np.mgrid[0:H, 0:W]
gt = np.zeros((H, W), int)
gt[(yy - 120) ** 2 + (xx - 90) ** 2 < 60 ** 2] = 1
gt[40:216, 178:184] = 2                       # 6 pixels wide
gt[196:216, 60:80] = 3                        # 20 x 20 pixels
K = 4

def iou_per_class(gt, pred, K):
    return np.array([((gt == k) & (pred == k)).sum() / ((gt == k) | (pred == k)).sum()
                     for k in range(K)])

onehot = np.stack([(gt == k).astype(float) for k in range(K)])
print("stride | nearest labels: mIoU (per class) | bilinear scores: mIoU (per class)")
for s in [4, 8, 16, 32]:
    # ideal stride-s scores: the class fractions inside each s x s cell
    coarse = onehot.reshape(K, H // s, s, W // s, s).mean(axis=(2, 4))
    # (a) decide at low resolution, then enlarge the label map
    lab_nn = np.kron(coarse.argmax(0), np.ones((s, s), int))
    # (b) enlarge the scores with bilinear interpolation, then decide per pixel
    up = np.stack([ndi.zoom(c, s, order=1, mode="nearest", grid_mode=True) for c in coarse])
    lab_bi = up.argmax(0)
    a, b = iou_per_class(gt, lab_nn, K), iou_per_class(gt, lab_bi, K)
    print(f"{s:6d} | {a.mean():.3f} {np.round(a, 2)} | {b.mean():.3f} {np.round(b, 2)}")
```

```
stride | nearest labels: mIoU (per class) | bilinear scores: mIoU (per class)
     4 | 0.906 [0.99 0.97 0.67 1.  ] | 0.994 [1.   1.   0.99 0.99]
     8 | 0.825 [0.98 0.93 0.75 0.64] | 0.883 [0.99 0.99 0.7  0.86]
    16 | 0.565 [0.95 0.89 0.   0.41] | 0.578 [0.97 0.97 0.   0.37]
    32 | 0.421 [0.92 0.76 0.   0.  ] | 0.473 [0.96 0.93 0.   0.  ]
```

Three lessons come out of this toy run (the numbers are from the code above, not from any paper). Bilinear upsampling of scores is almost free and always at least as good for the large disk. But no upsampling can bring back what the coarse grid never saw: at stride 16 the 6-pixel bar is a minority in every cell and vanishes, and at stride 32 the 20-pixel square vanishes too. Thin and small structures need a small output stride or a decoder that brings fine features back. And mean IoU punishes those losses heavily, because the bar and the square count as much as the background (Figure 2).

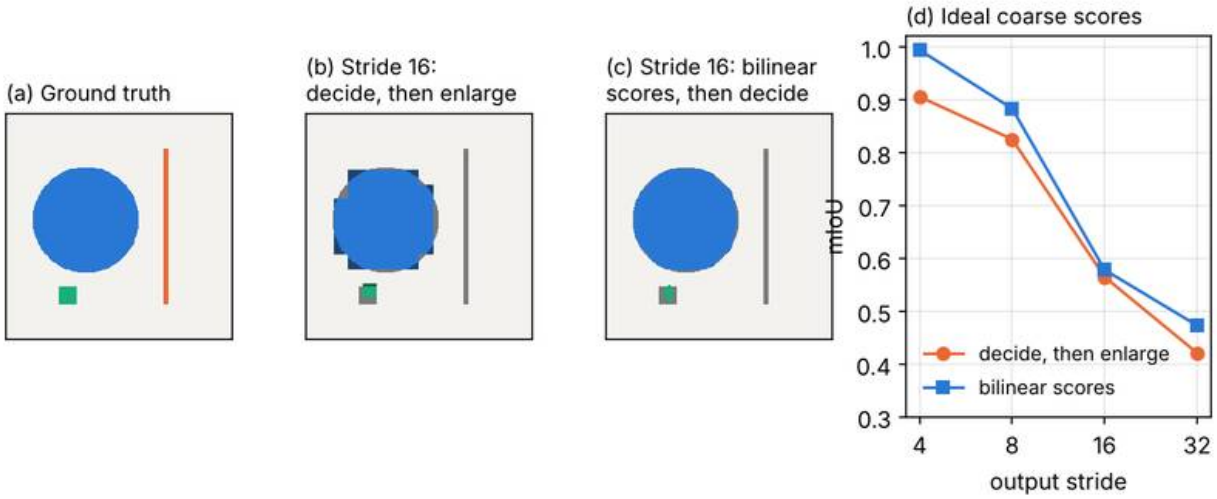



*Figure 2 — Output stride limits what a segmenter can recover, even with ideal coarse scores. (b, c) Stride 16; dark overlay marks wrong pixels. (d) mIoU of the toy map for both upsampling orders.*

### Encoder–decoders and skip connections

**Plain version.** A decoder increases resolution gradually and reuses matching encoder features through skips to recover detail.

**U-Net** [6] made this symmetric: a contracting **encoder** of convolutions and downsampling, an expanding **decoder** of upsampling and convolutions, and at each level a skip connection that concatenates the encoder's feature map to the decoder's. The encoder supplies *what*, the skips supply *where*. The paper's design choices (unpadded convolutions, the overlap-tile strategy, a weighted loss for touching cells and strong elastic augmentation) are covered in the U-Net reading guide of IP 10.

*Optional: U-Net: a reading guide to the paper (in the Paper guides of IP 10). What the paper claims, the output-size arithmetic of valid convolutions, and how to read its tables.*

The opposite design keeps the decoder light. **DeepLabv3+** [7] runs the encoder to output stride 16, upsamples its features by 4, concatenates low-level features from the backbone at stride 4 (after a $1 \times 1$ convolution that reduces their channels), refines with a few $3 \times 3$ convolutions and upsamples by 4 again. The paper motivates the decoder by noting that a plain 16-fold bilinear upsampling, a "naive decoder", may not recover object details. Figure 3 compares the three shapes: FCN's few fused score maps, U-Net's full ladder, and DeepLab's context module plus a single skip.

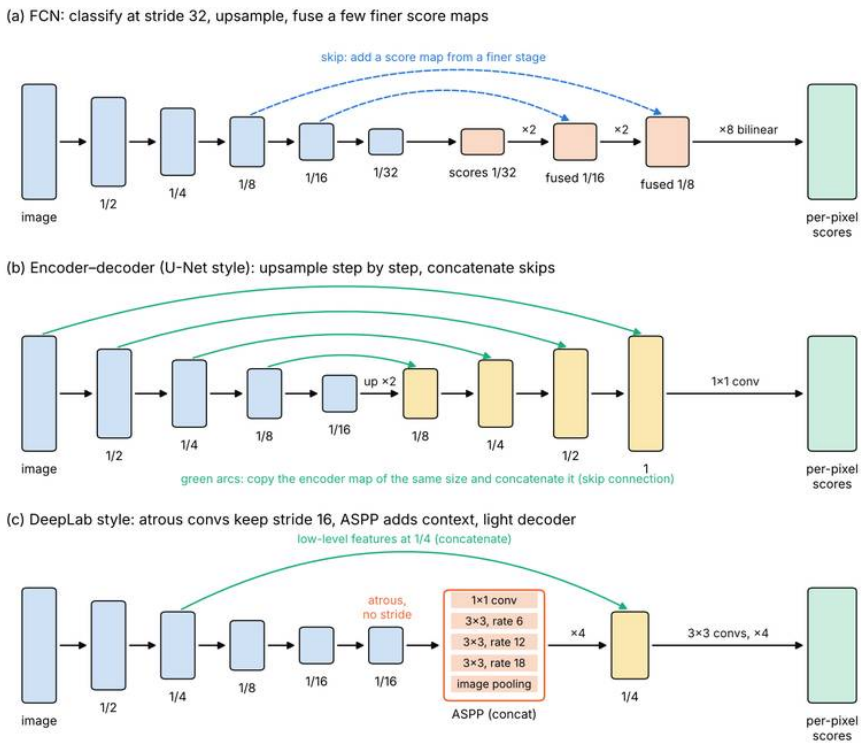



*Figure 3 — Three ways to get from coarse features back to pixels. (a) FCN fuses score maps from a few finer stages. (b) A U-Net-style encoder–decoder upsamples step by step and concatenates same-size encoder maps. (c) DeepLabv3+ keeps stride 16 with atrous convolutions, adds multi-scale context with ASPP, and uses one low-level skip.*

## Context without losing resolution

**Plain version.** To label a pixel you need two things that pull in opposite directions: a wide view (is this grey patch a road or a wall?) and fine resolution (where exactly does the road end?). Atrous convolution and pooling pyramids give the wide view without shrinking the map.

### Atrous (dilated) convolution

Downsampling is the usual way a CNN grows its receptive field, the region of the input that one output can see (DLCV 01 and ML 03 derive it). An **atrous** convolution ("à trous", with holes) spreads the taps of a small kernel apart instead. In 1-D, DeepLab [8] defines the output $y$ of an atrous convolution of input $x$ with a filter $w$ of length $K$ as

$$y[i] = \sum_{k=1}^{K} x[i + r \cdot k]\, w[k],$$

where $i$ indexes output positions and $r$ is the **rate**: the input is sampled every $r$ positions. Rate $r = 1$ is ordinary convolution. A $k \times k$ kernel with rate $r$ covers $k_e = k + (k-1)(r-1)$ pixels per side with the same $k^2$ weights and the same cost [8]. Yu and Koltun [9] called the same operation a dilated convolution and showed that stacking dilations that double from layer to layer grows the receptive field exponentially without losing resolution.

The DeepLab recipe is to set the stride of the last downsampling layer to 1 and replace the convolutions that follow by atrous convolutions with rate 2, so that they still see the same image region with the same pretrained weights; repeating this for one more stage doubles the density again [8]. Typical models output stride 8 or 16 instead of 32 [10]. The price is compute and memory: stride 8 has four times as many positions as stride 16 in the last stages.

```
import numpy as np

def atrous_1d(x, w, r):
    """y[i] = sum_k x[i + r*k] w[k] (DeepLab's Eq. 1), taps centred, zero padding."""
    K = len(w); c = K // 2
    xp = np.pad(x, r * c)
    return np.array([sum(xp[i + r * (k)] * w[k] for k in range(K)) for i in range(len(x))])

x = np.zeros(21); x[10] = 1.0                   # an impulse in the middle
w = np.array([1.0, 1.0, 1.0])
for r in [1, 2, 4]:
    print(f"rate {r}: taps at offsets {[(k - 1) * r for k in range(3)]},",
          "response", atrous_1d(x, w, r).astype(int))

def receptive_field(kernels, rates, strides):
    """Receptive field (in input pixels) of a stack of conv layers."""
    rf, jump = 1, 1                              # jump = distance between neighbouring
    ↪  outputs, in input pixels
    for k, r, s in zip(kernels, rates, strides):
        rf += (r * (k - 1)) * jump
        jump *= s
    return rf, jump

print("three 3x3, rates 1,1,1:", receptive_field([3, 3, 3], [1, 1, 1], [1, 1, 1]))
print("three 3x3, rates 1,2,4:", receptive_field([3, 3, 3], [1, 2, 4], [1, 1, 1]))
print("three 3x3, stride 2   :", receptive_field([3, 3, 3], [1, 1, 1], [2, 2, 2]))
```

```
rate 1: taps at offsets [-1, 0, 1], response [0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0]
rate 2: taps at offsets [-2, 0, 2], response [0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0]
rate 4: taps at offsets [-4, 0, 4], response [0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0]
three 3x3, rates 1,1,1: (7, 1)
three 3x3, rates 1,2,4: (15, 1)
three 3x3, stride 2   : (15, 8)
```

The last two lines are the whole argument for atrous convolution. Three layers with rates 1, 2, 4 see 15 input pixels, exactly as many as three stride-2 layers, but they keep one output per input pixel (jump 1) instead of one per 8 pixels (Figure 4).

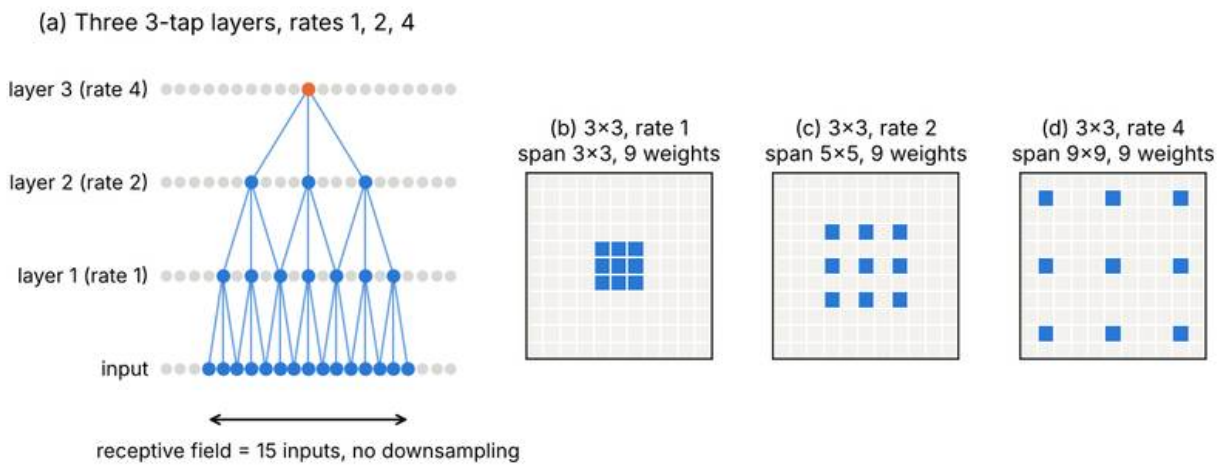



*Figure 4 — Atrous convolution. (a) Stacking rates 1, 2 and 4 covers 15 inputs at full resolution. (b–d) A 3×3 kernel at rates 1, 2 and 4 keeps 9 weights but spans 3×3, 5×5 and 9×9 pixels.*

Dilation has a catch. A single dilated layer only reads a sparse lattice of inputs, and stacking layers whose rates share a common factor leaves holes that no path ever reaches, a checkerboard artifact that Wang et al. call **gridding** [11]. The fix is to choose rates such as 1, 2, 5 that fill each other's gaps.

*Online only: Gridding in stacked dilated convolutions — https://blog.shyandram.dev/en/learn/deep-learning/image-segmentation/notes/gridding-in-stacked-dilated-convolutions/*

### Multi-scale context: ASPP and pyramid pooling

**Plain version.** Objects come at many sizes, and what a pixel is often depends on what surrounds it. So look at each position through several windows of different sizes at once, including one that sees the whole image, and let the network combine the answers.

**Atrous spatial pyramid pooling (ASPP)** [8] applies several atrous convolutions with different rates in parallel to the same feature map and fuses their outputs, which probes the image at several effective scales. DeepLabv3 [10] uses one $1\times1$ convolution and three $3\times3$ convolutions with rates 6, 12 and 18 at output stride 16 (rates doubled at stride 8), plus an **image-level feature**: global average pooling of the whole map, a $1\times1$ convolution, and bilinear upsampling back. The reason for the image-level branch is a boundary effect the paper points out: as the rate approaches the feature-map size, more and more of the $3\times3$ taps fall outside the map, and the filter degenerates to a $1\times1$ filter that only uses its centre weight. The snippet counts this on a $65\times65$ map.

```
import numpy as np

n = 65                                          # feature map is n x n
for r in [1, 6, 12, 18, 24, 36, 64]:
    pos = np.arange(n)
```

```
    inside_1d = np.array([[(0 <= p + r * d < n) for d in (-1, 0, 1)] for p in pos]).sum(1)
    mean_taps = np.mean(np.outer(inside_1d, inside_1d))   # taps inside the map, averaged
     ↪  over positions
    print(f"rate {r:2d}: mean valid taps {mean_taps:.2f} of 9")
```

```
rate  1: mean valid taps 8.82 of 9
rate  6: mean valid taps 7.93 of 9
rate 12: mean valid taps 6.92 of 9
rate 18: mean valid taps 5.98 of 9
rate 24: mean valid taps 5.11 of 9
rate 36: mean valid taps 3.58 of 9
rate 64: mean valid taps 1.06 of 9
```

At rate 64 a "3×3" filter is, on average, a single tap. Global pooling is the cheap and reliable way to get image-wide context.

The **Pyramid Scene Parsing Network** (**PSPNet**) [12] reaches the same goal with pooling instead of dilation. Its pyramid pooling module pools the final feature map into $1 \times 1$, $2 \times 2$, $3 \times 3$ and $6 \times 6$ grids, reduces each with a $1 \times 1$ convolution, upsamples all of them back and concatenates them with the original features. The paper motivates it with errors of an FCN baseline on scene parsing, such as a "mismatched relationship": a boat on a river labeled as a car, although a car is seldom over a river [12].

Early DeepLab versions also sharpened boundaries after the network with a fully connected conditional random field, which encourages pixels of similar colour and position to share a label [8]. DeepLabv3 improves on the earlier versions without it [10].

*Online only: Dense CRF post-processing — https://blog.shyandram.dev/en/learn/deep-learning/image-segmentation/notes/dense-crf-post-processing/*

### Attention as context

A transformer gives every token a view of every other token in each layer, so "context" no longer needs a special module (ML 05 explains the Vision Transformer, ViT [13]). The catch is resolution: plain ViT has one coarse token grid, and global attention over many tokens is costly. **SegFormer** [14] is a widely used answer for semantic segmentation. Its encoder is hierarchical, producing multi-scale features like a CNN, uses a reduced attention that shortens the key and value sequences, and needs no positional encoding, which avoids interpolating position codes when the test resolution differs from the training one. Its decoder is only a few MLP layers that fuse the multi-scale features. Hierarchical vision transformers are covered in ML 05.

## Losses for dense prediction

**Plain version.** A segmenter is trained by comparing its per-pixel probabilities with the true label map. Averaging a classification loss over pixels is the default, but when one class covers 99% of the image, that average mostly measures the easy class.

**Per-pixel cross-entropy.** With $p_{\mathbf{x}}(k)$ the softmax probability of class $k$ at pixel $\mathbf{x}$ and $y(\mathbf{x})$ the true class,

$$\mathcal{L}_{\mathrm{CE}} = -\frac{1}{|\Omega'|} \sum_{\mathbf{x} \in \Omega'} w_{y(\mathbf{x})} \log p_{\mathbf{x}}(y(\mathbf{x})),$$

where $\Omega'$ is the set of labeled (non-void) pixels and $w_k$ an optional weight per class, for example inversely proportional to the class frequency. Void pixels are simply left out of the sum; most frameworks implement this with an "ignore index".

**Dice loss.** Milletari et al. [15] proposed to maximize the Dice coefficient between a soft prediction and the binary ground truth directly, as an alternative to re-weighting samples, because the anatomy of interest often occupies only a small part of a medical volume and training then tends to predict background everywhere. For foreground probabilities $p_i$ and labels $g_i \in \{0, 1\}$ over $N$ pixels, their form is

$$D = \frac{2 \sum_{i=1}^{N} p_i g_i}{\sum_{i=1}^{N} p_i^2 + \sum_{i=1}^{N} g_i^2}, \qquad \mathcal{L}_{\mathrm{Dice}} = 1 - D.$$

A common variant uses $\sum p_i + \sum g_i$ in the denominator; with binary $p$ and $g$ both equal the overlap measure $2|A \cap B| / (|A| + |B|)$, a close relative of IoU. For several classes, the Dice loss is computed per class and averaged. Because $D$ is a ratio over the whole image, a small object counts as much as a large background.

**Focal loss.** The focal loss [16] multiplies the cross-entropy of each pixel by $(1 - p_t)^{\gamma}$, where $p_t$ is the probability of the true class, so that easy pixels contribute little; it is derived in DLCV 03. Combinations are common: MaskFormer, discussed below, trains each binary mask with a weighted sum of a focal loss and a Dice loss [17].

The experiment compares the losses on a 64×64 image whose foreground is a disk of 49 pixels, for three candidate predictions. "IoU@0.5" thresholds the probabilities at 0.5 and computes the IoU that an evaluation would report.

```
import numpy as np

H = W = 64
yy, xx = np.mgrid[0:H, 0:W]
r2 = (yy - 32) ** 2 + (xx - 32) ** 2
g = (r2 <= 4 ** 2).astype(float)                # small foreground disk
print(f"foreground pixels: {int(g.sum())} of {g.size} ({100 * g.mean():.1f}%)")

def bce(p, g):
    return -np.mean(g * np.log(p) + (1 - g) * np.log(1 - p))

def weighted_bce(p, g):                          # weight each class by inverse frequency
    wf, wb = 0.5 / g.mean(), 0.5 / (1 - g.mean())
    return -np.mean(wf * g * np.log(p) + wb * (1 - g) * np.log(1 - p))

def focal(p, g, gamma=2.0):
    pt = np.where(g == 1, p, 1 - p)
```

```
    return -np.mean((1 - pt) ** gamma * np.log(pt))

def soft_dice_loss(p, g, eps=1e-6):
    return 1 - (2 * (p * g).sum() + eps) / (p.sum() + g.sum() + eps)

def iou_at_half(p, g):                          # the metric we actually report
    m = p > 0.5
    return (m & (g == 1)).sum() / (m | (g == 1)).sum()

preds = {
    "all background (p=0.05)":  np.full((H, W), 0.05),
    "right shape, timid (0.4)": np.where(g == 1, 0.4, 0.05),
    "bloated, confident (0.9)": np.where(r2 <= 8 ** 2, 0.9, 0.05),
}
print(f"{'prediction':26s} {'BCE':>6s} {'wBCE':>6s} {'focal':>6s} {'Dice':>6s}
 ↪  {'IoU@0.5':>8s}")
for name, p in preds.items():
    print(f"{name:26s} {bce(p, g):6.3f} {weighted_bce(p, g):6.3f} "
          f"{focal(p, g):6.3f} {soft_dice_loss(p, g):6.3f} {iou_at_half(p, g):8.3f}")

# Where does the BCE gradient come from? dL/dz = p - g per pixel (z = logit)
p = preds["all background (p=0.05)"]
grad = np.abs(p - g)
print(f"share of |BCE gradient| from background pixels: {grad[g == 0].sum() /
 ↪  grad.sum():.2f}")
```

```
foreground pixels: 49 of 4096 (1.2%)
prediction                   BCE   wBCE  focal   Dice  IoU@0.5
all background (p=0.05)     0.087  1.524  0.032  0.981    0.000
right shape, timid (0.4)    0.062  0.484  0.004  0.855    0.000
bloated, confident (0.9)    0.133  0.119  0.068  0.791    0.249
share of |BCE gradient| from background pixels: 0.81
```

Read the table row by row (all numbers are from this toy run). Plain binary cross-entropy (BCE) prefers "all background" to the bloated prediction, the only one that finds the object at the 0.5 threshold, because the bloated mask is wrong on about 150 background pixels and those outnumber the object. The unweighted focal loss, without the class weight $\alpha$, makes the same choice. Class-weighted BCE and Dice both rank the bloated prediction best and "all background" worst, in the same order as the IoU. The last line shows why plain BCE behaves this way: 81% of its gradient comes from background pixels that are already almost right. Note also how harshly soft Dice treats the timid prediction: the 4,047 background pixels at $p = 0.05$ add about 202 to the denominator, more than four times the object's area, so a soft prediction must be confident on the background too.

In practice, cross-entropy (often with class weights or the focal modulation) is the default for natural-scene semantic segmentation, Dice or a Dice + cross-entropy sum is common for medical images with small structures, and mask-based models train each mask with Dice + focal. Losses that also penalize boundary or topology errors exist; three built from active-contour energies are summarized in IP 11's Supplementary notes.

*Online only: The gradient of the soft Dice loss — https://blog.shyandram.dev/en/learn/deep-learning/image-segmentation/notes/the-gradient-of-the-soft-dice-loss/*

## Measuring semantic segmentation

**Plain version.** Count, over the whole test set, how many pixels of each true class were given each predicted class. Every common score is a different summary of that table. The standard one, mean IoU, asks for each class "of all pixels that are this class or were called this class, what fraction got it right?", and then averages over classes.

Let $n_{ij}$ be the number of pixels of true class $i$ predicted as class $j$, accumulated over all test images, and $t_i = \sum_j n_{ij}$ the number of pixels of class $i$. FCN [5] defines four metrics, with $K$ classes:

$$\text{pixel accuracy} = \frac{\sum_i n_{ii}}{\sum_i t_i}, \qquad \text{mean accuracy} = \frac{1}{K}\sum_i \frac{n_{ii}}{t_i},$$

$$\text{mean IoU} = \frac{1}{K}\sum_i \frac{n_{ii}}{t_i + \sum_j n_{ji} - n_{ii}}, \qquad \text{frequency-weighted IoU} = \frac{1}{\sum_k t_k}\sum_i t_i \frac{n_{ii}}{t_i + \sum_j n_{ji} - n_{ii}}.$$

In the IoU of class $i$, the numerator $n_{ii}$ counts true positives and the denominator counts the union of true and predicted pixels: $t_i$ (true positives plus misses) plus $\sum_j n_{ji} - n_{ii}$ (false alarms). **Mean IoU** (**mIoU**) is the standard metric of PASCAL VOC, Cityscapes and ADE20K.

```
import numpy as np

def confusion(gt, pred, K, ignore=255):
    """K x K matrix: rows = true class, columns = predicted class; 'ignore' pixels are
     ↪  skipped."""
    keep = gt != ignore
    return np.bincount(K * gt[keep] + pred[keep], minlength=K * K).reshape(K, K)

def seg_metrics(C):
    tp = np.diag(C).astype(float)
    t = C.sum(1)                    # pixels of each true class
    p = C.sum(0)                    # pixels predicted as each class
    iou = tp / (t + p - tp)
    return {"pixel acc": tp.sum() / C.sum(), "mean acc": np.mean(tp / t),
            "mIoU": iou.mean(), "fwIoU": (t * iou).sum() / t.sum(), "IoU": np.round(iou, 3)}

# Toy: road (0) covers most of the image, car (1) is medium, pole (2) is thin
H, W = 120, 160
gt = np.zeros((H, W), int)
gt[60:100, 40:90] = 1
gt[10:110, 130:133] = 2
pred = gt.copy()
pred[60:100, 86:90] = 0                          # car slightly too short
pred[10:110, 130:133] = 0                        # the pole is missed entirely
m = seg_metrics(confusion(gt, pred, 3))
for k, v in m.items():
    print(f"{k:9s}: {v:.3f}" if np.isscalar(v) else f"{k:9s}: {v}")
```

```
pixel acc: 0.976
mean acc : 0.640
mIoU     : 0.631
fwIoU    : 0.953
IoU      : [0.974 0.92  0.   ]
```

Missing the pole entirely costs 1.6% of pixel accuracy and almost nothing in frequency-weighted IoU, but a third of the mean IoU. That is why mIoU, which weighs every class equally, is the headline number. Four practical details matter when you compare papers:

- The confusion matrix is accumulated over the **whole dataset**, then the IoUs are computed. Averaging per-image IoUs gives a different number.
- Void pixels are excluded through the ignore label, as in `confusion` above.
- mIoU still favours large objects *within* a class: one large car outweighs ten small ones. Cityscapes therefore also reports an instance-level **iIoU**, in which each pixel's contribution is weighted by the ratio of the class's average instance size to the size of its ground-truth instance [18].
- Region overlap is insensitive to boundary quality on large objects, where a few pixels of boundary error hardly change the IoU. **Boundary IoU** [19] measures overlap only in a thin band along the contours.

*Online only: Boundary IoU in detail — https://blog.shyandram.dev/en/learn/deep-learning/image-segmentation/notes/boundary-iou-in-detail/*

> **Interactive demonstration** (**online**): open the online edition of this tutorial to use this demonstration.Open demonstration

## Instance segmentation

**Plain version.** Instance segmentation is detection with an outline. The dominant approach finds each object first, as a detector would, and then segments the inside of its box.

### Detect, then segment: Mask R-CNN

**Mask R-CNN** [20] adds a small fully convolutional mask branch to the two-stage detector Faster R-CNN [21], which DLCV 03 explains together with RoIAlign. For each region of interest, the branch predicts $K$ binary masks of size $m \times m$, one per class ($28 \times 28$ in the paper's variant built on a feature pyramid network, FPN [22]), and the loss uses a per-pixel sigmoid and the average binary cross-entropy on the mask of the ground-truth class only. Mask and class prediction are therefore **decoupled**: the box branch chooses the class, and masks of different classes do not compete, unlike the per-pixel softmax of an FCN; the paper's ablation finds that the per-class

sigmoid gives large gains over a softmax [20]. At test time the $m \times m$ mask of the predicted class is resized to the box and binarized at 0.5.

*Optional: RoI pooling versus RoIAlign (in the Supplementary notes of DLCV 03). Why rounding the box to the feature grid shifts masks, and the bilinear sampling that fixes it.*

The fixed $28 \times 28$ grid is also Mask R-CNN's weakness: a large object's outline is drawn from a few hundred cells and comes out oversmoothed. **PointRend** [23] treats mask prediction like rendering in computer graphics: it starts from a coarse mask, repeatedly upsamples it bilinearly, and recomputes predictions only at the most uncertain points of the denser grid (for a binary mask, those with probabilities closest to 0.5), which lie mostly along the boundary.

Other routes exist. **Bottom-up** methods first predict per-pixel quantities, such as an embedding vector or a distance to the object centre, and then group pixels into instances; a watershed on a predicted distance map, with markers found by the network, is the learned version of the marker-controlled watershed of IP 10. And **query-based** methods predict a set of masks directly, as described in the mask-classification section below.

### Mask AP

Instance segmentation is scored exactly like detection, with the COCO average precision of DLCV 03, except that the IoU between a prediction and a ground-truth object is computed on masks instead of boxes [20]. The metric needs a confidence score per mask, because AP ranks predictions, and it allows overlapping masks: a detector may output several overlapping hypotheses and is only penalized for those that do not match. This is why instance outputs cannot be used as a panoptic map without further processing.

## Panoptic segmentation

**Plain version.** The panoptic task asks for one coherent picture: every pixel labeled, stuff and things alike, no overlaps, and a metric that treats a sky region and a car the same way.

### The task and its metric

Kirillov et al. [4] defined the task (the format is the one given in "Three tasks, one label map") together with a metric, **panoptic quality (PQ)**. PQ is computed in two steps.

**1. Matching.** A predicted segment $p$ and a ground-truth segment $g$ of the same class match only if their IoU is strictly greater than 0.5. Because segments of a panoptic map cannot overlap, this match is **unique**: if $p_1$ and $p_2$ are disjoint predictions, then $\mathrm{IoU}(p_1, g) + \mathrm{IoU}(p_2, g) \leq 1$, since each $|p_i \cap g|/|p_i \cup g| \leq |p_i \cap g|/|g|$ and the two intersections share no pixel. So at most one prediction can exceed 0.5 for each $g$, and vice versa [4]. No optimal assignment step is needed.

**2. Scoring.** For each class, the matches split the segments into true positives TP (matched pairs),

false positives FP (unmatched predictions) and false negatives FN (unmatched ground-truth segments). Then

$$\mathrm{PQ} = \frac{\sum_{(p,g)\in \mathrm{TP}} \mathrm{IoU}(p,g)}{|\mathrm{TP}| + \tfrac{1}{2}|\mathrm{FP}| + \tfrac{1}{2}|\mathrm{FN}|} = \underbrace{\frac{\sum_{(p,g)\in \mathrm{TP}} \mathrm{IoU}(p,g)}{|\mathrm{TP}|}}_{\text{segmentation quality (SQ)}} \times \underbrace{\frac{|\mathrm{TP}|}{|\mathrm{TP}| + \tfrac{1}{2}|\mathrm{FP}| + \tfrac{1}{2}|\mathrm{FN}|}}_{\text{recognition quality (RQ)}},$$

where the sums run over matched pairs and $|\cdot|$ counts elements of a set. **SQ** is the average IoU of matched segments, and **RQ** is the familiar F1 score of detection. PQ is computed per class and averaged over classes, so a rare class counts as much as a frequent one; every segment counts equally regardless of its area. $\mathrm{PQ}^{\mathrm{St}}$ and $\mathrm{PQ}^{\mathrm{Th}}$ average over stuff and thing classes only. Void pixels are removed from predictions before matching, and unmatched predictions that consist mostly of void (or of a "crowd" region of the same class) are not counted as false positives [4]. The two factors are not independent, since SQ is measured only on the matched segments, so always report all three.

The code implements this matching and scoring on the toy scene of Figure 5 (void handling reduced to its simplest form).

```
import numpy as np

def panoptic_quality(gt_seg, gt_cls, pr_seg, pr_cls):
    """gt_seg/pr_seg: integer segment-id maps (0 = void); *_cls: dict segment id -> class.
    Matches segments of the same class with IoU > 0.5 and returns per-class (PQ, SQ, RQ, TP,
     ↪  FP, FN)."""
    out = {}
    for c in sorted(set(gt_cls.values()) | set(pr_cls.values())):
        g_ids = [i for i, k in gt_cls.items() if k == c]
        p_ids = [i for i, k in pr_cls.items() if k == c]
        ious, matched_p = [], set()
        for gi in g_ids:
            G = gt_seg == gi
            for pj in p_ids:
                P = (pr_seg == pj) & (gt_seg != 0)          # ignore predicted pixels on void
                iou = (G & P).sum() / (G | P).sum()
                if iou > 0.5:                                # unique by the matching theorem
                    ious.append(iou); matched_p.add(pj)
        tp = len(ious); fn = len(g_ids) - tp; fp = len(p_ids) - len(matched_p)
        sq = np.mean(ious) if tp else 0.0
        rq = tp / (tp + 0.5 * fp + 0.5 * fn)
        out[c] = (sq * rq, sq, rq, tp, fp, fn)
    return out

H, W = 100, 200
SKY, ROAD, CAR, PERSON = 0, 1, 2, 3
def scene(boxes, sky_rows):
    seg = np.full((H, W), 2); seg[:sky_rows] = 1          # segment 1 = sky, 2 = road
    cls = {1: SKY, 2: ROAD}
    for sid, (c, r0, r1, c0, c1) in enumerate(boxes, start=3):
        seg[r0:r1, c0:c1] = sid; cls[sid] = c
    return seg, cls
```

```
gt_seg, gt_cls = scene([(CAR, 50, 80, 10, 55), (CAR, 50, 80, 65, 110),
                        (CAR, 38, 52, 125, 150), (PERSON, 30, 80, 165, 185)], sky_rows=30)
pr_seg, pr_cls = scene([(CAR, 52, 80, 13, 55),             # car A, slightly off
                        (CAR, 55, 80, 65, 96),             # car B, too small
                        (CAR, 84, 97, 110, 150),           # a car that is not there
                        (PERSON, 30, 52, 165, 185),        # the person, split in two
                        (PERSON, 52, 80, 165, 185)], sky_rows=33)   # car C is missed
names = ["sky", "road", "car", "person"]
res = panoptic_quality(gt_seg, gt_cls, pr_seg, pr_cls)
for c, (pq, sq, rq, tp, fp, fn) in res.items():
    print(f"{names[c]:6s} PQ {pq:.3f} = SQ {sq:.3f} x RQ {rq:.3f}  (TP {tp}, FP {fp}, FN
     ↪ {fn})")
pqs = np.array([v[0] for v in res.values()])
print(f"PQ {pqs.mean():.3f} | PQ_stuff {pqs[:2].mean():.3f} | PQ_things
 ↪ {pqs[2:].mean():.3f}")
```

```
sky    PQ 0.917 = SQ 0.917 x RQ 1.000  (TP 1, FP 0, FN 0)
road   PQ 0.805 = SQ 0.805 x RQ 1.000  (TP 1, FP 0, FN 0)
car    PQ 0.482 = SQ 0.723 x RQ 0.667  (TP 2, FP 1, FN 1)
person PQ 0.373 = SQ 0.560 x RQ 0.667  (TP 1, FP 1, FN 0)
PQ 0.644 | PQ_stuff 0.861 | PQ_things 0.428
```

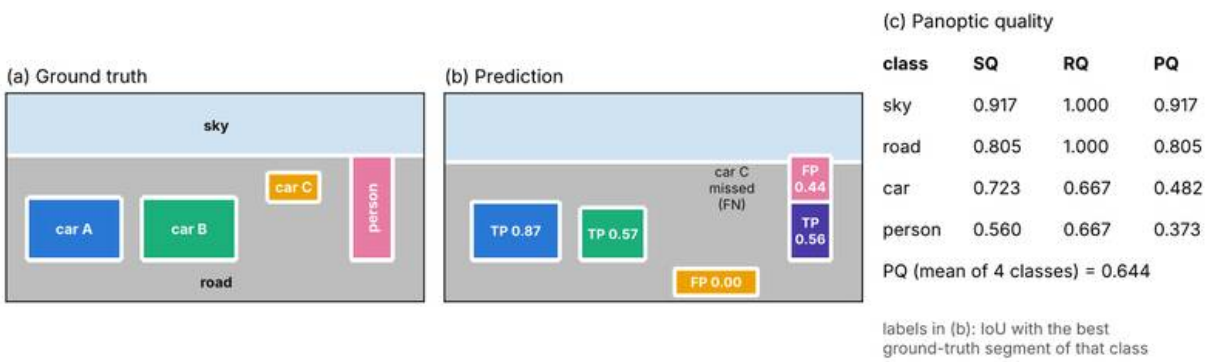



*Figure 5 — Panoptic quality on a toy scene. Each predicted segment is labeled with its best IoU against ground truth of its class; only IoU above 0.5 counts as a match.*

The toy shows how PQ reads errors (numbers from this run). The split person still gets a match, because the lower piece covers 56% of it, but the upper piece becomes a false positive: splitting an object costs recognition quality, not just segmentation quality. Car B, at IoU 0.57, counts as fully recognized, but its low IoU pulls the car's SQ down. A missed car and an invented car each cost half a segment in the denominator.

### Early panoptic systems: merge two outputs

**Plain version.** The first panoptic systems simply ran a semantic segmenter and an instance segmenter and pasted the results together.

The panoptic paper's own baseline [4] does this with a non-maximum-suppression-like procedure. It sorts the instance masks by confidence, drops low-scoring ones, and walks down the list: each mask loses the pixels already claimed by earlier masks and is kept only if enough of it remains. The surviving thing segments are then combined with the semantic map, and any pixel claimed by both a thing and a stuff class goes to the thing. **Panoptic FPN** [24] then put both tasks into one network: a Mask R-CNN with a feature pyramid network (FPN [22], see DLCV 03) plus a light semantic branch that upsamples and sums all pyramid levels at 1/4 scale. The semantic branch predicts the stuff classes plus a single "other" class for all thing pixels, so the two branches do not fight over objects.

*Online only: Panoptic Segmentation: a reading guide — https://blog.shyandram.dev/en/learn/deep-learning/image-segmentation/papers/panoptic-segmentation-a-reading-guide/*

## Mask classification: one model for all three tasks

**Plain version.** Instead of asking "what class is this pixel?", ask "which regions are there, and what is each region?". Predict a fixed number of candidate masks and one class (or "nothing") for each. A semantic map, a set of instances and a panoptic map are then just different ways of reading the same list.

### MaskFormer

Cheng, Schwing and Kirillov [17] call the FCN formulation **per-pixel classification** and the Mask R-CNN and DETR formulation **mask classification**. Their observation is that mask classification is general enough for semantic segmentation too. **MaskFormer** predicts a set of $N$ probability–mask pairs $\{(p_i, m_i)\}_{i=1}^{N}$, where $p_i$ is a distribution over $K$ classes plus a "no object" label $\emptyset$, and $m_i \in [0,1]^{H \times W}$ is a soft mask. Three modules produce them:

1. A backbone and a **pixel decoder** produce per-pixel embeddings $\mathcal{E}_{\text{pixel}} \in \mathbb{R}^{C_E \times H \times W}$, where $C_E$ is the embedding dimension.
2. A **transformer decoder** turns $N$ learnable queries into $N$ per-segment embeddings by attending to the image features, as in DETR [25].
3. A linear classifier gives $p_i$, and a small MLP gives a mask embedding $\mathcal{E}_{\text{mask}}[:, i] \in \mathbb{R}^{C_E}$. The mask is a dot product followed by a sigmoid,

$$m_i[h, w] = \sigma(\mathcal{E}_{\text{mask}}[:, i]^{\top} \, \mathcal{E}_{\text{pixel}}[:, h, w]),$$

where $\sigma$ is the logistic sigmoid and $(h, w)$ a pixel position.

Training matches predictions to ground-truth segments one to one, as DETR does for boxes (DLCV 03), but with a matching cost built from class probabilities and mask losses instead of boxes. Matched predictions get a cross-entropy class loss and a focal + Dice mask loss; unmatched ones are trained to predict $\emptyset$ [17].

At inference, two rules read the same output in different ways [17]. **Semantic inference** marginalizes over queries, $\hat{y}[h,w] = \arg\max_c \sum_i p_i(c)\, m_i[h,w]$, with $c$ ranging over the $K$ real classes. **Panoptic inference** assigns each pixel to one query, $\arg\max_{i:\, c_i \neq \emptyset} p_i(c_i)\, m_i[h,w]$, where $c_i$ is the most likely class of query $i$, so two queries of the same class give two instances. The snippet runs both rules on a tiny $6 \times 10$ example.

```
import numpy as np

rng = np.random.default_rng(0)
# Ground-truth layout of a tiny 6 x 10 image: sky on top, road below, two cars on the road
region = np.zeros((6, 10), int)               # 0 sky, 1 road, 2 car A, 3 car B
region[2:] = 1
region[3:5, 1:4] = 2
region[3:5, 6:9] = 3

# Pixel decoder output: a C_E = 4 dimensional embedding per pixel (here: noisy one-hot of the
 ↪  region)
E_pixel = 6 * np.eye(4)[region].transpose(2, 0, 1) - 3 + 0.5 * rng.standard_normal((4, 6,
 ↪  10))

# Transformer decoder output for N = 5 queries: a mask embedding and class probabilities
# over K = 3 classes (sky, road, car) plus "no object" (last column)
E_mask = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1], [.2, .2, .2, .2]],
 ↪  float)
p = np.array([[.90, .05, .03, .02],          # query 0: sky
              [.05, .85, .05, .05],          # query 1: road
              [.02, .08, .85, .05],          # query 2: a car
              [.03, .07, .75, .15],          # query 3: another car
              [.05, .05, .05, .85]])         # query 4: nothing

m = 1 / (1 + np.exp(-np.einsum("nc,chw->nhw", E_mask, E_pixel)))   # N masks, sigmoid(dot
 ↪  product)

# Semantic inference: sum over queries of p_i(c) * m_i, then argmax over the K real classes
sem = np.einsum("nk,nhw->khw", p[:, :3], m).argmax(0)
# Panoptic inference: each pixel goes to the query maximizing p_i(c_i) * m_i (no-object
 ↪  queries dropped)
c_i = p.argmax(1)
keep = c_i != 3
score = p[np.arange(5), c_i][:, None, None] * m
seg = np.where(keep[:, None, None], score, -1).argmax(0)
print("semantic (0 sky, 1 road, 2 car):\n", sem)
print("panoptic segment = query index:\n", seg)
print("query classes:", c_i, "-> car segments:", [int(i) for i in np.unique(seg) if c_i[i] ==
 ↪  2])
```

```
semantic (0 sky, 1 road, 2 car):
 [[0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1]
 [1 2 2 2 1 1 2 2 2 1]
 [1 2 2 2 1 1 2 2 2 1]
```

```
 [1 1 1 1 1 1 1 1 1 1]]
panoptic segment = query index:
 [[0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1]
 [1 2 2 2 1 1 3 3 3 1]
 [1 2 2 2 1 1 3 3 3 1]
 [1 1 1 1 1 1 1 1 1 1]]
query classes: [0 1 2 2 3] -> car segments: [2, 3]
```

The semantic map merges both cars into class 2; the panoptic map keeps them apart as segments 2 and 3, from exactly the same network output. The paper reports that MaskFormer is on par with per-pixel classification on Cityscapes (19 classes) and better on datasets with larger vocabularies [17].

### Mask2Former and its successors

**Mask2Former** [26] kept MaskFormer's meta-architecture (backbone, pixel decoder, transformer decoder) and changed the decoder so that it trains faster and segments instances well. Its key component is **masked attention**: each query attends only to the pixels inside the mask it predicted at the previous layer. In layer $l$, with query features $X_l \in \mathbb{R}^{N\times C}$,

$$X_l = \mathrm{softmax}(\mathcal{M}_{l-1} + Q_l K_l^\top) V_l + X_{l-1}, \qquad \mathcal{M}_{l-1}(x,y) = \begin{cases} 0 & \text{if } M_{l-1}(x,y) = 1, \\ -\infty & \text{otherwise,} \end{cases}$$

where $Q_l$ is a linear projection of $X_{l-1}$, $K_l$ and $V_l$ are projections of the image features, and $M_{l-1}$ is the previous layer's mask prediction, resized and binarized at 0.5. Adding $-\infty$ before the softmax gives zero attention weight outside the mask. The authors hypothesize that local features suffice to update a query, and that the slow convergence of earlier models came from cross-attention having to learn to focus on the object. Two further changes are a round-robin use of the pixel decoder's 1/32, 1/16 and 1/8 feature maps in successive decoder layers, for small objects, and computing the mask loss on sampled points instead of full masks, which the paper reports reduces training memory three-fold [26].

*Online only: Mask2Former: a reading guide — https://blog.shyandram.dev/en/learn/deep-learning/image-segmentation/papers/mask2former-a-reading-guide/*

Two later directions build on this design. **OneFormer** [27] trains a *single* model for all three tasks: it derives semantic and instance labels from panoptic annotations, samples one task per training image, and conditions the queries on a "task token", so that one set of weights serves every task. **EoMT** [28] removes the task-specific parts altogether. The authors argue that a large plain ViT with extensive pretraining can learn what the adapter, pixel decoder and transformer decoder supplied; they concatenate learnable queries to the patch tokens before the last ViT blocks, which then process both jointly, and report accuracy similar to the more complex models at much higher speed (up to four times faster with ViT-L).

## Promptable and open-vocabulary segmentation

**Plain version.** The models above segment a fixed list of classes they were trained on. Two newer families relax that list. Promptable models segment *whatever you point at*; open-vocabulary models segment *whatever you name*.

**Promptable segmentation.** The Segment Anything Model (SAM) [29] returns a mask for a point, box or rough-mask prompt, and for an ambiguous point (shirt or person?) it returns several valid masks with confidence scores; it was trained on SA-1B, a dataset of over 1 billion masks on 11 million images built with a model-in-the-loop data engine. SAM 2 [30] extends this to video with a streaming memory, and SAM 3 [31] adds **promptable concept segmentation**: given a short noun phrase or image exemplars, it returns masks and identities for *all* matching instances. These models are described in DLCV 01 and ML 10, and SAM 2 as a tracker in DLCV 07. In terms of this chapter's three tasks: SAM's masks carry **no class**, so on their own they are neither semantic nor panoptic output; a box from a detector plus SAM gives instance masks; and SAM 3's concept prompts make it an open-vocabulary instance segmenter. When you read "zero-shot" results with such models, check where the prompts came from: prompts derived from ground-truth boxes give the model a lot of information.

**Open-vocabulary segmentation.** The key ingredient is a vision–language model such as CLIP [32], whose text encoder maps any class name to an embedding in the same space as image features (ML 10 covers it). Replace a segmenter's fixed classifier weights with text embeddings of the class names and the vocabulary becomes a list you type at test time. **LSeg** [33] does exactly this: a transformer image encoder produces per-pixel embeddings, a text encoder embeds the label names, and training aligns each pixel's embedding with the text embedding of its class, so that unseen labels can be used at test time without retraining. **CAT-Seg** [34] instead builds a **cost volume** of cosine similarities between CLIP image and text embeddings and aggregates it, which the authors find lets them fine-tune CLIP's encoders without overfitting to the training classes, a problem that made most earlier methods freeze CLIP. Zhu and Chen's survey [35] warns that evaluating "unseen" classes is difficult because the image–text pretraining data may already contain them.

## Datasets and benchmarks

**Plain version.** Pixel labels are expensive, so a few public datasets carry the field. Each one fixes a class list, a stuff/thing split and an evaluation protocol.

| Dataset | Content | Scale and classes (from the papers) | Tasks |
|---|---|---|---|
| PASCAL VOC 2012 [36] | everyday objects | 20 object classes + background (as described in [8]) | semantic |

| Dataset | Content | Scale and classes (from the papers) | Tasks |
|---|---|---|---|
| Cityscapes [18] | street scenes from 50 cities | 5,000 finely and 20,000 coarsely annotated images; 30 classes, 19 used for evaluation; 8 with instance labels [4] | semantic, instance, panoptic |
| ADE20K [37] | diverse scenes, objects and parts | 20,210 / 2,000 / 3,000 train / val / test images; benchmark of the top 150 classes (SceneParse150); 100 things + 50 stuff for panoptic [4] | semantic, panoptic |
| COCO [38] + COCO-Stuff [39] | everyday scenes | instance masks for 80 thing classes [24]; COCO-Stuff adds pixel labels for 91 stuff classes to all 164K images of COCO 2017; 80 things + 53 stuff in COCO panoptic [17] | instance, semantic, panoptic |
| Mapillary Vistas [40] | street-level images worldwide | 25k images; 28 stuff + 37 thing classes [4] | semantic, instance, panoptic |

Three facts about these datasets shape every result. First, **annotation is costly and noisy**: the panoptic paper's human-consistency study, in which two annotations of the same images are scored against each other, gives PQ values well below 100 on all three datasets it examines [4]. Second, **class lists are small** compared with the vocabulary of the world, which motivates the open-vocabulary models above. Third, **domains differ**: Cityscapes was recorded in 50 cities, primarily in Germany, and deliberately not in adverse weather [18], so a model trained on it meets different road markings, weather and cameras elsewhere. Bird's-eye-view map segmentation for driving, which predicts labels on a ground-plane grid instead of the image, is covered in DLCV 03B.

## Failure modes and practical advice

**Plain version.** Segmenters fail in predictable places: where structures are thin or small, along boundaries, on rare classes, and on images unlike the training set.

- **Thin and small structures** (poles, wires, distant pedestrians) disappear at large output strides, as the first experiment showed. Use output stride 8 or a decoder with high-resolution skips, and evaluate with per-class IoU, not only mIoU.
- **Boundaries** come out smooth or blocky. Bilinear score upsampling, decoders, point-based refinement such as PointRend [23] and boundary-aware metrics [19] address this.
- **Class imbalance** biases the network toward frequent classes. Weighted cross-entropy, focal or Dice terms, and sampling images that contain rare classes help (see the loss experiment).
- **Context errors**: a region labeled with a class that makes no sense in its surroundings (the boat labeled as a car in [12]). Global context modules and attention reduce them.

- **Fragmented masks and holes**: per-pixel classifiers have no notion of shape. Morphological clean-up of predicted masks (opening, closing, hole filling; see IP 09) is a cheap and common post-processing step.
- **High-resolution inputs** exceed GPU memory. Run on overlapping tiles and blend the scores, the same overlap-tile idea as U-Net's.
- **Domain shift**: performance drops on new cities, weather, sensors or medical scanners. Measure it with a held-out domain, not only a held-out split.

## Modern view

### What the surveys say

**Minaee et al.** [1] review the CNN era of semantic and instance segmentation in ten families: fully convolutional networks, convolutional models with graphical models (such as CRFs), encoder–decoders, multi-scale and pyramid networks, R-CNN-based models for instances, dilated convolutions and the DeepLab family, recurrent networks, attention models, generative and adversarial models, and convolutional models combined with active contours. They also review datasets and reported results. Most families appear in this chapter's sections, with FCN, U-Net, PSPNet, DeepLab and Mask R-CNN as examples.

**Li et al.** [2] organize transformer-based segmentation around one **meta-architecture** with a feature extractor, object queries and a transformer decoder, and classify methods by which component they modify: better backbones, cross-attention designs in the decoder, ways of optimizing the queries, and so on. They also cover video and point-cloud segmentation, foundation-model tuning, efficiency and medical segmentation, and they re-evaluate many methods on common benchmarks.

**Zhou et al.** [3] review more than 300 methods of the foundation-model era, splitting the field into **generic** segmentation (semantic, instance, panoptic) and **promptable** segmentation (interactive, referring and few-shot). A central theme is that segmentation knowledge *emerges* in models trained without masks, such as CLIP, text-to-image diffusion models and self-supervised ViTs, which training-free methods exploit. **Zhu and Chen** [35] focus on open-vocabulary detection and segmentation and on how hard fair evaluation of "novel" classes is.

### The state of the art, 2023–2026

- **Universal query-based architectures** are the default for closed-vocabulary benchmarks: Mask2Former [26] and its descendants such as OneFormer [27] handle semantic, instance and panoptic segmentation with one design, and specialized per-pixel models remain common where speed or simplicity matters (SegFormer [14], DeepLab-style models [7]).
- **Simpler models on larger pretraining**: EoMT [28] argues that compute is better spent on scaling a plain ViT encoder than on adding decoders and adapters.
- **Promptable foundation models** (SAM [29], SAM 2 [30], SAM 3 [31]) turned class-agnostic

masks, video propagation and concept prompts into reusable components, and their data engines changed how segmentation datasets are built.
- **Open-vocabulary segmentation** builds on vision–language models (LSeg [33], CAT-Seg [34]); in many applications a model is now prompted with class names instead of retrained.

### Open problems

1. **Boundaries, small objects and thin structures.** Mask2Former's authors state that it still struggles with small objects [26], and region metrics hide boundary errors on large objects [19].
2. **One model, trained once.** Universal architectures are still often trained separately per task and dataset; Mask2Former trained only on panoptic labels is slightly worse on the instance and semantic metrics than when trained on each task [26].
3. **Open-vocabulary evaluation** depends on prompts and on what the pretraining data contained [35].
4. **Data.** Pixel labels are expensive; the surveys point to scalable data engines, synthetic image–mask pairs from generative models and weak or no supervision as directions, each with its own domain gap [3], [2].
5. **Efficiency.** Foundation-model segmenters are heavy; distillation, compression and parameter-efficient tuning are open engineering problems [3].
6. **Understanding emergence.** Why models trained without masks learn to segment is not well explained [3].

## Key takeaways

- Define whether the output needs classes, object identities or both.
- Output stride limits spatial detail; upsample scores before assigning labels.
- Skip-connected decoders restore detail, while dilation, pooling and attention provide context.
- Cross-entropy, focal and Dice losses weight errors differently; choose them for the task's imbalance and metric.
- mIoU, mask AP and panoptic quality evaluate different output structures and are not interchangeable.
- Mask classification can support several tasks; prompt and vocabulary interfaces change what must be evaluated.

## Exercises

1. **Output stride by hand.** A road scene is 1024 × 2048 pixels and a pole is 6 pixels wide. With ideal coarse scores (class fractions per cell) and argmax decisions, at which output strides can the pole survive at all, and why does its position relative to the grid matter? Check your answer by moving the bar in the first snippet by one or two pixels.

**Hint**

The pole wins a cell only if it covers more than half of the cell's width, so $s < 12$; with $s = 8$ it survives only in cells where at least 5 of its 6 columns fall inside one 8-pixel cell. At $s = 4$ it always covers at least one full cell. Shifting the bar changes which cells it dominates, which is why the bar's stride-4 IoU in the nearest-label column of the snippet is 0.67, not 1 (a tie at exactly half a cell goes to the background, the first class).

2. **Atrous arithmetic.** (a) What is the effective size of a $3 \times 3$ kernel at rate 12? (b) A backbone stage often starts with stride 2 followed by $3 \times 3$ convolutions. If you remove the stride and use rate 2 instead, what happens to the output stride, the receptive field of the following layers, and the cost of those layers?

**Hint**

(a) $k_e = 3 + 2 \cdot 11 = 25$. (b) The output stride halves (for example 32 to 16), every later layer covers the same image region as before (the rate compensates for the denser grid), and the later layers now run on four times as many positions, so their cost grows about four-fold.

3. **Losses with an empty mask.** Modify the loss snippet so that the image contains *no* foreground. What does each loss return for the "all background (p=0.05)" prediction, and what happens to the soft Dice loss without the `eps` term? How do implementations usually avoid the problem?

**Hint**

With $g = 0$ everywhere, the Dice numerator is 0 (plus `eps`) and the denominator is $\sum p_i = 204.8$, so the loss is almost 1 however good the prediction is; for $p = 0$ everywhere it is $0/0$ without `eps`, and with `eps` it becomes $1 - \text{eps}/\text{eps} = 0$. `iou_at_half` also divides 0 by 0 when neither mask has a pixel. Common fixes: add a smoothing constant to numerator and denominator, compute Dice over the whole batch instead of per image, or combine Dice with cross-entropy.

4. **PQ by hand.** In the PQ snippet, replace the two person pieces by a single prediction rows 32 to 80. Predict the new person SQ, RQ and PQ before running the code. Then make car B larger so that its IoU drops below 0.5 by over-segmenting; how do the car's TP, FP and FN counts change?

**Hint**

The single person prediction has IoU $48 \cdot 20/(50 \cdot 20) = 0.96$, so SQ = 0.96, RQ = 1 and PQ = 0.96. If car B's IoU falls to 0.5 or below, that pair is no longer a match: car TP drops from 2 to 1, and FP and FN each rise by one.

5. **Semantic from panoptic, and back.** Using the mask-classification snippet, (a) derive the semantic map from the panoptic segment map and check it equals `sem`; (b) make query 3 predict "road" instead of "car" with probability 0.6 and explain both outputs. (c) Why can a semantic map not be turned back into a panoptic map in general?

**Hint**

(a) Map each segment id to its class, `c_i[seg]`. (b) Query 3's mask still covers car B, so in the panoptic map car B stays segment 3, but that segment is now labeled road; because a stuff class has a single segment per image, the final output merges it with the road. In the semantic map car B becomes road too. Only one car is left in both. (c) A semantic map has no instance ids: touching objects of one class form one region, and splitting it needs information the map does not contain.

## References


1. S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz and D. Terzopoulos, "Image segmentation using deep learning: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 7, 2022. doi ·arXiv
2. X. Li, H. Ding, H. Yuan, W. Zhang, J. Pang, G. Cheng, K. Chen, Z. Liu and C. C. Loy, "Transformer-based visual segmentation: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 12, 2024. doi ·arXiv
3. T. Zhou, W. Xia, F. Zhang, B. Chang, W. Wang, Y. Yuan, E. Konukoglu and D. Cremers, "Image segmentation in foundation model era: A survey," arXiv:2408.12957, 2024. arXiv
4. A. Kirillov, K. He, R. Girshick, C. Rother and P. Dollár, "Panoptic segmentation," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2019. doi ·arXiv
5. J. Long, E. Shelhamer and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2015. doi · arXiv
6. O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention* (*MICCAI*), LNCS vol. 9351, 2015. doi
7. L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *Proc. European Conference on Computer Vision* (*ECCV*), 2018. doi ·arXiv
8. L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy and A. L. Yuille, "DeepLab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, 2018. doi · arXiv
9. F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," in *Proc. International Conference on Learning Representations* (*ICLR*), 2016. arXiv
10. L.-C. Chen, G. Papandreou, F. Schroff and H. Adam, "Rethinking atrous convolution for semantic image segmentation," arXiv:1706.05587, 2017. arXiv
11. P. Wang, P. Chen, Y. Yuan, D. Liu, Z. Huang, X. Hou and G. Cottrell, "Understanding convolution for semantic segmentation," in *Proc. IEEE Winter Conference on Applications of Computer Vision* (*WACV*), 2018. doi ·arXiv
12. H. Zhao, J. Shi, X. Qi, X. Wang and J. Jia, "Pyramid scene parsing network," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2017. doi ·arXiv
13. A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner et al., "An image is worth 16x16 words: Transformers for image recognition at scale," in *Proc. International Conference on Learning Representations* (*ICLR*), 2021. arXiv

14. E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez and P. Luo, "SegFormer: Simple and efficient design for semantic segmentation with transformers," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2021. NeurIPS ·arXiv
15. F. Milletari, N. Navab and S.-A. Ahmadi, "V-Net: Fully convolutional neural networks for volumetric medical image segmentation," in *Proc. Fourth International Conference on 3D Vision* (*3DV*), 2016. doi ·arXiv
16. T.-Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár, "Focal loss for dense object detection," in *Proc. IEEE International Conference on Computer Vision* (*ICCV*), 2017. doi
17. B. Cheng, A. G. Schwing and A. Kirillov, "Per-pixel classification is not all you need for semantic segmentation," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2021. NeurIPS · arXiv
18. M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth and B. Schiele, "The Cityscapes dataset for semantic urban scene understanding," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2016. doi ·arXiv
19. B. Cheng, R. Girshick, P. Dollár, A. C. Berg and A. Kirillov, "Boundary IoU: Improving object-centric image segmentation evaluation," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2021. doi ·arXiv
20. K. He, G. Gkioxari, P. Dollár and R. Girshick, "Mask R-CNN," in *Proc. IEEE International Conference on Computer Vision* (*ICCV*), 2017. doi ·arXiv
21. S. Ren, K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, 2017. doi
22. T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan and S. Belongie, "Feature pyramid networks for object detection," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2017. doi
23. A. Kirillov, Y. Wu, K. He and R. Girshick, "PointRend: Image segmentation as rendering," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2020. doi ·arXiv
24. A. Kirillov, R. Girshick, K. He and P. Dollár, "Panoptic feature pyramid networks," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2019. doi ·arXiv
25. N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov and S. Zagoruyko, "End-to-end object detection with transformers," in *Proc. European Conference on Computer Vision* (*ECCV*), 2020. doi ·arXiv
26. B. Cheng, I. Misra, A. G. Schwing, A. Kirillov and R. Girdhar, "Masked-attention mask transformer for universal image segmentation," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2022. doi ·arXiv
27. J. Jain, J. Li, M. Chiu, A. Hassani, N. Orlov and H. Shi, "OneFormer: One transformer to rule universal image segmentation," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2023. doi ·arXiv
28. T. Kerssies, N. Cavagnero, A. Hermans, N. Norouzi, G. Averta, B. Leibe, G. Dubbelman and D. de Geus, "Your ViT is secretly an image segmentation model," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2025. CVF ·arXiv
29. A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson et al., "Segment anything," in *Proc. IEEE/CVF International Conference on Computer Vision* (*ICCV*), 2023. doi ·arXiv
30. N. Ravi, V. Gabeur, Y.-T. Hu, R. Hu, C. Ryali, T. Ma et al., "SAM 2: Segment anything in images

and videos," in *Proc. International Conference on Learning Representations* (*ICLR*), 2025. OpenReview ·arXiv
31. N. Carion, L. Gustafson, Y.-T. Hu, S. Debnath, R. Hu, D. Suris et al., "SAM 3: Segment anything with concepts," in *Proc. International Conference on Learning Representations* (*ICLR*), 2026. OpenReview ·arXiv
32. A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal et al., "Learning transferable visual models from natural language supervision," in *Proc. International Conference on Machine Learning* (*ICML*), PMLR 139, 2021. PMLR
33. B. Li, K. Q. Weinberger, S. Belongie, V. Koltun and R. Ranftl, "Language-driven semantic segmentation," in *Proc. International Conference on Learning Representations* (*ICLR*), 2022. OpenReview ·arXiv
34. S. Cho, H. Shin, S. Hong, A. Arnab, P. H. Seo and S. Kim, "CAT-Seg: Cost aggregation for open-vocabulary semantic segmentation," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2024. doi ·arXiv
35. C. Zhu and L. Chen, "A survey on open-vocabulary detection and segmentation: Past, present, and future," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 12, 2024. doi
36. M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn and A. Zisserman, "The PASCAL Visual Object Classes (VOC) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, 2010. doi
37. B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso and A. Torralba, "Scene parsing through ADE20K dataset," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2017. doi ·CVF
38. T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan et al., "Microsoft COCO: Common objects in context," in *Proc. European Conference on Computer Vision* (*ECCV*), 2014. doi · arXiv
39. H. Caesar, J. Uijlings and V. Ferrari, "COCO-Stuff: Thing and stuff classes in context," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2018. doi ·arXiv
40. G. Neuhold, T. Ollmann, S. Rota Bulò and P. Kontschieder, "The Mapillary Vistas dataset for semantic understanding of street scenes," in *Proc. IEEE International Conference on Computer Vision* (*ICCV*), 2017. doi
41. P. Krähenbühl and V. Koltun, "Efficient inference in fully connected CRFs with Gaussian edge potentials," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2011. NeurIPS · arXiv

# DLCV 05 • Human Pose and Keypoint Estimation

How deep networks find the joints of every person in an image: skeletons and visibility, heatmaps versus regression and sub-pixel decoding, top-down and bottom-up multi-person pipelines, OKS-based AP and PCK, 3D pose from lifting, triangulation and body models, datasets and failure modes.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/pose-keypoints/

## What you'll learn

- Represent pose as named keypoints, visibility labels and skeleton connections.
- Compare coordinate regression and heatmaps, including sub-pixel decoding.
- Choose top-down, bottom-up or direct multi-person prediction.
- Interpret OKS-based AP, PCK/PCKh and 3D joint error.
- Understand 3D lifting, triangulation and body-model fitting, then diagnose common errors.

## The big picture

A detector puts a box around each person (DLCV 03). A box says *where* a person is, but not *what they are doing*: standing, waving, falling or kicking a ball all give similar boxes. **Pose estimation** goes one step further. It finds a small set of meaningful points on the body, the **keypoints** (nose, shoulders, elbows, wrists, hips, knees, ankles and so on), and connects them into a stick figure.

> **Explain like I'm five**
>
> Remember connect-the-dots pictures? Pose estimation is the opposite game. The computer looks at a photo of a person and has to *put the dots back*: one dot on each shoulder, elbow, wrist, hip, knee and ankle, each with the right name. Then it draws lines between the dots and you get a stick figure that moves the way the person moves. The hard parts are the ones you would also find hard: a hand hidden behind someone's back, two friends hugging so you cannot tell whose arm is whose, and remembering which side is *their* left.

**Why it matters.** A stick figure is a compact summary of what a body is doing. Sports analysis, physical therapy, fall detection and fitness apps measure joint angles from it. Animation, games and augmented reality drive avatars with it. Robots and cars use it to anticipate where people will move. Neuroscientists track the joints of mice and flies without attaching markers [1]. And the same ideas, heatmaps, keypoint grouping and keypoint metrics, are reused for faces, hands, animals and object parts.

This chapter builds on DLCV 03 for person detection, non-maximum suppression (NMS) and the AP machinery, and on DLCV 04 for dense prediction and upsampling; we link to them instead

of repeating them. Tracking poses over time uses the association methods of DLCV 07, and 3D triangulation uses the camera geometry of DLCV 11. Two surveys frame the field: Zheng et al. [2] cover 2D and 3D pose with deep learning, and Chen et al. [3] organize monocular methods by task. We return to both in the Modern view.

## What a pose is

**Plain version.** A pose consists of named body points, their coordinates and visibility labels, connected by a predefined skeleton.

**Precise version.** A **skeleton** is a graph $G = (V, E)$. The vertices $V = \{1, \dots, K\}$ are the $K$ keypoint *types* (for example "left wrist"), and the edges $E \subset V \times V$ are the **limbs** that connect them. A 2D pose of one person is

$$P = \{(\mathbf{x}_k, v_k)\}_{k=1}^{K}, \qquad \mathbf{x}_k = (x_k, y_k) \in \mathbb{R}^2,$$

where $\mathbf{x}_k$ is the image position of keypoint $k$ in pixels and $v_k$ is its **visibility flag**. The COCO keypoint format uses $v_k = 0$ for "not labeled", $v_k = 1$ for "labeled but not visible" (occluded, position guessed by the annotator) and $v_k = 2$ for "labeled and visible" [4]. A 3D pose replaces $\mathbf{x}_k$ by a point $\mathbf{X}_k \in \mathbb{R}^3$, usually in millimetres in a camera or world frame.

The COCO person skeleton has $K = 17$ keypoints: the nose, both eyes, ears, shoulders, elbows, wrists, hips, knees and ankles [4]. Figure 1 shows them on a synthetic stick figure. Two conventions trip up every beginner:

- **Left and right are the person's, not the viewer's.** A person facing the camera has their left shoulder on the *right* side of the image. Flipping an image horizontally therefore turns every left keypoint into a right one, and data augmentation must swap the labels as well as mirror the coordinates.
- **Not visible is not the same as not labeled.** An occluded wrist ($v = 1$) still has a position and is still evaluated; an unlabeled keypoint ($v = 0$) is simply ignored.

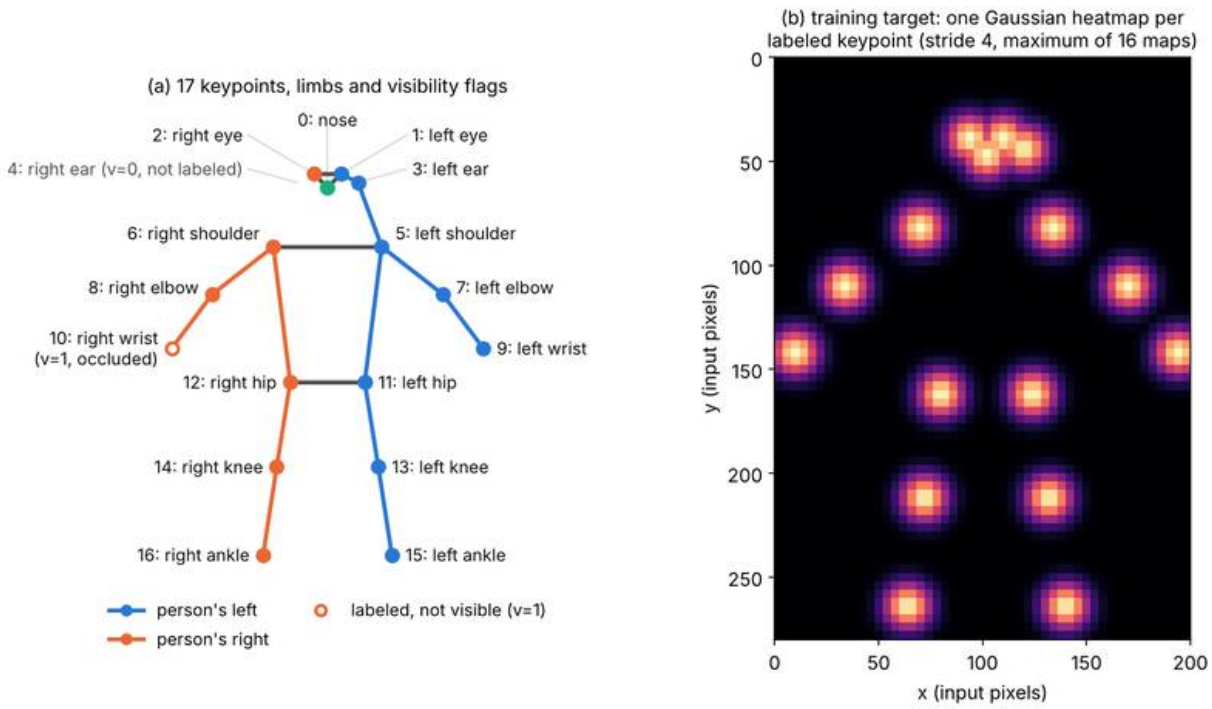



*Figure 1 — (a) The 17-keypoint COCO-style skeleton on a synthetic person, with the person's left in blue and right in orange, and the two non-visible cases. (b) The corresponding training target of a heatmap network: one Gaussian blob per labeled keypoint at output stride 4 (the 16 maps are shown as their pixel-wise maximum).*

### Pose estimation versus its neighbours

Four related tasks are easy to confuse:

| Task | Output | Where in this course |
|---|---|---|
| Generic keypoint detection | Corners and blobs that are easy to find again in another image, with descriptors; no semantic name | IP 12 |
| Object detection | One box, class and score per object | DLCV 03 |
| Instance segmentation | One pixel mask per object | DLCV 04 |
| Pose estimation | One set of *named* keypoints per object, connected by a fixed skeleton | this chapter |

The keypoints of IP 12 are chosen *by the image*: wherever a corner or blob happens to be. The keypoints here are chosen *by the task*: the left elbow must be found even when it lies on a smooth sleeve with no corner at all. That is why semantic keypoints are learned from labeled examples rather than detected by a formula.

The task comes in several flavors. **Single-person 2D** estimation assumes one person per input crop. **Multi-person 2D** estimation must also decide how many people there are and which keypoint belongs to whom. **3D** estimation recovers depth. **Whole-body** estimation adds feet, hands and face landmarks, and **mesh recovery** outputs a full body surface instead of a sparse skeleton [5].

## Two ways to output a keypoint

**Plain version.** A network can either *say the numbers* ("the wrist is at $x = 140$, $y = 128$"), which is regression, or *paint a picture* that glows where the wrist is, which is a heatmap. For most of the deep-learning era, painting won, because convolutional networks are better at painting than at saying numbers.

### Direct regression

DeepPose [6] was an early deep pose estimator built on regression. A convolutional network maps the image to a vector of $2K$ numbers, the keypoint coordinates normalized by the person's box, and is trained with a squared error,

$$\mathcal{L}_{\text{reg}} = \sum_{k=1}^{K} \|\hat{\mathbf{x}}_k - \mathbf{x}_k\|_2^2,$$

where $\hat{\mathbf{x}}_k$ is the predicted and $\mathbf{x}_k$ the ground-truth position of keypoint $k$ (in normalized coordinates). DeepPose refined its first guess with a cascade of further regressors applied to higher-resolution crops around each joint.

Regression is simple, fast and has no resolution limit, but the fully connected output throws away the spatial layout that convolutions are good at, and for years regression trailed heatmaps on accuracy.

### Gaussian heatmaps

Tompson et al. [7] instead asked the network to output one image per keypoint type, a **heatmap** that is high near the keypoint and low elsewhere. The target for keypoint $k$ is a Gaussian blob centred on the true position:

$$H_k^*(\mathbf{p}) = \exp\!\left(-\frac{\|\mathbf{p} - \mathbf{x}_k/s\|_2^2}{2\sigma^2}\right),$$

where $\mathbf{p}$ is a heatmap cell (integer coordinates), $s$ is the **output stride** (how many input pixels one heatmap cell covers, typically 4), $\mathbf{x}_k/s$ is the true keypoint position in heatmap units and $\sigma$ is the blob's width in cells. The network is trained with a pixel-wise mean squared error between the predicted heatmaps $\hat{H}_k$ and the targets,

$$\mathcal{L}_{\text{hm}} = \frac{1}{K\,|\Omega|} \sum_{k=1}^{K} \sum_{\mathbf{p}\in\Omega} w_k \big(\hat{H}_k(\mathbf{p}) - H_k^*(\mathbf{p})\big)^2,$$

where $\Omega$ is the set of heatmap cells and $w_k$ is 0 for unlabeled keypoints and 1 otherwise. The stacked hourglass network uses this supervision, following Tompson et al., with $\sigma = 1$ pixel [8].

Why does this work so much better than regression? A heatmap **matches what convolutions do**: it is dense and translation-equivariant (if the person moves, the blob moves), so every output cell answers a local question, "is there a wrist *here*?". It **can express doubt**: two plausible wrist positions become two weaker blobs, where a single regressed coordinate would land in between. And the peak height is a **free confidence score** for each keypoint.

The price is resolution: at stride 4 the peak can only sit on a grid of spacing 4 pixels, and $K$ maps cost far more memory than $2K$ numbers. How dense prediction networks recover resolution is covered in DLCV 04; the next section shows how much the grid costs and how to get most of it back.

### Decoding a heatmap: argmax, quantization error and sub-pixel refinement

**Plain version.** The brightest heatmap cell gives a coarse location. Neighboring responses help refine it between grid points.

The simplest decoder takes the brightest cell and maps it back to the image:

$$\hat{\mathbf{x}}_k = s \cdot \arg\max_{\mathbf{p}\in\Omega} \hat{H}_k(\mathbf{p}).$$

If the true position falls uniformly at random inside a cell, the error of this decoder in each axis is uniform on $[-s/2, s/2]$, and the expected Euclidean error is about $0.383\,s$ (the mean distance from the centre of a unit square to a random point in it, $(\sqrt{2} + \ln(1+\sqrt{2}))/6 \approx 0.383$, scaled by $s$). At stride 4 that is about 1.5 pixels of error *before the network makes any mistake*. On a person 100 pixels tall, that is enough to lose points on the strict thresholds of the OKS metric introduced later.

Three refinements are common:

- **The quarter offset.** Move the estimate a quarter of a cell from the peak toward its higher neighbour. The stacked hourglass paper added this to improve results at high precision thresholds [8], and SimpleBaseline uses it as standard practice [9].
- **A Taylor step on the log-heatmap.** The logarithm of a Gaussian is a quadratic (a parabola), so its maximum can be found exactly from the value, slope and curvature at the brightest cell: $\hat{\mathbf{p}} = \mathbf{m} - \mathbf{D}''(\mathbf{m})^{-1}\mathbf{D}'(\mathbf{m})$, where $\mathbf{m}$ is the brightest cell, and $\mathbf{D}'$ and $\mathbf{D}''$ are the gradient and Hessian of $\log \hat{H}_k$ there, estimated by finite differences. This is the decoding step of DARK [10]. DARK also smooths the predicted heatmap with a Gaussian before decoding, because real predictions are not perfect Gaussians, and centres the training targets on the exact (unrounded) keypoint position, as our formula for $H_k^*$ already does.
- **Soft-argmax (integral regression).** Treat the heatmap as a probability distribution and output its *mean*: $\hat{\mathbf{x}}_k = s \sum_{\mathbf{p}} \mathbf{p}\, \tilde{H}_k(\mathbf{p})$, with $\tilde{H}_k = \text{softmax}(\hat{H}_k)$ over all cells. Sun et al. [11] showed that

this turns the heatmap into a differentiable regression output, so the network can be trained end to end on the coordinates, without quantization.

DARK's ablation makes the size of the effect concrete: for an HRNet-W32 model with a small $128 \times 96$ input on COCO validation, AP was 61.2 with plain argmax, 66.9 with the quarter offset and 68.4 with its Taylor decoding [10]. The decoder alone moved AP by more than 7 points.

*Online only: Sub-pixel decoding by a Taylor step — https://blog.shyandram.dev/en/learn/deep-learning/pose-keypoints/notes/sub-pixel-decoding-by-a-taylor-step/*

The code encodes 500 random keypoints into heatmaps at strides 1, 2, 4 and 8, adds a little noise to imitate an imperfect prediction, and decodes them with each method. All numbers come from this synthetic run.

```
import numpy as np

def encode(xy, shape, stride, sigma=2.0):
    """Gaussian heatmap of size shape=(H, W) cells for a keypoint at image position xy."""
    cx, cy = xy[0] / stride, xy[1] / stride           # exact (unrounded) center in cells
    ys, xs = np.mgrid[0:shape[0], 0:shape[1]]
    return np.exp(-((xs - cx) ** 2 + (ys - cy) ** 2) / (2 * sigma ** 2))

def peak(h):
    y, x = np.unravel_index(np.argmax(h), h.shape)
    return x, y

def decode_argmax(h, stride):
    x, y = peak(h)
    return np.array([x, y], float) * stride

def decode_quarter(h, stride):
    """Move a quarter cell from the peak toward the higher neighbour on each axis."""
    x, y = peak(h)
    p = np.array([x, y], float)
    if 0 < x < h.shape[1] - 1: p[0] += 0.25 * np.sign(h[y, x + 1] - h[y, x - 1])
    if 0 < y < h.shape[0] - 1: p[1] += 0.25 * np.sign(h[y + 1, x] - h[y - 1, x])
    return p * stride

def decode_taylor(h, stride):
    """One Newton step on log h at the peak: offset = -Hessian^-1 * gradient."""
    x, y = peak(h)
    p = np.array([x, y], float)
    if 1 <= x < h.shape[1] - 1 and 1 <= y < h.shape[0] - 1:
        L = np.log(np.maximum(h[y - 1:y + 2, x - 1:x + 2], 1e-10))
        g = np.array([L[1, 2] - L[1, 0], L[2, 1] - L[0, 1]]) / 2
        hxx = L[1, 2] - 2 * L[1, 1] + L[1, 0]
        hyy = L[2, 1] - 2 * L[1, 1] + L[0, 1]
        hxy = (L[2, 2] - L[2, 0] - L[0, 2] + L[0, 0]) / 4
        Hs = np.array([[hxx, hxy], [hxy, hyy]])
        if np.linalg.det(Hs) > 0 and hxx < 0:          # a proper maximum
            p += -np.linalg.solve(Hs, g)
    return p * stride

def decode_soft(logits, stride):
```

```
    """Soft-argmax: the expected position under softmax(logits)."""
    w = np.exp(logits - logits.max()); w /= w.sum()
    ys, xs = np.mgrid[0:w.shape[0], 0:w.shape[1]]
    return np.array([(w * xs).sum(), (w * ys).sum()]) * stride

rng = np.random.default_rng(0)
W, H = 192, 256                                  # input crop size in pixels (width,
 ↪  height)
pts = rng.uniform([16, 16], [W - 16, H - 16], (500, 2))

def mean_error(decoder, s, noise=0.01, use_log=False):
    shape, e = (H // s, W // s), 0.0
    for xy in pts:
        h = encode(xy, shape, s)
        h = np.clip(h + rng.normal(0, noise, h.shape), 1e-6, None)   # an imperfect
         ↪  "prediction"
        e += np.linalg.norm(decoder(np.log(h) if use_log else h, s) - xy)
    return e / len(pts)

print("stride  argmax  quarter  taylor   (mean error in input pixels)")
for s in (1, 2, 4, 8):
    print(f"{s:6d}  {mean_error(decode_argmax, s):6.2f}  {mean_error(decode_quarter,
     ↪  s):7.2f}"
          f"  {mean_error(decode_taylor, s):6.2f}")
print(f"soft-argmax, stride 4: clean map {mean_error(decode_soft, 4, 0.0, True):.2f},"
      f" noisy map {mean_error(decode_soft, 4, 0.01, True):.2f}")
```

```
stride  argmax  quarter  taylor   (mean error in input pixels)
     1    0.39     0.20    0.05
     2    0.80     0.40    0.11
     4    1.54     0.78    0.21
     8    3.20     1.59    0.43
soft-argmax, stride 4: clean map 0.04, noisy map 24.42
```

Four lessons are visible, and Figure 2 plots the first three.

1. **Argmax error grows linearly with the stride** and follows the $0.383\,s$ prediction closely (0.38, 0.77, 1.53, 3.06 for strides 1 to 8).
2. **The quarter offset halves it**, at no cost.
3. **The Taylor step removes most of what remains**, because the log of a Gaussian really is a parabola; what is left comes from the added noise.
4. **Soft-argmax is exact on a clean map but fails badly on a noisy one.** With logits equal to $\log \hat{H}$, the softmax weights are proportional to $\hat{H}$, and a small noise floor spread over thousands of cells outweighs the narrow peak and pulls the mean toward the centre of the map. Integral regression trains the network *through* the soft-argmax, which pushes it to put almost no mass away from the keypoint; applied after the fact to a heatmap trained with MSE, soft-argmax is a poor decoder.

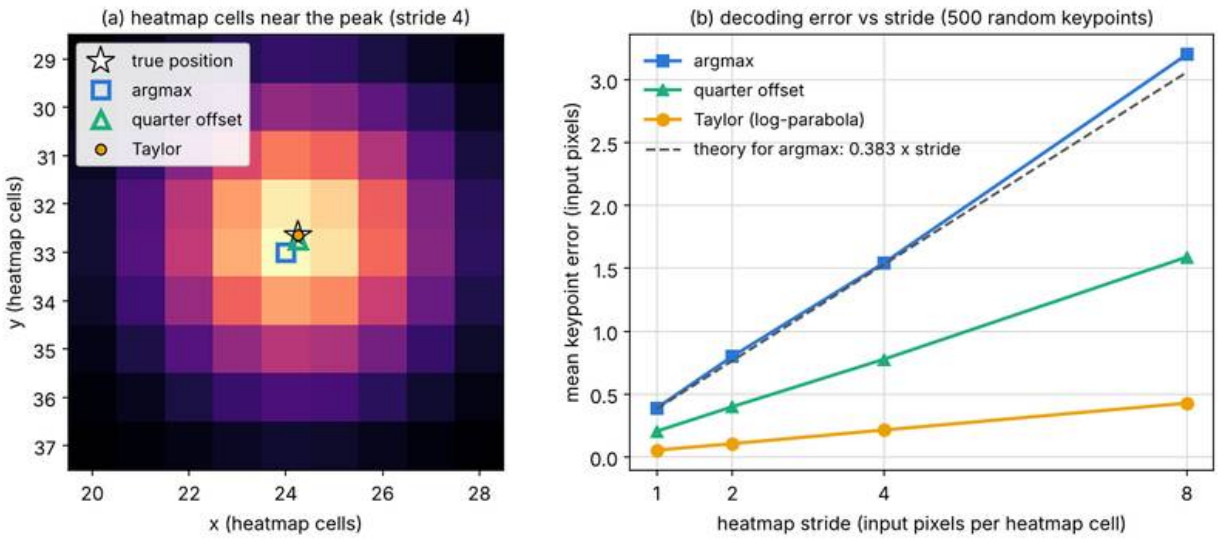



*Figure 2 — Decoding a heatmap. (a) Near the peak at stride 4, the brightest cell (square) misses the true position (star); the quarter offset (triangle) moves partway; the Taylor step (dot) lands on it. (b) Mean error of each decoder over 500 random keypoints in the synthetic run of the code above; the dashed line is the argmax prediction of 0.383 times the stride.*

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

### Other output representations

Two newer representations sit between heatmaps and plain regression. **Coordinate classification** (SimCC [12]) predicts two one-dimensional distributions per keypoint, over $x$ and over $y$, with several bins per pixel, which cuts quantization error without a large 2D map. **Likelihood-based regression** (RLE [13]) keeps direct regression but learns the shape of the error distribution during training, and its paper reports regression beating heatmap methods on multi-person pose. Both are the natural outputs for the one-stage and query-based models later in this chapter.

*Online only: Coordinate classification and likelihood-based regression — https://blog.shyandram.dev/en/learn/deep-learning/pose-keypoints/notes/coordinate-classification-and-likelihood-based-regression/*

## Networks for single-person heatmaps

**Plain version.** Precise localization needs high-resolution detail; identifying the correct joint also needs context from the body.

The input is a **person crop**: a box around one person, enlarged a little, padded to a fixed aspect ratio and resized to a fixed size such as 256 × 192 (height × width). The output is $K$ heatmaps at stride 4, 64 × 48 in this case.

- **Convolutional Pose Machines (CPM)** [14] stack several stages. Each stage sees the image features *and* the previous stage's heatmaps (belief maps), so later stages can use the positions of other joints as context, for example "an elbow lies between a shoulder and a wrist". Each stage is supervised (**intermediate supervision**), which counters vanishing gradients in a deep stack.
- **Stacked hourglass networks** [8] repeat a symmetric encoder–decoder ("hourglass") module several times, with intermediate supervision after each one. Every hourglass pools down to a very coarse resolution to gather whole-body context and upsamples back with skip connections, the same encoder–decoder idea described for segmentation in DLCV 04.
- **SimpleBaseline** [9] showed that much of this machinery was unnecessary. It takes an ImageNet-pretrained ResNet (DLCV 02) and adds three deconvolution (transposed convolution) layers on top of its last stage to produce $64 \times 48$ heatmaps for a $256 \times 192$ input. Its ablation found that three deconvolution layers (heatmaps at $64 \times 48$) beat two (at $32 \times 24$) by 2.5 AP, a direct measurement of the resolution cost discussed above.
- **HRNet** [15] keeps a high-resolution branch alive through the whole network instead of recovering resolution at the end. It adds lower-resolution branches one by one, runs them **in parallel** and repeatedly exchanges information between all resolutions (multi-scale fusion). The heatmaps are read from the high-resolution branch, which has seen coarse context many times. Several later methods in this chapter build on HRNet: DARK reports its decoding results with HRNet models, and HigherHRNet and DEKR extend it to bottom-up pose.
- **ViTPose** [16] replaces the convolutional backbone by a plain, non-hierarchical vision transformer (ML 05) with $16 \times 16$ patches, followed by a lightweight decoder of a few deconvolution layers (or even a single upsampling layer and a convolution). The paper scales it from 100M to 1B parameters and shows that simple design plus large pretrained transformers sets a new accuracy–throughput trade-off.

*Online only: Simple Baselines (Xiao et al., 2018) — https://blog.shyandram.dev/en/learn/deep-learning/pose-keypoints/papers/simple-baselines-xiao-et-al-2018/*

Two training details matter for every model in this list. **Data augmentation** rotates, scales and flips the crops; a horizontal flip must swap the left and right keypoint channels. **Flip testing** runs the model on the crop and on its mirror image, swaps the channels of the mirrored output back, and averages the heatmaps; the stacked hourglass paper reports about a 1% average improvement on validation from this [8], and later models adopted it as standard practice [9].

*Online only: Flip testing and left–right bookkeeping — https://blog.shyandram.dev/en/learn/deep-learning/pose-keypoints/notes/flip-testing-and-leftright-bookkeeping/*

## Many people: top-down versus bottom-up

**Plain version.** With several people in a photo there are two orders of work. **Top-down**: first find each person, then find the joints inside each person's box. **Bottom-up**: first find every joint in the picture, then work out which joints belong together. A third family does both at once.

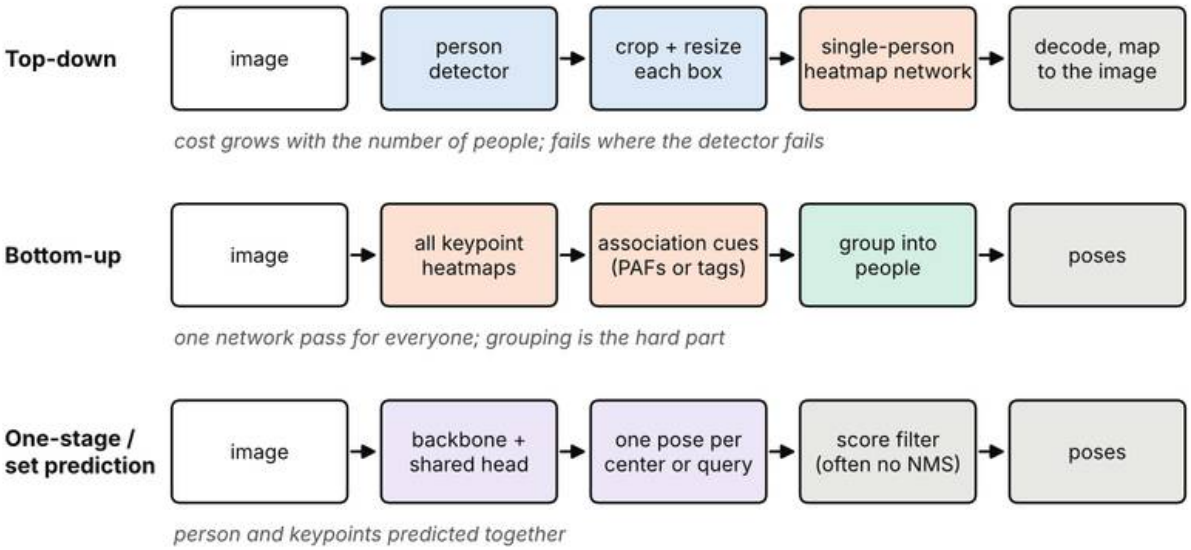



*Figure 3 — Three multi-person pipelines. Top-down runs a single-person model per detected box; bottom-up finds all keypoints at once and groups them; one-stage and set-prediction models output a whole pose per person in one pass.*

### Top-down: detect, then estimate

A person detector (DLCV 03) proposes boxes; each box is cropped, resized and passed to a single-person network such as SimpleBaseline, HRNet or ViTPose; the heatmaps are decoded and the keypoints mapped back to image coordinates. Mask R-CNN [17] builds the same idea into the detector: its keypoint branch predicts, inside each region of interest, $K$ masks of size $m \times m$, each trained as an $m^2$-way softmax whose target is a single "one-hot" pixel at the keypoint (see DLCV 03 for the RoI machinery).

Top-down methods are accurate because every person is resized to the same comfortable scale before the keypoint network sees them, which helps most for small and medium-sized people [18]. Their weaknesses follow from the pipeline:

- **Cost grows with the number of people**, since the keypoint network runs once per box.
- **They fail where the detector fails.** A missed person gets no pose; NMS (DLCV 03) may delete one of two overlapping people; and a box that contains two people gives the single-person network two left wrists to choose from. The OpenPose authors measured this on a COCO validation subset: the same single-person model scored 62.7 AP with ground-truth boxes but 52.7 AP with boxes from an SSD detector [18].

### Bottom-up: detect all keypoints, then group

A bottom-up model runs once on the whole image and outputs $K$ heatmaps that contain the keypoints of *all* people. Peaks are found with a local-maximum test, which is a per-keypoint NMS. The hard part is **grouping**: deciding which left elbow goes with which left wrist.

**Part affinity fields (PAFs).** OpenPose [18] predicts, besides the heatmaps, one 2D vector field per limb type. At a pixel on limb $c$ of person $k$, the target is the unit vector pointing along that limb; elsewhere it is zero:

$$\mathbf{L}^{*}_{c,k}(\mathbf{p}) = \begin{cases} \mathbf{v} = \dfrac{\mathbf{x}_{j_2,k} - \mathbf{x}_{j_1,k}}{\|\mathbf{x}_{j_2,k} - \mathbf{x}_{j_1,k}\|_2} & \text{if } \mathbf{p} \text{ lies on limb } c \text{ of person } k, \\ \mathbf{0} & \text{otherwise,} \end{cases}$$

where $j_1$ and $j_2$ are the two keypoint types joined by limb $c$ (for example left elbow and left wrist), $\mathbf{x}_{j,k}$ is the position of keypoint $j$ of person $k$, and "on the limb" means within a distance $\sigma_l$ (the limb width, in pixels) of the segment between them. Where limbs of several people overlap, the targets are averaged. To score a candidate connection between two detected keypoints $\mathbf{d}_{j_1}$ and $\mathbf{d}_{j_2}$, OpenPose integrates the predicted field $\mathbf{L}_c$ along the segment joining them:

$$E = \int_0^1 \mathbf{L}_c(\mathbf{p}(u)) \cdot \frac{\mathbf{d}_{j_2} - \mathbf{d}_{j_1}}{\|\mathbf{d}_{j_2} - \mathbf{d}_{j_1}\|_2}\, du, \qquad \mathbf{p}(u) = (1-u)\,\mathbf{d}_{j_1} + u\,\mathbf{d}_{j_2},$$

where $u$ runs along the segment, and the integral is approximated by sampling a few evenly spaced values of $u$. $E$ is high when the field along the segment points the same way as the segment, which happens only if the two keypoints really are the ends of one limb. Finding the best assignment for all limbs at once is NP-hard, so OpenPose relaxes it: it keeps only the edges of a tree skeleton and solves one **bipartite matching** per limb type, for example with the Hungarian algorithm (see DLCV 07), then joins limbs that share keypoints into people.

*Online only: OpenPose: part affinity fields (Cao et al., 2017) — https://blog.shyandram.dev/en/learn/deep-learning/pose-keypoints/papers/openpose-part-affinity-fields-cao-et-al-2017/*

The code builds the ground-truth PAFs of two people standing side by side whose left forearms cross, adds small noise to the keypoint positions, and groups the keypoints limb by limb, once with the PAF score and once with plain distance (the shortest total limb length). The coordinates are synthetic.

```
import numpy as np
from scipy.optimize import linear_sum_assignment

# 13 body keypoints and the 12 limbs of a tree that connects them
J = ["nose", "l_sho", "r_sho", "l_elb", "r_elb", "l_wri", "r_wri",
     "l_hip", "r_hip", "l_knee", "r_knee", "l_ank", "r_ank"]
LIMBS = [(0, 1), (0, 2), (1, 3), (3, 5), (2, 4), (4, 6),
         (1, 7), (2, 8), (7, 9), (9, 11), (8, 10), (10, 12)]

# two people side by side; their left forearms cross (synthetic coordinates)
A = np.array([[40, 20], [52, 38], [28, 38], [70, 50], [24, 62], [100, 74], [22, 86],
              [48, 80], [32, 80], [50, 110], [30, 110], [52, 140], [28, 140]], float)
```

```
B = np.array([[72, 24], [84, 42], [62, 42], [98, 58], [56, 64], [74, 76], [52, 86],
              [80, 84], [66, 84], [82, 114], [64, 114], [84, 144], [62, 144]], float)
people = [A, B]
H, W, width = 160, 120, 3.0

def render_paf(c):
    """Ground-truth part affinity field of limb c: unit vectors near each person's limb."""
    ys, xs = np.mgrid[0:H, 0:W]
    field, count = np.zeros((H, W, 2)), np.zeros((H, W))
    for P in people:
        a, b = P[LIMBS[c][0]], P[LIMBS[c][1]]
        v = (b - a) / np.linalg.norm(b - a)
        along = (xs - a[0]) * v[0] + (ys - a[1]) * v[1]
        across = np.abs(-(xs - a[0]) * v[1] + (ys - a[1]) * v[0])
        on = (along >= 0) & (along <= np.linalg.norm(b - a)) & (across <= width)
        field[on] += v; count[on] += 1
    return field / np.maximum(count, 1)[..., None]          # average where limbs overlap

def limb_score(field, p, q, n=10):
    """Line integral of the field along the segment p -> q, approximated by n samples."""
    d = (q - p) / np.linalg.norm(q - p)
    u = np.linspace(0, 1, n)[:, None]
    pts = np.round(p + u * (q - p)).astype(int)
    return np.mean(field[pts[:, 1], pts[:, 0]] @ d)

rng = np.random.default_rng(0)
cands = [np.stack([A[j], B[j]]) + rng.normal(0, 1.0, (2, 2)) for j in range(13)]  #
 ↪  detections

ok_paf = ok_near = 0
for c, (j1, j2) in enumerate(LIMBS):
    field = render_paf(c)
    S = np.array([[limb_score(field, p, q) for q in cands[j2]] for p in cands[j1]])
    D = np.linalg.norm(cands[j1][:, None] - cands[j2][None], axis=2)
    if (j1, j2) == (3, 5):                                  # the crossing forearms
        print("left elbow -> left wrist, PAF scores (rows: elbows A, B; cols: wrists A, B)")
        print(np.round(S, 2))
        print("distances in pixels"); print(np.round(D, 1))
    r, k = linear_sum_assignment(-S)                        # maximize total affinity
    ok_paf += np.sum(k == r)                                # candidate i belongs to person
     ↪  i
    r, k = linear_sum_assignment(D)                         # baseline: shortest total
     ↪  length
    ok_near += np.sum(k == r)
print(f"correct limbs with PAF scores:     {ok_paf}/{2 * len(LIMBS)}")
print(f"correct limbs with distance only:  {ok_near}/{2 * len(LIMBS)}")
```

```
left elbow -> left wrist, PAF scores (rows: elbows A, B; cols: wrists A, B)
[[0.67 0.04]
 [0.29 0.78]]
distances in pixels
[[41.1 27.2]
 [18.4 30.2]]
correct limbs with PAF scores:     24/24
```

```
correct limbs with distance only:  22/24
```

The printed matrices show the failure of geometry alone. Each elbow is *closer* to the other person's wrist (27.2 and 18.4 pixels) than to its own (41.1 and 30.2), so matching by distance connects both forearms wrongly. The PAF score is high only along the true limbs (0.67 and 0.78), because the field there points along the segment; along the wrong pairings it is low. Figure 4 draws both results. Note that the PAF here is the *ground truth*; a trained network predicts a blurrier field, and the method's accuracy depends on how well it does so.

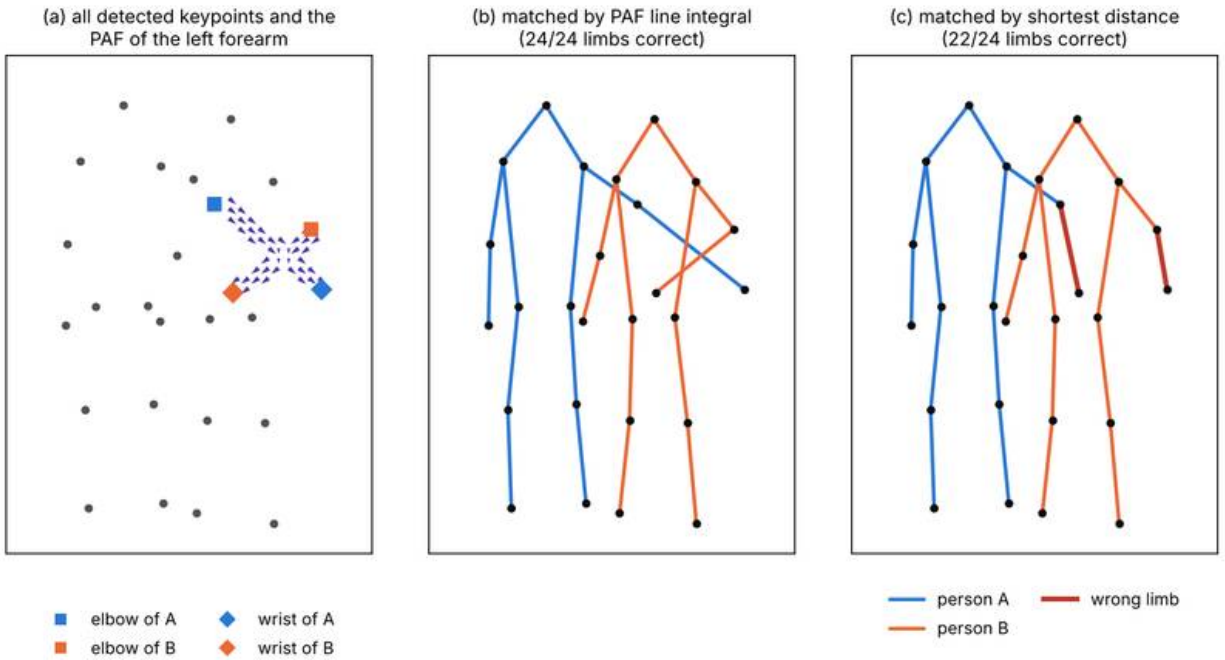



*Figure 4 — Bottom-up grouping on the synthetic scene of the code above. (a) Detected keypoints and the left-forearm PAF. (b) Matching by the PAF line integral recovers every limb. (c) Matching by distance alone swaps the two crossing forearms (red).*

**Associative embedding.** Newell et al. [19] avoid limb fields altogether. Alongside each keypoint heatmap the network outputs a **tag map**: one real number per pixel. Training pushes the tags of one person's keypoints to be equal and the tags of different people to be different. With $h_k(\mathbf{x})$ the tag of keypoint type $k$ at pixel $\mathbf{x}$, $\mathbf{x}_{nk}$ the ground-truth position of keypoint $k$ of person $n$, $N$ people and $\bar{h}_n = \frac{1}{K}\sum_k h_k(\mathbf{x}_{nk})$ the mean tag of person $n$, the grouping loss is

$$\mathcal{L}_g = \frac{1}{N}\sum_n \sum_k \big(\bar{h}_n - h_k(\mathbf{x}_{nk})\big)^2 + \frac{1}{N^2}\sum_n \sum_{n'} \exp\Big(-\frac{1}{2\sigma^2}(\bar{h}_n - \bar{h}_{n'})^2\Big),$$

where the first term pulls each person's tags toward their mean and the second pushes the means of different people apart, with a penalty that fades as the distance between them grows ($\sigma$ sets how far apart is "far enough"). The absolute tag values do not matter, only their differences. At

test time, keypoints are added to people joint by joint, matching tags that are close and detections that are confident. CornerNet borrowed exactly this idea to pair box corners (DLCV 03).

**Scale.** Bottom-up methods see all people at the image's own scale, so small people get few heatmap cells; OpenPose was less accurate than top-down methods on medium-sized people for this reason [18]. HigherHRNet [20] adds higher-resolution heatmaps (HRNet's output upsampled by a transposed convolution) and supervises and aggregates several resolutions; its paper reports results above all top-down methods on the CrowdPose test set, where boxes are unreliable. DEKR [21] returns to **dense regression**, with one branch per keypoint whose adaptive convolutions focus on that keypoint's region, and reports better results than detection-and-grouping methods on COCO and CrowdPose.

### One-stage and set-prediction pose

A third family predicts a whole pose per person directly, without a separate grouping step.

- **Center-based.** CenterNet ("Objects as Points") [22] detects each person as a centre point on a heatmap and regresses the $2K$ offsets from the centre to the keypoints; it then snaps each regressed keypoint to the nearest peak of a separate keypoint heatmap, so the centre acts as the grouping cue.
- **Real-time dense heads.** RTMO [23] puts a pose head into a YOLO-style one-stage detector and represents each keypoint with a pair of one-dimensional heatmaps (coordinate classification), aiming at top-down accuracy at one-stage speed.
- **Set prediction.** Transformer detectors in the DETR style (DLCV 03) output a fixed set of queries matched one-to-one to the ground truth, so no NMS or grouping is needed. ED-Pose [24] applies this to pose with two explicit box-detection stages: a human decoder detects people, and a human-to-keypoint decoder treats each keypoint as a small box to detect. Trained with an L1 regression loss and no heatmaps, it reports results above heatmap-based top-down methods with the same backbone on COCO.

### Choosing a pipeline

| | Top-down | Bottom-up | One-stage / set prediction |
|---|---|---|---|
| Main step | Detect people, then estimate each | Detect all keypoints, then group | Predict each person's pose directly |
| Cost vs number of people | Grows linearly | Nearly constant network cost; grouping grows | Nearly constant |
| Small people | Good (each crop is resized) | Harder (one image scale for all) | Depends on the feature pyramid |
| Crowds and overlap | Boxes overlap; NMS can remove people | Grouping can mix people | Queries or centres can collide |
| Typical use | High accuracy when people are few and well separated | Many people at once; real-time systems | End-to-end training; real-time |

## Measuring pose accuracy

**Plain version.** A predicted keypoint is "good" if it is close to the true one, but "close" must depend on the size of the person and on the keypoint: being 5 pixels off on a tiny person's nose is a big mistake; 5 pixels off on a large person's hip is not.

### PCK and PCKh

The **percentage of correct keypoints** (**PCK**) counts a predicted keypoint as correct if it lies within a threshold distance of the ground truth, where the threshold is a fraction of the person's size. Yang and Ramanan used a fraction of the person's bounding-box size [25]. MPII's **PCKh** [26] uses 50% of the **head segment length** instead, because head size does not change when the person bends or stretches. Formally,

$$\mathrm{PCKh@}\alpha = \frac{1}{\sum_{n,k} \delta(v_{nk} > 0)} \sum_{n,k} \delta(v_{nk} > 0)\; \mathbb{1}\big[\|\hat{\mathbf{x}}_{nk} - \mathbf{x}_{nk}\|_2 \le \alpha\, \ell_n\big],$$

where $n$ indexes people, $k$ keypoints, $\ell_n$ is the head segment length of person $n$, $\alpha$ is the fraction ($\alpha = 0.5$ for the standard PCKh), $\delta(\cdot)$ and $\mathbb{1}[\cdot]$ are 1 when their condition holds and 0 otherwise, and only labeled keypoints count. PCK assumes the person has been found; it does not penalize missed or invented people, which is why multi-person benchmarks use an AP-style metric.

### Object keypoint similarity and keypoint AP

COCO's metric mirrors box detection (DLCV 03). In box AP, a detection is matched to a ground-truth box if their IoU passes a threshold. In keypoint AP, a predicted pose is matched to a ground-truth person if their **object keypoint similarity** (**OKS**) passes a threshold [4]:

$$\mathrm{OKS} = \frac{\sum_i \exp\!\left(-d_i^2 / (2 s^2 \kappa_i^2)\right)\, \delta(v_i > 0)}{\sum_i \delta(v_i > 0)},$$

where $i$ runs over the keypoints, $d_i$ is the Euclidean distance between the predicted and ground-truth keypoint $i$, $v_i$ is the ground-truth visibility flag, $s$ is the object scale, defined as the square root of the person's segmentation area, and $\kappa_i$ is a per-keypoint constant. Each term is an unnormalized Gaussian of the error with standard deviation $s\kappa_i$, so it is 1 for a perfect keypoint and falls toward 0 as the error grows; OKS averages it over the labeled keypoints. The predicted visibility flags are not used.

**Where the constants come from.** COCO had 5,000 validation images annotated twice and measured, for each keypoint type, how much human annotators disagree relative to the person's scale, $\sigma_i^2 = E[d_i^2/s^2]$. It then set $\kappa_i = 2\sigma_i$ [4]. The values of $\sigma_i$ are 0.026 for the nose, 0.025 for the eyes, 0.035 for the ears, 0.079 for the shoulders, 0.072 for the elbows, 0.062 for the wrists, 0.107 for the hips, 0.087 for the knees and 0.089 for the ankles [4], [27]. Eyes are easy to place exactly; hips, hidden under clothing, are not, so the metric is more forgiving for hips. With $\kappa_i = 2\sigma_i$, errors of

one, two and three annotator standard deviations give similarities of $e^{-1/8} \approx 0.88$, $e^{-1/2} \approx 0.61$ and $e^{-9/8} \approx 0.32$, and human annotators reach a median OKS of about 0.91 against each other [4].

**From OKS to AP.** The rest is the AP of DLCV 03: rank all predicted poses by their person-level score, match each to the unmatched ground-truth person with the highest OKS, and compute AP at OKS thresholds $0.50, 0.55, \ldots, 0.95$. The primary metric, AP, averages these ten values. Companion metrics are $AP^{50}$, $AP^{75}$, $AP^{M}$ for medium people (area between $32^2$ and $96^2$) and $AP^{L}$ for large ones (area above $96^2$), plus the matching AR values. At most 20 poses per image are scored, and small people have no keypoint labels [4].

*Online only: Inside OKS: unlabeled people, crowds and the scale s — https://blog.shyandram.dev/en/learn/deep-learning/pose-keypoints/notes/inside-oks-unlabeled-people-crowds-and-the-scale-s/*

The code computes OKS with COCO's constants for a synthetic person about 200 pixels tall.

```
import numpy as np

NAMES = ["nose", "l_eye", "r_eye", "l_ear", "r_ear", "l_shoulder", "r_shoulder",
         "l_elbow", "r_elbow", "l_wrist", "r_wrist", "l_hip", "r_hip",
         "l_knee", "r_knee", "l_ankle", "r_ankle"]
SIGMAS = np.array([.26, .25, .25, .35, .35, .79, .79, .72, .72, .62, .62,
                   1.07, 1.07, .87, .87, .89, .89]) / 10.0       # COCO per-keypoint
                    ↪ constants

def oks(pred, gt, vis, area):
    """pred, gt: (17, 2) pixel coordinates; vis: (17,) COCO flags; area: segment area
     ↪ (s^2)."""
    d2 = ((pred - gt) ** 2).sum(1)
    kappa = 2 * SIGMAS
    ks = np.exp(-d2 / (2 * area * kappa ** 2))                # per-keypoint similarity
    labeled = vis > 0
    return ks[labeled].mean(), ks

# a standing person about 200 px tall, facing the camera (synthetic coordinates)
gt = np.array([[100, 30], [106, 25], [94, 25], [112, 28], [88, 28], [125, 60], [75, 60],
               [135, 95], [65, 95], [140, 128], [60, 128], [115, 130], [85, 130],
               [117, 175], [83, 175], [118, 220], [82, 220]], float)
vis = np.full(17, 2); vis[3] = 0                                  # left ear not labeled
area = 0.6 * (70 * 200)                                           # segment area: ~60% of its
 ↪ box

rng = np.random.default_rng(1)
jitter = gt + rng.normal(0, 3, gt.shape)                          # every keypoint off by ~3 px
 ↪ per axis
flipped = jitter.copy(); flipped[[9, 10]] = flipped[[10, 9]]    # left/right wrists swapped
print(f"perfect          OKS = {oks(gt, gt, vis, area)[0]:.3f}")
print(f"3 px jitter      OKS = {oks(jitter, gt, vis, area)[0]:.3f}")
print(f"+ wrists swapped OKS = {oks(flipped, gt, vis, area)[0]:.3f}")

# the same 5-pixel error costs very different amounts on different keypoints
for k in (0, 5, 11):
    p = gt.copy(); p[k, 0] += 5
    print(f"5 px error on {NAMES[k]:10s}: similarity {oks(p, gt, vis, area)[1][k]:.3f}")
```

```
perfect          OKS = 1.000
3 px jitter      OKS = 0.933
+ wrists swapped OKS = 0.810
5 px error on nose      : similarity 0.577
5 px error on l_shoulder: similarity 0.942
5 px error on l_hip     : similarity 0.968
```

Three lessons follow from this synthetic example. Small jitter everywhere costs little (0.933). A single **left–right swap** of the wrists sets both wrist similarities to essentially zero and drops OKS to 0.810: this pose still counts as correct at thresholds up to 0.80 but fails at 0.85 and above, so it loses about a third of the ten AP thresholds. And the same 5-pixel error costs 0.42 of similarity on the nose but only 0.03 on a hip. Figure 5 shows the tolerance radius of each keypoint and the similarity curves.

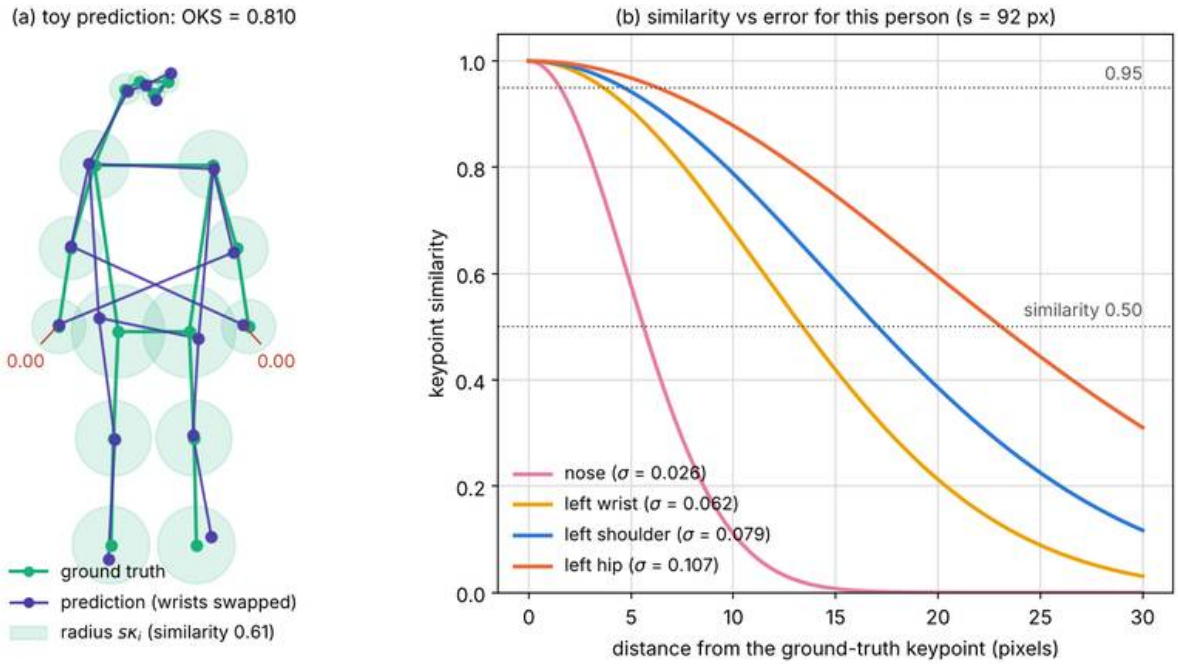



*Figure 5 — OKS on the synthetic person of the code above. (a) Circles of radius $s\kappa_i$, where a keypoint's similarity is 0.61; the prediction is jittered by about 3 pixels and has its wrists swapped. (b) Keypoint similarity as a function of the error for four keypoint types, for this person's scale $s \approx 92$ pixels.*

### Metrics for 3D pose

For 3D pose, the standard metric is the **mean per-joint position error** (**MPJPE**), the average Euclidean distance between predicted and ground-truth 3D joints,

$$\text{MPJPE} = \frac{1}{K}\sum_{k=1}^{K} \|\hat{\mathbf{X}}_k - \mathbf{X}_k\|_2,$$

reported in millimetres and averaged over all test frames, where $\hat{\mathbf{X}}_k$ and $\mathbf{X}_k$ are the predicted and true 3D positions of joint $k$. Two protocols are common on Human3.6M: errors after aligning the root (central hip) joints of prediction and ground truth, called Protocol #1, and errors after an additional rigid alignment of the predicted skeleton to the ground truth, called Protocol #2 [28]; the second metric is also known as PA-MPJPE or reconstruction error [2]. The second hides errors in global rotation and scale, so the two numbers must never be compared with each other.

### What the numbers do not tell you

Like box AP, keypoint AP mixes several kinds of error. Ronchi and Perona [29] split keypoint localization errors into four types: **jitter** (near the right keypoint but outside the human error margin), **inversion** (at the person's own left–right twin or another of their keypoints), **swap** (at a keypoint of a *different person*) and **miss** (near no keypoint). They found that errors depend most on the number of visible keypoints and on clutter from nearby people. Even COCO's doubly annotated data contain 1–2% outliers, mostly left–right confusions or the wrong person labeled [4]. Diagnose models by error type, not only by AP.

## Datasets and benchmarks

**Plain version.** A handful of datasets defined what "good" means: COCO for many people in everyday photos, MPII for a wide range of activities, CrowdPose for heavy overlap, Human3.6M for 3D, and newer sets for whole bodies, video and animals.

| Dataset | Content (from the papers or official pages) | Metric | Why it matters |
|---|---|---|---|
| COCO keypoints [30], [4] | 17 body keypoints on people in everyday photos; small people unlabeled | OKS AP | The main multi-person benchmark |
| MPII Human Pose [26] | 40,522 annotated people in images from YouTube videos, collected with queries for more than 800 activities | PCKh | Diverse activities; single- and multi-person tracks |
| CrowdPose [31] | 20,000 images, about 80,000 people, 14 keypoints, selected for a uniform spread of crowding | OKS AP | Crowded scenes |
| COCO-WholeBody [32] | 133 landmarks per person: 68 face, 42 hand, 23 body and feet | OKS AP | Face, hands and feet together |
| PoseTrack [33] | Videos with person tracks and articulated poses | AP, MOTA | Pose estimation and tracking in video |

| Dataset | Content (from the papers or official pages) | Metric | Why it matters |
|---|---|---|---|
| Human3.6M [34] | 3.6 million 3D poses of 11 actors in 17 scenarios, recorded by 4 calibrated cameras with motion capture | MPJPE | The most widely used 3D benchmark |
| AP-10K [35] | 10,015 images of 54 mammal species from 23 families, annotated in COCO style | AP | Animal pose and cross-species generalization |

CrowdPose deserves a closer look because it defines crowding precisely. For person $i$, let $N_a^i$ be the number of its own joints inside its box and $N_b^i$ the number of *other* people's joints inside the same box; the **Crowd Index** of an image with $n$ people is $\frac{1}{n}\sum_i N_b^i/N_a^i$ [31]. Common datasets are dominated by uncrowded images; CrowdPose was selected so that this index is spread nearly uniformly between 0 and 1, and its authors found the accuracy of strong models to be about 20 AP lower on crowded images than on uncrowded ones [31].

Two cautions apply when reading results. Human3.6M is an indoor dataset with a few actors [2], [34], so low MPJPE there does not imply good results on outdoor video. And on COCO, results depend on the person detector used by top-down methods (see the Simple Baselines guide (https://blog.shyandram.dev/en/learn/deep-learning/pose-keypoints/papers/simple-baselines-xiao-et-al-2018/)), on input resolution and on extra training data, so compare numbers only under the same settings.

## Beyond the human body

**Plain version.** Nothing in a heatmap network is specific to people. Change the list of keypoints and the training data, and the same machinery finds the corners of a face, the joints of a hand, or the paws of a dog.

- **Whole body.** COCO-WholeBody [32] adds feet, hands and dense face landmarks to the body keypoints, which creates a huge range of scales inside one person; its baseline model, ZoomNet, is built around the body's hierarchical structure to handle this. The journal version of OpenPose [36] added foot keypoints.
- **Animals.** AP-10K [35] benchmarks pose across 54 mammal species and tests generalization to unseen animal families. In neuroscience, DeepLabCut [1] uses transfer learning from a pre-trained network to track user-defined body parts of mice and flies from about 200 labeled frames in its experiments, replacing reflective markers.
- **Objects.** The same formulation works for any category with consistent landmarks, such as the wheels and lights of cars, provided the keypoints are defined consistently and the category's left–right symmetries are handled like the body's.

## From 2D to 3D pose

**Plain version.** A photo flattens the world: from one image, a hand pointing at the camera and a hand held at the hip can look almost the same. There are three ways to get the depth back: *guess* it from experience (lifting), *measure* it with two or more cameras (triangulation), or *fit* a model of the human body that only allows realistic shapes.

### Lifting 2D keypoints

Martinez et al. [28] showed that a surprisingly large part of monocular 3D pose estimation is solved once the 2D keypoints are known. Their model takes the $2K$ coordinates of a detected 2D pose and maps them with a simple fully connected network (linear layers with batch normalization, dropout and residual connections) to $3K$ coordinates. Trained on Human3.6M, it beat the previous best methods by a large margin when given ground-truth 2D keypoints, which suggests that much of the remaining difficulty lies in the 2D detection rather than in the lifting. Lifting is also *ambiguous*: several 3D poses project to the same 2D pose, and a network with a single output can return only one of them.

VideoPose3D [37] lifts a whole *sequence* of 2D poses with dilated temporal convolutions, so motion over neighbouring frames helps resolve depth and jitter. It also introduced **back-projection**, a semi-supervised loss that estimates 3D poses for unlabeled video and requires them to project back onto the detected 2D keypoints.

### Multi-view triangulation

With two or more calibrated cameras, depth is no longer a guess. Each camera's 2D keypoint defines a ray in space, and the 3D joint is the point where the rays (almost) meet. The geometry is the triangulation of DLCV 11, applied joint by joint: the camera matrices are the pinhole models of that chapter, and the linear (DLT) solution is derived in its Supplementary notes. Learnable triangulation [38] makes this step part of the network: it weights each view by a confidence predicted from the image, so that a view in which the joint is occluded contributes less, and its second variant aggregates 2D features from all views into a 3D volume refined by 3D convolutions.

The code projects a synthetic 13-joint skeleton, standing 4 m from two converging cameras, into both images, adds 2-pixel noise to the 2D keypoints, triangulates every joint with the DLT and reports MPJPE for three distances between the cameras (the **baseline**). The numbers come from this synthetic run, without root alignment.

```
import numpy as np

def camera(K, angle_deg, center):
    """3x4 projection P = K [R | t] for a camera at `center` rotated about the y axis."""
    a = np.deg2rad(angle_deg)
    R = np.array([[np.cos(a), 0, -np.sin(a)], [0, 1, 0], [np.sin(a), 0, np.cos(a)]])
    return K @ np.hstack([R, -R @ center[:, None]])
```

```
def project(P, X):
    x = (P @ np.hstack([X, np.ones((len(X), 1))]).T).T
    return x[:, :2] / x[:, 2:]

def triangulate(Ps, xs):
    """Linear (DLT) triangulation of one point seen in several views."""
    A = []
    for P, (u, v) in zip(Ps, xs):
        A += [u * P[2] - P[0], v * P[2] - P[1]]
    X = np.linalg.svd(np.array(A))[2][-1]
    return X[:3] / X[3]

# a 13-joint 3D skeleton in millimetres, standing 4 m in front of the cameras (synthetic)
skel = np.array([[0, -1650, 0], [170, -1400, 0], [-170, -1400, 0], [250, -1120, 50], [-250,
 ↪ -1120, 50],
                 [280, -860, 120], [-280, -860, 120], [100, -900, 0], [-100, -900, 0],
                 [110, -480, 30], [-110, -480, 30], [110, -60, 0], [-110, -60, 0]], float)
skel += [0, 900, 4000]
K = np.array([[1000, 0, 640], [0, 1000, 360], [0, 0, 1]], float)

rng = np.random.default_rng(0)
for baseline in (200, 1000, 3000):                          # distance between the cameras
 ↪ (mm)
    half = baseline / 2
    Ps = [camera(K, np.degrees(np.arctan2(half, 4000)), np.array([-half, 0, 0.])),
          camera(K, -np.degrees(np.arctan2(half, 4000)), np.array([half, 0, 0.]))]
    errs = []
    for trial in range(200):                                  # 2D detections with 2 px noise
        obs = [project(P, skel) + rng.normal(0, 2.0, (13, 2)) for P in Ps]
        est = np.array([triangulate(Ps, [o[j] for o in obs]) for j in range(13)])
        errs.append(np.linalg.norm(est - skel, axis=1).mean())   # MPJPE of this trial
    print(f"baseline {baseline:4d} mm: MPJPE = {np.mean(errs):6.1f} mm")
```

```
baseline  200 mm: MPJPE =  183.7 mm
baseline 1000 mm: MPJPE =   37.9 mm
baseline 3000 mm: MPJPE =   17.4 mm
```

With cameras 20 cm apart, a 2-pixel image error becomes about 18 cm in 3D, almost all of it in depth (a per-axis check of the same run gives 182 mm in depth against 9 and 18 mm across): the rays are nearly parallel, so a small shift moves their crossing point far along them. A wider baseline makes the rays cross at a larger angle and cuts the error by a factor of ten.

### Parametric body models and mesh recovery

A skeleton says nothing about body shape, and nothing stops a lifting network from predicting a forearm that changes length from frame to frame. **Parametric body models** fix both problems by describing a body with a small set of parameters. The most widely used is SMPL [39]. It starts from a template mesh with $N = 6890$ vertices and a skeleton with $K = 23$ joints, and generates a posed body as

$$M(\boldsymbol{\beta}, \boldsymbol{\theta}) = W(T_P(\boldsymbol{\beta}, \boldsymbol{\theta}), J(\boldsymbol{\beta}), \boldsymbol{\theta}, \mathcal{W}), \qquad T_P(\boldsymbol{\beta}, \boldsymbol{\theta}) = \bar{\mathbf{T}} + B_S(\boldsymbol{\beta}) + B_P(\boldsymbol{\theta}),$$

where $\boldsymbol{\beta}$ are **shape** parameters (coefficients of the main modes of body-shape variation), $\boldsymbol{\theta}$ are **pose** parameters (an axis–angle rotation for each joint relative to its parent, plus the global orientation: $3 \times 23 + 3 = 72$ numbers), $\bar{\mathbf{T}}$ is the template mesh, $B_S$ and $B_P$ are vertex offsets that model identity and pose-dependent deformations such as bulging muscles, $J(\boldsymbol{\beta})$ are the joint locations regressed from the shaped mesh, $\mathcal{W}$ are the blend weights and $W$ is a standard blend-skinning function that rotates each vertex around the joints. Keypoints are read off the mesh, so a fitted SMPL body always has consistent bone lengths.

**Human mesh recovery (HMR)** [40] regresses SMPL parameters and a camera directly from an image. Its main loss is the **reprojection error** of the model's joints against 2D keypoint labels, which allows training on in-the-wild images; an adversarial prior trained on real 3D bodies rejects implausible parameters. HMR 2.0, part of 4DHumans [41], makes the network fully transformer-based and tracks the reconstructed people through occlusions in video.

*Online only: SMPL in more detail — https://blog.shyandram.dev/en/learn/deep-learning/pose-keypoints/notes/smpl-in-more-detail/*

## Failure modes and practical advice

**Plain version.** Pose models fail where people would hesitate too: hidden joints, tangled people, mirror-image confusion, bodies cut off by the frame, poses rarely seen in training, and blur.

- **Occlusion.** An occluded joint ($v = 1$) is labeled but not shown, so the network must infer it from context; self-occlusion makes legs especially prone to left–right confusion [29].
- **Crowds.** Overlapping people break detectors, confuse grouping and cause **swap** errors [29]; accuracy falls as the Crowd Index rises [31].
- **Left–right flips.** Front and back views look alike, so a model can mirror a limb or a whole person (an **inversion**); a forgotten channel swap in flip augmentation causes the same error systematically.
- **Truncation.** Joints outside the frame are unlabeled or guessed, and crops of a partial person are poorly centred.
- **Unusual poses.** Gymnastics, dancing or lying people are rare in training data; HMR 2.0's authors single them out as a case where earlier methods struggled [41].
- **Motion blur, low resolution and domain shift.** Fast limbs smear, small people have few pixels per limb, and models trained on adult photos transfer poorly to infants or artwork [2].

Practical advice that follows from this chapter: use an unbiased encoder and a sub-pixel decoder; check that flipping swaps left and right channels; report the detector and input size with top-down results; evaluate crowded and uncrowded images separately; break errors down by type; and in video, smooth or track keypoints over time. Tracking-by-detection with the association methods of DLCV 07 carries over directly: SimpleBaseline, for example, propagates poses to the next frame with optical flow and matches them with an OKS-based similarity [9], and PoseTrack provides the benchmark [33]. Following individual surface points instead of named joints is the

point-tracking problem of DLCV 07.

## Modern view

### What the surveys say

**Zheng et al.** [2] review more than 250 deep-learning papers on 2D and 3D pose since 2014, organized as in this chapter: regression versus heatmaps, top-down versus bottom-up, and for 3D, lifting, mesh recovery and multi-view methods. Their future directions include domain adaptation, human–scene constraints and metrics for temporal smoothness.

**Chen et al.** [3] organize monocular methods into four tasks (2D or 3D, single- or multi-person), list the design tricks that proved effective (multi-stage structures, intermediate supervision, multi-scale fusion, body-structure constraints) and name occlusion and crowds as the main unresolved challenges.

**Tian et al.** [5] survey monocular 3D mesh recovery with body models, contrasting optimization-based fitting with regression networks, and point to the lack of in-the-wild 3D labels as the central difficulty.

*Online only: Guide to the Modern view surveys — https://blog.shyandram.dev/en/learn/deep-learning/pose-keypoints/papers/guide-to-the-modern-view-surveys/*

### The state of the art, 2023–2026

- **Scaled transformers for top-down pose.** ViTPose [16] showed that plain vision transformers with simple decoders scale well (from 100M to 1B parameters in the paper), and its largest model set a new state of the art on the COCO keypoint benchmark at publication.
- **Human-centric foundation models.** Sapiens [42] pretrains vision transformers from 0.3 to 2 billion parameters on over 300 million images of humans with self-supervision, then fine-tunes them for 2D pose, body-part segmentation, depth and surface normals; its authors report that this human-specific pretraining improves all four tasks at the same compute budget.
- **Real-time systems.** RTMPose [43] (top-down, coordinate classification, engineered for CPU, GPU and mobile deployment) and RTMO [23] (one-stage) target the latency budgets of applications rather than leaderboard peaks.
- **End-to-end multi-person models.** Query-based models such as ED-Pose [24] remove NMS, grouping and heatmaps; ED-Pose reports higher COCO accuracy than heatmap-based top-down methods with the same backbone.
- **3D humans in video.** Transformer mesh recovery such as HMR 2.0 [41] reconstructs and tracks full 3D bodies from monocular video, moving the field from sparse skeletons toward complete body surfaces.

### Open problems

1. **Crowds and heavy occlusion** remain the largest accuracy gap [31], [3].
2. **In-the-wild 3D.** Accurate 3D labels are scarce outside the lab [5], and the most widely used 3D benchmark, Human3.6M, is an indoor dataset [2], so monocular 3D methods are trained and tested far from where they are used.
3. **Temporal consistency.** Per-frame metrics such as MPJPE do not measure jitter or physical plausibility over time [2].
4. **Generalization across domains and species**: infants, artwork, unusual viewpoints and unseen animal families [2], [35].
5. **Evaluation.** OKS constants were measured from annotator disagreement on COCO people [4] and do not transfer automatically to other keypoint sets, and AP hides error types such as inversions and swaps [29].

## Key takeaways

- Keep keypoint names, left/right conventions and visibility labels consistent.
- Heatmap resolution and decoding affect localization; the brightest cell alone loses sub-pixel information.
- Precise joints require both fine spatial features and enough context to identify the body part.
- Top-down methods inherit detector errors and per-person cost; bottom-up methods require grouping.
- State each metric's scale normalization, visibility rules and coordinate convention.
- 3D lifting is ambiguous; triangulation needs calibration, and body models impose shape assumptions. Inspect errors by type.

## Exercises

1. **Quantization by hand.** A wrist lies at $x = 101.3$ pixels in a crop, and the heatmap stride is 4. Which heatmap column holds the peak? What does argmax decoding return, and what is its error? What does the quarter offset return? What would an exact Taylor step on a noise-free Gaussian return?

   **Hint**

   In heatmap units the wrist is at $101.3/4 = 25.325$, so the peak is in column 25 and argmax returns $25 \times 4 = 100$ pixels, an error of 1.3 pixels. The right neighbour (column 26) is closer to 25.325 than the left one, so the quarter offset moves to 25.25, which maps to 101.0 pixels (error 0.3). On a noise-free Gaussian, the log-heatmap is an exact parabola, so the Taylor step returns 25.325, that is 101.3 pixels.

2. **OKS by hand.** A person has segmentation area 10,000 pixels², so $s = 100$. A predicted knee is 10 pixels from the ground truth. Compute its keypoint similarity with COCO's constant. Repeat for a nose 10 pixels off. Which error would you rather make, and why does COCO treat them so differently?

**Hint**

Knee: $\kappa = 2 \times 0.087 = 0.174$, $s\kappa = 17.4$, similarity $= \exp(-100/(2 \times 17.4^2)) = \exp(-0.165) \approx 0.85$. Nose: $\kappa = 0.052$, $s\kappa = 5.2$, similarity $= \exp(-100/54.08) \approx 0.16$. Annotators place noses about three times more consistently than knees, and COCO scales the tolerance by that disagreement.

3. **When does top-down become too slow?** A person detector takes 20 ms per image and a single-person network takes 5 ms per crop. A bottom-up network takes 40 ms per image plus 1 ms of grouping per person. For how many people per image is bottom-up faster? Name two reasons, other than speed, to prefer one pipeline in a crowded train station.

**Hint**

Top-down costs $20 + 5N$ ms and bottom-up $40 + N$ ms, so bottom-up is faster when $4N > 20$, that is for more than 5 people. In a crowded station, top-down suffers from overlapping boxes and NMS removing people, but handles small distant people better because each crop is resized; bottom-up has a constant network cost but must group heavily overlapping limbs, and small people get few heatmap cells.

4. **Break the part affinity field.** In the grouping code, move person B so that its left forearm lies on the same line as person A's left forearm and overlaps it (one person standing behind the other). What happens to the PAF scores of the two correct and the two wrong pairings, and why? What extra cue could resolve the case?

**Hint**

Where the two limbs overlap, the target field is the average of two identical unit vectors, so it points along the shared line. Every candidate segment between an elbow and a wrist on that line, correct or not, is aligned with the field and gets a similar score, so the matching becomes ambiguous. Cues that can help: the lengths of the limbs relative to the rest of each body, tags as in associative embedding, the other limbs of each person (a global rather than per-limb parse), or temporal continuity in video.

5. **A swap the cameras cannot see.** Change the triangulation code so that both cameras aim at the centre of the skeleton, and in the second camera exchange the 2D positions of the left and right knees (an inversion error of the 2D detector). Triangulate the left knee and compute its mean reprojection error in the two views, with and without the swap. Then move the second camera 1 m *above* the first instead of beside it and repeat. Explain the difference.

**Hint**

In our run of this experiment, with the cameras side by side the reprojection error is about 1 pixel with or without the swap, yet the swapped knee lands more than half a metre from the truth. The two knees are at nearly the same height, and with a horizontal baseline the epipolar lines are nearly horizontal, so swapping the knees moves the observation *along* its epipolar line: the two rays still meet, just at the wrong depth. With the second camera above the first, the epipolar lines become nearly vertical, the swap moves the observation off its line, and the reprojection error jumps to tens of pixels, which flags the error. Camera layouts for multi-view pose should therefore not place all cameras along one horizontal line.

6. **Forgetting the channel swap.** You run flip testing but forget to swap the left and right channels of the mirrored heatmaps before averaging. Describe what the averaged left-wrist heatmap looks like for a person facing the camera with arms spread, and what argmax decoding returns.

**Hint**

Think about what the network sees in the mirrored image: a person facing the camera, so it labels the wrist on the image's right as "left wrist". That wrist is the person's *right* wrist, mirrored. Flipping the map back without swapping channels therefore puts a blob on the true *right* wrist in the left-wrist channel. The average has two half-height blobs, one on each wrist, and argmax picks whichever is slightly higher, so roughly half the time the left wrist is reported on the right wrist: an inversion error.

## References


1. A. Mathis, P. Mamidanna, K. M. Cury, T. Abe, V. N. Murthy, M. W. Mathis and M. Bethge, "DeepLabCut: Markerless pose estimation of user-defined body parts with deep learning," *Nature Neuroscience*, vol. 21, no. 9, 2018. doi ·arXiv
2. C. Zheng, W. Wu, C. Chen, T. Yang, S. Zhu, J. Shen, N. Kehtarnavaz and M. Shah, "Deep learning-based human pose estimation: A survey," *ACM Computing Surveys*, vol. 56, no. 1, 2024. doi ·arXiv
3. Y. Chen, Y. Tian and M. He, "Monocular human pose estimation: A survey of deep learning-based methods," *Computer Vision and Image Understanding*, vol. 192, 102897, 2020. doi ·arXiv
4. COCO Consortium, "Keypoint evaluation," COCO dataset official documentation (accessed 2026-11-01). cocodataset.org
5. Y. Tian, H. Zhang, Y. Liu and L. Wang, "Recovering 3D human mesh from monocular images: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 12, 2023. doi ·arXiv
6. A. Toshev and C. Szegedy, "DeepPose: Human pose estimation via deep neural networks," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014. doi ·arXiv
7. J. Tompson, A. Jain, Y. LeCun and C. Bregler, "Joint training of a convolutional network and a graphical model for human pose estimation," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2014. NeurIPS ·arXiv
8. A. Newell, K. Yang and J. Deng, "Stacked hourglass networks for human pose estimation," in *Proc. European Conference on Computer Vision (ECCV)*, LNCS, 2016. doi ·arXiv
9. B. Xiao, H. Wu and Y. Wei, "Simple baselines for human pose estimation and tracking," in *Proc. European Conference on Computer Vision (ECCV)*, LNCS, 2018. doi ·arXiv
10. F. Zhang, X. Zhu, H. Dai, M. Ye and C. Zhu, "Distribution-aware coordinate representation for human pose estimation," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. doi ·arXiv
11. X. Sun, B. Xiao, F. Wei, S. Liang and Y. Wei, "Integral human pose regression," in *Proc. European Conference on Computer Vision (ECCV)*, LNCS, 2018. doi ·arXiv
12. Y. Li, S. Yang, P. Liu, S. Zhang, Y. Wang, Z. Wang, W. Yang and S.-T. Xia, "SimCC: A simple coordinate classification perspective for human pose estimation," in *Proc. European Conference on Computer Vision (ECCV)*, LNCS, 2022. doi ·arXiv

13. J. Li, S. Bian, A. Zeng, C. Wang, B. Pang, W. Liu and C. Lu, "Human pose regression with residual log-likelihood estimation," in *Proc. IEEE/CVF International Conference on Computer Vision* (*ICCV*), 2021. doi ·arXiv
14. S.-E. Wei, V. Ramakrishna, T. Kanade and Y. Sheikh, "Convolutional pose machines," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2016. doi ·arXiv
15. K. Sun, B. Xiao, D. Liu and J. Wang, "Deep high-resolution representation learning for human pose estimation," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2019. doi ·arXiv
16. Y. Xu, J. Zhang, Q. Zhang and D. Tao, "ViTPose: Simple vision transformer baselines for human pose estimation," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2022. NeurIPS ·arXiv
17. K. He, G. Gkioxari, P. Dollár and R. Girshick, "Mask R-CNN," in *Proc. IEEE International Conference on Computer Vision* (*ICCV*), 2017. doi ·arXiv
18. Z. Cao, T. Simon, S.-E. Wei and Y. Sheikh, "Realtime multi-person 2D pose estimation using part affinity fields," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2017. doi ·arXiv
19. A. Newell, Z. Huang and J. Deng, "Associative embedding: End-to-end learning for joint detection and grouping," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2017. NeurIPS ·arXiv
20. B. Cheng, B. Xiao, J. Wang, H. Shi, T. S. Huang and L. Zhang, "HigherHRNet: Scale-aware representation learning for bottom-up human pose estimation," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2020. doi ·arXiv
21. Z. Geng, K. Sun, B. Xiao, Z. Zhang and J. Wang, "Bottom-up human pose estimation via disentangled keypoint regression," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2021. doi ·arXiv
22. X. Zhou, D. Wang and P. Krähenbühl, "Objects as points," arXiv:1904.07850, 2019. arXiv
23. P. Lu, T. Jiang, Y. Li, X. Li, K. Chen and W. Yang, "RTMO: Towards high-performance one-stage real-time multi-person pose estimation," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2024. doi ·arXiv
24. J. Yang, A. Zeng, S. Liu, F. Li, R. Zhang and L. Zhang, "Explicit box detection unifies end-to-end multi-person pose estimation," in *Proc. International Conference on Learning Representations* (*ICLR*), 2023. OpenReview ·arXiv
25. Y. Yang and D. Ramanan, "Articulated human detection with flexible mixtures of parts," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 12, 2013. doi
26. M. Andriluka, L. Pishchulin, P. Gehler and B. Schiele, "2D human pose estimation: New benchmark and state of the art analysis," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2014. doi ·CVF
27. COCO Consortium, `cocoeval.py` (keypoint OKS computation and the per-keypoint sigmas), official COCO API source code (accessed 2026-11-01). GitHub
28. J. Martinez, R. Hossain, J. Romero and J. J. Little, "A simple yet effective baseline for 3d human pose estimation," in *Proc. IEEE International Conference on Computer Vision* (*ICCV*), 2017. doi · arXiv
29. M. R. Ronchi and P. Perona, "Benchmarking and error diagnosis in multi-instance pose estimation," in *Proc. IEEE International Conference on Computer Vision* (*ICCV*), 2017. doi ·arXiv

30. T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Proc. European Conference on Computer Vision (ECCV)*, LNCS, 2014. doi ·arXiv
31. J. Li, C. Wang, H. Zhu, Y. Mao, H.-S. Fang and C. Lu, "CrowdPose: Efficient crowded scenes pose estimation and a new benchmark," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. doi ·arXiv
32. S. Jin, L. Xu, J. Xu, C. Wang, W. Liu, C. Qian, W. Ouyang and P. Luo, "Whole-body human pose estimation in the wild," in *Proc. European Conference on Computer Vision (ECCV)*, LNCS, 2020. doi ·arXiv
33. M. Andriluka, U. Iqbal, E. Insafutdinov, L. Pishchulin, A. Milan, J. Gall and B. Schiele, "PoseTrack: A benchmark for human pose estimation and tracking," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. doi ·arXiv
34. C. Ionescu, D. Papava, V. Olaru and C. Sminchisescu, "Human3.6M: Large scale datasets and predictive methods for 3D human sensing in natural environments," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 7, 2014. doi ·project page
35. H. Yu, Y. Xu, J. Zhang, W. Zhao, Z. Guan and D. Tao, "AP-10K: A benchmark for animal pose estimation in the wild," in *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2021. NeurIPS ·arXiv
36. Z. Cao, G. Hidalgo, T. Simon, S.-E. Wei and Y. Sheikh, "OpenPose: Realtime multi-person 2D pose estimation using part affinity fields," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 1, 2021. doi ·arXiv
37. D. Pavllo, C. Feichtenhofer, D. Grangier and M. Auli, "3D human pose estimation in video with temporal convolutions and semi-supervised training," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. doi ·arXiv
38. K. Iskakov, E. Burkov, V. Lempitsky and Y. Malkov, "Learnable triangulation of human pose," in *Proc. IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019. doi ·arXiv
39. M. Loper, N. Mahmood, J. Romero, G. Pons-Moll and M. J. Black, "SMPL: A skinned multi-person linear model," *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, vol. 34, no. 6, 2015. doi
40. A. Kanazawa, M. J. Black, D. W. Jacobs and J. Malik, "End-to-end recovery of human shape and pose," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. doi ·arXiv
41. S. Goel, G. Pavlakos, J. Rajasegaran, A. Kanazawa and J. Malik, "Humans in 4D: Reconstructing and tracking humans with transformers," in *Proc. IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023. doi ·arXiv
42. R. Khirodkar, T. Bagautdinov, J. Martinez, S. Zhaoen, A. James, P. Selednik, S. Anderson and S. Saito, "Sapiens: Foundation for human vision models," in *Proc. European Conference on Computer Vision (ECCV)*, LNCS, 2024. doi ·arXiv
43. T. Jiang, P. Lu, L. Zhang, N. Ma, R. Han, C. Lyu, Y. Li and K. Chen, "RTMPose: Real-time multi-person pose estimation based on MMPose," arXiv:2303.07399, 2023. arXiv

# DLCV 06 • Text in Images: OCR and Document Understanding

How machines read text in photos, scans and screenshots: the detection, recognition and layout pipeline and where each stage fails; regression and segmentation text detectors, CTC, attention and transformer recognizers, text spotting, LayoutLM-style and OCR-free document models; CER, WER and H-mean; the standard benchmarks and the failure modes that matter.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/text-in-images/

## What you'll learn

- Separate text detection, recognition, layout analysis and document understanding.
- Compare geometric and segmentation-based text detectors.
- Decode character sequences with CTC, attention or Transformer models.
- Understand layout-aware and OCR-free document models.
- Evaluate recognition, detection and downstream extraction with appropriate protocols.

## The big picture

Text is everywhere in images: street signs, product labels, license plates, receipts, forms, slides and screenshots. Reading it turns pixels into a string that a program can search, translate, check or store. This is **optical character recognition** (**OCR**). The hard version, text in ordinary photos, is called **scene text** reading; the structured version, scans and PDFs of pages, is **document** analysis.

> **Explain like I'm five**
>
> Think of how you read a poster on a wall. First your eyes find the patches that have writing on them (that is **detection**). Then you look closely at one patch and sound out the letters, using what words usually look like to guess a smudged one (that is **recognition**). Finally you work out what the poster means: the big line at the top is the title, the small line at the bottom is the date, and the two columns are read one after the other (that is **layout and understanding**). A computer does the same three jobs, and it can make a mistake at each one.

Why a separate chapter? A word is long, thin, often rotated or curved, and packed close to its neighbors; its "class" is not a label from a fixed list but any string over an alphabet that may hold tens of thousands of symbols; and one wrong character can change the meaning ("1046" versus "1O46" on an invoice). The field therefore has its own detectors, sequence recognizers, metrics and benchmarks. It is now being reshaped by models that read a whole page and write structured text directly, which makes the pipeline optional but adds a new risk: fluent output that the image does not support. Figure 1 shows the pipeline and where each stage typically fails.

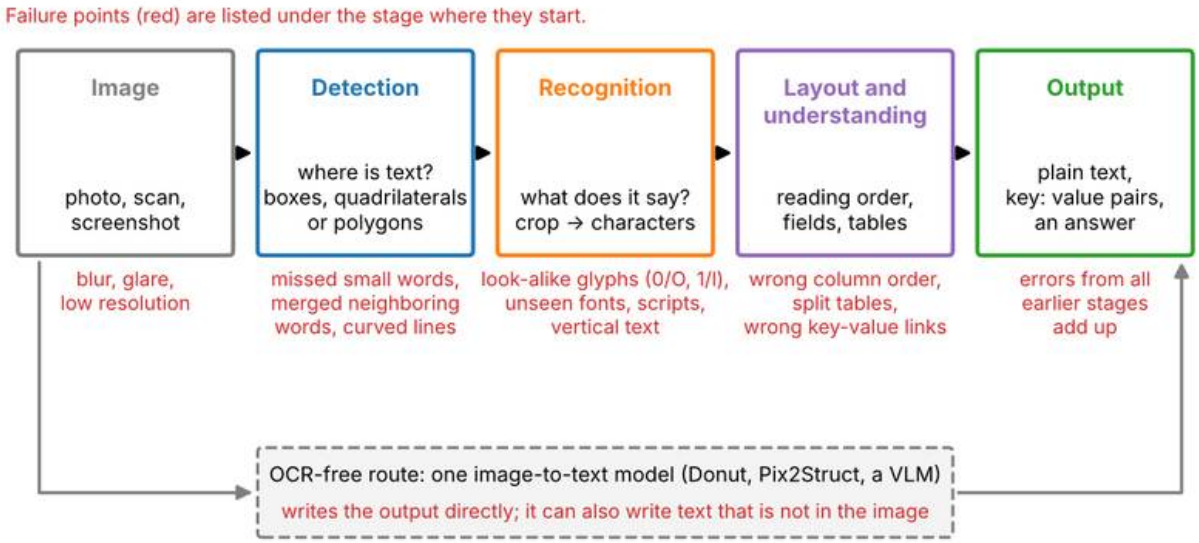



*Figure 1 — The reading pipeline. Each stage has its own typical failures (red), and errors made early cannot be repaired later. OCR-free models replace the middle stages with one image-to-text model.*

This chapter assumes DLCV 03 for boxes, IoU, NMS and precision–recall. Other ideas are linked rather than repeated: the CTC algorithm (PR 06's CTC note), edit distance (PR 06), attention (NLP 07) and vision-language models (ML 10). Surveys by Long et al. [1] and Chen et al. [2] frame scene text; Cui et al. [3] and Zhang et al. [4] cover documents.

## The reading pipeline and where it breaks

**Plain version.** A classical OCR system is a chain: find text regions, read each one, then order the readings and attach meaning. Every link passes its mistakes to the next.

### Scene text versus document text

The two settings differ in what is hard.

| | Scene text | Document text |
|---|---|---|
| Source | photos, video, street views | scans, PDFs, screenshots, phone photos of pages |
| Amount | a few words per image | hundreds to thousands of words per page |
| Geometry | any orientation, perspective, curves | mostly horizontal lines; columns, tables |
| Appearance | decorative fonts, clutter, glare, blur | small print, noise, stamps, handwriting |
| Main difficulty | finding and reading each word | reading order, structure and meaning |

Benchmarks follow the split. The Robust Reading competitions held with ICDAR (the International Conference on Document Analysis and Recognition) defined many of the standard scene-text benchmarks: ICDAR 2013 with focused, mostly horizontal English text [5], and ICDAR 2015's *Incidental Scene Text* challenge [6], recorded with wearable cameras (Google Glass), where words are often small, blurred and rotated [7]. Document benchmarks ask for layout regions, form fields or answers about a page (Document understanding).

### Four places where errors start

Errors are easier to fix when you know which stage produced them.

1. **Image.** Blur, glare, low resolution and compression remove the evidence. No later stage can recover a character that is not in the pixels; a generative model that "recovers" it is guessing (see Failure modes).
2. **Detection.** A missed word is lost for good; two words merged into one box are read as one string; a box that cuts off the last letter produces a deletion.
3. **Recognition.** Look-alike glyphs (0/O, 1/l/I, rn/m), unseen fonts and scripts, and vertical text cause substitutions.
4. **Layout and understanding.** Correct words in the wrong order, a table read across instead of down, or a value attached to the wrong key give a wrong answer from perfect characters.

A useful habit: evaluate each stage on its own (recognition on ground-truth crops, for example) before the whole chain, so you know which stage to improve.

## Before deep learning: thresholds and connected components

**Plain version.** On clean pages, threshold dark pixels, group connected regions and fit boxes. Uneven backgrounds and touching characters expose this baseline's limits.

The classical recipe has three steps. **Binarization** labels each pixel as ink or paper by comparing it with a threshold. A single global threshold, for example the one chosen by Otsu's method [8] (derived in IP 10), fails under uneven lighting, so document systems use a local threshold computed in a window around each pixel. Sauvola's method [9], designed for document images, is a common choice:

$$T(x,y) = m(x,y)\left[1 + k\left(\frac{s(x,y)}{R} - 1\right)\right],$$

where $T(x,y)$ is the threshold at pixel $(x,y)$, $m(x,y)$ and $s(x,y)$ are the mean and standard deviation of the gray levels in a $w \times w$ window centered on the pixel, $R$ is the dynamic range of the standard deviation, and $k > 0$ (scikit-image's default is 0.2) controls how much local contrast lowers the threshold. A pixel darker than $T$ is ink. **Grouping** then joins ink pixels into **connected components** (IP 09), usually after a horizontal dilation that bridges the gaps between the letters of a word. **Box extraction** takes the bounding box of each component.

The snippet renders a small invoice with matplotlib, darkens its left side, adds noise, and runs this recipe with a global and a local threshold. It scores the boxes with the precision, recall and H-mean used by text benchmarks (defined precisely in Measuring OCR): a predicted box counts as correct if its IoU with a still-unmatched ground-truth word is at least 0.5.

```
import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from scipy import ndimage as ndi
from skimage.filters import threshold_otsu, threshold_sauvola

H, W = 200, 560
WORDS = [("INVOICE", 20, 18, 22), ("No.", 200, 24, 14), ("1046", 245, 24, 14),
         ("Date:", 330, 24, 14), ("2026-11-01", 385, 24, 14),
         ("Item", 20, 80, 13), ("Qty", 260, 80, 13), ("Price", 400, 80, 13),
         ("Paper", 20, 110, 13), ("12", 265, 110, 13), ("3.50", 405, 110, 13),
         ("Total", 20, 160, 15), ("42.00", 395, 160, 15)]     # (word, x, y, font size)

def render(words):
    """Draw words with matplotlib; return a gray image in [0, 1] and word boxes."""
    fig = plt.figure(figsize=(W / 100, H / 100), dpi=100)
    texts = [fig.text(x / W, 1 - y / H, s, fontsize=fs, va="top") for s, x, y, fs in words]
    fig.canvas.draw()
    img = np.asarray(fig.canvas.buffer_rgba())[..., :3].mean(axis=2) / 255.0
    r = fig.canvas.get_renderer()
    boxes = [[b.x0, H - b.y1, b.x1, H - b.y0]
             for b in (t.get_window_extent(r) for t in texts)]
    plt.close(fig)
    return img, np.array(boxes)

def iou(a, b):
    w = min(a[2], b[2]) - max(a[0], b[0]); h = min(a[3], b[3]) - max(a[1], b[1])
    inter = max(w, 0) * max(h, 0)
    area = lambda r: (r[2] - r[0]) * (r[3] - r[1])
    return inter / (area(a) + area(b) - inter)

def detect(img, thresh, gap=9):
    ink = img < thresh                                    # dark pixels = ink
    ink = ndi.binary_opening(ink, np.ones((2, 2)))       # drop isolated noise pixels
    words = ndi.binary_dilation(ink, np.ones((3, gap)))  # bridge gaps between letters
    lab, _ = ndi.label(words)
    out = []
    for y, x in ndi.find_objects(lab):
        if (y.stop - y.start) * (x.stop - x.start) > 40:   # ignore tiny blobs
            out.append([x.start, y.start, x.stop, y.stop])
    return np.array(out, float)

def h_mean(pred, gt, thr=0.5):
    used, tp = set(), 0
    for p in pred:                                        # greedy one-to-one matching
        cand = [(iou(p, g), j) for j, g in enumerate(gt) if j not in used]
        best = max(cand, default=(0, -1))
        if best[0] >= thr:
```

```
            used.add(best[1]); tp += 1
    P, R = tp / max(len(pred), 1), tp / len(gt)
    return P, R, 2 * P * R / max(P + R, 1e-9)

rng = np.random.default_rng(0)
clean, gt = render(WORDS)
shade = 0.45 + 0.55 * np.linspace(0, 1, W)[None, :]      # light falls off to the left
img = np.clip(clean * shade + rng.normal(0, 0.04, clean.shape), 0, 1)

t_sau = threshold_sauvola(img, window_size=31)
for name, t, gap in [("Otsu, gap 9", threshold_otsu(img), 9), ("Sauvola, gap 3", t_sau, 3),
                     ("Sauvola, gap 9", t_sau, 9), ("Sauvola, gap 25", t_sau, 25)]:
    pred = detect(img, t, gap)
    P, R, F = h_mean(pred, gt)
    print(f"{name:16s} boxes={len(pred):2d}  P={P:.2f}  R={R:.2f}  H-mean={F:.2f}")
```

```
Otsu, gap 9      boxes= 8  P=0.62  R=0.38  H-mean=0.48
Sauvola, gap 3   boxes=33  P=0.15  R=0.38  H-mean=0.22
Sauvola, gap 9   boxes=13  P=1.00  R=1.00  H-mean=1.00
Sauvola, gap 25  boxes=11  P=0.64  R=0.54  H-mean=0.58
```

Figure 2 shows what happened. The global threshold marks the whole dark left side as ink, so the words there vanish into one large false box. The local threshold recovers every word, but only for one hand-chosen gap: a narrow dilation splits words into letters, a wide one merges “No.” with “1046” and “Date:” with the date. All numbers here come from this synthetic toy.

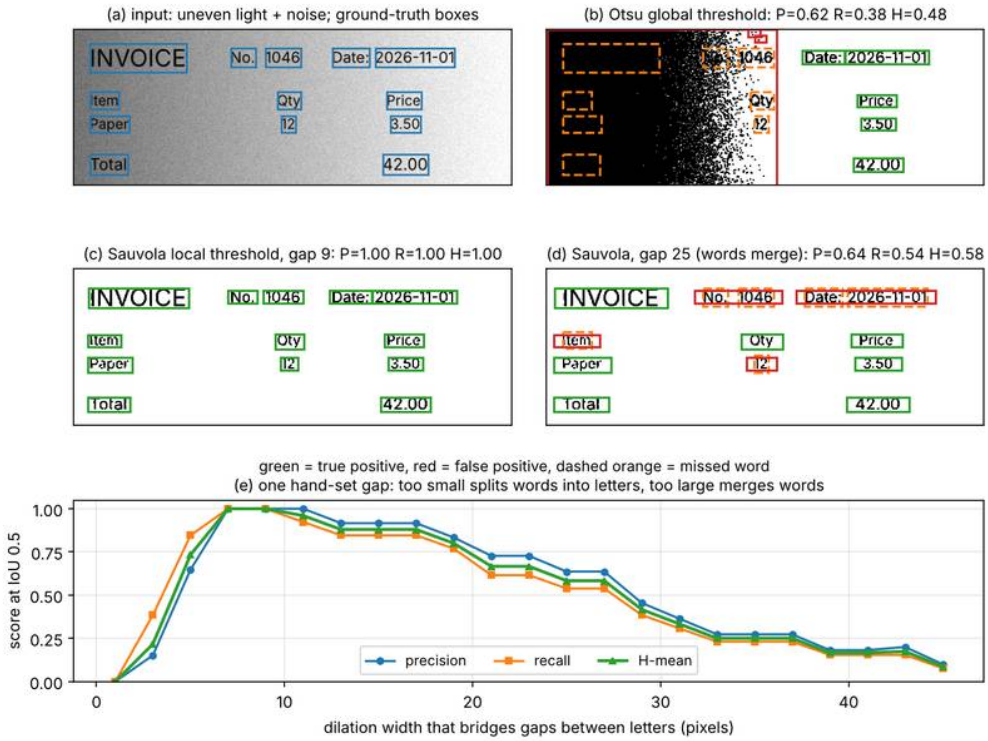


*Figure 2 — Classical detection on a synthetic invoice (toy data). (a) Input with ground truth. (b) A global threshold fails under uneven light. (c) A local (Sauvola) threshold with a 9-pixel gap finds every word. (d) A 25-pixel gap merges neighbors. (e) Scores at IoU 0.5 against the gap: one fixed rule cannot suit every font size and spacing.*

This experiment is the whole motivation for learned text detectors. Real images have every font size at once, text on textured backgrounds, rotation, and words closer together than letters inside other words. A network can learn *where text is* from context and appearance, and it can learn to keep neighbors apart.

## Text detection

**Plain version.** Text detectors must handle elongated, rotated, curved and closely spaced regions, using geometric predictions or text masks.

### Why text is not just another object

The detectors of DLCV 03 predict upright (axis-aligned) boxes. For text this breaks in three ways.

- **Extreme aspect ratios.** A text line can be many times wider than it is tall, far outside anchor shapes tuned for ordinary objects.
- **Orientation and curvature.** Words on signs and packages are rotated, seen in perspective, or follow a curve. Benchmarks therefore annotate quadrilaterals (ICDAR 2015 [6]) or polygons (Total-Text [10], CTW1500 [11]), and compute IoU between polygons instead of rectangles.
- **Dense neighbors.** Words on a line are a few pixels apart; merging them, or letting NMS delete one, loses recall.

The first two problems combine badly. The snippet computes the IoU between a long thin word and the tightest upright box around it, as the word rotates.

```
import numpy as np
from skimage.draw import polygon

def rotated_word(cx, cy, w, h, deg):
    """Corners of a w x h word box rotated by deg around its center (x, y)."""
    t = np.deg2rad(deg)
    R = np.array([[np.cos(t), -np.sin(t)], [np.sin(t), np.cos(t)]])
    corners = np.array([[-w, -h], [w, -h], [w, h], [-w, h]]) / 2
    return corners @ R.T + [cx, cy]

def mask(quad, shape=(300, 300)):
    m = np.zeros(shape, bool)
    rr, cc = polygon(quad[:, 1], quad[:, 0], shape)
    m[rr, cc] = True
    return m

for deg in [0, 10, 20, 30, 45]:
```

```
    quad = rotated_word(150, 150, 160, 24, deg)          # a long, thin word
    x0, y0 = quad.min(axis=0); x1, y1 = quad.max(axis=0)
    aabb = np.array([[x0, y0], [x1, y0], [x1, y1], [x0, y1]])  # tightest upright box
    a, b = mask(quad), mask(aabb)
    print(f"angle {deg:2d} deg: IoU(word, its upright box) = {(a & b).sum() / (a |
     ↪  b).sum():.2f}")
```

```
angle  0 deg: IoU(word, its upright box) = 1.00
angle 10 deg: IoU(word, its upright box) = 0.47
angle 20 deg: IoU(word, its upright box) = 0.31
angle 30 deg: IoU(word, its upright box) = 0.25
angle 45 deg: IoU(word, its upright box) = 0.22
```

For a $w \times h$ rectangle rotated by $\theta$ the exact value is $wh/[(w\cos\theta + h\sin\theta)(w\sin\theta + h\cos\theta)]$, where $w$ and $h$ are the word's width and height and the denominator is the area of its upright bounding box. Already at 10 degrees the *best possible* upright box fails the usual IoU 0.5 test (Figure 3). Text detectors therefore output rotated rectangles, quadrilaterals or polygons; how IoU is computed for rotated shapes is the same polygon-clipping problem solved for bird's-eye-view boxes in DLCV 03B.

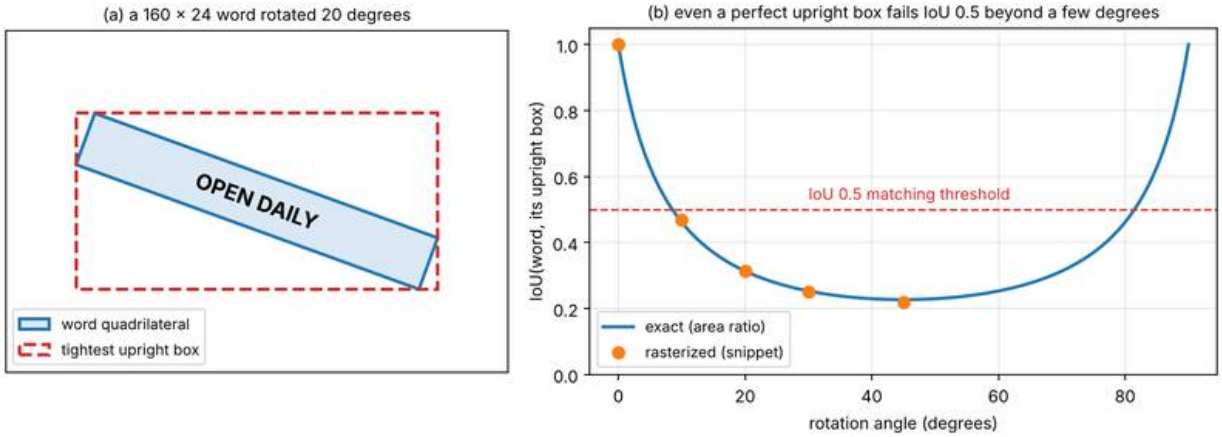



*Figure 3 — Why text detectors predict quadrilaterals. (a) A 160 × 24 word rotated 20° and its tightest upright box. (b) Their IoU against the angle: beyond about 8.5° even a perfect upright box is counted as a miss at IoU 0.5.*

### Regression-based detectors: EAST

**Plain version.** Each text pixel predicts confidence and word geometry. Merge consistent high-confidence predictions into detections.

EAST (Efficient and Accurate Scene Text detector) [12] is a fully convolutional network with a U-shaped feature-merging branch that outputs, at each location of a downsampled map, a **score** in $[0, 1]$ (is this pixel inside a shrunk version of a word?) and a **geometry**. Two geometries were

proposed. **RBOX** has five channels: the distances from the pixel to the four sides of a rotated rectangle and the rectangle's rotation angle. **QUAD** has eight channels: the offsets $(\Delta x_i, \Delta y_i)$, $i = 1, \dots, 4$, from the pixel to the four corners of a quadrilateral. Because tens of thousands of pixels vote, EAST first merges neighboring geometries row by row, averaging them with their scores as weights (**locality-aware NMS**), and only then runs standard NMS [12]. One network pass replaces the candidate-grouping steps of earlier systems. Its weakness is reach: a pixel must predict edges that may be hundreds of pixels away, and a quadrilateral cannot follow a curved word.

### Segmentation-based detectors: PSENet and DBNet

**Plain version.** Instead of asking each pixel for the whole word's shape, ask only "is this pixel text?" That gives a mask that can follow any curve. The difficulty is splitting a mask into words when the words touch, so these detectors predict a *thinner* core for each word, keep the cores apart, and grow them back afterwards.

The network is an encoder–decoder like those of DLCV 04, predicting a per-pixel text **probability map** $P$; the trick is in the labels. Each ground-truth polygon $G$ with area $A$ and perimeter $L$ is shrunk inward by an offset

$$D = \frac{A\,(1 - r^2)}{L},$$

where $r \in (0, 1)$ is a shrink ratio; the shrunk polygon (the word's **kernel**) is the positive region. Kernels of adjacent words are separated by a margin even when the words touch, so connected components of the predicted kernel map are word instances; each is then dilated back to full size.

**PSENet** (Progressive Scale Expansion Network) [13] predicts several kernels per word at different shrink ratios and grows the smallest, well-separated ones into the larger ones with a breadth-first search, so a pixel claimed by one word cannot be taken by its neighbor. The price is a slow, sequential post-process.

**DBNet** [14] keeps a single kernel and makes the binarization step itself trainable. A standard binarization $B = \mathbf{1}[P \geq t]$ with a fixed threshold $t$ has zero gradient almost everywhere, so it cannot be trained. DBNet predicts a second map, a **threshold map** $T$, and uses the approximate step

$$\hat{B}_{i,j} = \frac{1}{1 + e^{-k\,(P_{i,j} - T_{i,j})}},$$

where $\hat{B}_{i,j}$ is the approximate binary value at pixel $(i, j)$, $P_{i,j}$ the predicted text probability, $T_{i,j}$ the learned threshold and $k$ an amplifying factor set to 50 [14]. Writing $x = P_{i,j} - T_{i,j}$ and $f(x) = \hat{B}_{i,j}$, the binary cross-entropy for a text pixel, $\ell_+ = -\log f(x)$, has derivative $\partial \ell_+ / \partial x = -k\,(1 - f(x))$, and for a background pixel, $\ell_- = -\log(1 - f(x))$, $\partial \ell_- / \partial x = k f(x)$. The gradient is scaled by $k$ and is largest where a pixel is on the wrong side of its threshold. The threshold map is supervised to be high in a band around each word's border, so the binarization is strict where words meet and relaxed inside them.

The snippet checks the derivative numerically and shows, on a 1-D profile, a threshold that is high between two words separating them where a fixed threshold merges them.

```
import numpy as np

def db(P, T, k=50.0):
    """Differentiable binarization: an approximate step at the learned threshold T."""
    return 1.0 / (1.0 + np.exp(-k * (P - T)))

x = np.array([-0.10, -0.02, 0.0, 0.02, 0.10])            # x = P - T at five pixels
for k in [1.0, 50.0]:
    f = db(x, 0.0, k)
    g_pos = -k * (1 - f)       # d/dx of -log f       (pixel labeled text)
    g_neg = k * f              # d/dx of -log(1 - f)  (pixel labeled background)
    # check the text-pixel gradient by finite differences
    eps = 1e-6
    num = (-np.log(db(x + eps, 0, k)) + np.log(db(x - eps, 0, k))) / (2 * eps)
    assert np.allclose(num, g_pos, rtol=1e-4)
    print(f"k={k:4.0f}  B_hat={np.round(f, 3)}  dl+/dx={np.round(g_pos, 2)}
     ↪  dl-/dx={np.round(g_neg, 2)}")

# Two words whose probability map P does not drop to 0 between them.
P = np.array([0.1, 0.9, 0.95, 0.9, 0.55, 0.5, 0.55, 0.9, 0.95, 0.9, 0.1])
T = np.array([0.3, 0.3, 0.3, 0.3, 0.7, 0.7, 0.7, 0.3, 0.3, 0.3, 0.3])   # high at the gap
print("fixed t=0.3:", (P >= 0.3).astype(int))
print("DB, k=50   :", (db(P, T) > 0.5).astype(int))
```

```
k=   1  B_hat=[0.475 0.495 0.5   0.505 0.525]  dl+/dx=[-0.52 -0.5  -0.5  -0.5  -0.48]
 ↪  dl-/dx=[0.48 0.5  0.5  0.5  0.52]
k=  50  B_hat=[0.007 0.269 0.5   0.731 0.993]  dl+/dx=[-49.67 -36.55 -25.   -13.45  -0.33]
 ↪  dl-/dx=[ 0.33 13.45 25.   36.55 49.67]
fixed t=0.3: [0 1 1 1 1 1 1 1 1 1 0]
DB, k=50   : [0 1 1 1 0 0 0 1 1 1 0]
```

With $k = 1$ the "binary" map is a gray blur near 0.5 and every pixel gets the same small gradient; with $k = 50$ it is nearly binary, and a text pixel 0.1 below its threshold gets a gradient of about $-50$ while a correct one gets almost none. Figure 4 repeats the separation in 2-D.

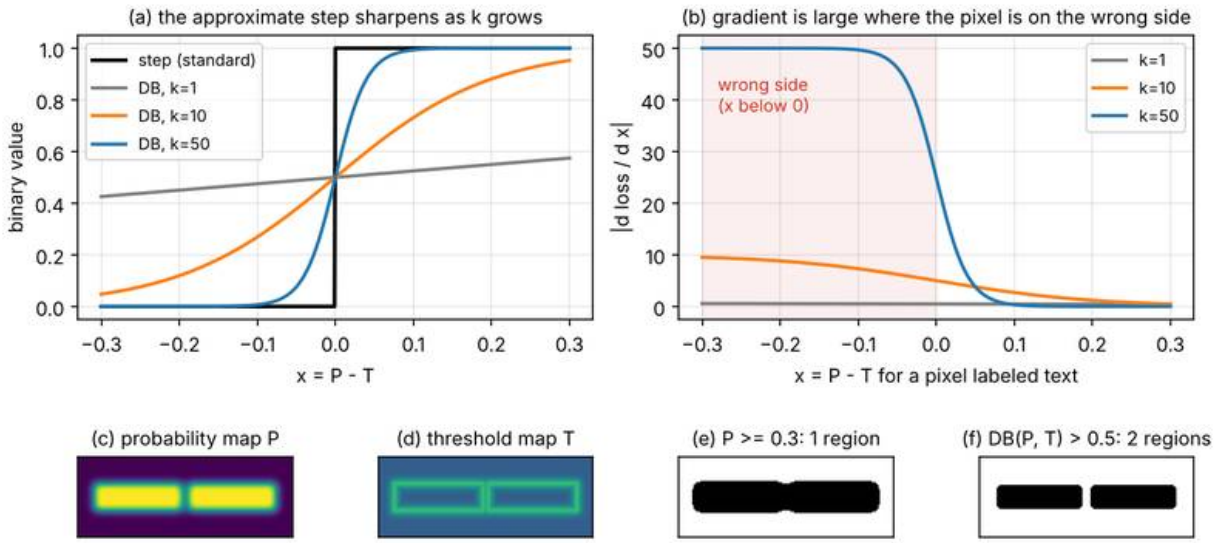



*Figure 4 — Differentiable binarization.* (*a*) *The approximate step for several k.* (*b*) *The text-pixel gradient is amplified by k on the wrong side of the threshold.* (*c*)–(*f*) *Toy maps: a probability map that blurs across a 4-pixel gap merges two words at a fixed threshold, while a threshold map that is high at borders* (*hand-made here, learned in DBNet*) *keeps them apart.*

At inference DBNet drops the threshold branch: it binarizes $P$ at a constant 0.2, takes connected components, and expands each region by $D' = A'r'/L'$, with $A'$ and $L'$ the region's area and perimeter and $r' = 1.5$ [14]. The threshold map is thus mainly a training device that makes $P$ itself sharp, and the simple post-processing keeps the detector fast.

*Online only: Shrinking and expanding text polygons — https://blog.shyandram.dev/en/learn/deep-learning/text-in-images/notes/shrinking-and-expanding-text-polygons/*

*Online only: DBNet: a reading guide — https://blog.shyandram.dev/en/learn/deep-learning/text-in-images/papers/dbnet-a-reading-guide/*

### Character-level and curved-text detectors

Some detectors find letters first and glue neighbors into words. CRAFT [15] predicts a **region score** (center of a character) and an **affinity score** (between two characters of one word) and links characters into words of any shape; because real datasets label only words, it estimates character labels on real images with an interim model. Polygon detectors such as the one introduced with CTW1500 [11] instead regress many boundary points along a curved line. Richer shape models follow irregular text better but need more complex outputs or post-processing.

## Text recognition

**Plain version.** Recognition takes one cropped word or line and outputs its characters. The modern approach reads the crop like a sequence: slide along it from left to right, describe each thin slice, and turn the sequence of slices into a sequence of characters, without ever cutting the word into separate letters.

### Recognition as sequence prediction

Early systems cut a word into characters and classified each, which fails when letters touch or break. **CRNN** (Convolutional Recurrent Neural Network) [16] made recognition segmentation-free and end-to-end trainable. A crop is resized to a fixed height; a CNN turns it into a feature map whose columns form a **feature sequence**, each column describing a thin vertical strip of the image; a deep bidirectional LSTM (see ML 04) adds left and right context; and a transcription layer turns per-column predictions into a label sequence. With rectangular pooling windows, a $100 \times 32$ input yields a sequence of 25 frames, longer than most English words [16]. Nothing tells the network which frame belongs to which character; CTC handles that missing alignment.

### CTC: training without alignments

**Plain version.** At every frame the network may say a character or "nothing new" (a **blank**). To read the output, merge repeated characters and delete the blanks. To train, add up the probability of every frame-by-frame story that reads out as the correct word.

**Connectionist temporal classification** (CTC) [17] defines, for per-frame probabilities $q_t(s)$ of each symbol $s$ (a character of the alphabet or the blank $\emptyset$) at frame $t$, a collapsing map $\mathcal{B}$ (merge repeats, then remove blanks) and the probability of a label sequence $y$ given the input $x$,

$$p(y \mid x) = \sum_{z:\ \mathcal{B}(z)=y} \prod_{t=1}^{T} q_t(z_t),$$

where $z = (z_1, \dots, z_T)$ is an alignment (one symbol per frame), $T$ the number of frames, and the sum runs over every alignment that collapses to $y$. Training minimizes $-\log p(y \mid x)$, computed by a forward recursion in $O(TL)$ for a label of length $L$. The full derivation, with a brute-force check, is in PR 06's CTC note; here we only need what it implies for decoding.

The cheapest decoder is **greedy** (best-path) decoding: take the most likely symbol at each frame and collapse. CRNN uses exactly this approximation in its lexicon-free mode [16]. But the most likely *path* need not collapse to the most likely *text*, because a text's probability is spread over many paths. The snippet builds a five-frame toy where this matters.

```
import itertools
import numpy as np

alphabet = ["-", "c", "a", "t"]                # index 0 is the CTC blank
```

```
# Per-frame probabilities from a recognizer (rows: 5 frames, columns: symbols).
probs = np.array([[0.10, 0.80, 0.05, 0.05],
                  [0.20, 0.05, 0.70, 0.05],
                  [0.60, 0.05, 0.30, 0.05],
                  [0.55, 0.02, 0.03, 0.40],
                  [0.55, 0.02, 0.03, 0.40]])

def collapse(path):
    """CTC map B: merge repeated symbols, then remove blanks."""
    merged = [s for i, s in enumerate(path) if i == 0 or s != path[i - 1]]
    return "".join(alphabet[s] for s in merged if s != 0)

greedy = probs.argmax(axis=1)                    # best symbol at every frame
print("greedy path:", "".join(alphabet[s] for s in greedy), "->", repr(collapse(greedy)))

# Exact: add up every alignment that collapses to the same text (4**5 = 1024 paths).
total = {}
for path in itertools.product(range(4), repeat=len(probs)):
    p = np.prod(probs[np.arange(len(probs)), path])
    total[collapse(path)] = total.get(collapse(path), 0.0) + p
for text, p in sorted(total.items(), key=lambda kv: -kv[1])[:3]:
    print(f"p({text!r}) = {p:.3f}")
print("collapse('caa-at') =", repr(collapse([1, 2, 2, 0, 2, 3])),
      " collapse('caaat') =", repr(collapse([1, 2, 2, 2, 3])))
```

```
greedy path: ca--- -> 'ca'
p('cat') = 0.363
p('ca') = 0.179
p('ct') = 0.091
collapse('caa-at') = 'caat'  collapse('caaat') = 'cat'
```

The "t" is never the single most likely symbol in any frame, so greedy decoding drops it, yet the alignments that contain a "t" in frame 4 or 5 add up to twice the probability of "ca" (Figure 5). The last line shows the blank's second job: a doubled letter survives only with a blank between its copies. Real decoders use **prefix beam search**, which keeps the best few partial transcriptions and merges alignments that collapse to the same prefix.

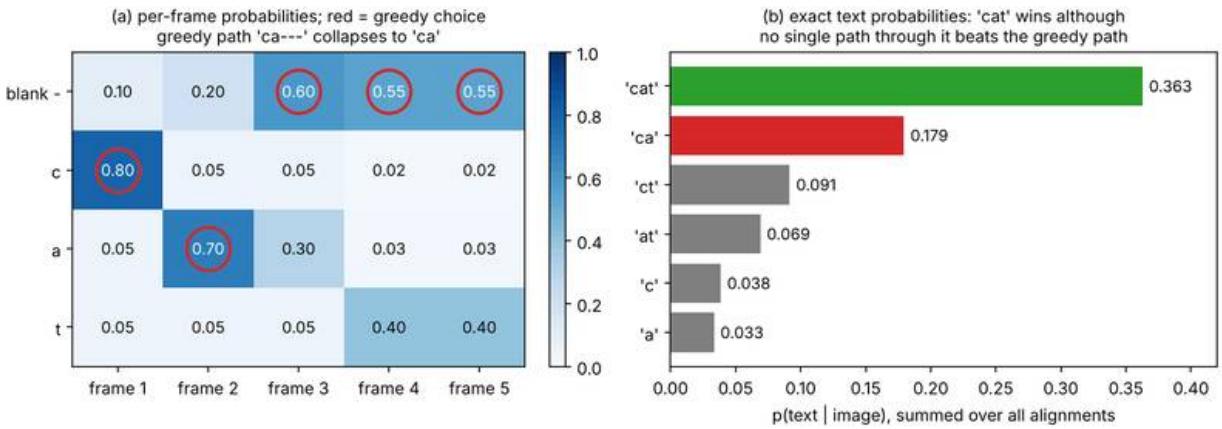


*Figure 5 — CTC decoding on a toy matrix. (a) Greedy decoding picks 'ca—', which collapses to 'ca'. (b) Summing over all 1,024 alignments, 'cat' is twice as likely as 'ca', although every single path that reads 'cat' is less likely than the greedy path.*

*Online only: CTC prefix beam search, step by step — https://blog.shyandram.dev/en/learn/deep-learning/text-in-images/notes/ctc-prefix-beam-search-step-by-step/*

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

CTC has two built-in limits (see PR 06's note): frame outputs are independent given the image, so the model cannot condition on characters it has already emitted, and the alignment is monotonic, so reading order must match frame order. Vertical or curved text breaks the second unless it is rectified first.

### Attention decoders and rectification

**Plain version.** An attention decoder writes the word one character at a time. Before each character it looks back at the image features and chooses where to look; it also remembers what it has already written, so it behaves like a small language model of spelling.

An **attention decoder** is the encoder–decoder of NLP 07 applied to image features: at step $i$ it attends to the feature sequence and predicts character $y_i$ from the result and from $y_1, \dots, y_{i-1}$, until an end token. It drops CTC's independence assumption at the price of sequential decoding. **ASTER** [18] pairs such a decoder with a **rectification network**: a thin-plate spline (TPS) transformation, predicted from the image and trained only through the recognition loss, warps a curved or perspective word into a straight horizontal one before reading it.

Baek et al. [7] showed that most recognizers of the time fit one four-stage template — **transformation** (none or TPS), **feature extraction**, **sequence modeling** (none or BiLSTM) and **prediction** (CTC or attention) — and, retraining all combinations on the same data, that attention mainly costs speed and that earlier comparisons had been distorted by different training sets and test subsets.

*Online only: Baek et al., "What Is Wrong With Scene Text Recognition Model Comparisons?" (ICCV 2019) — https://blog.shyandram.dev/en/learn/deep-learning/text-in-images/papers/baek-et-al-what-is-wrong-with-scene-text-recognition-model-comparisons-iccv-2019/*

### Transformer recognizers

**TrOCR** [19] replaces both the CNN and the RNN with pre-trained transformers. The image is resized to 384 × 384 and split into 16 × 16 patches, as in a Vision Transformer (ML 05); an image encoder initialized from DeiT or BEiT encodes the patches, and a text decoder initialized from RoBERTa or MiniLM generates **wordpieces** (subword tokens) autoregressively, decoded with

beam search [19]. It is pre-trained on synthetic text images and fine-tuned on printed, handwritten and scene text: recognition now borrows its backbones and decoder from general vision and language pre-training.

**PARSeq** [20] trains one transformer decoder on many random orders of the output positions (**permutation language modeling**), so the same weights can decode left to right, decode all characters in parallel without context, or refine a first guess using context on both sides.

### Language models: help and harm

A recognizer that "knows English" can fix a smudged letter: "PLEA_E" is almost certainly "PLEASE". The knowledge can be **implicit** (an autoregressive decoder conditions on its previous characters), a **lexicon** (CRNN can restrict the output to a word list [16]) or an **explicit language model**: in ABINet [21] a separate bidirectional language model iteratively corrects the vision model's output.

The same prior can hurt. Text that is not language — serial numbers, codes, prices, names, mixed scripts — is exactly where OCR matters most and where a language model "corrects" correct readings. PARSeq's authors note this risk for external language models that do not see the image [20], and OCRBench reports that large multimodal models are weak on **non-semantic text** (random character strings) as well as on handwriting, multilingual text and mathematical expressions [22]. When the content is not natural language, measure with and without the language prior.

*Online only: Where the language model sits in a recognizer — https://blog.shyandram.dev/en/learn/deep-learning/text-in-images/notes/where-the-language-model-sits-in-a-recognizer/*

### Training data: synthetic text

Labeled real crops are scarce, and recognizers trained only on **synthetic text** generalize well across scene-text datasets [1]. MJSynth [23] renders single words with varied fonts, shadows, distortions and noise (8.9 million word images); SynthText [24] pastes text into natural photos following the scene geometry (5.5 million cropped words) [7]. Training on both beats either alone, and a combined 20% sample of each beats either full set, so diversity can matter more than size [7].

## End-to-end text spotting

**Plain version.** Instead of running a detector and then a separate recognizer on each crop, one network does both and shares its features. Reading helps finding (a region that reads as a word is probably text), and finding helps reading.

A **text spotter** outputs (shape, transcription) pairs directly. FOTS [25] shares convolutional features between a dense per-pixel detection branch and a recognition branch decoded with CTC, through **RoIRotate**, an operator that crops and rotates the feature region of each detected quadrilateral into an upright, fixed-height strip. Mask TextSpotter [26] extends Mask R-CNN (see instance segmentation in DLCV 04) with masks for whole words and for individual characters, so it

can read curved text without a sequence model. ABCNet [27] fits each curved text instance with parametric Bézier curves and samples features along them with **BezierAlign**, a curved analogue of RoIAlign. End-to-end evaluation then asks whether each ground-truth word was both located and read correctly, which we define in Measuring OCR.

## Document understanding

**Plain version.** On a page, reading the words is often the easy part. The questions are which words belong together, in which order to read them, and what they mean: which number is the total, which line is the title, which cell is in which column.

### Layout analysis and reading order

**Layout analysis** finds the regions of a page — paragraphs, titles, lists, tables, figures — usually by object detection or instance segmentation on page images. PubLayNet [28] matched the XML and PDF versions of over a million PubMed Central articles to label over 360 thousand page images automatically with five classes (text, title, list, table, figure). DocLayNet [29] has 80,863 manually annotated pages from diverse sources with 11 classes, in COCO format, because models trained only on scientific articles transfer poorly to reports, manuals and forms.

Given regions and words, the system must also produce a **reading order**. The obvious rule, "top to bottom, then left to right", fails on the simplest multi-column page, as the snippet shows on a two-column notice.

```
import numpy as np

# Word boxes (x0, y0, x1, y1) on a two-column page: each column is one sentence.
left = "Pay the invoice within thirty days of receipt".split()
right = "Late payments carry a two percent monthly fee".split()
words, boxes = [], []
for col, text in [(0, left), (1, right)]:
    for k, w in enumerate(text):
        row, pos = divmod(k, 3)                       # three words per line
        x0 = 40 + 300 * col + 80 * pos
        y0 = 100 + 30 * row + 3 * col                 # columns not perfectly aligned
        words.append(w); boxes.append([x0, y0, x0 + 70, y0 + 20])
boxes = np.array(boxes, float)

def naive_order(boxes, line_tol=10):
    """Top-to-bottom, then left-to-right, treating boxes within line_tol as one line."""
    line = np.round(boxes[:, 1] / line_tol)
    return np.lexsort((boxes[:, 0], line))

def column_order(boxes, line_tol=10):
    """Split the page at the widest empty vertical strip, then read column by column."""
    x_min, x_max = int(boxes[:, 0].min()), int(boxes[:, 2].max())
    covered = np.zeros(x_max - x_min, bool)
    for x0, _, x1, _ in boxes.astype(int):
```

```
        covered[x0 - x_min:x1 - x_min] = True
    edges = np.flatnonzero(np.diff(np.r_[1, covered.astype(int), 1]))  # empty-run bounds
    starts, ends = edges[::2], edges[1::2]
    widest = np.argmax(ends - starts)
    split = x_min + (starts[widest] + ends[widest]) / 2
    col = (boxes[:, 0] > split).astype(int)
    line = np.round(boxes[:, 1] / line_tol)
    return np.lexsort((boxes[:, 0], line, col))

print("naive :", " ".join(words[i] for i in naive_order(boxes)))
print("column:", " ".join(words[i] for i in column_order(boxes)))
```

```
naive : Pay the invoice Late payments carry within thirty days a two percent of receipt
 ↪  monthly fee
column: Pay the invoice within thirty days of receipt Late payments carry a two percent
 ↪  monthly fee
```

Every word in the naive output is correct, and the text is still useless. The column rule fixes this page but is another hand-made rule that fails on tables or figures (Figure 6); learned layout and document models exist largely to replace such rules.

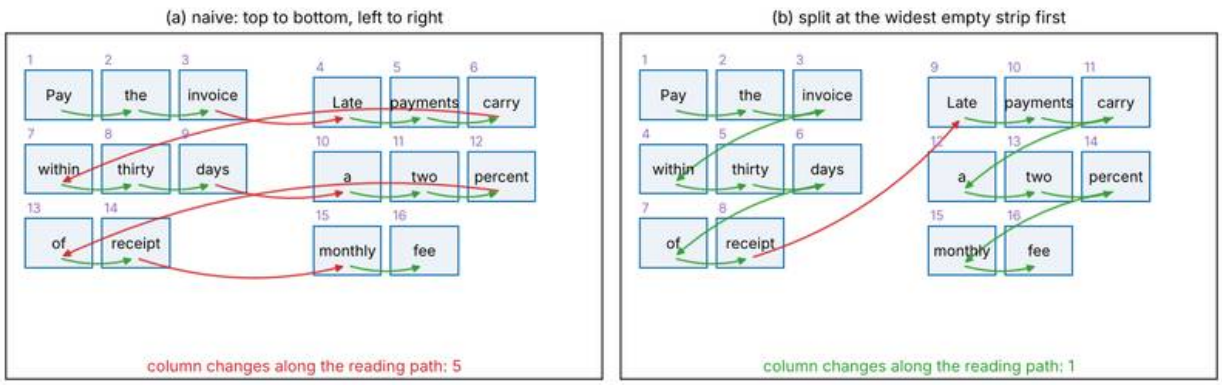



*Figure 6 — Reading order on a synthetic two-column page. (a) Sorting by line then by x interleaves the columns (5 column changes). (b) Splitting at the widest empty vertical strip first gives the intended order (1 change), but only for this kind of layout.*

### Text plus layout: the LayoutLM family

**Plain version.** Give a language model not only the words of a page but also where each word sits. Then "Total" followed by a number at the bottom right of a receipt means something different from the same word in the middle of a paragraph.

LayoutLM [30] starts from BERT (NLP 10) and adds a **2-D position embedding** to each word's token embedding. Each word comes with its OCR box $(x_0, y_0, x_1, y_1)$ — upper-left and lower-right corners — rescaled to integer "virtual" coordinates from 0 to 1,000 so that pages of different sizes

share one scale. Four embedding lookups are added, with one table shared by $x_0$ and $x_1$ and another by $y_0$ and $y_1$:

$$\mathbf{h}_i = \mathbf{e}_{\text{tok}}(w_i) + \mathbf{e}_{\text{pos}}(i) + \mathbf{X}(x_{0,i}) + \mathbf{Y}(y_{0,i}) + \mathbf{X}(x_{1,i}) + \mathbf{Y}(y_{1,i}),$$

where $\mathbf{h}_i$ is the input vector of the $i$-th word $w_i$, $\mathbf{e}_{\text{tok}}$ and $\mathbf{e}_{\text{pos}}$ are BERT's token and 1-D position embeddings, and $\mathbf{X}$, $\mathbf{Y}$ are the learned tables for horizontal and vertical coordinates. Pre-training masks some words but keeps their 2-D positions, and the model predicts them from text and layout context (a **masked visual-language model**), on about 11 million scanned pages OCRed with the open-source Tesseract engine [30], [31]. LayoutLM also adds visual features of each word's image region, computed with a Faster R-CNN model, for downstream tasks [30]. LayoutLMv3 [32] instead feeds the page image as patches, masks both text tokens and image patches and adds a word–patch alignment objective, so one model serves text-centric tasks (forms, receipts, question answering) and image-centric ones (document classification, layout analysis).

**Key information extraction** — pulling "invoice number", "date" and "total" out of a page — is then token classification over the OCR words, as in NLP 06's sequence labeling, plus linking keys to values. FUNSD [33] is a small benchmark of this kind: 199 real, noisy scanned forms whose semantic entities are labeled *question*, *answer*, *header* or *other*, with links between them.

The family's weakness is its input: it cannot fix a word the OCR engine missed or misread, and it inherits the engine's reading order.

### OCR-free models: Donut and Pix2Struct

**Plain version.** Skip the OCR engine. Train one model to look at the page image and *write* the desired output — the full text, a JSON record, or an answer — token by token, the way an image captioner writes a caption.

**Donut** (Document understanding transformer) [34] is a Swin Transformer encoder (ML 05) and a BART decoder. It is pre-trained to read all the text of a page in reading order — a "pseudo-OCR" task on 11 million scanned English documents plus synthetic pages in Chinese, Japanese, Korean and English from its SynthDoG generator — and fine-tuned to emit a token sequence that converts one-to-one into JSON, for classification, information extraction or question answering [34]. It avoids the cost of running OCR, the inflexibility of OCR engines across languages and document types, and the propagation of OCR errors. **Pix2Struct** [35] is pre-trained to parse masked screenshots of web pages into simplified HTML, which mixes OCR, language modeling and captioning in one objective; it accepts variable-resolution inputs and renders the question directly onto the image.

The price is resolution and verifiability. Donut uses 2560 × 1920 inputs and reports that accuracy rises quickly with input size on documents with tiny text, at a growing compute cost [34]; and because the model *generates* rather than *transcribes*, nothing ties each output character to a region of the image.

*Online only: Donut: a reading guide — https://blog.shyandram.dev/en/learn/deep-learning/text-in-images/papers/donut-a-reading-guide/*

### Vision-language models for documents

General multimodal large language models (ML 10) now answer questions about documents and transcribe pages; reading small text depends on tiling or dynamic resolution (What the diagram hides). Three cautions apply. First, a fluent answer is not a transcription: check it against detections or ask for coordinates and verify them. Second, generative readers have their own failure modes: the academic-document model Nougat [36] reports that it sometimes degenerates into repeating a sentence over and over (on 1.5% of its test pages, more often out of domain), and that numbers in reference lists and section titles are sometimes skipped or hallucinated. Third, measure the specific skill: OCRBench [22] gathers 29 datasets to test text recognition, scene-text and document question answering, key information extraction and handwritten mathematical expressions separately, instead of hiding OCR inside one aggregate score.

## Handwriting, many scripts and CJK text

**Plain version.** Handwriting varies from writer to writer and joins letters together; many scripts have far more symbols than the Latin alphabet; and some, such as Chinese, can also be written vertically. Each of these breaks an assumption that English scene-text models make quietly.

### Handwriting

Offline handwriting recognition reads lines of handwritten text from scans. The IAM database [37] of handwritten English sentences is, in the TrOCR authors' words, the most popular dataset for handwritten text recognition; TrOCR uses a split of 6,161 training, 966 validation and 2,915 test lines and reports case-sensitive CER [19]. Line-level recognizers with CTC or attention apply unchanged; the difficulty is variation between writers, connected letters and limited labeled data, which is why synthetic pre-training and large pre-trained decoders help. For Chinese handwriting, the CASIA online and offline databases [38] provide data for both pen-trajectory (online) and scanned (offline) recognition.

### Large character sets and vertical text

Chinese and Japanese use alphabets of tens of thousands of symbols, against a few dozen for Latin script [1]. RCTW-17 lists what makes Chinese text different: a much larger character set, no spaces between words, and many characters made of several unconnected parts [39]. Each has a consequence. The recognizer's output layer grows with the alphabet and most characters are rare: the Chinese Text in the Wild dataset labels 1,018,402 character instances in 32,285 street-view images, covering 3,850 distinct characters, most of them rarely used [40]. Unconnected parts break connected-component grouping. And without spaces, systems work on lines and characters rather than words (see NLP 01).

**Vertical text** is the other difficulty: Chinese can be set vertically as well as horizontally, and street-view datasets such as CTW contain vertical text [40]. The W3C's *Requirements for Chinese Text*

*Layout* describes vertical composition: characters run from top to bottom and lines from right to left, while embedded Latin words are either set upright or rotated 90° clockwise [41]. A recognizer that assumes left-to-right frames on a horizontal crop reads such a column as garbage; Baek et al. list vertical text among the failure cases common to all 24 recognizers they tested [7]. The remedy is to decide each detected line's orientation first and rotate or rearrange it before reading.

*Online only: Reading vertical and mixed-direction text — https://blog.shyandram.dev/en/learn/deep-learning/text-in-images/notes/reading-vertical-and-mixed-direction-text/*

### Many scripts at once

Multilingual systems must also decide *which* script they are looking at. The ICDAR 2019 multilingual challenge (MLT-2019) [42] has 20,000 real images with text in 10 languages, and includes a script-identification task; its languages cover seven scripts [2].

## Measuring OCR

**Plain version.** Count mistakes the way a proofreader would: how many characters must be inserted, deleted or changed to turn the output into the truth. For detection, count how many found boxes are right and how many real words were found, as in object detection.

### CER and WER from edit distance

The **edit distance** (Levenshtein distance) between a reference string $r$ and a recognized string $h$ is the minimum number of insertions, deletions and substitutions that turn one into the other; the Wagner–Fischer dynamic program computes it [43] (derived in PR 06). Following an optimal alignment back through the table splits the total into $S$ substitutions, $D$ deletions and $I$ insertions. Then

$$\mathrm{CER} = \frac{S + D + I}{N}, \qquad \mathrm{WER} = \frac{S_w + D_w + I_w}{N_w},$$

where $N$ is the number of characters in the reference, and $S_w$, $D_w$, $I_w$ and $N_w$ are the same quantities counted over words (sequences split at spaces). Both are error rates: 0 is perfect, and they can exceed 1 when the output contains many insertions. Other conventions exist: scene-text recognition on cropped words usually reports **word accuracy** (the fraction of crops read exactly), some papers call one minus that the word error rate, and multilingual competitions report one minus a **normalized edit distance** (NED) [2]. Always state which one you report.

```
import numpy as np

def edit_ops(ref, hyp):
    """Levenshtein alignment of two sequences; returns (S, D, I) counts."""
    n, m = len(ref), len(hyp)
    E = np.zeros((n + 1, m + 1), int)
```

```
    E[:, 0], E[0, :] = np.arange(n + 1), np.arange(m + 1)
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            E[i, j] = min(E[i - 1, j] + 1, E[i, j - 1] + 1,
                          E[i - 1, j - 1] + (ref[i - 1] != hyp[j - 1]))
    S = D = I = 0
    i, j = n, m
    while i > 0 or j > 0:                       # walk back along one optimal path
        if i > 0 and j > 0 and E[i, j] == E[i - 1, j - 1] + (ref[i - 1] != hyp[j - 1]):
            S += ref[i - 1] != hyp[j - 1]; i, j = i - 1, j - 1
        elif i > 0 and E[i, j] == E[i - 1, j] + 1:
            D += 1; i -= 1
        else:
            I += 1; j -= 1
    return S, D, I

def rates(ref, hyp):
    c = edit_ops(ref, hyp)
    w = edit_ops(ref.split(), hyp.split())
    return c, sum(c) / len(ref), w, sum(w) / len(ref.split())

ref = "Total due 25.80 EUR by 1 May"
for hyp in ["Total due 25.80 EUR by 1 May",
            "Tota1 due 25.8O EUR by l May",      # three look-alike glyphs
            "Total due25.80 EUR by 1 May",       # one missing space
            "Total due 25.80 EUR"]:              # last words not read
    c, cer, w, wer = rates(ref, hyp)
    print(f"{hyp:30s} chars S,D,I={c} CER={cer:.3f} | words S,D,I={w} WER={wer:.3f}")
```

```
Total due 25.80 EUR by 1 May   chars S,D,I=(0, 0, 0) CER=0.000 | words S,D,I=(0, 0, 0)
 ↪  WER=0.000
Tota1 due 25.8O EUR by l May   chars S,D,I=(3, 0, 0) CER=0.107 | words S,D,I=(3, 0, 0)
 ↪  WER=0.429
Total due25.80 EUR by 1 May    chars S,D,I=(0, 1, 0) CER=0.036 | words S,D,I=(1, 1, 0)
 ↪  WER=0.286
Total due 25.80 EUR            chars S,D,I=(0, 9, 0) CER=0.321 | words S,D,I=(0, 3, 0)
 ↪  WER=0.429
```

Three look-alike substitutions cost 3 of 28 characters (CER 0.107) but ruin 3 of 7 words (WER 0.429), and one turns an amount into a non-number. One missing space costs one character but two word errors; losing the last three words costs nine characters, spaces included. CER says how much must be fixed; WER, or better a task check such as “is the total correct?”, says whether the output is usable (Figure 7).

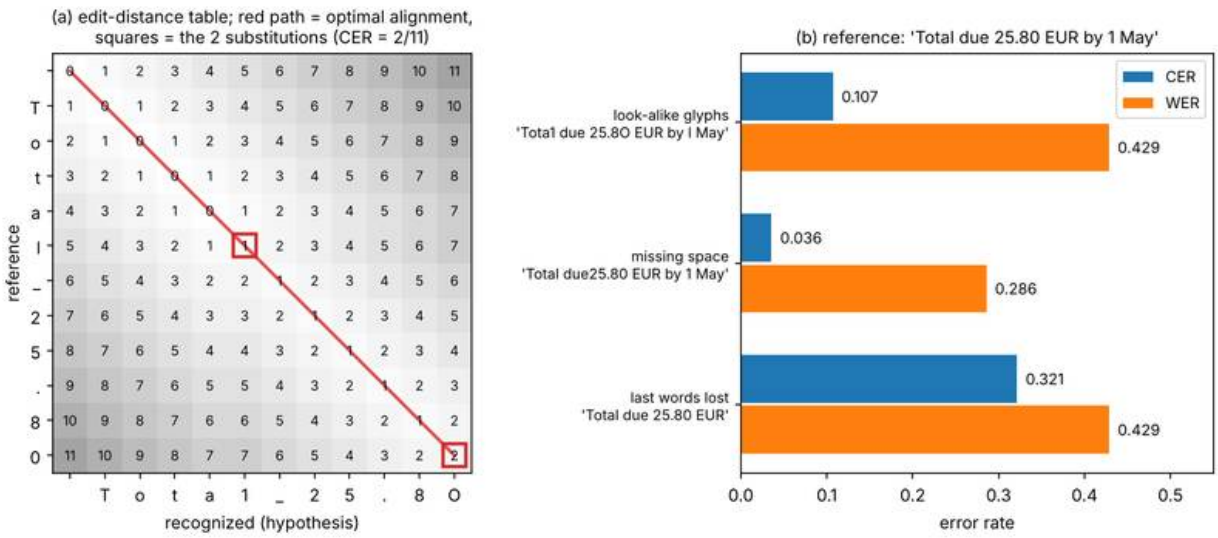



*Figure 7 — CER and WER. (a) The edit-distance table for a short pair; the red path is an optimal alignment with two substitutions, so CER = 2/11. (b) For the same reference, small character errors can produce large word error rates.*

### Detection: precision, recall and H-mean at IoU 0.5

Text detection uses the precision and recall of DLCV 03 with one matching rule and one summary number. In the protocol used by the Robust Reading competitions, as described for MLT-2019, a detection is a true positive if its IoU with a ground-truth quadrilateral exceeds 0.5, each ground truth and each detection may be matched at most once, and detections that overlap a region marked "do not care" (illegible or out-of-scope text) by more than half of that region's area are removed before matching [42]. Counts are accumulated over the whole test set, and

$$P = \frac{\mathrm{TP}}{|\mathcal{D}|}, \qquad R = \frac{\mathrm{TP}}{|\mathcal{G}|}, \qquad H = \frac{2PR}{P + R},$$

where TP is the number of matched pairs, $|\mathcal{D}|$ the number of detections after filtering, $|\mathcal{G}|$ the number of ground-truth words that are not "do not care", and $H$ the **H-mean** (harmonic mean of precision and recall, the F-score). Unlike COCO AP, H-mean is a single point of the precision–recall curve, at a confidence threshold chosen by the submitter; Long et al. recommend also reporting AP [1]. ICDAR 2013 used a different, overlap-based protocol, so its numbers are not comparable with IoU-based ones [1].

### End-to-end and document metrics

End-to-end spotting first matches detections to ground truth as above and then compares the transcriptions, either for every word (**end-to-end recognition**) or only for words in a given vocabulary

(**word spotting**); results depend strongly on the size of that vocabulary [2]. Document benchmarks use task metrics: **ANLS** (average normalized Levenshtein similarity), an edit-distance-based score for DocVQA answers, chosen so that small OCR slips are not scored as total failures [44]; field-level F1 and a tree-edit-distance accuracy for extracted JSON records [34]; and COCO-style mAP for layout regions [29].

*Online only: Evaluation protocols in detail — https://blog.shyandram.dev/en/learn/deep-learning/text-in-images/notes/evaluation-protocols-in-detail/*

## Datasets and benchmarks

The table lists the benchmarks named in this chapter. Sizes are as stated in the cited papers.

| Benchmark | Task | Content |
|---|---|---|
| ICDAR 2013 [5] | detection, recognition, end-to-end | focused scene text, mostly horizontal English; 462 images [10] |
| ICDAR 2015 Incidental [6] | detection, recognition, end-to-end | wearable-camera images; 1,000 train / 500 test for detection [12]; quadrilateral labels |
| Total-Text [10] | detection, spotting | 1,555 images (1,255 / 300), 9,330 words, polygons; horizontal, multi-oriented and curved |
| CTW1500 [11] | curved-text detection | 1,500 images (1,000 / 500), over 10,000 text annotations as polygons |
| IIIT5K [45] | cropped-word recognition | words from web image search; 2,000 train / 3,000 test [7] |
| SVT [46] | cropped-word recognition | Google Street View; 257 train / 647 test words [7] |
| MLT-2019 [42] | multilingual detection, script ID, end-to-end | 20,000 images, 10 languages |
| RCTW-17 [39], CTW [40] | Chinese detection and recognition | 12,263 images; 32,285 images with character-level labels |
| IAM [37], CASIA-HWDB [38] | handwriting | English sentences; Chinese online and offline handwriting |
| FUNSD [33] | form understanding | 199 scanned forms; question / answer / header / other entities with links |
| PubLayNet [28], DocLayNet [29] | layout analysis | over 360k auto-labeled article pages (5 classes); 80,863 hand-labeled pages (11 classes) |
| DocVQA [44] | document question answering | 50,000 questions on over 12,000 document images; ANLS |
| OCRBench [22], OmniDocBench [47] | OCR skills of multimodal models; page parsing | 29 datasets; 9 document sources, 19 layout categories |

Two cautions apply to all of them. Different papers have evaluated on different subsets of the same benchmark (Baek et al. found two versions each of IC03 and IC15 in use), so check the subset before comparing numbers [7]. And the labels are not perfect: Baek et al. estimated the mislabeling rate of the common recognition benchmarks at 1.3% ignoring special characters, rising to 6.1% when special characters count and 24.1% when case counts [7].

## Failure modes

**Plain version.** OCR fails where the pixels stop carrying the answer, where the shape or script differs from training, and where a model fills a gap with something plausible. The last kind is the most dangerous, because the output looks perfect.

**Look-alike glyphs and low resolution.** Many characters differ by a few pixels: 0 and O, 1, l and I, 5 and S, 8 and B, “rn” and “m”. The snippet makes a tiny template reader for 15 characters and shows how it degrades as the number of pixels per character drops.

```
import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from skimage.transform import resize

CHARS = "0O8B5S1lIZ2aekx"

def glyph(c):
    """Render one character (ink = 1, paper = 0) on a 48 x 48 canvas."""
    fig = plt.figure(figsize=(0.48, 0.48), dpi=100)
    fig.text(0.5, 0.5, c, fontsize=26, ha="center", va="center")
    fig.canvas.draw()
    img = 1 - np.asarray(fig.canvas.buffer_rgba())[..., :3].mean(axis=2) / 255.0
    plt.close(fig)
    return img

def ncc(a, b):
    a, b = a - a.mean(), b - b.mean()
    return (a * b).sum() / np.sqrt((a * a).sum() * (b * b).sum())

templates = {c: glyph(c) for c in CHARS}
rng = np.random.default_rng(0)
for h in [48, 12, 8, 6]:                          # pixels per character cell
    errors, total = {}, 0
    for c, img in templates.items():
        for _ in range(20):
            low = resize(img, (h, h), anti_aliasing=True)       # camera sees few pixels
            low = low + rng.normal(0, 0.05, low.shape)          # sensor noise
            back = resize(low, (48, 48), order=1)               # what the reader gets
            guess = max(CHARS, key=lambda t: ncc(back, templates[t]))
            total += 1
            if guess != c:
                errors[(c, guess)] = errors.get((c, guess), 0) + 1
    top = sorted(errors.items(), key=lambda kv: -kv[1])[:3]
    print(f"{h:2d}px  accuracy={1 - sum(errors.values()) / total:.2f}  most confused: "
          + ", ".join(f"{a}->{b} ({n})" for (a, b), n in top))
```

```
48px  accuracy=1.00  most confused:
12px  accuracy=0.93  most confused: l->I (20)
 8px  accuracy=0.84  most confused: I->1 (13), l->1 (12), 0->O (6)
 6px  accuracy=0.53  most confused: B->8 (20), 0->8 (15), S->8 (12)
```

Lowercase "l" and capital "I" are confused already at 12 pixels, because this sans-serif font draws both as a plain bar (Figure 8); at 6 pixels half the readings are wrong. A deep recognizer does far better than template matching, but no recognizer can separate glyphs that the font draws identically; only context can. Super-resolution can help (DLCV 09A), but see the last item below.

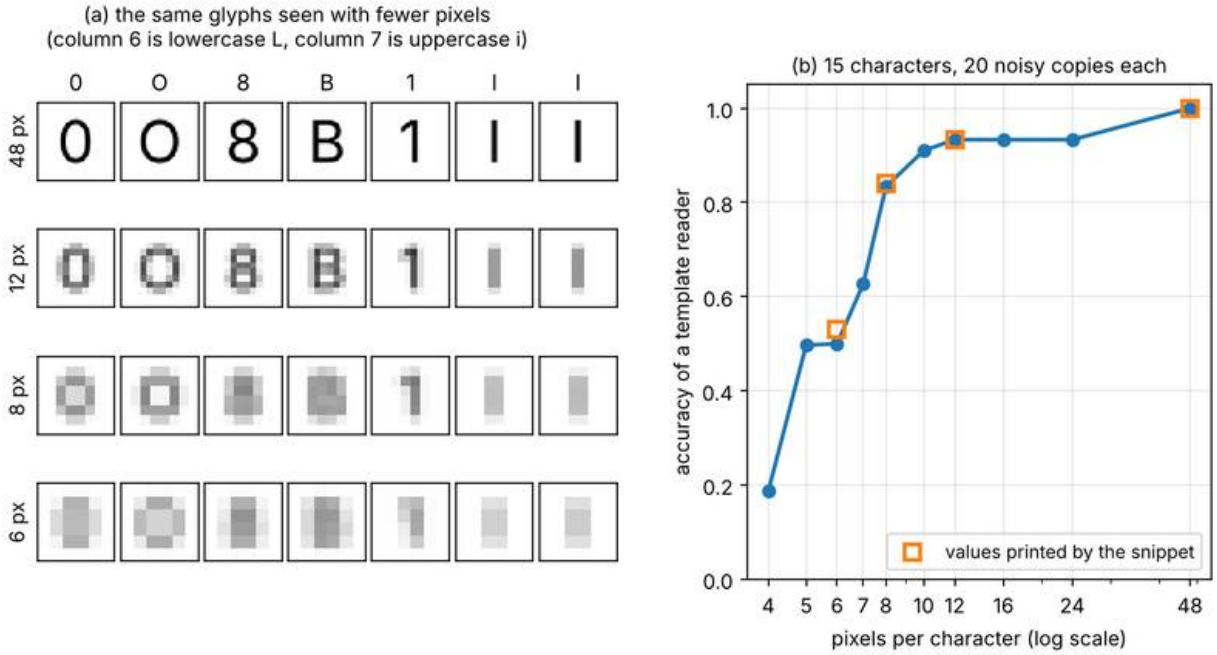



*Figure 8 — Look-alike glyphs at low resolution (toy template reader, 15 characters, 20 noisy copies each). (a) The same glyphs with fewer pixels. (b) Accuracy against resolution; squares mark the values printed by the snippet.*

**Other common failures**, each with the stage where it starts (Figure 1):

- **Curved, rotated and perspective text** (detection and recognition): upright boxes miss it (Figure 3); horizontal recognizers misread it unless it is rectified.
- **Unusual fonts and styles** (recognition): calligraphic brand names and shop signs were among the cases that all 24 models in Baek et al.'s study failed on, along with vertical text, special characters, heavy occlusion and low resolution [7].
- **Merged or split instances** (detection): Figure 2's gap trade-off in learned form; segmentation detectors fight it with shrunk kernels and learned thresholds.
- **Wrong layout** (understanding): correct words in the wrong order (Figure 6), tables read across rows, a value linked to the wrong key.
- **Distribution shift** (all stages): layout models trained only on article pages degrade on more diverse documents [29]; DLCV 14 treats shift in general.
- **Hallucinated text** (any generative stage): a generative reader can output text that is not in the image — repeated sentences, invented section numbers [36], a "corrected" serial number. The same happens upstream: a generative restoration or super-resolution model can turn unreadable letters into clean, legible, wrong ones, as discussed in DLCV 09B; and text in generated

images is often misspelled (DLCV 12). Never run OCR on a generatively enhanced image for anything that must be true, such as an amount, a name or a license plate, without checking against the original.

## Modern view

### What the surveys say

Four reviews cover this chapter's ground. **Long et al.** [1] organize scene-text detection by how much of a word each prediction covers (whole words versus characters, segments or kernels) and recognition into CTC-based and encoder–decoder methods; they see generalization across datasets and large alphabets as the open problems. **Chen et al.** [2] focus on recognition, organized by pipeline stage, and collect the datasets and evaluation protocols. **Cui et al.** [3] review document tasks from rule-based analysis to pre-trained LayoutLM-style models, and **Zhang et al.** [4] split document parsing into modular pipelines and unified VLM-based models, naming complex layouts, the reliability of VLM parsing and inference cost as the main challenges. OCRBench [22] complements them with an evaluation of large multimodal models on text-related tasks.

*Online only: Guide to the Modern view surveys — https://blog.shyandram.dev/en/learn/deep-learning/text-in-images/papers/guide-to-the-modern-view-surveys/*

### The state of the art, 2023–2026

- **Pipelines remain the natural choice** when coordinates, speed or auditability matter: a DBNet-style detector, an attention or transformer recognizer and a layout model, each evaluated and corrected separately.
- **End-to-end page readers are maturing fast.** Models that turn a page into Markdown or structured text in one pass now cover formulas, tables and charts: Nougat for academic papers [36]; GOT, which treats plain text, formulas, tables, charts and sheet music as "characters" for one encoder–decoder model [48]; olmOCR, a fine-tuned vision-language model for linearizing PDFs at scale, released with its own benchmark of hard pages [49]; and DeepSeek-OCR, which studies how few vision tokens a page can be compressed into while remaining readable [50]. All of these reports come from their authors.
- **Evaluation is catching up.** OmniDocBench [47] (CVPR 2025) evaluates pipelines and end-to-end models on pages from nine document sources with 19 layout categories and 15 attribute labels, so that results can be broken down by document type and difficulty instead of one score.

### Open problems

1. **Faithfulness.** Generative readers can produce text that is not on the page, and their errors look like correct text. Character-level grounding (which pixels support each output token) and calibrated confidence are not yet standard.

2. **Non-semantic and rare text.** Codes, numbers, names, rare characters in large alphabets and mixed scripts are where language priors help least and errors cost most [22].
3. **Layout and reading order** on complex pages — multi-column articles with floating figures, nested tables, forms with handwritten entries — remain hard for pipelines and end-to-end models alike; robustness to complex layouts is among the main challenges named by Zhang et al. [4].
4. **Generalization across domains and scripts**, including vertical and curved text, without collecting new labels for each domain [1].
5. **Comparable evaluation.** Different subsets, metrics (CER, word accuracy, NED, ANLS) and output normalizations make published numbers hard to compare [7].

## Key takeaways

- Evaluate each reading stage and the complete pipeline; upstream errors propagate.
- Rotated, curved and crowded text needs geometry that ordinary upright boxes may not capture.
- Shrunk text regions help separate touching instances; thresholding and region expansion affect the final detections.
- Recognition is sequence prediction; decoding and language priors can help words but distort codes or numbers.
- Layout and OCR-free models change the error pathway, while resolution limits and hallucinated text remain concerns.
- State CER/WER variants, detection matching rules, ignored regions and downstream task metrics.

## Exercises

1. **Gap and font size.** In the detection snippet, scale every font size in `WORDS` by 0.6 and then by 1.6, and rerun the sweep over the dilation gap. How does the best gap move, and what does that tell you about one fixed post-processing rule for an image that contains both sizes?

   **Hint**

   The best gap scales with letter spacing, hence with font size, so an image with both sizes has no single good gap. Learned detectors avoid the problem by predicting word regions, or shrunk kernels, directly.

2. **Upright boxes by hand.** Show that the IoU between a $w \times h$ rectangle rotated by $\theta$ and its tightest upright box is $wh/[(w\cos\theta + h\sin\theta)(w\sin\theta + h\cos\theta)]$. For $w = 160$ and $h = 24$, find the angle at which it drops to 0.5, and compare with a square word ($w = h$).

   **Hint**

   The rotated rectangle lies inside its upright box, so the intersection is $wh$ and the union is the box area. Solving numerically for $w/h = 160/24$ gives about 8.5°. For a square, the minimum at 45° is $1/2$ exactly, so a square never falls below 0.5: the problem is specific to elongated shapes such as text lines.

3. **Greedy versus exact CTC.** Change the last two rows of the CTC matrix so that the blank and "t" have probabilities 0.65 and 0.30 in both frames. Is "cat" still the most probable text? Then let the "t" probability $p_t$ in frames 4 and 5 run from 0.05 to 0.45, with the blank taking $0.95 - p_t$. For which values do greedy and exact decoding agree?

   **Hint**

   Reuse the brute-force loop. At $p_t = 0.30$, "cat" (about 0.294) still beats "ca" (about 0.249). Greedy decoding returns "ca" until $p_t$ exceeds the blank (above 0.475), but the exact answer switches to "cat" already at $p_t \approx 0.27$; between those values the two disagree. Below the crossover they agree on "ca".

4. **CER that hides a disaster.** Construct a reference of 40 characters and two recognitions with the same CER of 0.05 such that one is harmless for an invoice-processing system and the other is fatal. Propose a field-level metric that separates them.

   **Hint**

   Two errors in a long description word ("Paper A4 recyled") versus two errors in the total ("42.00" read as "4Z.0O"). A field-level exact match on the key fields (invoice number, date, total), or the F1 over extracted fields that Donut reports, distinguishes them; CER and WER weight every character or word equally.

5. **Reading order breaks again.** Add a heading "PAYMENT TERMS" that spans both columns above the two-column notice of the reading-order snippet. What does `column_order` produce? Modify it so that full-width lines are read first, and describe a layout that still defeats your rule.

   **Hint**

   The heading crosses the split and is assigned to the left column by its left edge, which happens to work at the top but not for a full-width line in the middle. Treat boxes that overlap the split as a separate band and read band by band. A table, or a figure inside one column, still breaks rules based on empty strips.

## References

1. S. Long, X. He and C. Yao, "Scene text detection and recognition: The deep learning era," *International Journal of Computer Vision*, vol. 129, pp. 161–184, 2021. doi
2. X. Chen, L. Jin, Y. Zhu, C. Luo and T. Wang, "Text recognition in the wild: A survey," *ACM Computing Surveys*, vol. 54, no. 2, 2021. doi
3. L. Cui, Y. Xu, T. Lv and F. Wei, "Document AI: Benchmarks, models and applications," arXiv: 2111.08609, 2021. arXiv
4. Q. Zhang, B. Wang, V. S.-J. Huang, J. Zhang, Z. Wang, H. Liang, C. He and W. Zhang, "Document parsing unveiled: Techniques, challenges, and prospects for structured information extraction," arXiv:2410.21169, 2024. arXiv

5. D. Karatzas, F. Shafait, S. Uchida, M. Iwamura, L. Gomez i Bigorda, S. Robles Mestre, J. Mas et al., "ICDAR 2013 Robust Reading Competition," in *Proc. International Conference on Document Analysis and Recognition (ICDAR)*, pp. 1484–1493, 2013. doi
6. D. Karatzas, L. Gomez-Bigorda, A. Nicolaou, S. Ghosh, A. Bagdanov, M. Iwamura, J. Matas et al., "ICDAR 2015 competition on Robust Reading," in *Proc. International Conference on Document Analysis and Recognition (ICDAR)*, pp. 1156–1160, 2015. doi
7. J. Baek, G. Kim, J. Lee, S. Park, D. Han, S. Yun, S. J. Oh and H. Lee, "What is wrong with scene text recognition model comparisons? Dataset and model analysis," in *Proc. IEEE/CVF ICCV*, pp. 4714–4722, 2019. doi
8. N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, pp. 62–66, 1979. doi
9. J. Sauvola and M. Pietikäinen, "Adaptive document image binarization," *Pattern Recognition*, vol. 33, no. 2, pp. 225–236, 2000. doi
10. C. K. Ch'ng and C. S. Chan, "Total-Text: A comprehensive dataset for scene text detection and recognition," in *Proc. ICDAR*, pp. 935–942, 2017. doi
11. Y. Liu, L. Jin, S. Zhang, C. Luo and S. Zhang, "Curved scene text detection via transverse and longitudinal sequence connection," *Pattern Recognition*, vol. 90, pp. 337–345, 2019. doi
12. X. Zhou, C. Yao, H. Wen, Y. Wang, S. Zhou, W. He and J. Liang, "EAST: An efficient and accurate scene text detector," in *Proc. IEEE CVPR*, pp. 2642–2651, 2017. doi
13. W. Wang, E. Xie, X. Li, W. Hou, T. Lu, G. Yu and S. Shao, "Shape robust text detection with progressive scale expansion network," in *Proc. IEEE/CVF CVPR*, pp. 9328–9337, 2019. doi
14. M. Liao, Z. Wan, C. Yao, K. Chen and X. Bai, "Real-time scene text detection with differentiable binarization," in *Proc. AAAI Conference on Artificial Intelligence*, vol. 34, pp. 11474–11481, 2020. doi
15. Y. Baek, B. Lee, D. Han, S. Yun and H. Lee, "Character region awareness for text detection," in *Proc. IEEE/CVF CVPR*, pp. 9357–9366, 2019. doi
16. B. Shi, X. Bai and C. Yao, "An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition," *IEEE TPAMI*, vol. 39, no. 11, pp. 2298–2304, 2017. doi
17. A. Graves, S. Fernández, F. Gomez and J. Schmidhuber, "Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks," in *Proc. ICML*, pp. 369–376, 2006. doi
18. B. Shi, M. Yang, X. Wang, P. Lyu, C. Yao and X. Bai, "ASTER: An attentional scene text recognizer with flexible rectification," *IEEE TPAMI*, vol. 41, no. 9, pp. 2035–2048, 2019. doi
19. M. Li, T. Lv, J. Chen, L. Cui, Y. Lu, D. Florencio, C. Zhang, Z. Li and F. Wei, "TrOCR: Transformer-based optical character recognition with pre-trained models," in *Proc. AAAI Conference on Artificial Intelligence*, vol. 37, pp. 13094–13102, 2023. doi
20. D. Bautista and R. Atienza, "Scene text recognition with permuted autoregressive sequence models," in *Proc. ECCV*, pp. 178–196, 2022. doi
21. S. Fang, H. Xie, Y. Wang, Z. Mao and Y. Zhang, "Read like humans: Autonomous, bidirectional and iterative language modeling for scene text recognition," in *Proc. IEEE/CVF CVPR*, pp. 7094–7103, 2021. doi
22. Y. Liu, Z. Li, M. Huang, B. Yang, W. Yu, C. Li, X.-C. Yin, C.-L. Liu, L. Jin and X. Bai, "OCRBench: On the hidden mystery of OCR in large multimodal models," *Science China Information Sciences*,

vol. 67, 2024. doi

23. M. Jaderberg, K. Simonyan, A. Vedaldi and A. Zisserman, “Synthetic data and artificial neural networks for natural scene text recognition,” arXiv:1406.2227, 2014. arXiv
24. A. Gupta, A. Vedaldi and A. Zisserman, “Synthetic data for text localisation in natural images,” in *Proc. IEEE CVPR*, pp. 2315–2324, 2016. doi
25. X. Liu, D. Liang, S. Yan, D. Chen, Y. Qiao and J. Yan, “FOTS: Fast oriented text spotting with a unified network,” in *Proc. IEEE/CVF CVPR*, pp. 5676–5685, 2018. doi
26. P. Lyu, M. Liao, C. Yao, W. Wu and X. Bai, “Mask TextSpotter: An end-to-end trainable neural network for spotting text with arbitrary shapes,” in *Proc. ECCV*, pp. 71–88, 2018. doi
27. Y. Liu, H. Chen, C. Shen, T. He, L. Jin and L. Wang, “ABCNet: Real-time scene text spotting with adaptive Bezier-curve network,” in *Proc. IEEE/CVF CVPR*, pp. 9806–9815, 2020. doi
28. X. Zhong, J. Tang and A. Jimeno Yepes, “PubLayNet: Largest dataset ever for document layout analysis,” in *Proc. ICDAR*, pp. 1015–1022, 2019. doi
29. B. Pfitzmann, C. Auer, M. Dolfi, A. S. Nassar and P. Staar, “DocLayNet: A large human-annotated dataset for document-layout segmentation,” in *Proc. ACM SIGKDD*, pp. 3743–3751, 2022. doi
30. Y. Xu, M. Li, L. Cui, S. Huang, F. Wei and M. Zhou, “LayoutLM: Pre-training of text and layout for document image understanding,” in *Proc. ACM SIGKDD*, pp. 1192–1200, 2020. doi
31. R. Smith, “An overview of the Tesseract OCR engine,” in *Proc. ICDAR*, vol. 2, pp. 629–633, 2007. doi
32. Y. Huang, T. Lv, L. Cui, Y. Lu and F. Wei, “LayoutLMv3: Pre-training for document AI with unified text and image masking,” in *Proc. ACM International Conference on Multimedia*, pp. 4083–4091, 2022. doi
33. G. Jaume, H. K. Ekenel and J.-P. Thiran, “FUNSD: A dataset for form understanding in noisy scanned documents,” in *Proc. ICDAR Workshops (OST)*, 2019. doi
34. G. Kim, T. Hong, M. Yim, J. Nam, J. Park, J. Yim, W. Hwang, S. Yun, D. Han and S. Park, “OCR-free document understanding transformer,” in *Proc. ECCV*, pp. 498–517, 2022. doi
35. K. Lee, M. Joshi, I. R. Turc, H. Hu, F. Liu, J. M. Eisenschlos, U. Khandelwal, P. Shaw, M.-W. Chang and K. Toutanova, “Pix2Struct: Screenshot parsing as pretraining for visual language understanding,” in *Proc. ICML*, PMLR vol. 202, pp. 18893–18912, 2023. PMLR
36. L. Blecher, G. Cucurull, T. Scialom and R. Stojnic, “Nougat: Neural optical understanding for academic documents,” in *Proc. ICLR*, 2024. OpenReview ·arXiv
37. U.-V. Marti and H. Bunke, “The IAM-database: An English sentence database for offline handwriting recognition,” *International Journal on Document Analysis and Recognition*, vol. 5, pp. 39–46, 2002. doi
38. C.-L. Liu, F. Yin, D.-H. Wang and Q.-F. Wang, “CASIA online and offline Chinese handwriting databases,” in *Proc. ICDAR*, pp. 37–41, 2011. doi
39. B. Shi, C. Yao, M. Liao, M. Yang, P. Xu, L. Cui, S. Belongie, S. Lu and X. Bai, “ICDAR2017 competition on reading Chinese text in the wild (RCTW-17),” in *Proc. ICDAR*, pp. 1429–1434, 2017. doi ·arXiv
40. T.-L. Yuan, Z. Zhu, K. Xu, C.-J. Li, T.-J. Mu and S.-M. Hu, “A large Chinese text dataset in the wild,” *Journal of Computer Science and Technology*, vol. 34, pp. 509–521, 2019. doi ·arXiv
41. F. Xue and R. Ishida (eds.), “Requirements for Chinese Text Layout,” W3C Group Note (draft of 7 October 2026). W3C

42. N. Nayef, Y. Patel, M. Busta, P. N. Chowdhury, D. Karatzas, W. Khlif, J. Matas et al., “ICDAR2019 Robust Reading Challenge on multi-lingual scene text detection and recognition — RRC-MLT-2019,” in *Proc. ICDAR*, pp. 1582–1587, 2019. doi ·arXiv
43. R. A. Wagner and M. J. Fischer, “The string-to-string correction problem,” *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974. doi
44. M. Mathew, D. Karatzas and C. V. Jawahar, “DocVQA: A dataset for VQA on document images,” in *Proc. IEEE/CVF WACV*, pp. 2199–2208, 2021. doi
45. A. Mishra, K. Alahari and C. V. Jawahar, “Scene text recognition using higher order language priors,” in *Proc. BMVC*, 2012. doi
46. K. Wang, B. Babenko and S. Belongie, “End-to-end scene text recognition,” in *Proc. IEEE ICCV*, pp. 1457–1464, 2011. doi
47. L. Ouyang, Y. Qu, H. Zhou, J. Zhu, R. Zhang, Q. Lin, B. Wang, Z. Zhao et al., “OmniDocBench: Benchmarking diverse PDF document parsing with comprehensive annotations,” in *Proc. IEEE/CVF CVPR*, pp. 24838–24848, 2025. CVF
48. H. Wei, C. Liu, J. Chen, J. Wang, L. Kong, Y. Xu, Z. Ge, L. Zhao et al., “General OCR theory: Towards OCR-2.0 via a unified end-to-end model,” arXiv:2409.01704, 2024. arXiv
49. J. Poznanski, A. Rangapur, J. Borchardt, J. Dunkelberger, R. Huff, D. Lin, C. Wilhelm et al., “olmOCR: Unlocking trillions of tokens in PDFs with vision language models,” arXiv:2502.18443, 2025. arXiv
50. H. Wei, Y. Sun and Y. Li, “DeepSeek-OCR: Contexts optical compression,” arXiv:2510.18234, 2025. arXiv
51. C. Wolf and J.-M. Jolion, “Object count/area graphs for the evaluation of object detection and segmentation algorithms,” *International Journal of Document Analysis and Recognition*, vol. 8, pp. 280–296, 2006. doi

# DLCV 07 • Object Tracking: From Single-Object to Multi-Object

How computer vision follows objects through video: re-identification, single-object trackers from correlation filters to transformers and SAM 2, and multi-object tracking with Kalman filters, Hungarian matching, end-to-end models and the MOTA/IDF1/HOTA metrics.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/object-tracking/

## What you'll learn

- Separate motion estimation from identity association.
- Distinguish single-object, multi-object, multi-camera tracking and re-identification.
- Compare template matching, tracking-by-detection and learned query approaches.
- Choose metrics that expose detection and identity errors separately.
- Run a small tracker and diagnose missed detections, fragmentation and identity switches.

## The big picture

A detector looks at one picture and says "there are three people here". A tracker watches a video and says "this is the *same* person as one second ago, and that one just walked behind a pillar and will come out again". Tracking adds time and memory to detection.

> **Explain like I'm five**
>
> Imagine you are watching a soccer game with your friend, and you each pick a favorite player. Finding every player on the field in one photo is **detection**. Keeping your eyes on *your* player as she runs, turns and gets hidden behind others for a moment is **single-object tracking**. Putting a numbered sticker on *every* player and making sure the stickers never get swapped, even when two players crash into each other, is **multi-object tracking**. And recognizing your player again when the TV switches to a different camera is **re-identification**.

Why it matters: tracking is how a self-driving car knows that a pedestrian is *walking toward* the road rather than just standing near it, how sports analytics measures how far each player ran, how a warehouse counts forklifts and robots, and how a video editor keeps a mask glued to a moving object. Each of these needs not just *where* things are, but *which one is which* over time.

> **About this tutorial**
>
> This tutorial is expanded from an earlier Medium post (written in Traditional Chinese), "物件追蹤 Object Tracking 簡介" [1], keeping its structure and adding the technical depth, a runnable tracker and an up-to-date survey. You can read the original post here.

**Related chapters.** DLCV 05 estimates human pose and keypoints, which trackers often use as appearance-free cues, and DLCV 08 recognizes what people are doing over time (action recognition).

## From classification to detection to tracking

**Plain version.** Classification answers *what* is in an image. Detection answers *what and where*. Tracking answers *what, where, and which one, over time*.

The progression in [1] is a useful map of computer vision tasks:

1. **Image classification** maps an image $I$ to a label $y$.
2. **Object detection** maps an image to a set of boxes with labels and confidence scores, $\mathcal{D} = \{(b_j, c_j, s_j)\}_{j=1}^{N}$, where $b_j$ is a bounding box, $c_j$ a class and $s_j \in [0, 1]$ a score. Detection imitates the human ability to locate targets in a scene; it is covered in depth in DLCV 03 ·Object Detection.
3. **Object tracking** takes a sequence of frames $I_1, \dots, I_T$ and outputs **trajectories**: for each object $k$, a sequence of boxes (or masks) $(b_t^k)_{t \in \mathcal{T}_k}$ over the set of frames $\mathcal{T}_k$ in which it is visible, all carrying the same identity $k$.

The key new ingredient is the identity label. Detection is stateless: frame $t$ and frame $t + 1$ are processed independently. Tracking is stateful: it carries information forward.

## What tracking does

**Plain version.** A tracker predicts motion and maintains identity through appearance changes and temporary occlusion.

More precisely, every tracker solves two coupled sub-problems:

- **Localization over time** (motion perception): estimate where each target is in each frame, ideally using the past to predict the future. This is what lets a tracker survive blur, missed detections and short occlusions.
- **Identity maintenance** (data association): decide which observation in frame $t$ belongs to which target from frame $t-1$. This is what prevents two people who cross paths from swapping labels.

Single-object tracking puts most of its effort into the first problem, since there is only one identity. Multi-object tracking spends most of its effort on the second. Re-identification is the second problem with *no* motion information at all.

The difficulties that [1] lists for each family come from these two jobs: appearance changes (pose, lighting, scale, viewpoint), occlusion by other objects or the scene, similar-looking distractors, fast or non-linear motion, camera motion, and targets entering and leaving the scene.

## A taxonomy of tracking

**Plain version.** You can sort trackers by *what* they follow (one object or many, one camera or many) and by *how* they keep identity (by appearance alone, or by following motion frame to frame).

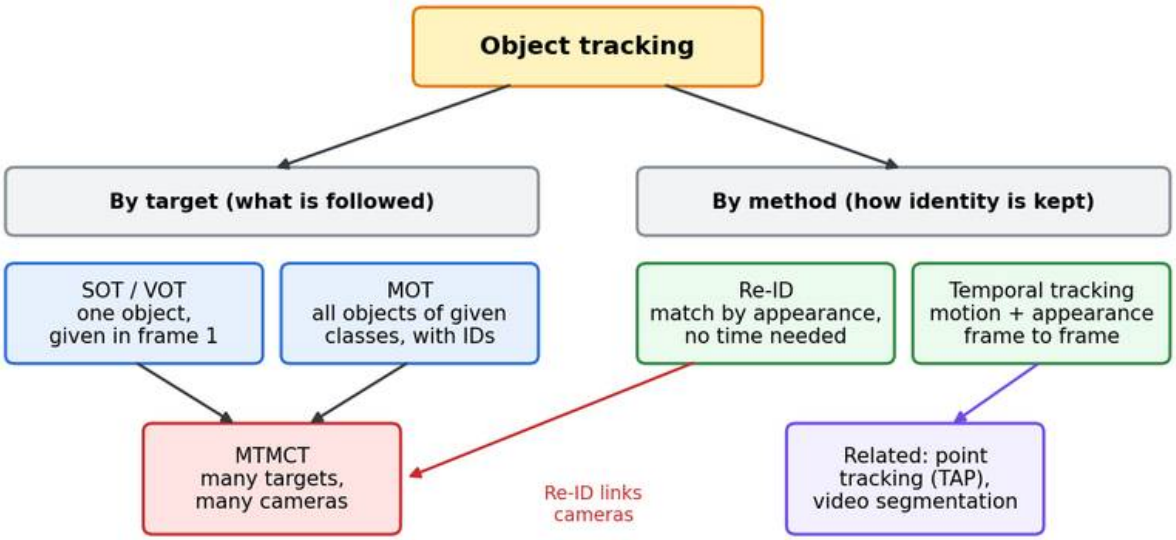



*Figure 1 — A taxonomy of tracking. By target: single-object (SOT, also called VOT) and multi-object (MOT). By method: appearance-only re-identification and temporal tracking. The hardest combined setting is multi-target multi-camera tracking (MTMCT).*

Following [1]:

- **By method.** *Re-identification* (*Re-ID*) matches a target across images that may be far apart in time or come from different cameras, using appearance (and sometimes pose) alone. *Temporal object tracking* follows targets frame by frame in one video stream, where motion continuity is available.
- **By target.** *Visual object tracking* (*VOT*), also called *single-object tracking* (*SOT*), follows one arbitrary object specified by a box in the first frame. *Multi-object tracking* (*MOT*) follows all objects of given classes (for example, every pedestrian) through consecutive frames and maintains a consistent ID for each.
- **The combined goal.** *Multi-target multi-camera tracking* (*MTMCT*) tracks many targets across a network of cameras and keeps the same identity when a person leaves one camera's view and appears in another. It combines MOT within each camera with Re-ID between cameras.

Most of the classical literature, and most benchmarks, use **pedestrians** as targets [1], because surveillance and autonomous driving made people the most valuable class to track. Modern benchmarks add dancers, athletes, vehicles and open-vocabulary categories.

## Person re-identification

**Plain version.** Re-ID is a search problem: given a photo of a person from one camera, find the same person in photos taken by other cameras, ranked from most to least likely.

### Problem formulation

Re-ID is posed as **retrieval** [7]. A *query* image $q$ and a *gallery* $\mathcal{G} = \{g_1, \dots, g_M\}$ of detected person crops from other cameras are mapped by a network $f_\theta$ to embeddings. The gallery is ranked by a distance, usually cosine distance

$$d(q, g) = 1 - \frac{f_\theta(q)^\top f_\theta(g)}{\|f_\theta(q)\| \, \|f_\theta(g)\|},$$

where $f_\theta(\cdot) \in \mathbb{R}^D$ is the $D$-dimensional embedding produced by the network with parameters $\theta$. Training makes embeddings of the same identity close and of different identities far apart. A typical recipe combines an identity classification loss (cross-entropy over the $C$ training identities) with a **triplet loss**

$$\mathcal{L}_{\text{tri}} = \left[\, d(a, p) - d(a, n) + m \,\right]_+,$$

where $a$ is an anchor image, $p$ a positive (same identity), $n$ a negative (different identity), $m > 0$ a margin and $[x]_+ = \max(x, 0)$. Test identities never appear in training, so Re-ID is an open-set problem: the network must learn a general notion of "same person".

### Challenges

[1] groups the difficulties into **data factors** and **environment factors**. Data factors: limited labeled identities, noisy boxes from the detector, partial bodies, low resolution. Environment factors: different viewpoints and camera color responses, lighting, occlusion, background clutter, and people who change clothes. Re-ID also raises clear **privacy** concerns, which is part of why surveillance-style datasets have been withdrawn (below). The original post names city-scale surveillance systems as the main application; today Re-ID is just as common as the appearance module inside MOT trackers and in retail and sports analytics.

### Representative models

The three models below all keep the retrieval setup above and change how the embedding network $f_\theta$ is trained. A plain ResNet-50 with a set of training "tricks" is a strong, reproducible baseline [14]; the centroid triplet loss compares images with identity *centroids* rather than with single images [15]; and CLIP-ReID adapts the vision-language model CLIP to Re-ID, where no text labels exist [16]. The point to remember: careful training of a standard backbone already goes a long way, so new Re-ID ideas should be compared against such a baseline.

*Online only: More Re-ID models: a strong baseline, centroids and CLIP-ReID — https://blog.shyandram.dev/en/learn/deep-learning/object-tracking/notes/more-re-id-models-a-strong-baseline-centroids-and-clip-reid/*

### Datasets

| Dataset | Content | Status |
|---|---|---|
| Market-1501 [12] | 1,501 identities, 6 cameras, detector-cropped boxes | available |
| MSMT17 [13] | 4,101 identities, 15 indoor and outdoor cameras | available |
| DukeMTMC [10] (and DukeMTMC-reID) | 8 synchronized campus cameras | **withdrawn in 2019** |

DukeMTMC was introduced with the IDF1 metric for multi-camera tracking [10], and a Re-ID subset was widely used. Its creators withdrew it in 2019 after privacy and ethics concerns about recording people on a campus, and it should no longer be downloaded or used; Peng et al. [11] trace how derived copies kept circulating afterwards and argue for better dataset stewardship.

### Metrics: CMC rank-$k$ and mAP

The **cumulative matching characteristic** at rank $k$ is the fraction of queries whose first correct match appears in the top $k$ of the ranked gallery:

$$\mathrm{CMC}(k) = \frac{1}{|Q|} \sum_{q \in Q} \mathbb{1}[\, r_q \le k \,],$$

where $Q$ is the set of queries and $r_q$ is the rank of the first true match for query $q$. *Rank-1 accuracy* is $\mathrm{CMC}(1)$. Because a query usually has several true matches in the gallery, rank-$k$ alone ignores how well the *others* are ranked. **Mean average precision** fixes this:

$$\mathrm{AP}(q) = \frac{1}{G_q} \sum_{i=1}^{G_q} \frac{i}{r_{q,i}}, \qquad \mathrm{mAP} = \frac{1}{|Q|} \sum_{q} \mathrm{AP}(q),$$

where $G_q$ is the number of true matches of $q$ in the gallery and $r_{q,i}$ is the rank of the $i$-th one. So $i / r_{q,i}$ is the precision at the moment the $i$-th true match is found.

```
import numpy as np

def cmc_map(dist, q_ids, g_ids, max_rank=5):
    """dist: (Q, G) distances between query and gallery embeddings."""
    order = np.argsort(dist, axis=1)                # closest gallery items first
    match = g_ids[order] == q_ids[:, None]          # (Q, G) True where same person
    cmc = (np.cumsum(match, axis=1) > 0).mean(axis=0)[:max_rank]
```

```
    aps = []
    for row in match:
        hits = np.flatnonzero(row)                  # ranks (0-based) of true matches
        precision_at_hits = np.arange(1, len(hits) + 1) / (hits + 1)
        aps.append(precision_at_hits.mean())
    return cmc, np.mean(aps)

rng = np.random.default_rng(0)
centres = rng.normal(size=(50, 64))                 # 50 identities, 64-D embeddings
g_ids = np.repeat(np.arange(50), 4)                 # 4 gallery images per identity
q_ids = np.arange(50)
gallery = centres[g_ids] + 1.6 * rng.normal(size=(200, 64))
query = centres[q_ids] + 1.6 * rng.normal(size=(50, 64))
norm = lambda x: x / np.linalg.norm(x, axis=1, keepdims=True)
dist = 1 - norm(query) @ norm(gallery).T            # cosine distance
cmc, mAP = cmc_map(dist, q_ids, g_ids)
print("rank-1 %.2f  rank-5 %.2f  mAP %.2f" % (cmc[0], cmc[4], mAP))
```

On this synthetic gallery (4 images per identity) the script prints `rank-1 0.76  rank-5 0.92  mAP 0.57`: most queries find *one* correct image near the top, but the other three are scattered, which only mAP reveals.

## Single-object tracking

**Plain version.** Given an initial target box, single-object tracking must locate that same target in later frames.

### Problem formulation

Given frames $I_1, \dots, I_T$ and an initial box $b_1$, a single-object tracker outputs $\hat{b}_2, \dots, \hat{b}_T$. The target is **class-agnostic**: it could be a person, a bird or a coffee cup. This makes SOT a form of *one-shot* learning: the only example of the target is the first-frame patch, the **template** $z$. In each new frame the tracker searches a region $x$ around the last position. The challenges [1] lists are complex appearance changes, unusual motion patterns, and environmental change such as lighting and occlusion.

### Classical: correlation filters

A correlation filter is a template slid over the search window; the place where it lines up best gives the highest score. Before deep learning, the fastest trackers learned such a **correlation filter** online. MOSSE [17] learns a filter $H$ in the Fourier domain so that correlating it with training patches $F_i$ produces sharp Gaussian peaks $G_i$ at the target center:

$$\min_{H^*} \sum_i |F_i \odot H^* - G_i|^2 \quad \Rightarrow \quad H^* = \frac{\sum_i G_i \odot F_i^*}{\sum_i F_i \odot F_i^*},$$

where capital letters are 2-D discrete Fourier transforms, $\odot$ is element-wise multiplication, $^*$ is complex conjugation and the division is element-wise. Because correlation becomes multiplication in the frequency domain (see IP 04), training and detection cost only a few FFTs, so these trackers run at hundreds of frames per second on a CPU. KCF [18] showed that training on *all cyclic shifts* of a patch gives a circulant data matrix that the DFT diagonalizes, which turns kernel ridge regression into element-wise operations and allows multi-channel features such as HOG. Their weaknesses are a fixed search window, boundary effects from the cyclic assumption, and hand-crafted features.

*Online only: MOSSE: a correlation filter learned in the Fourier domain — https://blog.shyandram.dev/en/learn/deep-learning/object-tracking/notes/mosse-a-correlation-filter-learned-in-the-fourier-domain/*

### Siamese trackers

**Plain version.** Train one network to answer a single question — "where in this search image does the template appear?" — on huge amounts of video offline, then use it frozen at test time.

SiamFC [19] applies the *same* fully convolutional network $\varphi$ to the template $z$ and the search region $x$ and cross-correlates the two feature maps:

$$f(z, x) = \varphi(z) \star \varphi(x) + b\,\mathbb{1},$$

where $\star$ is cross-correlation (the template features act as a convolution kernel), $b$ is a learned bias and $\mathbb{1}$ is a map of ones. The output is a score map whose peak gives the new position. It is trained with a logistic loss on score-map positions labeled positive near the true center and negative elsewhere. Because nothing is updated online, it is fast and robust to drift, but it only predicts position, with scale handled by searching a small pyramid.

SiamRPN [20] adds a **region proposal network** after the Siamese features: one branch classifies anchors as target or background, another regresses box offsets. Tracking becomes "one-shot detection" with the template as the class definition, which yields accurate boxes with changing aspect ratios.

### Transformer one-stream trackers

From 2022 the mainstream moved from Siamese CNNs to transformers. The key change is *where* the template and the search region interact (Figure 2). In a two-stream design they are encoded separately and compared once. In a **one-stream** design such as OSTrack [21], the image patches of $z$ and $x$ are concatenated into one token sequence and processed by a single Vision Transformer, so that every attention layer both extracts features and relates template to search. OSTrack also drops search tokens that are clearly background in early layers to save computation. SeqTrack [22] goes further in simplicity: an encoder-decoder transformer generates the four box coordinates as a sequence of discrete tokens, autoregressively, trained with plain cross-entropy — the same recipe as language modeling.

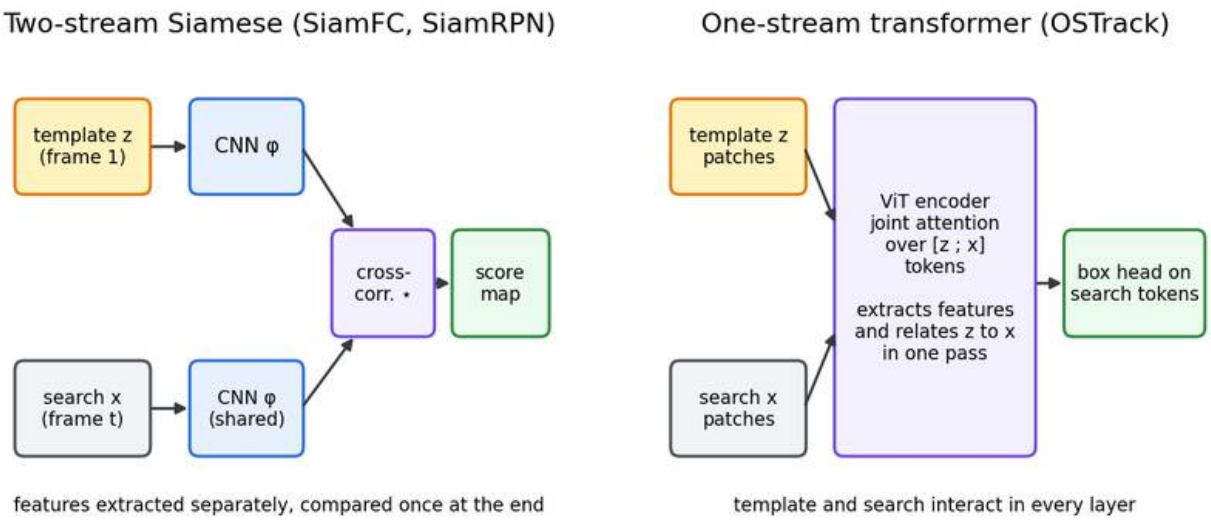



*Figure 2 — Two-stream Siamese trackers compare template and search features once, at the end. One-stream transformer trackers let them interact in every layer.*

### Tracking with video segmentation models

**Plain version.** Prompted video segmentation can track a selected object with pixel masks instead of boxes; identity and temporal consistency still matter.

SAM 2 [23] extends the Segment Anything model to video with a **streaming memory**: features and predicted masks from past frames, plus the prompted frames, are stored in a memory bank that the current frame attends to. Prompting one frame with a click, box or mask yields a "masklet" through the whole video, so it behaves like a mask-output SOT tracker. SAMURAI [24] adapts SAM 2 for tracking without any retraining: it adds a Kalman-filter motion model to choose among SAM 2's candidate masks and a motion-aware rule for which past frames enter memory, which helps in crowded scenes with similar distractors. SAM 3 [25] adds **promptable concept segmentation**: given a short noun phrase such as "yellow school bus" or an image exemplar, it detects, segments and tracks *all* matching instances in a video and gives each its own identity. That is a form of open-vocabulary MOT, and it blurs the old SOT/MOT boundary.

### Benchmarks and metrics

SOT benchmarks differ in video length, scale and whether test object classes appear in training. The two classic scores ask a simple question each frame: does the predicted box overlap the true box enough, and is its center close enough?

| Benchmark | What makes it distinct |
|---|---|
| OTB [26] | early standard benchmark with per-sequence attribute labels (occlusion, fast motion, ...); introduced the success and precision plots |
| VOT challenge [53] | yearly challenge series since 2013, with reset-based protocols; now VOTS, tracking *and* segmentation |
| LaSOT [27] | long-term: 1,400 sequences, more than 3.5 million frames, each frame manually annotated |
| GOT-10k [28] | one-shot protocol: training and test object classes do not overlap |
| TrackingNet [29] | large-scale, in-the-wild videos with a held-out evaluation server |

The two classic measures, from OTB [26], are:

$$\text{Success}(\tau) = \frac{1}{T}\sum_{t=1}^{T} \mathbb{1}[\,\text{IoU}(\hat{b}_t, b_t) > \tau\,], \qquad \text{AUC} = \int_0^1 \text{Success}(\tau)\,d\tau,$$

$$\text{Precision}(\delta) = \frac{1}{T}\sum_{t=1}^{T} \mathbb{1}[\|\hat{c}_t - c_t\|_2 \le \delta],$$

where $\hat{b}_t, b_t$ are the predicted and ground-truth boxes, $\tau$ is an IoU threshold, $\hat{c}_t, c_t$ are their centers and $\delta$ is a pixel distance, conventionally 20. The AUC of the success plot is the headline number on OTB and LaSOT. GOT-10k reports **average overlap** (the mean IoU) and success rates at $\tau = 0.5$ and 0.75.

```
import numpy as np

def iou_xywh(a, b):
    """IoU between rows of a and b, boxes as [x, y, w, h] (top-left corner)."""
    x1 = np.maximum(a[:, 0], b[:, 0]); y1 = np.maximum(a[:, 1], b[:, 1])
    x2 = np.minimum(a[:, 0] + a[:, 2], b[:, 0] + b[:, 2])
    y2 = np.minimum(a[:, 1] + a[:, 3], b[:, 1] + b[:, 3])
    inter = np.clip(x2 - x1, 0, None) * np.clip(y2 - y1, 0, None)
    return inter / (a[:, 2] * a[:, 3] + b[:, 2] * b[:, 3] - inter)

def success_auc(pred, gt, thresholds=np.linspace(0, 1, 21)):
    ious = iou_xywh(pred, gt)
    curve = np.array([(ious > t).mean() for t in thresholds])
    return curve.mean()                              # area under the success plot

def precision_at(pred, gt, px=20):
    c_pred = pred[:, :2] + pred[:, 2:] / 2; c_gt = gt[:, :2] + gt[:, 2:] / 2
    return (np.linalg.norm(c_pred - c_gt, axis=1) <= px).mean()

rng = np.random.default_rng(0)
gt = np.c_[np.linspace(50, 400, 300), np.full(300, 80.0), np.full(300, 40.0), np.full(300,
 ↪  60.0)]
pred = gt + rng.normal(0, 4, gt.shape)
pred[200:230, :2] += 120                             # 30 frames lost on a distractor
```

```
print("success AUC %.3f   precision@20px %.3f" % (success_auc(pred, gt), precision_at(pred,
 ↪  gt)))
```

This prints `success AUC 0.647   precision@20px 0.900`: the 30 frames spent on a distractor cost precision 10 points, while the small jitter on the other frames costs AUC much more, because AUC also rewards tight boxes.

## Multi-object tracking

**Plain version.** Find every object of interest in every frame, and give each one a number that stays the same for as long as it is in view — even when objects cross, hide behind each other, or leave and come back.

MOT tracks multiple targets **across consecutive frames** and maintains a consistent ID for each; a single frame on its own only supports detection. The challenges [1] lists are frequent occlusion, deciding when trajectories start and end, similar appearance between targets, and interaction between targets (people in a crowd move together and occlude each other).

Formally, given detections $\mathcal{D}_t$ in each frame, MOT seeks a set of trajectories $\mathcal{T} = \{\tau_k\}$ that partitions the true detections, discards false positives, and fills in missed ones. Offline methods solve this over the whole video, often as a graph or network-flow problem; **online** methods, which this section focuses on, must decide at frame $t$ using only frames $1..t$, which is what real-time applications require.

### The tracking-by-detection pipeline

The dominant design for a decade has been **tracking-by-detection** [2], [4]: run a detector on every frame, then link detections over time (Figure 3). Each frame goes through four steps:

1. **Detect.** A detector (Faster R-CNN, YOLO-family, DETR-family; see DLCV 03) gives boxes and scores.
2. **Predict.** A motion model moves every existing track forward to where it should be now.
3. **Describe.** Optionally, a Re-ID network embeds each detection crop.
4. **Associate.** Build a cost matrix between predicted tracks and detections and solve the assignment problem. Matched tracks are updated; unmatched detections may start new tracks (*birth*); tracks unmatched for too long are deleted (*death*).

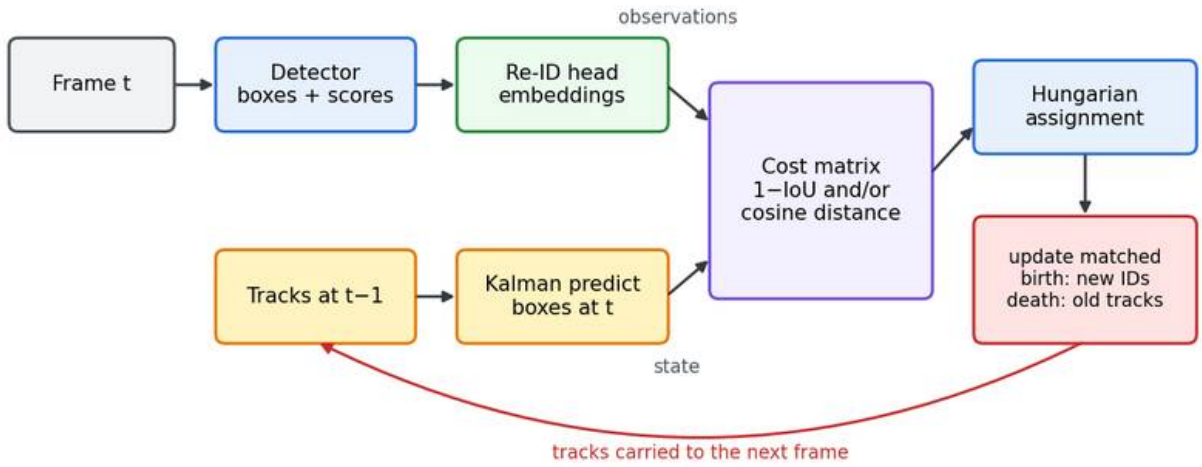



*Figure 3 — Online tracking-by-detection. Observations (boxes, embeddings) and state (predicted tracks) meet in a cost matrix; the assignment result updates the state for the next frame.*

### Motion model: the Kalman filter

**Plain version.** Guess where the object is now from where it was and how fast it was moving; then nudge that guess toward the detection, trusting whichever of the two is less uncertain.

Nearly all tracking-by-detection systems use a **linear Kalman filter** [30] with a constant-velocity model. In our code the state of one track is

$$\mathbf{x} = [\, c_x,\ c_y,\ w,\ h,\ v_x,\ v_y \,]^\top,$$

where $(c_x, c_y)$ is the box center, $w, h$ its width and height, and $(v_x, v_y)$ the center velocity in pixels per frame. (SORT uses the center, area and aspect ratio instead of $w, h$ [32]; BoT-SORT returns to $w$ and $h$ [36].) The model has two steps.

**Predict** (time update):

$$\begin{aligned} \hat{\mathbf{x}}_{t|t-1} &= \mathbf{F}\,\hat{\mathbf{x}}_{t-1|t-1}, \\ \mathbf{P}_{t|t-1} &= \mathbf{F}\,\mathbf{P}_{t-1|t-1}\,\mathbf{F}^\top + \mathbf{Q}, \end{aligned}$$

where $\mathbf{F}$ is the $6 \times 6$ transition matrix (identity plus 1s that add $v_x, v_y$ to $c_x, c_y$, since $\Delta t = 1$ frame), $\mathbf{P}$ is the state covariance (our uncertainty) and $\mathbf{Q}$ is the process noise covariance (how much the true motion can deviate from constant velocity).

**Update** (measurement update) with a matched detection $\mathbf{z}_t = [c_x, c_y, w, h]^\top$:

$$\begin{aligned} \mathbf{S}_t &= \mathbf{H}\,\mathbf{P}_{t|t-1}\,\mathbf{H}^\top + \mathbf{R}, \qquad \mathbf{K}_t = \mathbf{P}_{t|t-1}\,\mathbf{H}^\top\,\mathbf{S}_t^{-1}, \\ \hat{\mathbf{x}}_{t|t} &= \hat{\mathbf{x}}_{t|t-1} + \mathbf{K}_t(\mathbf{z}_t - \mathbf{H}\,\hat{\mathbf{x}}_{t|t-1}), \qquad \mathbf{P}_{t|t} = (\mathbf{I} - \mathbf{K}_t\mathbf{H})\,\mathbf{P}_{t|t-1}, \end{aligned}$$

where $\mathbf{H} = [\, \mathbf{I}_4 \ \ \mathbf{0} \,]$ picks the observed part of the state, $\mathbf{R}$ is the detector's measurement noise

covariance, $\mathbf{z}_t - \mathbf{H}\hat{\mathbf{x}}_{t|t-1}$ is the **innovation** (surprise), $\mathbf{S}_t$ its covariance, and $\mathbf{K}_t$ the **Kalman gain**. If the prediction is very uncertain ($\mathbf{P}$ large), the update moves the estimate almost all the way to the detection; if the detector is noisy ($\mathbf{R}$ large), it trusts the prediction. When a detection is missing, the tracker runs only the predict step ("coasting"), and $\mathbf{P}$ grows (Figure 4).

```
import numpy as np
from scipy.optimize import linear_sum_assignment

# ---------- 1. constant-velocity Kalman filter ----------
class KalmanBox:
    """State x = [cx, cy, w, h, vx, vy]; measurement z = [cx, cy, w, h]."""
    def __init__(self, z, q=1.0, r=4.0):
        self.x = np.r_[z, 0.0, 0.0]
        self.P = np.diag([r, r, r, r, 100.0, 100.0])    # unknown velocity: large variance
        self.F = np.eye(6); self.F[0, 4] = self.F[1, 5] = 1.0   # dt = 1 frame
        self.H = np.eye(4, 6)
        self.Q = q * np.diag([1, 1, 0.1, 0.1, 0.5, 0.5])
        self.R = r * np.eye(4)

    def predict(self):
        self.x = self.F @ self.x
        self.P = self.F @ self.P @ self.F.T + self.Q
        return self.x[:4]

    def update(self, z):
        S = self.H @ self.P @ self.H.T + self.R          # innovation covariance
        K = self.P @ self.H.T @ np.linalg.inv(S)          # Kalman gain
        self.x = self.x + K @ (z - self.H @ self.x)
        self.P = (np.eye(6) - K @ self.H) @ self.P
```

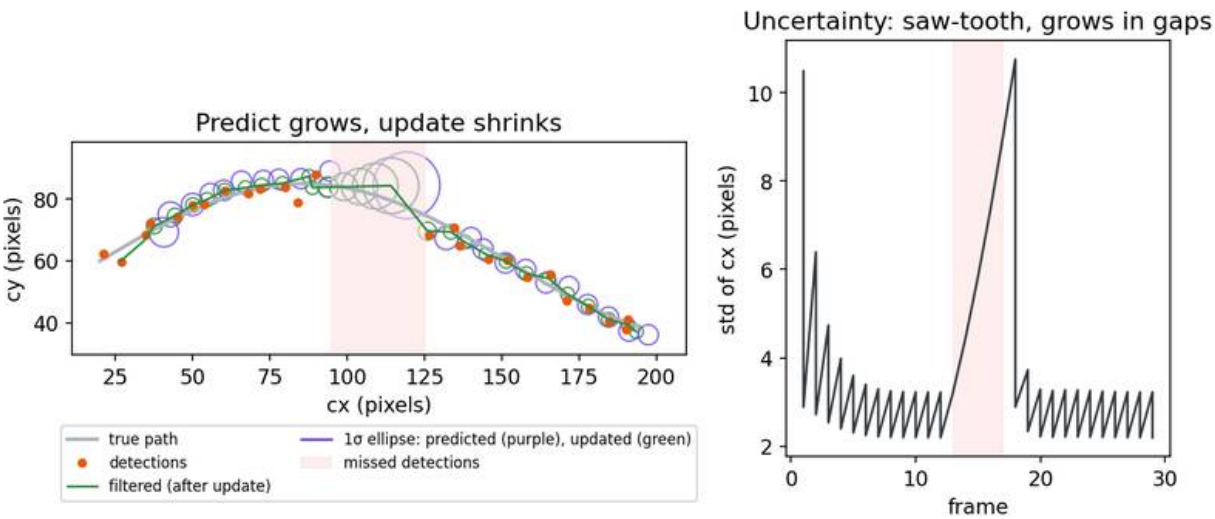



*Figure 4 — Kalman predict and update on a synthetic track. Every predict step inflates the uncertainty, every update shrinks it. During five missed detections (pink) the filter coasts in a straight line and its uncertainty grows; this is exactly when identity errors happen.*

The squared **Mahalanobis distance** $d_M^2 = \boldsymbol{\nu}^\top \mathbf{S}^{-1}\boldsymbol{\nu}$ of an innovation $\boldsymbol{\nu}$ follows a $\chi^2$ distribution with 4 degrees of freedom under the model, so DeepSORT [33] uses it to *gate* (forbid) implausible matches. The constant-velocity assumption is also the main weakness: dancers, athletes and anything filmed by a moving camera violate it.

### Appearance embeddings

A Re-ID network (previous section) turns each detection crop into a unit vector $\mathbf{e}_j$. Each track keeps an appearance memory, for example an exponential moving average $\mathbf{m}_k \leftarrow \alpha\,\mathbf{m}_k + (1-\alpha)\,\mathbf{e}_j$ followed by renormalization, with $\alpha \in [0,1)$ (we use 0.9). The appearance cost is the cosine distance $1 - \mathbf{m}_k^\top \mathbf{e}_j$. Appearance is what lets a tracker recover an identity after a long occlusion, when motion prediction has become useless.

### Association: cost matrices and the Hungarian algorithm

**Plain version.** Make a table of "how bad would it be to match track $k$ with detection $j$" for every pair, then choose the one-to-one matching with the smallest total badness.

With $K$ predicted tracks and $N$ detections, build a cost matrix $\mathbf{C} \in \mathbb{R}^{K\times N}$, for example

$$C_{kj} = \lambda\,(1 - \mathrm{IoU}(\hat{b}_k, b_j)) + (1-\lambda)\,(1 - \mathbf{m}_k^\top \mathbf{e}_j),$$

where $\hat{b}_k$ is the predicted box of track $k$, $b_j$ detection $j$'s box and $\lambda \in [0,1]$ the weight of motion versus appearance ($\lambda = 1$ is pure IoU, as in SORT). Then solve the **linear assignment problem**

$$\min_{\mathbf{X}\in\{0,1\}^{K\times N}} \sum_{k,j} C_{kj} X_{kj} \quad \text{s.t.} \quad \sum_j X_{kj} \le 1,\ \sum_k X_{kj} \le 1,$$

so each track takes at most one detection and vice versa. The **Hungarian algorithm** [31] solves it exactly in polynomial time ($O(n^3)$ for $n = \max(K,N)$); SciPy's `linear_sum_assignment` implements a fast variant. Finally, any matched pair whose cost exceeds a threshold is rejected, which is how gating is applied in practice.

*Online only: The Hungarian algorithm: optimal one-to-one assignment — https://blog.shyandram.dev/en/learn/deep-learning/object-tracking/notes/the-hungarian-algorithm-optimal-one-to-one-assignment/*

```
# ---------- 2. IoU and Hungarian association ----------
def iou_matrix(a, b):
    """a: (N,4), b: (M,4) boxes as [cx, cy, w, h] -> (N,M) IoU."""
    a1, a2 = a[:, None, :2] - a[:, None, 2:] / 2, a[:, None, :2] + a[:, None, 2:] / 2
    b1, b2 = b[None, :, :2] - b[None, :, 2:] / 2, b[None, :, :2] + b[None, :, 2:] / 2
    wh = np.clip(np.minimum(a2, b2) - np.maximum(a1, b1), 0, None)
    inter = wh[..., 0] * wh[..., 1]
    area = lambda x: x[..., 2] * x[..., 3]
    return inter / (area(a)[:, None] + area(b)[None, :] - inter + 1e-9)

def associate(cost, max_cost):
    """Hungarian assignment; pairs whose cost exceeds max_cost are rejected."""
```

```
    if cost.size == 0:
        return [], list(range(cost.shape[0])), list(range(cost.shape[1]))
    rows, cols = linear_sum_assignment(cost)
    pairs = [(r, c) for r, c in zip(rows, cols) if cost[r, c] <= max_cost]
    mr = {r for r, _ in pairs}; mc = {c for _, c in pairs}
    return (pairs, [r for r in range(cost.shape[0]) if r not in mr],
            [c for c in range(cost.shape[1]) if c not in mc])
```

### A complete tracker on a synthetic scene

To see what each idea buys, we generate a scene with five pedestrians over 80 frames, designed to contain the classic failure cases:

- objects 0 and 1 **cross**, and while they overlap, object 1 (behind) gets only **low-confidence** detections;
- objects 2 and 3 meet and **bounce back** (non-linear motion, like dancers), with the rear one also low-confidence during the overlap;
- object 4 walks **behind a pole** for 8 frames, so its detections are low-confidence there;
- every detection is dropped with 5% probability, and there are occasional low-score false positives.

Each detection also carries a 16-D appearance vector: its object's fixed unit vector plus noise.

```
# ---------- 3. synthetic scene ----------
def make_scene(T=80, seed=0):
    rng = np.random.default_rng(seed)
    W, H = 30.0, 70.0
    t = np.arange(T, dtype=float)
    tr = {}
    # pair 1: plain crossing, B is behind A (occluded while they overlap)
    tr[0] = np.c_[80 + 5 * t, np.full(T, 120.0)]
    tr[1] = np.c_[480 - 5 * t, np.full(T, 128.0)]
    # pair 2: they meet and bounce back (non-linear motion, like dancers)
    m = 40
    x2 = np.where(t < m, 120 + 4 * t, 120 + 4 * m - 4 * (t - m))
    x3 = np.where(t < m, 440 - 4 * t, 440 - 4 * m + 4 * (t - m))
    tr[2] = np.c_[x2 + 10, np.full(T, 260.0)]
    tr[3] = np.c_[x3 - 10, np.full(T, 266.0)]
    # walker passing behind a pole (frames 30-37): detections become low-score
    tr[4] = np.c_[60 + 3 * t, 40 + 1.0 * t]
    gt = []                                     # per frame list of (id, box)
    dets = []                                   # per frame list of (box, score, gt_id)
    feats = {k: (lambda v: v / np.linalg.norm(v))(rng.normal(size=16)) for k in tr}
    for f in range(T):
        g, d = [], []
        boxes = {k: np.r_[tr[k][f], W, H] for k in tr}
        for k, b in boxes.items():
            g.append((k, b))
            score = rng.uniform(0.6, 0.95)
            if k == 1 and iou_matrix(b[None], boxes[0][None])[0, 0] > 0.2:
```

```
            score = rng.uniform(0.15, 0.45)       # occluded by object 0
        if k == 3 and iou_matrix(b[None], boxes[2][None])[0, 0] > 0.2:
            score = rng.uniform(0.15, 0.45)       # occluded by object 2
        if k == 4 and 30 <= f <= 37:
            score = rng.uniform(0.15, 0.45)       # behind the pole
        if rng.random() < 0.05:
            continue                               # random missed detection
        noisy = b + np.r_[rng.normal(0, 2.0, 2), rng.normal(0, 1.0, 2)]
        e = feats[k] + 0.15 * rng.normal(size=16)
        d.append((noisy, score, k, e / np.linalg.norm(e)))
    if rng.random() < 0.3:                         # low-score false positive
        fp = np.r_[rng.uniform(50, 550), rng.uniform(50, 320), W, H]
        e = rng.normal(size=16)
        d.append((fp, rng.uniform(0.1, 0.4), -1, e / np.linalg.norm(e)))
    gt.append(g); dets.append(d)
return gt, dets
```

The tracker below has three switches. With both off it is **SORT-style**: Kalman prediction, IoU cost, Hungarian matching on high-score detections (score $\geq 0.5$), tracks deleted after 2 missed frames. `use_low=True` adds a **BYTE**-style second pass: tracks left unmatched try the low-score detections, with IoU only. `use_app=True` adds the **appearance** cost to the first pass and a third pass that re-identifies tracks lost for up to 30 frames by appearance alone.

```
# ---------- 4. SORT-style tracker with optional BYTE and appearance ----------
class Track:
    def __init__(self, tid, z, emb):
        self.id, self.kf, self.emb = tid, KalmanBox(z), emb
        self.lost = 0

def run_tracker(dets, use_low=False, use_app=False, hi=0.5, max_age=2,
                app_age=30, lam=0.5):
    tracks, out, next_id = [], [], 0
    for d in dets:
        for tk in tracks:
            tk.kf.predict()
        boxes = np.array([x[0] for x in d]).reshape(-1, 4)
        scores = np.array([x[1] for x in d])
        embs = np.array([x[3] for x in d]).reshape(-1, 16)
        high = np.where(scores >= hi)[0]
        low = np.where((scores < hi) & (scores >= 0.1))[0]
        active = [tk for tk in tracks if tk.lost <= max_age]
        pred = np.array([tk.kf.x[:4] for tk in active]).reshape(-1, 4)
        # stage 1: high-score detections vs active tracks
        cost = 1 - iou_matrix(pred, boxes[high])
        gate = 0.7
        if use_app and len(active) and len(high):
            tE = np.array([tk.emb for tk in active])
            app = 1 - tE @ embs[high].T            # cosine distance
            cost = np.where(cost > 0.9, 9.0, lam * cost + (1 - lam) * app)
            gate = 0.6
        pairs, ut, ud = associate(cost, gate)
        matched = [(active[r], high[c]) for r, c in pairs]
```

```
        rest_tracks = [active[r] for r in ut]
        rest_high = [high[c] for c in ud]
        # stage 2 (BYTE): remaining tracks vs low-score detections, IoU only
        if use_low and rest_tracks and len(low):
            p2 = np.array([tk.kf.x[:4] for tk in rest_tracks])
            pairs2, ut2, _ = associate(1 - iou_matrix(p2, boxes[low]), 0.5)
            matched += [(rest_tracks[r], low[c]) for r, c in pairs2]
            rest_tracks = [rest_tracks[r] for r in ut2]
        # stage 3 (appearance only): long-lost tracks vs leftover high-score dets
        if use_app and rest_high:
            lost = [tk for tk in tracks if max_age < tk.lost <= app_age]
            if lost:
                app = 1 - np.array([tk.emb for tk in lost]) @ embs[rest_high].T
                pairs3, _, ud3 = associate(app, 0.3)
                matched += [(lost[r], rest_high[c]) for r, c in pairs3]
                rest_high = [rest_high[c] for c in ud3]
        for tk, j in matched:
            tk.kf.update(boxes[j]); tk.lost = 0
            tk.emb = 0.9 * tk.emb + 0.1 * embs[j]; tk.emb /= np.linalg.norm(tk.emb)
        hit = {id(tk) for tk, _ in matched}
        for tk in tracks:
            if id(tk) not in hit:
                tk.lost += 1
        for j in rest_high:                            # birth: unmatched high-score
            tracks.append(Track(next_id, boxes[j], embs[j])); next_id += 1
        tracks = [tk for tk in tracks if tk.lost <= (app_age if use_app else max_age)]
        out.append([(tk.id, tk.kf.x[:4].copy()) for tk in tracks if tk.lost == 0])
    return out
```

The metric code (MOTA, IDF1 and HOTA, explained below) and the experiment:

```
# ---------- 5. metrics: MOTA, IDF1, HOTA ----------
def frame_match(g, p, thr):
    if not g or not p:
        return []
    iou = iou_matrix(np.array([b for _, b in g]), np.array([b for _, b in p]))
    r, c = linear_sum_assignment(-iou)
    return [(g[i][0], p[j][0]) for i, j in zip(r, c) if iou[i, j] >= thr]

def mota(gt, hyp, thr=0.5):
    fn = fp = idsw = 0; ngt = 0; last = {}
    for g, p in zip(gt, hyp):
        m = frame_match(g, p, thr)
        ngt += len(g); fn += len(g) - len(m); fp += len(p) - len(m)
        for gi, pi in m:
            if gi in last and last[gi] != pi:
                idsw += 1
            last[gi] = pi
    return 1 - (fn + fp + idsw) / ngt, idsw

def idf1(gt, hyp, thr=0.5):
    gids = sorted({i for g in gt for i, _ in g}); pids = sorted({i for p in hyp for i, _ in
    ↪  p})
```

```
    gi = {k: n for n, k in enumerate(gids)}; pi = {k: n for n, k in enumerate(pids)}
    ov = np.zeros((len(gids), len(pids)))
    for g, p in zip(gt, hyp):
        if g and p:
            iou = iou_matrix(np.array([b for _, b in g]), np.array([b for _, b in p]))
            for a in range(len(g)):
                for b in range(len(p)):
                    ov[gi[g[a][0]], pi[p[b][0]]] += iou[a, b] >= thr
    r, c = linear_sum_assignment(-ov)
    idtp = ov[r, c].sum()
    n_gt = sum(len(g) for g in gt); n_p = sum(len(p) for p in hyp)
    return 2 * idtp / (n_gt + n_p)

def hota(gt, hyp, alphas=np.arange(0.05, 0.96, 0.05)):
    res = []
    for a in alphas:
        pairs = []; n_gt = n_p = 0
        for g, p in zip(gt, hyp):
            pairs += frame_match(g, p, a); n_gt += len(g); n_p += len(p)
        tp = len(pairs)
        if tp == 0:
            res.append((0, 0, 0)); continue
        cnt = {}
        for q in pairs:
            cnt[q] = cnt.get(q, 0) + 1
        gcount = {}; pcount = {}
        for g in gt:
            for i, _ in g: gcount[i] = gcount.get(i, 0) + 1
        for p in hyp:
            for i, _ in p: pcount[i] = pcount.get(i, 0) + 1
        ass = sum(cnt[q] / (gcount[q[0]] + pcount[q[1]] - cnt[q]) for q in pairs) / tp
        det = tp / (n_gt + n_p - tp)
        res.append((np.sqrt(det * ass), det, ass))
    return np.array(res).mean(axis=0)   # HOTA, DetA, AssA averaged over alpha

gt, dets = make_scene(seed=3)
for name, kw in [("IoU only", {}), ("+ low-score pass", dict(use_low=True)),
                 ("+ appearance", dict(use_low=True, use_app=True))]:
    hyp = run_tracker(dets, **kw)
    m, sw = mota(gt, hyp)
    print(f"{name:18s} MOTA {m:.2f}  IDF1 {idf1(gt, hyp):.2f}  "
          f"HOTA {hota(gt, hyp)[0]:.2f}  ID switches {sw}")
```

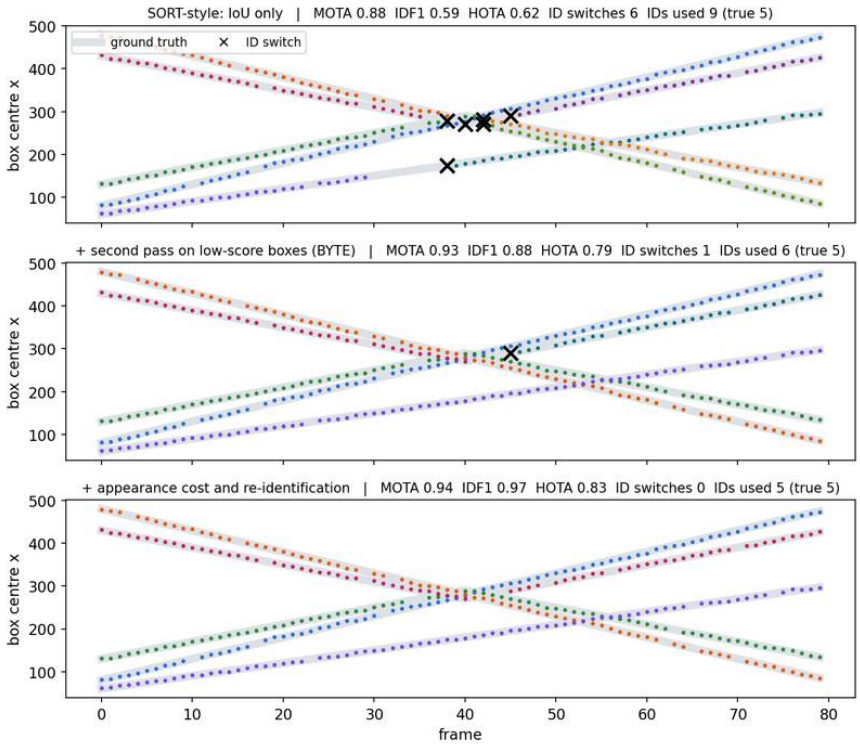



*Figure 5 — The same synthetic detections tracked three ways. Colors are predicted IDs; black crosses mark ID switches. Low-score association removes fragmentation from occlusion; appearance fixes the bounce, where constant-velocity prediction points the wrong way.*

The script prints:

| Tracker | MOTA | IDF1 | HOTA | ID switches | IDs used (true: 5) |
|---|---|---|---|---|---|
| SORT-style, IoU only | 0.88 | 0.59 | 0.62 | 6 | 9 |
| + BYTE low-score pass | 0.93 | 0.88 | 0.79 | 1 | 6 |
| + appearance and re-identification | 0.94 | 0.97 | 0.83 | 0 | 5 |

Read it carefully. The IoU-only tracker loses the occluded objects because their detections fall below the score threshold, so their tracks die and come back with **new IDs** (fragmentation). The low-score pass keeps them alive, which fixes almost everything, but it cannot fix the bounce: the Kalman filter predicts the two dancers will keep going, so after the overlap one of them is not matched and later restarts with a new ID. Only appearance resolves that. Notice also how little MOTA moves (0.88 → 0.94) compared with IDF1 (0.59 → 0.97): MOTA is mostly a detection measure, a point we return to in the metrics section.

**Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

### The SORT family

These ideas map directly onto the most cited online trackers:

| Tracker | Key idea |
|---|---|
| SORT [32] | Kalman filter + IoU cost + Hungarian assignment; deliberately minimal and very fast. Showed that detector quality dominates tracking quality. |
| DeepSORT [33] | Adds a CNN appearance descriptor trained on a person Re-ID dataset, Mahalanobis gating, and a *matching cascade* that gives priority to recently seen tracks. Fewer ID switches through longer occlusions. |
| ByteTrack [34] | Associates **every** detection box: high-score boxes first, then the remaining tracks against **low-score** boxes using IoU only. Occluded objects often get low scores, so they are recovered instead of discarded, while low-score background boxes stay unmatched and are thrown away. |
| OC-SORT [35] | "Observation-centric": when a lost track is re-found, it re-runs the Kalman updates along a virtual trajectory between the last and new observations (fixing error accumulated while coasting), and adds a motion-direction consistency term to the cost. Strong on non-linear motion such as DanceTrack. |
| BoT-SORT [36] | ByteTrack-style association plus camera-motion compensation (global image registration), a Kalman state with $w, h$ instead of aspect ratio, and a fused IoU/Re-ID cost. |

The lesson of this family is that *carefully engineered association on top of a strong detector* is very hard to beat on pedestrian benchmarks. The 2025 survey of Adžemović [4] reaches the same conclusion: heuristic methods lead on dense scenes with mostly linear motion, while learned association does better when motion is complex.

### Joint detection and embedding, and detector-based tracking

Running a separate Re-ID network on every crop is expensive. **JDE** [37] adds an embedding head to a one-stage detector so that boxes and appearance vectors come out of one forward pass. **FairMOT** [38] argued that this sharing had been "unfair" to Re-ID: anchor-based detectors produce features that are ambiguous for identity, and detection dominates training. It uses an anchor-free, CenterNet-style detector with two equal branches (detection and Re-ID) on a high-resolution feature map. **Tracktor** [39], which the original post also cites, removes explicit association for most cases: it feeds the previous frame's boxes to the detector's box-regression head, which pulls each box onto the object's new position and keeps its identity.

### Tracking by query: end-to-end MOT

**Plain version.** Instead of hand-writing "predict, compare, match", give a transformer a memory slot per object and let it learn to update the slot each frame.

DETR-style detectors (see DLCV 03) represent objects as **queries**. MOTR [40] extends this to video: each detected object's output embedding becomes a **track query** that is passed to the next frame and keeps predicting the same object, while new *detect queries* find newborn objects. Identity is carried implicitly by the query; there is no Kalman filter and no Hungarian matching at inference. Training uses a tracklet-aware label assignment and a loss averaged over clips. The weakness was detection quality: one decoder must both detect and associate, and the two tasks conflict. MOTRv2 [41] bootstraps MOTR with proposals from a separate pretrained detector (YOLOX) used as anchors, which eases that conflict. MOTIP [42] (CVPR 2025) reformulates association as **ID prediction**: given the trajectories so far, each with an ID label drawn from a learnable ID dictionary, a decoder classifies each new detection into one of the existing IDs or "new". This is in-context, end-to-end trainable, and works with plain object-level features.

### Open-vocabulary MOT

Classic MOT only tracks the classes in its training set. **OVTrack** [43] (CVPR 2023) tracks arbitrary categories by distilling knowledge from a vision-language model (CLIP) for both classification and association, and by learning robust appearance features from *hallucinated* image pairs produced with a diffusion model. SAM 3's concept prompts [25] approach the same goal from the segmentation side.

### Datasets

| Dataset | Domain | What it tests |
|---|---|---|
| MOT17 [44] | pedestrians, static and moving cameras | the standard pedestrian benchmark; the MOT16 sequences re-released as MOT17 |
| MOT20 [45] | very crowded pedestrian scenes | heavy occlusion, small targets |
| DanceTrack [46] | group dancing | uniform appearance and diverse, non-linear motion; appearance cannot separate people |
| SportsMOT [47] | basketball, volleyball, football (240 sequences) | fast, variable-speed motion with similar but distinguishable appearance |
| BDD100K MOT [48] | driving video | multi-class tracking (cars, pedestrians, cyclists, ...) from a moving vehicle |

The shift from MOT17 to DanceTrack and SportsMOT changed the field's priorities: on MOT17 a good detector and IoU matching go far, but on DanceTrack the motion model and association learning matter most.

### Metrics: MOTA, IDF1, HOTA

**Plain version.** A tracker can fail in two ways: it can miss objects or invent them (detection errors), or it can mix up who is who (association errors). Different metrics weigh these two differently.

All three metrics first match predictions to ground truth in each frame at an IoU threshold. Let $\mathrm{GT}_t$, $\mathrm{FN}_t$, $\mathrm{FP}_t$ and $\mathrm{IDSW}_t$ be the number of ground-truth objects, misses, false positives and identity switches in frame $t$. An identity switch occurs when a ground-truth object is matched to a different predicted ID than at its previous match.

**MOTA** (CLEAR MOT [9]):

$$\mathrm{MOTA} = 1 - \frac{\sum_t (\mathrm{FN}_t + \mathrm{FP}_t + \mathrm{IDSW}_t)}{\sum_t \mathrm{GT}_t}.$$

It ranges from $-\infty$ to 1. Because there are usually far more detections than switches, FN and FP dominate it: MOTA is mostly a detection score. A switch is also counted only at the moment it happens, so a tracker that swaps two IDs once and keeps them swapped for 1,000 frames pays the same as one that swaps back after 1 frame.

**IDF1** [10] computes a *global* one-to-one matching between ground-truth trajectories and predicted trajectories (again with the Hungarian algorithm) that maximizes the number of frames where they agree, then

$$\mathrm{IDF1} = \frac{2\,\mathrm{IDTP}}{2\,\mathrm{IDTP} + \mathrm{IDFP} + \mathrm{IDFN}},$$

where IDTP counts detections matched to the trajectory assigned to their ground-truth identity, and IDFP, IDFN the remaining predicted and ground-truth detections. IDF1 measures how long the tracker keeps the right identity, so it is mostly an association score — but because the matching is global, it can behave unintuitively, for example dropping when detection improves.

**HOTA** (Higher Order Tracking Accuracy [8]) was introduced because neither metric is balanced or decomposable. For a localization threshold $\alpha$, with per-frame matches giving true positives (TP), FN and FP:

$$\mathrm{DetA}_\alpha = \frac{|\mathrm{TP}|}{|\mathrm{TP}| + |\mathrm{FN}| + |\mathrm{FP}|}, \qquad \mathrm{AssA}_\alpha = \frac{1}{|\mathrm{TP}|} \sum_{c \in \mathrm{TP}} \frac{|\mathrm{TPA}(c)|}{|\mathrm{TPA}(c)| + |\mathrm{FNA}(c)| + |\mathrm{FPA}(c)|},$$

$$\mathrm{HOTA}_\alpha = \sqrt{\mathrm{DetA}_\alpha \cdot \mathrm{AssA}_\alpha}, \qquad \mathrm{HOTA} = \frac{1}{19} \sum_{\alpha \in \{0.05,\, 0.10,\, \ldots,\, 0.95\}} \mathrm{HOTA}_\alpha.$$

For a matched pair $c$ (ground-truth ID $g$, predicted ID $p$), $\mathrm{TPA}(c)$ is the set of all TPs in the video that also match $g$ to $p$; $\mathrm{FNA}(c)$ the detections of $g$ matched to another ID or missed; $\mathrm{FPA}(c)$ the detections of $p$ matched to another object or to nothing. The geometric mean means a tracker must be good at *both* detection and association to score well, and DetA and AssA (and a localization accuracy, LocA) can be reported separately to diagnose failures. Averaging over $\alpha$ also rewards precise boxes. (Our code uses a simplified matching that maximizes IoU per frame; the official definition matches with a score that also favors pairs that have been associated before.)

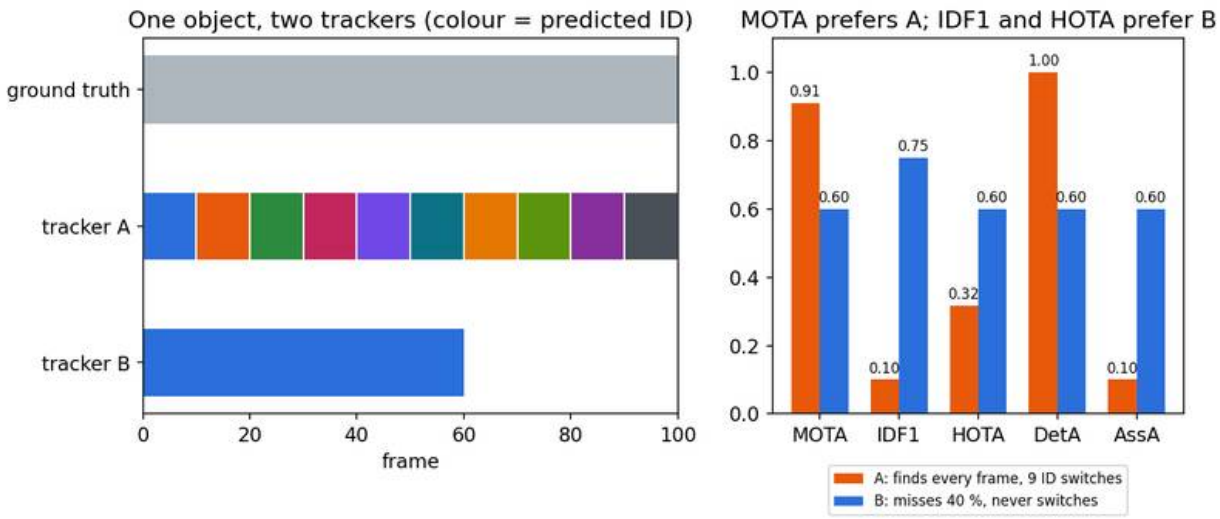



*Figure 6 — Why HOTA was introduced. Tracker A detects every frame but switches identity nine times; tracker B misses 40% of frames but never switches. MOTA ranks A far ahead; IDF1 and HOTA rank B ahead, and DetA/AssA show exactly why.*

Figure 6 is computed with the functions above: tracker A gets MOTA $= 1 - 9/100 = 0.91$ and AssA $= 0.10$, giving HOTA $= \sqrt{1.0 \times 0.10} \approx 0.32$; tracker B gets 0.60 on MOTA, DetA, AssA and HOTA, and IDF1 $= 2 \cdot 60/(100 + 60) = 0.75$. Today, papers on MOT17, MOT20, DanceTrack and SportsMOT report HOTA, IDF1 and MOTA together, and HOTA is usually the primary ranking metric.

## Point tracking: a related trend

**Plain version.** Instead of following whole objects, follow individual *points* — a freckle on a cheek, a corner of a box — through the whole video, even when they are briefly hidden.

**Tracking any point (TAP)** asks, for a query point $(x, y)$ in frame $t_0$, for its position and visibility in every other frame. It generalizes optical flow (two frames) to long, occlusion-aware trajectories, and is a natural successor to the classical keypoint tracking from IP 12. Representative models are TAPIR [49] and the CoTracker line [50], [51], which tracks many points *jointly*, so that a point that becomes occluded can be inferred from its visible neighbors. Point trackers are now used as motion cues inside object trackers, video editing and robotics.

*Online only: Point tracking models: TAPIR and the CoTracker line — https://blog.shyandram.dev/en/learn/deep-learning/object-tracking/notes/point-tracking-models-tapir-and-the-cotracker-line/*

## Multi-target multi-camera tracking

**Plain version.** Many cameras watch one place. Track everyone in each camera, then figure out that the woman leaving camera 3 is the same woman entering camera 5.

MTMCT is the ultimate goal named in [1]. A typical pipeline runs single-camera MOT to produce *tracklets*, embeds each tracklet with Re-ID features, and clusters tracklets across cameras using appearance, camera topology (which cameras connect, and travel times between them) and, with calibrated cameras, 3D positions on a ground plane. IDF1 was originally proposed for this setting [10]. The **AI City Challenge** is the main recurring benchmark. Its ninth edition (2025) [52] made Track 1 a multi-class **3D** multi-camera tracking task in warehouse-style environments, with people, humanoid robots, autonomous mobile robots and forklifts, calibrated cameras and 3D box annotations. That shift — from image boxes to calibrated 3D tracks across many views — is where MTMCT is heading.

## Modern view

**Plain version.** Two kinds of tracker now coexist: carefully engineered detect-predict-match pipelines, and models that learn association end to end. Around them, tracking is reaching out to any category named in text, to individual points, and to many calibrated cameras.

### What the surveys say

- **Luo et al. [3]** (*Artificial Intelligence*, 2021): the classical MOT review; a formal problem definition and a component-by-component analysis, independent of any network.
- **Ciaparrone et al. [2]** (*Neurocomputing*, 2020): deep MOT in four stages (detection, feature extraction/motion prediction, affinity, association); detection quality and learned appearance features drive most gains.
- **Adžemović [4]** (arXiv, 2025): tracking-by-detection families against end-to-end trackers; heuristic trackers lead on crowded, mostly linear scenes, learned association wins on complex motion.
- **Marvasti-Zadeh et al. [5]** (*IEEE T-ITS*, 2022) and **Thangavel (Kugarajeevan) et al. [6]** (*IEEE Access*, 2023): deep and transformer SOT; the one-stream (OSTrack-style) design emerges as the dominant pattern.
- **Ye et al. [7]** (*IEEE TPAMI*, 2022): closed-world versus open-world Re-ID, with the AGW baseline and the mINP metric.
- **Luiten et al. [8]** (*IJCV*, 2020): not a survey, but its re-examination of MOTA and IDF1 is the reason current leaderboards rank by HOTA.

*Online only: The tracking surveys in more detail — https://blog.shyandram.dev/en/learn/deep-learning/object-tracking/papers/the-tracking-surveys-in-more-detail/*

### State of the art, 2024–2026

- **SOT** is dominated by one-stream transformer trackers in the OSTrack/SeqTrack lineage and, increasingly, by segmentation foundation models: SAM 2 with motion-aware memory (SAMURAI) gives strong zero-shot tracking without tracking-specific training [23], [24].

- **MOT** has two coexisting camps. Heuristic tracking-by-detection (ByteTrack, OC-SORT, BoT-SORT and descendants) remains the default in practice: simple, fast and modular, with a strong detector doing most of the work. End-to-end trackers (MOTR, MOTRv2, MOTIP) lead where motion is complex and appearance is ambiguous, such as DanceTrack and SportsMOT [4], [42].
- **Open-vocabulary and promptable tracking** is moving into mainstream models: OVTrack established the task, and SAM 3 detects, segments and tracks instances of a text concept with identities [25], [43].
- **Point tracking** (TAPIR, CoTracker, CoTracker3) matured into a general motion primitive [49]–[51].
- **MTMCT** is moving to calibrated, multi-class 3D tracking (AI City Challenge 2025) [52].

### Open problems

- **Long-term occlusion and re-entry.** Motion models are useless after a few seconds; appearance must carry identity, but appearance is unreliable for uniform clothing (DanceTrack, sports) and across cameras.
- **Non-linear motion and camera motion.** Constant-velocity Kalman filters remain the default and are the weak point on dance, sports and driving data; learned motion models are not yet clearly better everywhere.
- **End-to-end versus heuristic association.** End-to-end trackers are elegant and win on complex motion, but need lots of video training data and compute and still lose to simple heuristics on crowded pedestrian scenes. A unified design that is best on both is open.
- **Open-vocabulary and long-tail tracking.** Rare classes, and appearance features that generalize to classes never seen in training.
- **Evaluation.** HOTA fixed much, but benchmarks are pedestrian-heavy, and ID-level evaluation over hours of video and many cameras remains hard. **Privacy and dataset ethics** (the DukeMTMC lesson) constrain what data can be collected at all.

## Key takeaways

- Tracking follows locations while preserving identities; re-identification instead ranks appearance matches.
- An initialized single-object tracker and an online multi-object tracker solve different input/output problems.
- Tracking-by-detection combines prediction, matching, updates and track birth/death decisions.
- Motion, appearance and low-confidence detections provide complementary evidence, each with failure cases.
- Separate detection quality from association quality using the metrics appropriate to the tracking task.
- Inspect occlusion, long gaps, camera changes and identity switches rather than relying on one aggregate score.

## Exercises

1. **Kalman gain intuition.** For a 1-D constant-position model with prior variance $P$ and measurement noise $R$, show that the Kalman gain is $K = P/(P + R)$ and that the posterior variance is $PR/(P + R)$. What happens to $K$ after five frames without detections if each predict step adds $Q$ to $P$?

   **Hint**

   With $H = 1$, $S = P + R$ and $K = P/S$. The posterior variance is $(1 - K)P = PR/(P + R)$, smaller than both $P$ and $R$. After five coasting steps the prior is $P + 5Q$, so $K$ moves toward 1: the filter will trust the next detection much more, which is why a wrong match after an occlusion moves the track a lot.

2. **Hungarian by hand.** Tracks A and B have predicted IoUs with detections 1 and 2 of $\begin{bmatrix} 0.6 & 0.5 \\ 0.55 & 0.0 \end{bmatrix}$. What does greedy matching (highest IoU first) return, and what does the Hungarian algorithm on cost $1 - \text{IoU}$ return? Which total IoU is larger?

   **Hint**

   Greedy takes A–1 (0.6) first, leaving B–2 with IoU 0 (rejected), total 0.6. Hungarian picks A–2 and B–1, total $0.5 + 0.55 = 1.05$. Greedy leaves track B unmatched and starts a new ID for detection 2; this is one way ID switches appear.

3. **Break the tracker.** In the synthetic scene, change the bounce in `make_scene` so that objects 2 and 3 pass through each other instead (no velocity reversal). Which of the three trackers improves the most, and why?

   **Hint**

   With linear motion the constant-velocity prediction is correct through the overlap. With `seed=3`, the low-score tracker's IDF1 rises from 0.88 to about 0.96 and it uses the correct 5 IDs, almost matching the appearance tracker (about 0.96 as well); the IoU-only tracker still fragments occluded objects. Appearance helps most exactly when motion is non-linear, which is the design rationale of DanceTrack.

4. **Metric arithmetic.** A video has one object for 200 frames. A tracker detects it in all frames but uses ID 1 for frames 1–100 and ID 2 for frames 101–200. Compute MOTA, IDF1, DetA, AssA and HOTA at a single threshold, assuming perfect boxes.

   **Hint**

   MOTA $= 1 - 1/200 = 0.995$. The best global ID match covers 100 frames: IDF1 $= 2 \cdot 100/(200 + 200) = 0.5$. DetA $= 1$. For each TP, $|\text{TPA}| = 100$, $|\text{FNA}| = 100$, $|\text{FPA}| = 0$, so AssA $= 0.5$ and HOTA $= \sqrt{0.5} \approx 0.71$. One swap barely moves MOTA but halves AssA.

5. **Re-ID metrics.** In the Re-ID snippet, change the noise level from 1.6 to 2.2 and then to 1.0. How do rank-1 and mAP change, and which one is more sensitive? Explain using the definition of AP.

   **Hint**

   mAP falls faster than rank-1 as noise grows, because rank-1 only needs the single easiest true match at the top, while AP penalizes every true match that is pushed down the list. With 4 true matches per query, the hardest one usually decides how much AP is lost.

6. **Design question.** You must track forklifts and workers across 12 calibrated cameras in a warehouse. Sketch a pipeline using components from this tutorial, and say which metric you would report and why.

   **Hint**

   Per-camera detection plus a ByteTrack/BoT-SORT-style tracker; project boxes to the ground plane using calibration; cross-camera association by 3D position and time first, Re-ID appearance second (workers may wear identical vests); handle the two classes separately. Report HOTA (balanced, decomposable into DetA/AssA) and IDF1 (long-term identity across cameras), as in the AI City Challenge setting.

## References


1. Shyandram, "物件追蹤 Object Tracking 簡介" (Introduction to object tracking), Medium post (in Traditional Chinese), 2023; updated 2026. link
2. G. Ciaparrone, F. Luque Sánchez, S. Tabik, L. Troiano, R. Tagliaferri and F. Herrera, "Deep learning in video multi-object tracking: A survey," *Neurocomputing*, 2020. arXiv:1907.12740
3. W. Luo, J. Xing, A. Milan, X. Zhang, W. Liu and T.-K. Kim, "Multiple object tracking: A literature review," *Artificial Intelligence*, vol. 293, 2021 (arXiv 2014). arXiv:1409.7618
4. M. Adžemović, "Deep learning-based multi-object tracking: A comprehensive survey from foundations to state-of-the-art," arXiv:2506.13457, 2025. arXiv
5. S. M. Marvasti-Zadeh, L. Cheng, H. Ghanei-Yakhdan and S. Kasaei, "Deep learning for visual tracking: A comprehensive survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, 2022. arXiv:1912.00535
6. J. Thangavel (Kugarajeevan), T. Kokul, A. Ramanan and S. Fernando, "Transformers in single object tracking: An experimental survey," *IEEE Access*, vol. 11, 2023. arXiv:2302.11867
7. M. Ye, J. Shen, G. Lin, T. Xiang, L. Shao and S. C. H. Hoi, "Deep learning for person re-identification: A survey and outlook," *IEEE TPAMI*, vol. 44, 2022. arXiv:2001.04193
8. J. Luiten, A. Ošep, P. Dendorfer, P. Torr, A. Geiger, L. Leal-Taixé and B. Leibe, "HOTA: A higher order metric for evaluating multi-object tracking," *International Journal of Computer Vision*, 2020. arXiv:2009.07736
9. K. Bernardin and R. Stiefelhagen, "Evaluating multiple object tracking performance: The CLEAR MOT metrics," *EURASIP Journal on Image and Video Processing*, 2008. doi

10. E. Ristani, F. Solera, R. S. Zou, R. Cucchiara and C. Tomasi, “Performance measures and a data set for multi-target, multi-camera tracking,” *ECCV Workshop on Benchmarking Multi-Target Tracking*, 2016. arXiv:1609.01775
11. K. Peng, A. Mathur and A. Narayanan, “Mitigating dataset harms requires stewardship: Lessons from 1000 papers,” arXiv:2108.02922, 2021. arXiv
12. L. Zheng, L. Shen, L. Tian, S. Wang, J. Wang and Q. Tian, “Scalable person re-identification: A benchmark,” *ICCV*, 2015. doi
13. L. Wei, S. Zhang, W. Gao and Q. Tian, “Person transfer GAN to bridge domain gap for person re-identification,” *CVPR*, 2018. arXiv:1711.08565
14. H. Luo, Y. Gu, X. Liao, S. Lai and W. Jiang, “Bag of tricks and a strong baseline for deep person re-identification,” *CVPR Workshops*, 2019. arXiv:1903.07071
15. M. Wieczorek, B. Rychalska and J. Dąbrowski, “On the unreasonable effectiveness of centroids in image retrieval,” arXiv:2104.13643, 2021. arXiv
16. S. Li, L. Sun and Q. Li, “CLIP-ReID: Exploiting vision-language model for image re-identification without concrete text labels,” *AAAI*, 2023. arXiv:2211.13977
17. D. S. Bolme, J. R. Beveridge, B. A. Draper and Y. M. Lui, “Visual object tracking using adaptive correlation filters,” *CVPR*, 2010. doi
18. J. F. Henriques, R. Caseiro, P. Martins and J. Batista, “High-speed tracking with kernelized correlation filters,” *IEEE TPAMI*, vol. 37, 2015. arXiv:1404.7584
19. L. Bertinetto, J. Valmadre, J. F. Henriques, A. Vedaldi and P. H. S. Torr, “Fully-convolutional Siamese networks for object tracking,” arXiv:1606.09549, 2016 (ECCV 2016 workshops). arXiv
20. B. Li, J. Yan, W. Wu, Z. Zhu and X. Hu, “High performance visual tracking with Siamese region proposal network,” *CVPR*, 2018. doi
21. B. Ye, H. Chang, B. Ma, S. Shan and X. Chen, “Joint feature learning and relation modeling for tracking: A one-stream framework” (OSTrack), *ECCV*, 2022. arXiv:2203.11991
22. X. Chen, H. Peng, D. Wang, H. Lu and H. Hu, “SeqTrack: Sequence to sequence learning for visual object tracking,” *CVPR*, 2023 (the arXiv entry was later extended as SeqTrackv2). arXiv:2304.14394
23. N. Ravi, V. Gabeur, Y.-T. Hu, R. Hu, C. Ryali, T. Ma et al., “SAM 2: Segment anything in images and videos,” *ICLR*, 2025. arXiv:2408.00714
24. C.-Y. Yang, H.-W. Huang, W. Chai, Z. Jiang and J.-N. Hwang, “SAMURAI: Adapting segment anything model for zero-shot visual tracking with motion-aware memory,” arXiv:2411.11922, 2024. arXiv
25. N. Carion, L. Gustafson, Y.-T. Hu et al., “SAM 3: Segment anything with concepts,” *ICLR*, 2026. arXiv:2511.16719
26. Y. Wu, J. Lim and M.-H. Yang, “Object tracking benchmark,” *IEEE TPAMI*, vol. 37, 2015. doi
27. H. Fan, L. Lin, F. Yang, P. Chu, G. Deng, S. Yu et al., “LaSOT: A high-quality benchmark for large-scale single object tracking,” *CVPR*, 2019. arXiv:1809.07845
28. L. Huang, X. Zhao and K. Huang, “GOT-10k: A large high-diversity benchmark for generic object tracking in the wild,” *IEEE TPAMI*, vol. 43, 2021. arXiv:1810.11981
29. M. Müller, A. Bibi, S. Giancola, S. Al-Subaihi and B. Ghanem, “TrackingNet: A large-scale dataset and benchmark for object tracking in the wild,” arXiv:1803.10794, 2018 (ECCV 2018). arXiv
30. R. E. Kalman, “A new approach to linear filtering and prediction problems,” *Journal of Basic*

*Engineering*, vol. 82, no. 1, pp. 35–45, 1960. doi
31. H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955. doi
32. A. Bewley, Z. Ge, L. Ott, F. Ramos and B. Upcroft, "Simple online and realtime tracking," *ICIP*, 2016. arXiv:1602.00763
33. N. Wojke, A. Bewley and D. Paulus, "Simple online and realtime tracking with a deep association metric," *ICIP*, 2017. doi
34. Y. Zhang, P. Sun, Y. Jiang, D. Yu, F. Weng, Z. Yuan, P. Luo, W. Liu and X. Wang, "ByteTrack: Multi-object tracking by associating every detection box," *ECCV*, 2022. arXiv:2110.06864
35. J. Cao, J. Pang, X. Weng, R. Khirodkar and K. Kitani, "Observation-centric SORT: Rethinking SORT for robust multi-object tracking," *CVPR*, 2023. arXiv:2203.14360
36. N. Aharon, R. Orfaig and B.-Z. Bobrovsky, "BoT-SORT: Robust associations multi-pedestrian tracking," arXiv:2206.14651, 2022. arXiv
37. Z. Wang, L. Zheng, Y. Liu, Y. Li and S. Wang, "Towards real-time multi-object tracking" (JDE), *ECCV*, 2020. arXiv:1909.12605
38. Y. Zhang, C. Wang, X. Wang, W. Zeng and W. Liu, "FairMOT: On the fairness of detection and re-identification in multiple object tracking," *IJCV*, 2021. arXiv:2004.01888
39. P. Bergmann, T. Meinhardt and L. Leal-Taixé, "Tracking without bells and whistles" (Tracktor), *ICCV*, 2019. arXiv:1903.05625
40. F. Zeng, B. Dong, Y. Zhang, T. Wang, X. Zhang and Y. Wei, "MOTR: End-to-end multiple-object tracking with transformer," *ECCV*, 2022. arXiv:2105.03247
41. Y. Zhang, T. Wang and X. Zhang, "MOTRv2: Bootstrapping end-to-end multi-object tracking by pretrained object detectors," *CVPR*, 2023. arXiv:2211.09791
42. R. Gao, J. Qi and L. Wang, "Multiple object tracking as ID prediction" (MOTIP), *CVPR*, 2025. arXiv:2403.16848
43. S. Li, T. Fischer, L. Ke, H. Ding, M. Danelljan and F. Yu, "OVTrack: Open-vocabulary multiple object tracking," *CVPR*, 2023. arXiv:2304.08408
44. A. Milan, L. Leal-Taixé, I. Reid, S. Roth and K. Schindler, "MOT16: A benchmark for multi-object tracking," arXiv:1603.00831, 2016 (MOT16/MOT17). arXiv
45. P. Dendorfer, H. Rezatofighi, A. Milan, J. Shi, D. Cremers, I. Reid, S. Roth, K. Schindler and L. Leal-Taixé, "MOT20: A benchmark for multi object tracking in crowded scenes," arXiv:2003.09003, 2020. arXiv
46. P. Sun, J. Cao, Y. Jiang, Z. Yuan, S. Bai, K. Kitani and P. Luo, "DanceTrack: Multi-object tracking in uniform appearance and diverse motion," *CVPR*, 2022. arXiv:2111.14690
47. Y. Cui, C. Zeng, X. Zhao, Y. Yang, G. Wu and L. Wang, "SportsMOT: A large multi-object tracking dataset in multiple sports scenes," *ICCV*, 2023. arXiv:2304.05170
48. F. Yu, H. Chen, X. Wang, W. Xian, Y. Chen, F. Liu, V. Madhavan and T. Darrell, "BDD100K: A diverse driving dataset for heterogeneous multitask learning," *CVPR*, 2020. arXiv:1805.04687
49. C. Doersch, Y. Yang, M. Vecerik, D. Gokay, A. Gupta, Y. Aytar et al., "TAPIR: Tracking any point with per-frame initialization and temporal refinement," *ICCV*, 2023. arXiv:2306.08637
50. N. Karaev, I. Rocco, B. Graham, N. Neverova, A. Vedaldi and C. Rupprecht, "CoTracker: It is better to track together," *ECCV*, 2024. arXiv:2307.07635
51. N. Karaev, I. Makarov, J. Wang, N. Neverova, A. Vedaldi and C. Rupprecht, "CoTracker3: Simpler and better point tracking by pseudo-labelling real videos," arXiv:2410.11831, 2024. arXiv

52. Z. Tang, S. Wang, D. C. Anastasiu et al., “The 9th AI City Challenge,” *ICCV Workshops*, 2025. arXiv:2508.13564
53. VOT Challenge organizers, “The Visual Object Tracking (VOT) challenge series,” website, 2013–present. votchallenge.net
54. R. Jonker and A. Volgenant, “A shortest augmenting path algorithm for dense and sparse linear assignment problems,” *Computing*, vol. 38, no. 4, pp. 325–340, 1987. doi
55. D. F. Crouse, “On implementing 2D rectangular assignment algorithms,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 52, no. 4, pp. 1679–1696, 2016. doi

# DLCV 08 • Video Understanding and Action Recognition

How deep networks recognize what happens in a video: the tasks (clip classification, temporal and spatio-temporal action localization, retrieval and question answering), why time is hard to learn, two-stream, 3D, (2+1)D, SlowFast, TSN/TSM and video-transformer architectures, masked video pretraining, evaluation protocols, datasets and failure modes such as scene bias.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/video-understanding/

## What you'll learn

- Match video tasks to clip, interval, person-level or language outputs.
- Distinguish temporal reasoning from appearance shortcuts.
- Compare optical-flow streams, temporal convolutions, sampling and attention.
- Understand masked video pretraining and video–language transfer.
- Control frames, views, resolution and pretraining when evaluating performance and cost.

## The big picture

An image classifier looks at one picture and names what is in it. A video model must also understand what *happens*: who does what, in which order, for how long. The extra input is a time axis, and with it come two problems that images do not have. First, time is expensive: a 10-second clip at 30 frames per second is 300 images. Second, time is easy to ignore: a network can often guess "playing tennis" from the court alone, without ever looking at the swing.

> **Explain like I'm five**
>
> Look at one photo of a door. Is someone opening it or closing it? You cannot tell. Now flip through a few photos taken one after another, like a flip-book, and the answer is obvious. A video model has to learn to read the flip-book, not just one page. The tricky part is that it can often cheat: if every photo of "swimming" shows a pool, it can just look for blue water and never learn what swimming looks like.

Video understanding matters because most visual data is video: phone recordings, surveillance and traffic cameras, sports broadcasts, robots, cars and medical procedures. Recognizing actions is a building block for searching video archives, analyzing sports, monitoring safety, assisting people at home and controlling robots. The field also produced ideas that spread to the rest of deep learning, such as inflating a pretrained 2D network into 3D, factorizing space and time, and masked pretraining at very high masking ratios.

This tutorial assumes image classification, its training and its evaluation from DLCV 02. It reuses detection metrics from DLCV 03 and attention from ML 05. Five surveys frame the field: Zhu et al. on deep action recognition [1], Kong and Fu on recognition and prediction [2], Selva et al. on

video transformers [3], Madan et al. on video foundation models [4] and Tang et al. on video understanding with large language models [5]. We return to them in the Modern view.

**Related chapters.** Following objects over time is DLCV 07 ·Object Tracking; body keypoints, which skeleton-based recognition uses as input, are DLCV 05; optical flow is estimated in DLCV 11A; *generating* video with spatiotemporal attention and video autoencoders is DLCV 12A; and predicting the future of a scene for planning is DLCV 13 ·World Models.

## Video understanding tasks

**Plain version.** Video outputs may be a clip label, action intervals, person-level detections or a language answer. Define the required output before choosing a model.

**Precise version.** A video is a tensor $V \in \mathbb{R}^{T\times H\times W\times 3}$, where $T$ is the number of frames, $H \times W$ the frame size and 3 the color channels. A model rarely sees all of $V$. It sees a **clip** $x \in \mathbb{R}^{T'\times H'\times W'\times 3}$ of $T' \ll T$ sampled frames, resized and cropped to $H' \times W'$. Figure 1 shows the four task families.

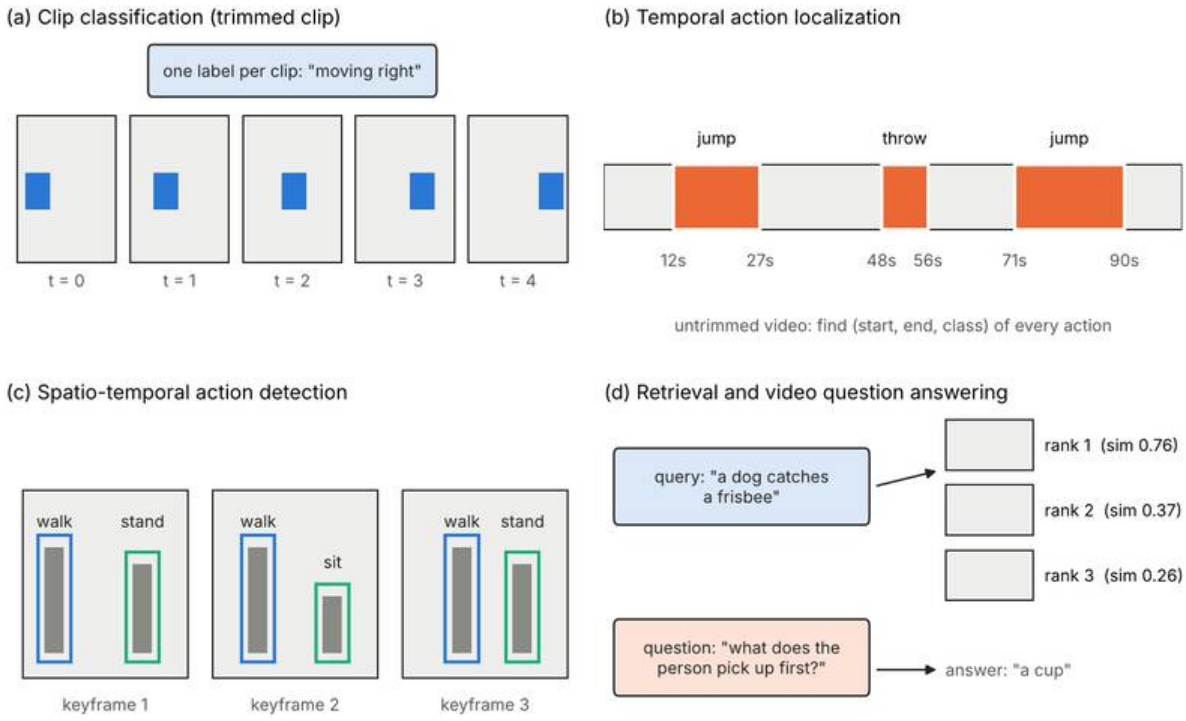



*Figure 1 — The main video tasks: (a) one label per trimmed clip; (b) start, end and class of each action in an untrimmed video; (c) a box and action labels for every person in each keyframe; (d) ranking videos for a text query, or answering a question about a video.*

1. **Clip (action) classification.** Given a *trimmed* clip that contains one action, predict its class $y \in \{1, \dots, K\}$. Trained with cross-entropy and evaluated with top-1 and top-5 accuracy, exactly

as in DLCV 02. Some datasets are **multi-label** (several actions at once) and are evaluated with mean average precision (mAP) over classes.
2. **Temporal action localization** (also called temporal action detection). Given an *untrimmed* video, output a set $\{(s_i, e_i, c_i, p_i)\}$ of segments with start time $s_i$, end time $e_i$, class $c_i$ and confidence $p_i$. It is object detection along one dimension, and it is evaluated with mAP at several **temporal IoU** thresholds.
3. **Spatio-temporal action detection.** Find *who* does *what*, *where* and *when*: a person box in each annotated keyframe, with one or more action labels per box. It is evaluated with frame-level mAP at a box-IoU threshold.
4. **Video–language tasks. Text-to-video retrieval** ranks a collection of videos by their similarity to a sentence, evaluated by Recall@$k$ (the fraction of queries whose correct video is among the top $k$). **Video question answering** produces a short answer, often chosen from multiple choices and scored by accuracy.

Neighboring problems are covered elsewhere. Linking the same object across frames is tracking (DLCV 07). Predicting what happens next is action anticipation or world modeling (DLCV 13). Labeling every pixel of every frame is video segmentation.

## Why video is hard

**Plain version.** Three things make video harder than images: the label often depends on *order* and *motion*, the datasets let models skip that work, and the input is huge.

### Time matters, but the data does not always force you to use it

Some actions can only be told apart by their motion: "opening a door" versus "closing a door", "pushing something left" versus "pushing something right", "sitting down" versus "standing up". A single frame of these is ambiguous. Many other actions are recognizable from one frame because of the **scene** and the **objects**: a swimming pool, a horse, a guitar. Datasets collected from the web contain many actions of the second kind.

This was visible early. Karpathy et al. trained CNNs on a dataset of 1 million YouTube videos in 487 sports classes and compared several ways of fusing information over time; their best spatio-temporal network improved only modestly over a single-frame model [6]. Later studies measured the problem directly. Li et al. defined the **representation bias** of a dataset and showed that common action datasets are biased toward static cues (objects, scenes and people); they also built Diving48, a dataset of competitive diving with 48 fine-grained dive classes, to reduce that bias [7]. Sevilla-Lara et al. asked people to recognize actions from videos whose frames had been shuffled, and kept only the classes that became unrecognizable; this "temporal" subset is where a model's ability to use time actually shows [8]. Something-Something was built to capture detailed physical aspects of actions rather than high-level concepts: its 174 classes are caption templates such as "pushing something from left to right" [9], which the VideoMAE authors describe as motion-centric [10].

The general phenomenon, a network solving the training task with the easiest cue available, is the **shortcut learning** of DLCV 02. In video the easiest cue is usually the scene.

### A dataset where one frame says nothing

The code below makes this concrete with synthetic clips. A small square moves left or right; the label is the direction. The square's position in the middle frame has the same distribution for both labels, so **the middle frame alone carries no information about the label**. We then add a "scene": a background brightness that agrees with the label in 90% of the training clips. Three logistic-regression classifiers see (1) the pixels of the middle frame, (2) a motion feature, and (3) both.

The motion feature is the classical brightness-constancy estimate (see DLCV 11A). If a pattern shifts by $u$ pixels per frame, then $I_{t+1}(x) \approx I_t(x-u)$, so the temporal difference is $I_{t+1} - I_t \approx -u\,\partial_x I_t$. A least-squares fit of $u$ over the whole frame gives

$$\hat{u}_t = -\frac{\sum_{x,y} \left(I_{t+1}(x,y) - I_t(x,y)\right) \partial_x I_t(x,y)}{\sum_{x,y} \left(\partial_x I_t(x,y)\right)^2},$$

where $I_t(x,y)$ is the brightness of frame $t$ at column $x$ and row $y$, $\partial_x I_t$ its horizontal derivative, and $\hat{u}_t$ the estimated horizontal motion between frames $t$ and $t+1$ (positive means rightward). It is a product of a temporal derivative filter and a spatial derivative filter, which is exactly the kind of pattern a 3D convolution can learn.

```
import numpy as np
from sklearn.linear_model import LogisticRegression

def make_clips(n, T=8, S=20, noise=0.5, scene_bias=0.5, seed=0):
    """A 3x3 square moves 1 px/frame left (y=0) or right (y=1) on a noisy background.
    scene_bias = P(background brightness agrees with the label); 0.5 = no bias."""
    rng = np.random.default_rng(seed)
    y = rng.integers(0, 2, n)
    X = noise * rng.standard_normal((n, T, S, S))
    for i in range(n):
        step = 1 if y[i] == 1 else -1
        c_mid = rng.integers(T // 2, S - 3 - T // 2 + 1)   # middle-frame position: same for
        ↪ both labels
        r = rng.integers(0, S - 3 + 1)
        agree = rng.random() < scene_bias
        X[i] += 0.3 * (y[i] if agree else 1 - y[i])       # the "scene": background
        ↪ brightness
        for t in range(T):
            c = c_mid + step * (t - T // 2)
            X[i, t, r:r + 3, c:c + 3] += 1.0
    return X, y

def frame_features(X):     # appearance only: the pixels of the middle frame
    return X[:, X.shape[1] // 2].reshape(len(X), -1)
```

```
def motion_features(X):    # temporal difference times horizontal gradient, per frame pair
    dt = X[:, 1:] - X[:, :-1]
    dx = np.gradient(X[:, :-1], axis=3)
    u = -(dt * dx).sum(axis=(2, 3)) / ((dx ** 2).sum(axis=(2, 3)) + 1e-6)
    return u                                              # one motion estimate per frame pair

data = {"train": make_clips(2000, scene_bias=0.9, seed=1),  # scene agrees with the label 90%
        "same": make_clips(1000, scene_bias=0.9, seed=2),   # test: same distribution
        "no cue": make_clips(1000, scene_bias=0.5, seed=3), # test: scene says nothing
        "shift": make_clips(1000, scene_bias=0.1, seed=4)}  # test: scene misleads 90%
for name, f in [("one frame", frame_features), ("motion", motion_features),
                ("frame+motion", lambda X: np.c_[frame_features(X), motion_features(X)])]:
    clf = LogisticRegression(C=0.1, max_iter=5000).fit(f(data["train"][0]), data["train"][1])
    acc = {k: clf.score(f(X), y) for k, (X, y) in data.items() if k != "train"}
    print(f"{name:13s}", "  ".join(f"{k} {v:.3f}" for k, v in acc.items()))
```

```
one frame     same 0.867  no cue 0.495  shift 0.131
motion        same 0.945  no cue 0.929  shift 0.941
frame+motion  same 0.889  no cue 0.522  shift 0.163
```

(All numbers in this block and in Figure 2 come from this toy run.) On test data from the training distribution, the single-frame model looks good (0.867), because the scene predicts the label 90% of the time. Remove the scene cue and it drops to chance (0.495); make the scene misleading and it falls to 0.131. The motion model is about 0.93–0.94 in every case, because it uses the cue that actually defines the label. The combined model is the warning: it *has* the motion feature, yet it learned to rely on the hundreds of brightness pixels, and it fails under the shift (0.163). How strongly it leans on the scene depends on feature scaling and regularization (Exercise 1), but a benchmark drawn from the training distribution would never reveal the problem.

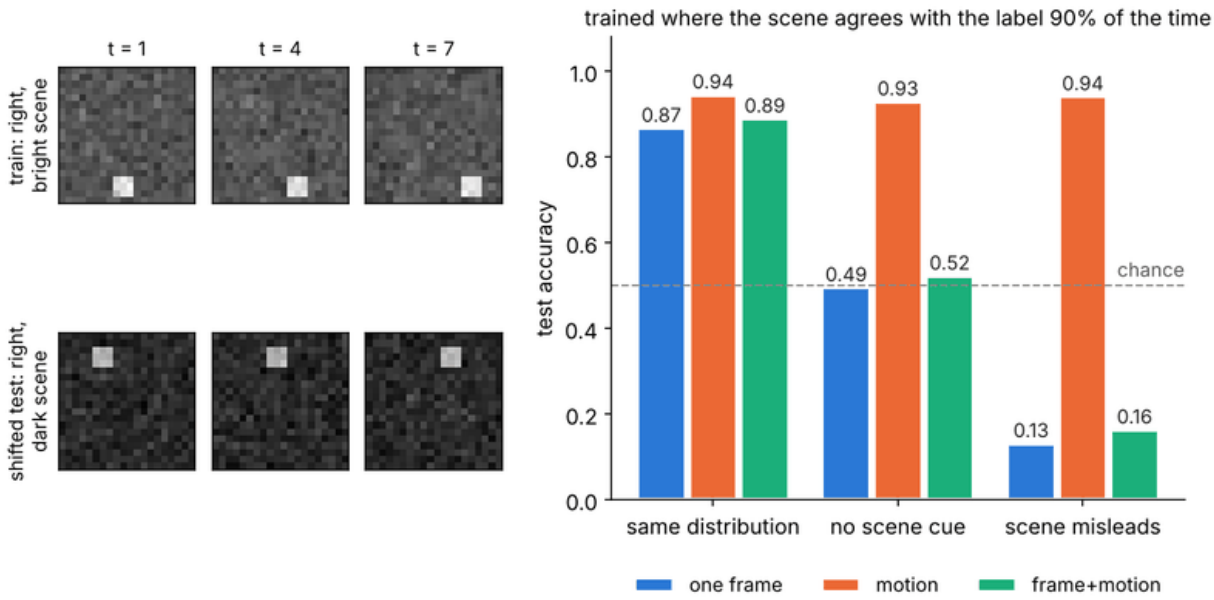


*Figure 2 — The scene-bias toy. Left: frames of a training clip (bright scene) and a shifted test clip (dark scene); shown with less noise than in the experiment so the square is visible. Right: accuracy of the three classifiers on the three test sets. Only the motion model survives when the scene stops agreeing with the label.*

### Compute and memory

A clip of $T'$ frames costs roughly $T'$ times as much as one image for a per-frame network, and more for models that mix frames. For a convolution with $C_{\text{in}}$ input channels, $C_{\text{out}}$ output channels and a kernel of $k_t \times k_h \times k_w$ (time × height × width), stride 1 and "same" padding on a $T \times H \times W$ input,

$$\text{params} = C_{\text{in}}\, C_{\text{out}}\, k_t k_h k_w, \qquad \text{MACs} = \text{params} \times THW,$$

where MACs counts multiply–accumulate operations (one multiplication plus one addition each). A $3\times3\times3$ kernel therefore has 3 times the parameters and 3 times the work of a $3\times3$ kernel applied to the same frames. For a transformer the problem is the number of tokens $N$: self-attention compares $N^2$ query–key pairs, and $N$ grows linearly with the number of frames. The code in the next section puts numbers on both.

## How to add time to an image network

**Plain version.** Video models combine frame features through pooling, optical flow, temporal convolutions, multiple frame rates or space–time attention.

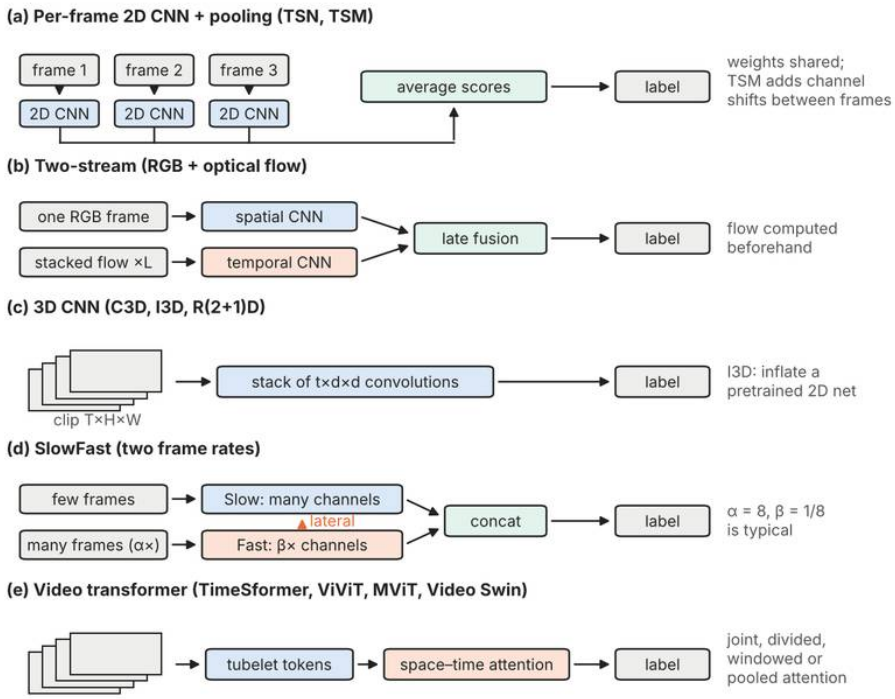



*Figure 3 — Five ways to add time to an image network. Families (a)–(d) are convolutional; (e) is attention-based. Real systems combine them, for example two-stream I3D or a transformer that uses SlowFast-style multiscale features.*

### Per-frame networks and late fusion

The simplest video model applies an image classifier $f$ to $K$ frames $x_{t_1}, \dots, x_{t_K}$ and averages the class scores:

$$\hat{y} = \operatorname{softmax}\Big(\frac{1}{K}\sum_{k=1}^{K} f(x_{t_k})\Big),$$

where $f(x) \in \mathbb{R}^C$ is the vector of class scores (logits) for one frame and $C$ the number of classes. This is order-blind: shuffling the frames gives the same answer, so it cannot tell "opening" from "closing". It is still a strong baseline on scene-heavy datasets, and it is cheap and can start from any pretrained image backbone (DLCV 02). A recurrent network such as an LSTM [11] over per-frame features adds order; this design is discussed in ML 04.

### Two-stream networks: appearance plus optical flow

**Plain version.** A two-stream model combines appearance from color frames with motion from optical-flow fields.

Simonyan and Zisserman's two-stream ConvNet [12] has a **spatial stream**, an image CNN on one RGB frame, and a **temporal stream**, a CNN whose input is the dense optical flow of $L$ consecutive frame pairs. Each flow field $d_t$ has a horizontal component $d_t^x$ and a vertical component $d_t^y$, so the stacked input for frame $\tau$ has $2L$ channels:

$$I_\tau(u,v,2k-1) = d^x_{\tau+k-1}(u,v), \qquad I_\tau(u,v,2k) = d^y_{\tau+k-1}(u,v), \qquad k = 1,\dots,L,$$

where $(u,v)$ is a pixel position and $d_t^x(u,v)$, $d_t^y(u,v)$ are the horizontal and vertical displacements of that pixel from frame $t$ to frame $t+1$. The two streams are trained separately and fused late, by averaging their softmax scores or with a linear SVM on the stacked scores [12]. The paper showed that a ConvNet on stacked flow performs well despite limited training data, and that the two streams are complementary [12].

The weakness is cost: dense optical flow must be computed (and usually stored) for every frame before training, with a classical solver or a flow network (see DLCV 09A.3 for warping with flow). The flow is also not learned for the recognition task. Temporal Segment Networks [13] and I3D [14] still used a flow stream; SlowFast (below) later learned motion end to end from raw frames, without flow, and its authors found it empirically more effective than the two-stream approach [15].

### 3D convolutions: C3D and I3D inflation

**Plain version.** Make every filter a small cube that slides over time as well as space. Then the network can learn motion detectors, such as "a bright edge that moves right", directly from the pixels.

A 3D convolution with kernel $w \in \mathbb{R}^{k_t \times k_h \times k_w}$ computes, for one input and one output channel,

$$y(t,i,j) = \sum_{a=0}^{k_t-1} \sum_{b=0}^{k_h-1} \sum_{c=0}^{k_w-1} w(a,b,c)\, x(t+a, i+b, j+c),$$

where $x(t,i,j)$ is the input at frame $t$, row $i$ and column $j$ (in practice summed over all input channels and with padding at the borders). **C3D** [16] trained a network of such layers on a large video dataset and found that a homogeneous architecture with small $3\times3\times3$ kernels in all layers was among the best; its features with a linear classifier were strong on several benchmarks.

Training 3D networks from scratch needs much more data than existed in 2015. Carreira and Zisserman solved both problems at once in 2017 [14]. They introduced **Kinetics**, a dataset of 400 human action classes with at least 400 YouTube clips per class, each about 10 seconds long [17], and the **Inflated 3D ConvNet (I3D)**: take a successful 2D image architecture and *inflate* every $N \times N$ filter and pooling kernel into $N \times N \times N$. To start from weights pretrained on ImageNet [18], they made the inflated network give the same response on a "boring video" (one image repeated in time) as the 2D network on the image: repeat each 2D filter $N$ times along time and divide by $N$ [14]. With $w_{2D}$ the pretrained filter,

$$w_{3D}(a,b,c) = \frac{1}{N}\, w_{2D}(b,c), \qquad a = 0,\dots,N-1,$$

so on a constant-in-time input the $N$ temporal taps add up to exactly the 2D response. The check below confirms it numerically.

```
import numpy as np
from scipy.ndimage import correlate

rng = np.random.default_rng(0)
w2d = rng.standard_normal((3, 3))            # a "pretrained" 2D filter
N = 3                                        # temporal size of the inflated filter
w3d = np.repeat(w2d[None] / N, N, axis=0)    # (N, 3, 3): copy N times, divide by N

img = rng.standard_normal((32, 32))
boring = np.repeat(img[None], 8, axis=0)     # an 8-frame "video" of one still image

r2d = correlate(img, w2d, mode="nearest")
r3d = correlate(boring, w3d, mode="nearest")  # 3D filtering over (t, y, x)
print("max |3D response on any frame - 2D response| =", np.abs(r3d - r2d[None]).max())
```

```
max |3D response on any frame - 2D response| = 1.7763568394002505e-15
```

The difference is at the level of floating-point rounding. Training then moves the filters away from "constant in time" as they learn motion. The paper's final model is **two-stream I3D**, one I3D on RGB and one on flow with predictions averaged, because the authors still found the flow stream valuable [14]. After pretraining on Kinetics, I3D improved considerably on the small UCF101 and HMDB51 benchmarks [14], which helped make "pretrain on Kinetics, fine-tune on the target" the standard video recipe. The same inflation idea reappears in video generators (DLCV 12A), and I3D features are the feature network behind the Fréchet Video Distance [19] (DLCV 12A).

*Online only: Quo Vadis: I3D and the Kinetics dataset (Carreira and Zisserman, 2017) — https://blog.shyandram.dev/en/learn/deep-learning/video-understanding/papers/quo-vadis-i3d-and-the-kinetics-dataset-carreira-and-zisserman-2017/*

### Factorized (2+1)D convolutions

**Plain version.** A cube-shaped filter can be split into two cheaper steps: first a flat filter that looks within each frame, then a thin filter that looks across frames at each position. With the right width in between, the split has the same number of parameters but an extra nonlinearity, and it trains more easily.

Tran et al. [20] compared several spatiotemporal convolutions inside residual networks: 2D only, 3D everywhere, mixed (3D in early layers only, or late layers only) and a factorized **(2+1)D** block. The (2+1)D block replaces $N_i$ filters of size $N_{i-1}\times t\times d\times d$ with $M_i$ spatial filters of size $N_{i-1}\times 1\times d\times d$, a ReLU, and $N_i$ temporal filters of size $M_i \times t \times 1 \times 1$. Here $N_{i-1}$ and $N_i$ are the input and output channel counts of layer $i$, $t$ and $d$ the temporal and spatial kernel sizes, and $M_i$ the width of the intermediate space, chosen so that the parameter count matches the 3D layer [20]:

$$M_i = \left\lfloor \frac{t\,d^2\,N_{i-1}\,N_i}{d^2\,N_{i-1} + t\,N_i} \right\rfloor .$$

The paper names two benefits: the extra ReLU doubles the number of nonlinearities for the same parameter count, and the factorization makes optimization easier, visible as lower *training* error than an equally sized 3D network [20]. It is the video analogue of the depthwise-separable trick in image CNNs (ML 03): split one big operation into cheaper ones along different axes.

The code computes parameters and multiply–adds for one 64-to-64-channel layer on a 16-frame clip with $56 \times 56$ feature maps, and the token count of a video transformer for comparison.

```
import math

def conv_cost(c_in, c_out, kt, kh, kw, T, H, W):
    """Parameters and multiply-adds of one convolution (stride 1, 'same' padding, no
     ↪  bias)."""
    params = c_in * c_out * kt * kh * kw
    return params, params * T * H * W

c, t, d = 64, 3, 3                          # channels in = out, temporal and spatial kernel
 ↪  size
T, H, W = 16, 56, 56                        # a 16-frame clip, 56x56 feature maps
M = math.floor(t * d * d * c * c / (d * d * c + t * c))   # R(2+1)D's matched middle width

layers = {
    "2D  (1x3x3), per frame": [conv_cost(c, c, 1, d, d, T, H, W)],
    "3D  (3x3x3)":            [conv_cost(c, c, t, d, d, T, H, W)],
    f"(2+1)D, M={M}":         [conv_cost(c, M, 1, d, d, T, H, W), conv_cost(M, c, t, 1, 1, T,
     ↪  H, W)],
}
for name, parts in layers.items():
    p = sum(x[0] for x in parts); f = sum(x[1] for x in parts)
    print(f"{name:24s} params {p:>8,d}   GMACs {f / 1e9:5.2f}")

# a video transformer's token count with 2x16x16 tubelets on a 16x224x224 clip
n_tok = (16 // 2) * (224 // 16) ** 2
print(f"tokens {n_tok}, joint attention pairs {n_tok**2:,}, "
      f"divided (time, then space) pairs {n_tok * 8 + n_tok * 196:,}")
```

```
2D  (1x3x3), per frame   params   36,864   GMACs  1.85
3D  (3x3x3)              params  110,592   GMACs  5.55
(2+1)D, M=144            params  110,592   GMACs  5.55
tokens 1568, joint attention pairs 2,458,624, divided (time, then space) pairs 319,872
```

With equal input and output widths, $M = 144$ makes the (2+1)D block match the 3D layer exactly in parameters and work; the gain is in expressiveness and optimization, not in cost. A cheaper split is possible by choosing a smaller $M$. Figure 4 draws the factorization.

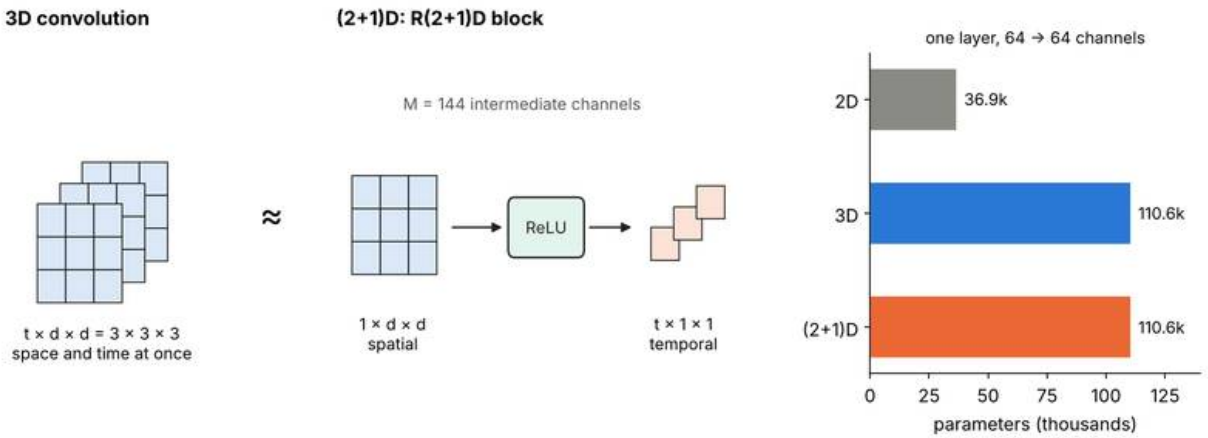



*Figure 4 — A 3D convolution and its $(2+1)D$ factorization. With the matched middle width $M$ from the formula above, both have the same parameter count; the $(2+1)D$ block adds a nonlinearity between the spatial and temporal steps.*

*Online only: Choosing the middle width of a $(2+1)D$ block — https://blog.shyandram.dev/en/learn/deep-learning/video-understanding/notes/choosing-the-middle-width-of-a-21d-block/*

### Cheap temporal modeling: TSN sampling and the temporal shift module

**Plain version.** Two tricks let a plain 2D network handle long videos cheaply. TSN looks at a few frames spread over the whole video instead of many frames from one moment. TSM lets neighboring frames swap a few feature channels inside the network, which mixes information across time at almost no cost.

**Temporal Segment Networks (TSN)** [13] divide a video into $K$ equal segments, take one short snippet from each (a random position during training), score every snippet with the same network, and combine the scores with a **segmental consensus** such as averaging before the softmax. The loss is computed on the consensus, so the whole video supervises the network, not just one short window [13]. The paper's default was $K = 3$ segments [13]. This **sparse sampling** covers long-range structure with a fixed, small number of frames (Figure 5).

```
import numpy as np

def tsn_indices(n_frames, K, train, rng=None):
    """TSN-style sparse sampling: split the video into K equal segments, take one frame
    per segment (random position when training, the segment centre when testing)."""
    edges = np.linspace(0, n_frames, K + 1)
    if train:
        return np.floor(edges[:-1] + rng.random(K) * np.diff(edges)).astype(int)
    return np.floor((edges[:-1] + edges[1:]) / 2).astype(int)

def dense_clip(n_frames, length, stride, start):
```

```
    """A dense clip: `length` frames, `stride` apart, starting at `start`."""
    return start + stride * np.arange(length)

rng = np.random.default_rng(0)
n = 300                                          # a 10-second video at 30 fps
print("TSN train:", tsn_indices(n, 8, True, rng))
print("TSN test :", tsn_indices(n, 8, False))
print("dense 8x8:", dense_clip(n, 8, 8, start=100))   # 8 frames, every 8th frame
# multi-view testing: 10 clip start positions spread uniformly over the video
starts = np.linspace(0, n - 8 * 8, 10).astype(int)
print("10 clip starts:", starts)
```

```
TSN train: [ 23  47  76 113 180 221 247 289]
TSN test : [ 18  56  93 131 168 206 243 281]
dense 8x8: [100 108 116 124 132 140 148 156]
10 clip starts: [  0  26  52  78 104 131 157 183 209 236]
```

The eight sparse frames span the whole video; the dense clip covers about 2 seconds in detail. (Taking segment centres at test time is a common choice in later implementations; the original TSN paper tested on 25 frames with 10 crops each [13].) Dense clips are what 3D networks and SlowFast consume; the last line, ten clips spread over the video, is the multi-view testing protocol discussed under evaluation below.

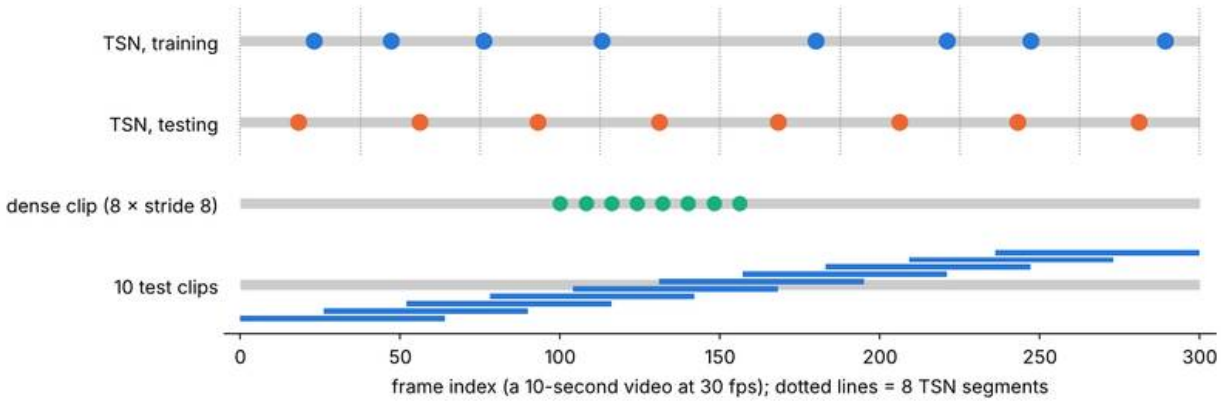



*Figure 5 — Frame sampling, from the code above: TSN-style sampling takes one frame per segment (random when training, centred when testing); a dense clip takes consecutive frames at a fixed stride; multi-view testing averages predictions over several clips spread across the video.*

The **Temporal Shift Module (TSM)** [21] adds real temporal interaction to a 2D network with zero extra parameters and zero extra multiply–adds. Inside each residual block, a fraction of the channels of the feature tensor is shifted by one frame forward in time, another fraction one frame backward, and the rest stay in place; the following 2D convolution then mixes channels that come from three different frames. Shifting most channels hurt both latency (data movement) and accuracy (less spatial information per frame), so the paper shifts 1/4 of the channels (1/8 in each direction) and places the shift on the residual branch [21]. For live video, a uni-directional version shifts only from past to current frames [21].

*Online only: The temporal shift as a sparse temporal convolution — https://blog.shyandram.dev/en/learn/deep-learning/video-understanding/notes/the-temporal-shift-as-a-sparse-temporal-convolution/*

### SlowFast: two frame rates

**Plain version.** What things are changes slowly; how they move can change fast. So use two pathways: a heavy one that looks at few frames to understand the scene, and a light one that looks at many frames to catch the motion.

**SlowFast networks** [15] have a **Slow pathway** that samples $T$ frames with a large temporal stride $\tau$, and a **Fast pathway** that samples $\alpha T$ frames with stride $\tau/\alpha$, so it is $\alpha$ times denser in time. The Fast pathway has only a fraction $\beta$ of the Slow pathway's channels. Typical values are $\alpha = 8$ and $\beta = 1/8$ [15]. Because the cost of a layer grows roughly with the square of its channel count, the Fast pathway takes only about 20% of the total computation in the paper's instantiations [15]. **Lateral connections** feed Fast features into the Slow pathway at every stage; to match shapes they convert the $\alpha T$ Fast frames to $T$, for example with a time-strided 3D convolution, the paper's default [15].

Two design details carry a general lesson. The Slow pathway uses temporal kernels larger than 1 only in its last two stages, because the authors found that temporal convolutions in early layers *hurt* when the frame stride is large: fast-moving content has little correlation across a large time step unless the spatial receptive field is also large [15]. And SlowFast uses **no optical flow**: it learns motion end to end from raw frames [15]. It reported strong results on Kinetics, Charades and AVA classification and detection [15].

*Online only: SlowFast networks (Feichtenhofer et al., 2019) — https://blog.shyandram.dev/en/learn/deep-learning/video-understanding/papers/slowfast-networks-feichtenhofer-et-al-2019/*

Two related ideas complete the CNN family. **Non-local blocks** [22] compute each position's response as a weighted sum over *all* space–time positions, an attention layer inserted into a 3D CNN to capture long-range dependencies. **X3D** [23] starts from a tiny 2D image network and expands it one axis at a time (in space, time, width and depth) to find accurate networks at low cost; one surprising finding was that networks with high spatiotemporal resolution can be extremely thin [23].

### Video transformers: TimeSformer, ViViT, MViT and Video Swin

**Plain version.** Cut the clip into small space–time patches, turn each into a token, and let a transformer decide which patches should look at which. The hard part is that a clip has thousands of tokens, so the designs differ mainly in *which* tokens each token may attend to.

A Vision Transformer [24] (ML 05) turns an image into patches. A video transformer does the same in space–time. With **tubelet embedding** [25], non-overlapping tubes of $t \times p \times p$ pixels are linearly projected (equivalently, a 3D convolution with stride equal to its kernel), so a $T \times H \times W$ clip becomes

$$N = \frac{T}{t} \cdot \frac{H}{p} \cdot \frac{W}{p}$$

tokens, where $t$ is the tubelet length in frames and $p$ the patch side in pixels. VideoMAE's ViT-Base, for example, takes a $16 \times 224 \times 224$ clip with $2 \times 16 \times 16$ cubes, giving $8 \times 196 = 1568$ tokens [10]. **Joint** space–time attention compares all $N^2$ pairs, which the code above counted as about 2.46 million per layer. Four designs tame this:

- **TimeSformer** [26] compared several self-attention schemes and found **divided attention** best among those it tried: within each block, a token first attends to the tokens at the same spatial position in other frames (temporal attention) and then to the tokens of its own frame (spatial attention) [26]. In the configuration above that is about 0.32 million pairs instead of 2.46 million. It can also process clips over one minute long [26]. The cost ratio of joint versus divided attention is derived in DLCV 12A, which also explains why video *generators* often prefer full attention.
- **ViViT** [25] proposed four models: joint spatio-temporal attention; a **factorised encoder** (a spatial transformer per frame, then a temporal transformer over the frame embeddings, a form of late fusion); factorised self-attention inside each block; and factorised dot-product attention, where half of the heads attend over space and half over time. It also showed how to regularize these models and initialize them from pretrained image models so they can be trained on comparatively small datasets [25].
- **MViT** [27] brings back the multiscale hierarchy of CNNs: stages start at high spatiotemporal resolution with few channels and progressively expand channels while reducing resolution, using **pooling attention**, which pools the query, key and value sequences inside attention [27].
- **Video Swin Transformer** [28] argues for **locality**: attention is computed only inside local 3D windows, which are shifted between consecutive layers so information can cross window borders. It extends the image Swin Transformer [29] (DLCV 02) and starts from pretrained image weights, giving a better speed–accuracy trade-off than global attention [28].

Attention is not the only way to model long clips. **VideoMamba** [30] adapts the Mamba state-space model [31], whose cost grows linearly with the number of tokens, to video; the underlying selective scan is explained in ML 04.

## Learning video representations without labels

**Plain version.** Labeling video is slow and expensive, but unlabeled video is plentiful. So hide most of a clip and train the network to fill it in; to do that well, it must learn how things look and how they move.

**VideoMAE** [10] applies masked autoencoding (MAE [32], see ML 06) to video. It masks space–time cubes, encodes only the visible ones with a ViT, and reconstructs the pixels of the masked ones with a light decoder. Two things are specific to video. First, consecutive frames are highly redundant, so the masking ratio can be extremely high: 90% to 95% still works well [10]. Second, if masks are drawn independently per frame, a masked patch can often be copied from the same position in a neighboring frame, an **information leakage** that makes the task too easy. VideoMAE

therefore uses **tube masking**: the same spatial mask in every frame [10]. The snippet measures the leakage.

```
import numpy as np

rng = np.random.default_rng(0)
T, P, ratio = 8, 196, 0.9              # 8 token frames of 14x14 patches, 90% masked

def leak(mask):
    """Fraction of masked tokens whose same-position token in a neighbouring frame is
     ↪  visible."""
    vis = ~mask
    nb = np.zeros_like(mask)
    nb[1:] |= vis[:-1]; nb[:-1] |= vis[1:]
    return (mask & nb).sum() / mask.sum()

random_mask = rng.random((T, P)) < ratio                  # independent per token
tube_mask = np.repeat((rng.random(P) < ratio)[None], T, 0) # same spatial mask in every frame
print(f"random masking: {random_mask.mean():.2f} masked, leak {leak(random_mask):.2f}")
print(f"tube masking  : {tube_mask.mean():.2f} masked, leak {leak(tube_mask):.2f}")
```

```
random masking: 0.90 masked, leak 0.16
tube masking  : 0.89 masked, leak 0.00
```

With independent masks, about one masked token in six has a visible twin in an adjacent frame (for interior frames the expected value is $1 - 0.9^2 = 0.19$; the first and last frames have only one neighbor). Tube masking removes that shortcut entirely, which is the point: a pretext task is only useful if its shortcuts are closed. VideoMAE also reported strong results on small datasets of a few thousand videos without extra data, found that data quality matters more than data quantity for this kind of pretraining, and noted that the domain shift between pretraining and target data is an important issue [10]. **VideoMAE V2** [33] scaled the approach to a billion-parameter ViT with **dual masking**, which also masks the decoder's input to save computation.

A second line predicts *features* instead of pixels. **V-JEPA** [34] trains video models only with a feature-prediction objective, without pretrained image encoders, text, negative examples or reconstruction, on about 2 million videos, and evaluates them with a frozen backbone on both motion- and appearance-based tasks [34]. Its successor V-JEPA 2 [35] and the JEPA idea are covered in ML 06 and DLCV 13.

## Video and language, briefly

**Plain version.** Instead of a fixed list of action names, describe videos with free text. A model that places videos and sentences in the same space can search videos by description; a model that feeds video tokens into a language model can answer questions about them.

**Retrieval.** CLIP-style image–text models [36] (ML 10) transfer to video: **CLIP4Clip** [37] transfers CLIP to text–video retrieval end to end and studies whether image features are enough, how

post-pretraining on video–text data helps and how to model the dependency between frames, on benchmarks such as **MSR-VTT**, a collection of 10K web video clips with 200K clip–sentence pairs [38]. **Video foundation models** such as **InternVideo2** [39] combine masked video modeling, video–text contrastive learning and next-token prediction in one progressive training scheme, with a video encoder of up to 6 billion parameters [39].

**Question answering.** Multimodal large language models (ML 10) accept sampled frames as visual tokens. Long videos stress them: **Video-MME** [40] evaluates them on 900 manually annotated videos from 11 seconds to 1 hour long (2,700 question–answer pairs), with subtitles and audio as optional inputs, and found that handling longer sequences remained a weakness [40]. A practical caution, by analogy with scene bias: check whether a question can be answered from a single frame, or from the question text alone, before crediting a model with temporal understanding; a single-frame baseline is a cheap sanity check.

## Localizing actions in time and space

**Plain version.** Real videos are not trimmed. A cooking video contains "cut", "stir" and "pour" with long boring stretches in between. Temporal localization finds where each action starts and ends; spatio-temporal detection also says which person is doing it.

### Temporal action localization

The task mirrors object detection along the time axis (DLCV 03), and so do its methods. A clip-level backbone (often frozen, such as I3D or SlowFast features) produces a feature sequence along time; a localization head turns it into scored segments.

- **Proposal-based (two-stage).** First generate class-agnostic candidate segments, then classify them. **BMN** [41] represents a proposal as a pair of start and end boundaries and scores all densely distributed pairs at once in a **boundary-matching confidence map**; combined with an action classifier it gives detections [41].
- **Anchor-free, single-stage. ActionFormer** [42] classifies *every moment* in time and regresses the distances to the action's start and end, with no proposals and no predefined anchor windows. It uses a multiscale transformer with local self-attention and a light decoder [42], the temporal counterpart of the anchor-free detectors of DLCV 03. Duplicate segments are removed with non-maximum suppression in time, as for boxes (ActionFormer uses a soft variant [42]).

**Temporal IoU and mAP.** For a predicted segment $\hat{g} = [\hat{s}, \hat{e}]$ and a ground-truth segment $g = [s, e]$,

$$\mathrm{tIoU}(\hat{g}, g) = \frac{\max\left(0, \min(\hat{e}, e) - \max(\hat{s}, s)\right)}{(\hat{e} - \hat{s}) + (e - s) - \max\left(0, \min(\hat{e}, e) - \max(\hat{s}, s)\right)},$$

where the numerator is the length of the overlap and the denominator the length of the union. For one class and threshold $\theta$, predictions are sorted by score; a prediction is a true positive if its tIoU with a not-yet-matched ground truth of that class is at least $\theta$. Average precision (AP) is the

area under the resulting precision–recall curve, mAP averages AP over classes, and an **average mAP** averages over several thresholds. Two conventions are common: THUMOS14 averages over tIoU $\{0.3, 0.4, \dots, 0.7\}$ and ActivityNet 1.3 over $\{0.5, 0.55, \dots, 0.95\}$ [42]. The code computes AP for one class on synthetic detections (85% of actions found, with boundary errors, plus random false alarms).

```
import numpy as np

def tiou(seg, segs):
    """Temporal IoU of one segment [s, e] with an array of segments (n, 2)."""
    inter = np.clip(np.minimum(seg[1], segs[:, 1]) - np.maximum(seg[0], segs[:, 0]), 0, None)
    union = (seg[1] - seg[0]) + (segs[:, 1] - segs[:, 0]) - inter
    return inter / union

def average_precision(preds, gts, thr):
    """preds: list of (video, start, end, score); gts: dict video -> (m, 2) array.
    Greedy matching in score order; each ground truth can be matched once."""
    n_gt = sum(len(g) for g in gts.values())
    used = {v: np.zeros(len(g), bool) for v, g in gts.items()}
    tp = []
    for v, s, e, _ in sorted(preds, key=lambda p: -p[3]):
        ious = tiou(np.array([s, e]), gts[v])
        ious[used[v]] = -1                      # already matched ground truths are off
        ↪ limits
        j = int(np.argmax(ious))
        hit = ious[j] >= thr
        if hit:
            used[v][j] = True
        tp.append(hit)
    tp = np.array(tp, float)
    recall = np.cumsum(tp) / n_gt
    precision = np.cumsum(tp) / np.arange(1, len(tp) + 1)
    precision = np.maximum.accumulate(precision[::-1])[::-1]   # monotone envelope
    return np.sum(np.diff(np.r_[0, recall]) * precision)

rng = np.random.default_rng(0)
gts, preds = {}, []
for v in range(20):                             # 20 untrimmed videos, 1-3 actions each
    starts = np.sort(rng.uniform(0, 100, rng.integers(1, 4)))
    g = np.c_[starts, starts + rng.uniform(4, 15, len(starts))]
    gts[v] = g
    for s, e in g:
        if rng.random() < 0.85:                 # the detector finds 85% of actions...
            j = rng.normal(0, 0.2 * (e - s), 2)  # ...with boundary errors
            preds.append((v, s + j[0], e + j[1], rng.uniform(0.5, 1.0)))
    for _ in range(rng.integers(0, 3)):         # plus some false alarms
        s = rng.uniform(0, 100)
        preds.append((v, s, s + rng.uniform(3, 12), rng.uniform(0.0, 0.7)))

thrs = [0.3, 0.4, 0.5, 0.6, 0.7]
aps = [average_precision(preds, gts, t) for t in thrs]
print("AP at tIoU:", "  ".join(f"{t}: {a:.3f}" for t, a in zip(thrs, aps)))
print(f"average over [0.3:0.1:0.7]: {np.mean(aps):.3f}")
```

```
AP at tIoU: 0.3: 0.731  0.4: 0.696  0.5: 0.696  0.6: 0.604  0.7: 0.284
average over [0.3:0.1:0.7]: 0.602
```

(Toy numbers.) Up to tIoU 0.5 the AP is limited mainly by the 15% missed actions and the false alarms; at 0.7 it collapses because boundary errors of about 20% of the action length are no longer good enough. Boundaries of actions are genuinely ambiguous (when does "pour" start?), which is why strict thresholds are hard even for people, and why benchmarks report several.

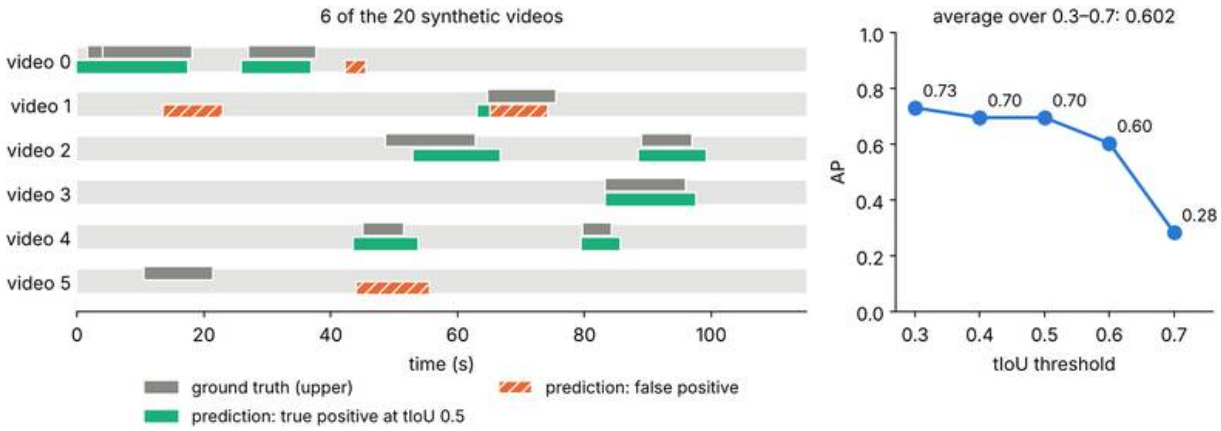



*Figure 6 — The temporal-AP toy from the code above. Left: ground truth (gray) and predictions on six of the 20 videos, colored by whether they match at tIoU 0.5. Right: AP falls as the tIoU threshold becomes stricter.*

*Online only: Average precision for temporal localization, step by step — https://blog.shyandram.dev/en/learn/deep-learning/video-understanding/notes/average-precision-for-temporal-localization-step-by-step/*

### Spatio-temporal action detection

Here the model must say which person does which action. In **AVA** [43], every person in keyframes sampled at 1 Hz from 15-minute movie clips gets a box and one or more labels from 80 atomic actions (14 pose classes, 49 person–object and 17 person–person interactions) [43]. A common pipeline is detection plus video classification, as in SlowFast's AVA detector [15]: an off-the-shelf person detector (DLCV 03) proposes boxes on the keyframe, a video backbone computes a spatiotemporal feature map for a clip around it, the features inside each box are pooled, and a per-class sigmoid classifier gives multi-label predictions, because one person can "stand", "talk to" and "hold" at once. The metric is **frame-mAP at IoU 0.5**: AP per action class over all keyframe detections, a detection counting only if its box overlaps a ground-truth box of that class with IoU at least 0.5, averaged over classes [43]. Linking detections into tubes over time borrows the association methods of tracking (DLCV 07).

### Skeleton-based action recognition

Instead of pixels, a model can read the **pose**: the 2D or 3D coordinates of body joints over time, estimated as in DLCV 05 or captured with depth sensors, as in the NTU RGB+D dataset of 60 action classes [44]. **ST-GCN** [45] builds a graph whose nodes are the joints in every frame, with *spatial* edges along the bones and *temporal* edges connecting the same joint in consecutive frames, and applies graph convolutions over it [45]. Skeletons are compact, ignore clothing and background, and are therefore less prone to scene bias: on the **Mimetics** dataset of mimed actions without the usual objects and scenes, a pose-based model held up much better than 3D CNNs [46]. The price is that skeletons miss objects ("drinking" versus "eating") and depend on the pose estimator's errors.

*Online only: Graph convolution on a skeleton sequence (ST-GCN) — https://blog.shyandram.dev/en/learn/deep-learning/video-understanding/notes/graph-convolution-on-a-skeleton-sequence-st-gcn/*

## Training and evaluating video classifiers

**Plain version.** How you pick frames, how many views you average at test time and which dataset you report on change the numbers as much as many architecture changes. Read the protocol before comparing two papers.

### Sampling and training

Three choices define the input: the **number of frames** $T'$, the **stride** between them (frame rate), and dense versus sparse (TSN-style) sampling. Motion-centric datasets reward dense, fine-grained sampling; scene-centric datasets tolerate sparse sampling. VideoMAE, for example, uses TSN-style uniform sampling on Something-Something V2 and dense sampling on Kinetics-400 [10]. Spatial augmentation is the same as for images (random resized crops, flips), with one video-specific trap: **horizontal flipping changes the label** of direction-dependent classes such as "pushing something from left to right". Training otherwise follows DLCV 02. Initialization usually comes from an image model (inflation, or a pretrained ViT) or from a self-supervised video model.

### Multi-view testing

At test time, one clip rarely covers the whole video, so predictions are averaged over $V_t$ temporal clips and $V_s$ spatial crops:

$$\hat{p}(y \mid \text{video}) = \frac{1}{V_t V_s} \sum_{i=1}^{V_t} \sum_{j=1}^{V_s} \text{softmax}\left(f(x_{ij})\right)_y,$$

where $x_{ij}$ is the $i$-th clip cropped at the $j$-th spatial position. SlowFast, for example, samples 10 clips uniformly along the video and takes 3 crops of 256 × 256 from each, 30 views in total, and

reports cost as **GFLOPs per view × number of views**, noting that some earlier works used more than 100 views [15]. VideoMAE evaluated with 2 clips × 3 crops on Something-Something V2 and 5 × 3 on Kinetics-400 [10]. The total inference cost is the per-view cost times the number of views, so a model that looks cheap per view may be expensive in practice. Report both, and compare models only under the same protocol.

### Datasets and benchmarks

| Dataset | Task | Content (as described by its authors) |
|---|---|---|
| UCF101 [47] | clip classification | 101 action classes, over 13k clips, 27 hours of user-uploaded video |
| HMDB51 [48] | clip classification | 51 action classes from movies and web videos; small and deliberately hard |
| Kinetics-400 [17] | clip classification | 400 human action classes, at least 400 clips each, about 10 s per clip, from YouTube |
| Something-Something V2 [9], [10] | clip classification | 174 motion-centric classes defined as caption templates; about 169k training and 20k validation videos |
| Diving48 [7] | clip classification | over 18,000 clips of 48 fine-grained dive classes, built to reduce static bias |
| Charades [49] | multi-label classification, localization | 9,848 videos of everyday indoor activities recorded by 267 people, 157 action classes |
| THUMOS14 [50] | temporal localization | untrimmed videos, including background videos without the target actions |
| ActivityNet [51] | untrimmed classification, localization | a large-scale benchmark of human activities in untrimmed web videos |
| AVA [43] | spatio-temporal detection | 80 atomic actions in 430 15-minute movie clips, 1.58M action labels |
| EPIC-KITCHENS-100 [52], [53] | egocentric recognition, detection, anticipation | 100 hours of unscripted head-mounted video in 45 environments, 90K actions |
| NTU RGB+D [44] | skeleton / RGB-D recognition | over 56 thousand samples, 60 action classes, 40 subjects |
| MSR-VTT [38] | retrieval, captioning | 10K web clips, 200K clip–sentence pairs |
| Video-MME [40] | video QA for multimodal LLMs | 900 videos from 11 s to 1 h, 2,700 question–answer pairs |

UCF101 and HMDB51 report average accuracy over **three official train/test splits** [12], [13]; Kinetics and Something-Something report top-1 and top-5 on their validation sets; Charades and AVA report mAP. Two warnings. First, Kinetics and Something-Something reward different things: the VideoMAE authors describe Kinetics videos as mostly stationary and scene-related, while Something-Something V2 is motion-centric [10]. A model's ranking can change between them. Second, the authors of EPIC-KITCHENS-100 built in a "test of time": testing models trained

on footage collected in 2018 on new footage collected two years later, and an unsupervised domain adaptation challenge [53]. Benchmarks like these measure robustness that a single held-out split cannot.

## Failure modes

**Plain version.** Video models fail in a few predictable ways. Most of them come from the model using a cue that happened to work on the training data.

1. **Scene and object bias.** The model predicts "playing tennis" from a tennis court. Evidence: representation bias toward static cues in common datasets [7]; models that fail on mimed actions without the usual objects and scenes [46]. Remedies include debiasing losses, such as an adversarial scene classifier and a loss that penalizes confident predictions on videos with the person masked out [54], motion-centric training data [9] and pose input [46].
2. **Temporal shortcuts.** The model ignores frame order. Test: shuffle or reverse the frames. If accuracy barely moves, the model (or the benchmark) does not use time; Sevilla-Lara et al. built their temporal subset of classes exactly this way [8].
3. **Short temporal context.** Most clip models see only a few seconds at a time. Actions defined over minutes ("assembling furniture") or questions about a whole film need long-range memory, which remains a weakness even for large multimodal models [40].
4. **Domain shift.** Third-person YouTube video, egocentric kitchens, surveillance at night and medical endoscopy look very different. Models trained on one transfer imperfectly to another; EPIC-KITCHENS-100 measures this with unseen kitchens and footage recorded later [52], [53]. Distribution-shift tests from DLCV 02 apply.
5. **Ambiguous labels and boundaries.** Actions overlap, have fuzzy starts and ends and depend on intent. Multi-label datasets (Charades, AVA) and several tIoU thresholds acknowledge this; single-label accuracy hides it.
6. **Unfair comparisons.** Different numbers of views, frame counts, resolutions and pretraining data make reported numbers incomparable [1]. Always check the protocol line of a results table.

Two cheap diagnostics catch most of these: a **single-frame baseline** (how much of the score needs no time at all?) and a **shuffled-frames test** (does order matter to the model?). Run both before trusting a video model.

## Modern view

### What the surveys say

Five surveys cover the field from different angles. **Zhu et al.** [1] review over 200 papers on deep action recognition, introduce 17 datasets, present the models in chronological order (early attempts, two-stream networks, 3D convolutions, compute-efficient models), benchmark popular methods

under one protocol with released code, and name the field's main challenges: long-range temporal modeling, compute cost and results made incomparable by different datasets and protocols. **Kong and Fu** [2] cover both recognition (labeling a complete action) and prediction (anticipating an action from its first part), including datasets and evaluation protocols. **Selva et al.** [3] organize video transformers by how they tokenize the input, how they reduce redundancy and reintroduce inductive biases, how they capture long-term dynamics and how they are trained, and find that they outperform 3D CNNs on action classification even at lower computational cost. **Madan et al.** [4] analyze over 200 video foundation models in three groups (image-based models adapted to video, video-based models, and universal models across modalities) and report that image-based models outperform video-based ones on most video understanding tasks, while universal multimodal models do best. **Tang et al.** [5] categorize video understanding with large language models by how the video reaches the language model (video analyzer, video embedder or both) and by the role of the language model (summarizer, manager, text decoder, regressor or hidden layer).

*Online only: Guide to the Modern view surveys — https://blog.shyandram.dev/en/learn/deep-learning/video-understanding/papers/guide-to-the-modern-view-surveys/*

### The state of the art, 2023–2026

- **Self-supervised pretraining is a common starting point.** Masked video modeling (VideoMAE [10], VideoMAE V2 [33]) and feature prediction (V-JEPA [34]) produce encoders that are fine-tuned or probed for downstream tasks; the VideoMAE V2 authors call scale the primary factor for such foundation models [33].
- **Video foundation models unify tasks.** Models such as InternVideo2 [39] train one encoder with masked modeling, video–text contrast and next-token prediction, and are evaluated across recognition, retrieval and dialogue. The finding of Madan et al. that adapted image models remain very competitive [4] is a reminder that much of video understanding is still appearance.
- **Multimodal LLMs take over open-ended understanding.** Video question answering, captioning and temporal grounding are increasingly handled by language models with visual tokens, evaluated on benchmarks such as Video-MME [40]. The bottleneck is the number of visual tokens for long videos.
- **Efficient long-range models.** Linear-cost sequence models such as VideoMamba [30] and streaming designs (in the spirit of online TSM [21]) target long and live video.
- **Localization builds on pretrained encoders.** Temporal localizers run on features from pretrained video models; ActionFormer, for example, uses two-stream I3D features pretrained on Kinetics for THUMOS14 [42], so better encoders and better localization heads improve results together.

### Open problems

1. **Genuine temporal reasoning.** Benchmarks where one frame or the language prior is not enough, and models that use order, causality and duration.

2. **Long video.** Memory and token budgets for hours of video, and evaluation of long-range understanding.
3. **Fine-grained and egocentric actions.** Hands, small objects, and intent; transfer across viewpoints and environments.
4. **Efficient and online recognition.** Low latency on devices, with honest cost reporting (FLOPs × views, real latency).
5. **Evaluation.** Comparable protocols, contamination of benchmarks by web-scale pretraining data, and metrics that reward temporal understanding rather than appearance.

## Key takeaways

- Choose metrics from the requested output: clip labels, action intervals, person boxes or retrieval results.
- A strong score can reflect scene cues rather than motion understanding.
- Flow, temporal convolutions and attention incorporate time differently; inspect what information each can access.
- Sampling and multi-rate pathways trade temporal coverage against computation.
- Masking must prevent easy temporal leakage if pretraining is meant to learn useful structure.
- Compare matched testing views and pretraining, then test a single-frame baseline and shuffled-frame inputs.

## Exercises

1. **How much does the combined model cheat?** In the scene-bias code, standardize the features (`sklearn.preprocessing.StandardScaler` in a pipeline) and vary `C` over 0.01, 0.1 and 1. How do the "no cue" and "shift" accuracies of the frame+motion model change? Why does standardization help it use the motion feature?

   **Hint**

   The motion features have a small numeric range compared with the 400 pixel features, so with an L2 penalty the model finds it cheaper to spread weight over many pixels. In our runs, standardizing gave the frame+motion model "no cue"/"shift" accuracies of 0.752/0.547 at `C = 0.01`, 0.824/0.685 at `C = 0.1` and 0.842/0.708 at `C = 1`, still well below the motion-only model. Scaling reduces the shortcut but does not remove it while the scene remains predictive in training.

2. **Inflation by hand.** A 2D filter $w_{2D} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$ (a horizontal derivative) is inflated to $N = 3$ frames. Write $w_{3D}$, and compute its response at one position for (a) a boring video where every frame has the row $[5, 3, 1]$ and (b) a video whose three frames have rows $[5, 3, 1]$, $[3, 1, 0]$ and $[1, 0, 0]$. What does the difference tell you about what training must change?

   **Hint**

$w_{3D}$ has three temporal slices, each $\frac{1}{3}[1, 0, -1]$. (a) Each slice gives $\frac{1}{3}(5 - 1) = \frac{4}{3}$, total 4, the same as the 2D response. (b) The slices give $\frac{4}{3}$, $\frac{3}{3}$ and $\frac{1}{3}$, total $\frac{8}{3}$. The inflated filter averages the frames, so it responds to motion only through blurring; to become a motion detector, training must make the temporal slices *different* (for example, opposite signs on the first and last).

3. **(2+1)D with unequal widths.** For a layer with $N_{i-1} = 64$ input and $N_i = 128$ output channels, $t = 3$, $d = 3$, compute $M_i$, the 3D parameter count and the (2+1)D parameter count. Then choose $M = 64$ and compute the savings.

   **Hint**

   $M_i = \lfloor 3 \cdot 9 \cdot 64 \cdot 128/(9 \cdot 64 + 3 \cdot 128) \rfloor = \lfloor 221184/960 \rfloor = 230$. 3D: $64 \cdot 128 \cdot 27 = 221{,}184$. (2+1)D with $M = 230$: $64 \cdot 230 \cdot 9 + 230 \cdot 128 \cdot 3 = 132{,}480 + 88{,}320 = 220{,}800$ (slightly less, because of the floor). With $M = 64$: 36,864 + 24,576 = 61,440, about 28% of the 3D layer.

4. **Views and cost.** Model A costs 36 GFLOPs per view and is tested with 10 × 3 views; model B costs 180 GFLOPs per view and is tested with 1 × 3 views. Which is more expensive per video? Design a fair comparison.

   **Hint**

   A: $36 \times 30 = 1080$ GFLOPs; B: $180 \times 3 = 540$ GFLOPs, so A costs twice as much per video despite being five times cheaper per view. A fair comparison plots accuracy against total inference cost for several view counts of each model, on the same frames and resolution.

5. **A shuffled-frames test.** Extend the scene-bias code: shuffle the frame order of each test clip, with a new permutation per clip (`rng = np.random.default_rng(5)`, then `np.stack([x[rng.permutation(8)] for x in X])`), and re-evaluate the three models trained on ordered clips. Predict the result before you run it.

   **Hint**

   The one-frame model still reads only the scene, so it barely changes (0.868 / 0.499 / 0.127 on the three test sets in our run). The motion model's per-pair estimates now have random signs, and it drops to chance (0.522 / 0.500 / 0.466). The frame+motion model does not move (0.864 / 0.497 / 0.133), which exposes that it never relied on order. A model whose accuracy survives shuffling is not using time.

6. **tIoU by hand.** A ground-truth action runs from 10 s to 20 s. Predictions: P1 [12, 22] with score 0.9, P2 [9, 19] with score 0.8, P3 [30, 35] with score 0.7. Compute each tIoU, mark TP/FP at $\theta = 0.5$ and $\theta = 0.75$, and compute AP at both thresholds.

   **Hint**

   P1: overlap 8, union 12, tIoU 0.667. P2: overlap 9, union 11, tIoU 0.818. P3: 0. At 0.5: P1 is TP (first in score order), P2 is a duplicate (FP), P3 FP; precision at recall 1 is 1, so AP = 1. At 0.75: P1 is FP, P2 is TP (precision 1/2 at recall 1), P3 FP; AP = 0.5. A worse box with a higher score can cost half the AP at a strict threshold.

## References


1. Y. Zhu, X. Li, C. Liu, M. Zolfaghari, Y. Xiong, C. Wu, Z. Zhang, J. Tighe et al., "A comprehensive study of deep video action recognition," technical report, arXiv:2012.06567, 2020. arXiv
2. Y. Kong and Y. Fu, "Human action recognition and prediction: A survey," *International Journal of Computer Vision*, vol. 130, no. 5, pp. 1366–1401, 2022. doi
3. J. Selva, A. S. Johansen, S. Escalera, K. Nasrollahi, T. B. Moeslund and A. Clapés, "Video transformers: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 11, pp. 12922–12943, 2023. doi
4. N. Madan, A. Moegelmose, R. Modi, Y. S. Rawat and T. B. Moeslund, "Foundation models for video understanding: A survey," arXiv:2405.03770, 2024. arXiv
5. Y. Tang, J. Bi, S. Xu, L. Song, S. Liang, T. Wang, D. Zhang, J. An et al., "Video understanding with large language models: A survey," arXiv:2312.17432, 2023 (accepted to *IEEE Transactions on Circuits and Systems for Video Technology* according to the arXiv record). arXiv
6. A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proc. IEEE CVPR*, pp. 1725–1732, 2014. CVF
7. Y. Li, Y. Li and N. Vasconcelos, "RESOUND: Towards action recognition without representation bias," in *Proc. ECCV*, 2018. CVF
8. L. Sevilla-Lara, S. Zha, Z. Yan, V. Goswami, M. Feiszli and L. Torresani, "Only time can tell: Discovering temporal data for temporal modeling," in *Proc. IEEE WACV*, 2021. doi
9. R. Goyal, S. Ebrahimi Kahou, V. Michalski, J. Materzyńska, S. Westphal, H. Kim, V. Haenel, I. Fruend et al., "The 'something something' video database for learning and evaluating visual common sense," in *Proc. IEEE ICCV*, 2017. CVF
10. Z. Tong, Y. Song, J. Wang and L. Wang, "VideoMAE: Masked autoencoders are data-efficient learners for self-supervised video pre-training," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2022. NeurIPS ·arXiv
11. S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. doi
12. K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2014. NeurIPS ·arXiv
13. L. Wang, Y. Xiong, Z. Wang, Y. Qiao, D. Lin, X. Tang and L. Van Gool, "Temporal segment networks: Towards good practices for deep action recognition," in *Proc. ECCV*, pp. 20–36, 2016. doi ·arXiv
14. J. Carreira and A. Zisserman, "Quo vadis, action recognition? A new model and the Kinetics dataset," in *Proc. IEEE CVPR*, 2017. CVF ·arXiv
15. C. Feichtenhofer, H. Fan, J. Malik and K. He, "SlowFast networks for video recognition," in *Proc. IEEE/CVF ICCV*, 2019. CVF ·arXiv
16. D. Tran, L. Bourdev, R. Fergus, L. Torresani and M. Paluri, "Learning spatiotemporal features with 3D convolutional networks," in *Proc. IEEE ICCV*, pp. 4489–4497, 2015. CVF
17. W. Kay, J. Carreira, K. Simonyan, B. Zhang, C. Hillier, S. Vijayanarasimhan, F. Viola, T. Green et al., "The Kinetics human action video dataset," arXiv:1705.06950, 2017. arXiv
18. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE CVPR*, 2009. doi

19. T. Unterthiner, S. van Steenkiste, K. Kurach, R. Marinier, M. Michalski and S. Gelly, "Towards accurate generative models of video: A new metric & challenges," arXiv:1812.01717, 2018. arXiv
20. D. Tran, H. Wang, L. Torresani, J. Ray, Y. LeCun and M. Paluri, "A closer look at spatiotemporal convolutions for action recognition," in *Proc. IEEE/CVF CVPR*, 2018. CVF ·arXiv
21. J. Lin, C. Gan and S. Han, "TSM: Temporal shift module for efficient video understanding," in *Proc. IEEE/CVF ICCV*, 2019. CVF ·arXiv
22. X. Wang, R. Girshick, A. Gupta and K. He, "Non-local neural networks," in *Proc. IEEE/CVF CVPR*, 2018. CVF
23. C. Feichtenhofer, "X3D: Expanding architectures for efficient video recognition," in *Proc. IEEE/CVF CVPR*, 2020. CVF
24. A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer et al., "An image is worth 16x16 words: Transformers for image recognition at scale," in *Proc. ICLR*, 2021. arXiv
25. A. Arnab, M. Dehghani, G. Heigold, C. Sun, M. Lučić and C. Schmid, "ViViT: A video vision transformer," in *Proc. IEEE/CVF ICCV*, 2021. CVF ·arXiv
26. G. Bertasius, H. Wang and L. Torresani, "Is space-time attention all you need for video understanding?," in *Proc. ICML*, PMLR 139, 2021. PMLR ·arXiv
27. H. Fan, B. Xiong, K. Mangalam, Y. Li, Z. Yan, J. Malik and C. Feichtenhofer, "Multiscale vision transformers," in *Proc. IEEE/CVF ICCV*, 2021. CVF ·arXiv
28. Z. Liu, J. Ning, Y. Cao, Y. Wei, Z. Zhang, S. Lin and H. Hu, "Video Swin Transformer," in *Proc. IEEE/CVF CVPR*, 2022. CVF ·arXiv
29. Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin and B. Guo, "Swin Transformer: Hierarchical vision transformer using shifted windows," in *Proc. IEEE/CVF ICCV*, 2021. doi
30. K. Li, X. Li, Y. Wang, Y. He, Y. Wang, L. Wang and Y. Qiao, "VideoMamba: State space model for efficient video understanding," in *Proc. ECCV*, pp. 237–255, 2024. doi ·arXiv
31. A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," in *Proc. Conference on Language Modeling* (*COLM*), 2024. arXiv
32. K. He, X. Chen, S. Xie, Y. Li, P. Dollár and R. Girshick, "Masked autoencoders are scalable vision learners," in *Proc. IEEE/CVF CVPR*, 2022. doi
33. L. Wang, B. Huang, Z. Zhao, Z. Tong, Y. He, Y. Wang, Y. Wang and Y. Qiao, "VideoMAE V2: Scaling video masked autoencoders with dual masking," in *Proc. IEEE/CVF CVPR*, 2023. CVF ·arXiv
34. A. Bardes, Q. Garrido, J. Ponce, X. Chen, M. Rabbat, Y. LeCun, M. Assran and N. Ballas, "Revisiting feature prediction for learning visual representations from video," *Transactions on Machine Learning Research* (*TMLR*), 2024. OpenReview ·arXiv
35. M. Assran, A. Bardes, D. Fan, Q. Garrido, R. Howes et al., "V-JEPA 2: Self-supervised video models enable understanding, prediction and planning," arXiv:2506.09985, 2025. arXiv
36. A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal et al., "Learning transferable visual models from natural language supervision," in *Proc. ICML*, PMLR 139, 2021. PMLR
37. H. Luo, L. Ji, M. Zhong, Y. Chen, W. Lei, N. Duan and T. Li, "CLIP4Clip: An empirical study of CLIP for end to end video clip retrieval and captioning," *Neurocomputing*, vol. 508, pp. 293–304, 2022. doi ·arXiv
38. J. Xu, T. Mei, T. Yao and Y. Rui, "MSR-VTT: A large video description dataset for bridging video

and language," in *Proc. IEEE CVPR*, 2016. CVF
39. Y. Wang, K. Li, X. Li, J. Yu, Y. He, G. Chen, B. Pei, R. Zheng et al., "InternVideo2: Scaling foundation models for multimodal video understanding," in *Proc. ECCV*, pp. 396–416, 2024. doi ·arXiv
40. C. Fu, Y. Dai, Y. Luo, L. Li, S. Ren, R. Zhang, Z. Wang, C. Zhou et al., "Video-MME: The first-ever comprehensive evaluation benchmark of multi-modal LLMs in video analysis," in *Proc. IEEE/CVF CVPR*, 2025. CVF ·arXiv
41. T. Lin, X. Liu, X. Li, E. Ding and S. Wen, "BMN: Boundary-matching network for temporal action proposal generation," in *Proc. IEEE/CVF ICCV*, 2019. CVF
42. C.-L. Zhang, J. Wu and Y. Li, "ActionFormer: Localizing moments of actions with transformers," in *Proc. ECCV*, pp. 492–510, 2022. doi ·arXiv
43. C. Gu, C. Sun, D. A. Ross, C. Vondrick, C. Pantofaru, Y. Li, S. Vijayanarasimhan, G. Toderici et al., "AVA: A video dataset of spatio-temporally localized atomic visual actions," in *Proc. IEEE/CVF CVPR*, 2018. CVF ·arXiv
44. A. Shahroudy, J. Liu, T.-T. Ng and G. Wang, "NTU RGB+D: A large scale dataset for 3D human activity analysis," in *Proc. IEEE CVPR*, 2016. doi ·arXiv
45. S. Yan, Y. Xiong and D. Lin, "Spatial temporal graph convolutional networks for skeleton-based action recognition," in *Proc. AAAI*, vol. 32, no. 1, 2018. doi ·arXiv
46. P. Weinzaepfel and G. Rogez, "Mimetics: Towards understanding human actions out of context," *International Journal of Computer Vision*, vol. 129, no. 5, pp. 1675–1690, 2021. doi ·arXiv
47. K. Soomro, A. R. Zamir and M. Shah, "UCF101: A dataset of 101 human actions classes from videos in the wild," technical report CRCV-TR-12-01, arXiv:1212.0402, 2012. arXiv
48. H. Kuehne, H. Jhuang, E. Garrote, T. Poggio and T. Serre, "HMDB: A large video database for human motion recognition," in *Proc. IEEE ICCV*, pp. 2556–2563, 2011. doi
49. G. A. Sigurdsson, G. Varol, X. Wang, A. Farhadi, I. Laptev and A. Gupta, "Hollywood in homes: Crowdsourcing data collection for activity understanding," in *Proc. ECCV*, pp. 510–526, 2016. doi ·arXiv
50. H. Idrees, A. R. Zamir, Y.-G. Jiang, A. Gorban, I. Laptev, R. Sukthankar and M. Shah, "The THUMOS challenge on action recognition for videos 'in the wild'," *Computer Vision and Image Understanding*, vol. 155, pp. 1–23, 2017. doi
51. F. Caba Heilbron, V. Escorcia, B. Ghanem and J. C. Niebles, "ActivityNet: A large-scale video benchmark for human activity understanding," in *Proc. IEEE CVPR*, pp. 961–970, 2015. CVF
52. D. Damen, H. Doughty, G. M. Farinella, S. Fidler, A. Furnari, E. Kazakos, D. Moltisanti, J. Munro et al., "Scaling egocentric vision: The EPIC-KITCHENS dataset," in *Proc. ECCV*, 2018. CVF
53. D. Damen, H. Doughty, G. M. Farinella, A. Furnari, E. Kazakos, J. Ma, D. Moltisanti, J. Munro et al., "Rescaling egocentric vision: Collection, pipeline and challenges for EPIC-KITCHENS-100," *International Journal of Computer Vision*, vol. 130, no. 1, 2022. doi ·arXiv
54. J. Choi, C. Gao, J. C. E. Messou and J.-B. Huang, "Why can't I dance in the mall? Learning to mitigate scene bias in action recognition," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2019. NeurIPS ·arXiv
55. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE CVPR*, 2015. doi

# DLCV 09 • Deep Image Restoration: Low-Level Vision

One degradation model, many tasks: how deep networks undo blur, noise, low resolution, haze, rain, snow and darkness, from SRCNN and DnCNN to Restormer, all-in-one models and diffusion priors.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/

## What you'll learn

- Describe restoration through an observation model and its unknowns.
- Build paired data and synthetic degradations, then assess the real-world gap.
- Compare task-specific backbones and models conditioned on degradation or instructions.
- Choose losses and distinguish fidelity, perceptual quality and measurement consistency.
- Evaluate results with appropriate datasets, protocols and uncertainty about missing detail.

## The big picture

Low-level vision is about the pixels themselves. Before a system can recognize a face, read a sign or detect a car, the image has to be clear enough to look at. Restoration models are the part of the pipeline that takes a damaged photo (too small, noisy, blurred, foggy, rainy, dark) and returns a better estimate of the clean scene.

> **Explain like I'm five**
>
> Imagine someone hands you a photo that went through the washing machine: it is smudged, speckled and faded. You have seen thousands of clean photos in your life, so you can guess what this one *used to* look like: faces have two eyes, edges of buildings are straight, the sky is smooth. A restoration network does the same thing. It has looked at millions of "before and after" pairs, so it learned what clean pictures look like and how washing-machine damage changes them, and it uses both to undo the damage.

Why it matters: every phone camera runs denoising, demosaicking and super-resolution before you ever see a picture. Satellite, microscope, medical and surveillance images are restored before they are measured. Video streaming services upscale low-bitrate frames. Self-driving cars need their detectors to work in rain and fog. And from a research point of view, restoration is where many architectural ideas were tested first: residual learning, channel attention, efficient transformers and diffusion priors all have well-known restoration variants.

> **About this tutorial**

This tutorial is expanded from an earlier post, Low-Level Vision Task — Image Restoration 簡介 (written in Traditional Chinese) [1]. It keeps the post's order and viewpoints, folds in its 2026 updates and corrections, and adds the math, code and survey literature.

We assume you know CNNs and backpropagation (DLCV 01) and the classical restoration chapter (IP 05), which covers noise models, Wiener filtering and the degradation model in detail. We link to it rather than repeat it.

**Related chapters.** Three chapters go deeper into parts of this one: DLCV 09A (single-image super-resolution), DLCV 09A.3 (video super-resolution) and DLCV 09B (generative restoration with GAN and diffusion priors). DLCV 10 covers image fusion. DLCV 01B explains why restoration networks are built from blocks differently than classifiers.

## What is low-level vision?

**Plain version.** High-level vision asks "what is in the picture?" (classification, detection). Low-level vision asks "what does the picture actually look like, pixel by pixel?" One is about *understanding*, the other about *perceiving*.

The analogy in the earlier post [1] comes from how people interact with the world: we first *sense* (light hits the retina and we see edges, colors and contrast) and then *interpret* (that is a cup). Both are necessary. If the sensing is poor (the cup is blurry), the interpretation suffers. Low-level vision tasks exist to make the sensing good again: when we say an object looks "blurry", our perception of it is not sharp, and a deblurring task is created to make it "clear".

**Precise version.** A low-level vision task maps an image to an image of (usually) the same content:

$$f_\theta : \mathbb{R}^{H\times W\times C} \to \mathbb{R}^{H'\times W'\times C'},$$

where the output is a pixel-wise estimate rather than a label. $H, W, C$ are the input height, width and channels, and $H', W', C'$ those of the output ($H' = sH$ for super-resolution by a factor $s$, otherwise usually equal). This has three consequences that shape the whole field:

1. **Dense, high-resolution outputs.** Every pixel matters, so architectures cannot downsample aggressively and must keep fine detail, which makes them expensive at high resolution.
2. **Supervision is cheap to synthesize.** Take a clean image, damage it with a known process, and you have an input–target pair. This is a huge advantage over labeled data for recognition, and also the source of the field's main weakness (synthetic damage is not real damage).
3. **"Correct" is ambiguous.** Many clean images are consistent with one damaged image. Which one is "best" depends on whether you value fidelity to the truth or natural appearance, as we will see in the evaluation section.

The family of tasks is broad: deblurring, super-resolution, denoising, inpainting, color calibration, infrared–visible fusion and more. Papers with Code used to be the standard place to browse these task lists and leaderboards, but it has shut down and its address now redirects to Hugging Face's trending-papers page, so its per-task leaderboards are no longer available. A reliable way

to follow progress today is the yearly **NTIRE** challenge reports (New Trends in Image Restoration and Enhancement, a CVPR workshop), which we discuss near the end [2], [3].

## One problem, many faces: the degradation model

**Plain version.** Every task in this tutorial can be written as "the camera saw a clean scene $x$, something damaged it, and we got $y$." Only the damage changes. If you can write the damage down, you can name the task.

### The unified view

$$y = \mathcal{D}(x) + n,$$

where $x$ is the unknown clean image, $\mathcal{D}$ is a (possibly nonlinear, possibly unknown) degradation operator, $n$ is noise, and $y$ is what we observe. Restoration means estimating $x$ from $y$. Here are the specific forms used in the post [1] and in the rest of this tutorial. Figure 1 shows each one applied to the same photo.

**Blur (the textbook model).** As in IP 05,

$$g(u,v) = (h * f)(u,v) + n(u,v),$$

where $f$ is the sharp image, $h$ the point spread function (blur kernel), $*$ 2-D convolution, $n$ additive noise and $g$ the observed image, all indexed by pixel coordinates $(u,v)$. Denoising is the special case $h = \delta$ (a unit impulse, so no blur).

**Super-resolution.** The low-resolution image is a blurred, subsampled and noisy version of the high-resolution one:

$$y = (x * k)\downarrow_s + n,$$

where $k$ is a blur (anti-aliasing) kernel, $\downarrow_s$ keeps every $s$-th pixel in each direction, and $s$ is the scale factor (2, 3, 4, ...). The classic "bicubic" benchmark setting replaces all of this with MATLAB-style bicubic downsampling and no noise.

**Compression.** $y = \mathrm{JPEG}_q(x)$, where $q$ is the quality factor. JPEG quantizes $8 \times 8$ DCT coefficients (see IP 08), producing blocking and ringing. It is not additive and not linear, so it is usually just added to the pipeline as a black box.

**Haze (the atmospheric scattering model).**

$$I(p) = J(p)\,t(p) + A\,(1 - t(p)), \qquad t(p) = e^{-\beta d(p)},$$

where $p$ is a pixel, $I$ the hazy image, $J$ the haze-free scene radiance, $A$ the global atmospheric light (the color of the haze), $t \in (0,1]$ the transmission, $\beta$ the scattering coefficient and $d$ the scene depth. The first term is the attenuated scene; the second is "airlight" scattered toward the camera. Far objects (large $d$) get small $t$ and fade into $A$.

**Rain.** For rain streaks the common model is additive,

$$I = B + R,$$

where $B$ is the rain-free background and $R$ the rain layer. Raindrops stuck on a lens or windshield behave differently: they block and refract the background, so a popular model is $I = (1-M)\odot B+R$, where $M$ is a binary raindrop mask, $\odot$ the element-wise product and $R$ the light passing through the drops [39].

**Low light (Retinex).** Retinex theory [5] separates an image into reflectance and illumination:

$$S = R \cdot L,$$

where $S$ is the observed image, $R$ the reflectance (the intrinsic color and texture of surfaces, what we want), $L$ the illumination and $\cdot$ the element-wise product. A dark photo has small $L$; enhancement estimates $L$ and brightens it, while noise that was hidden in the dark gets amplified.




*Figure 1 — One clean image x pushed through the degradation models of this section. Each restoration task tries to undo one (or several) of these.*

Although none of these models is exact, they all have the same shape: a clean image goes in, an operator plus noise comes out. That is the post's central point [1]: seen through the degradation model, these are all **the same kind of problem**, and the same tools (and often the same networks) apply to all of them. Figure 2 organizes them by where the damage comes from.

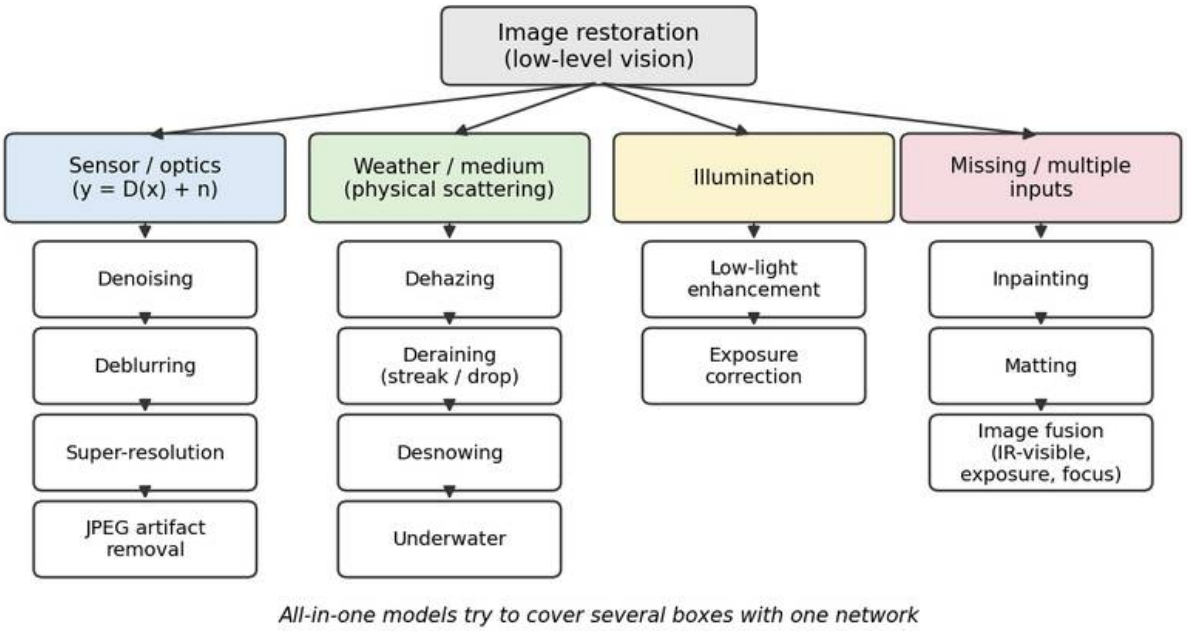



*Figure 2 — A task taxonomy of low-level vision by the source of degradation. All-in-one models try to cover several boxes with a single network.*

### Why the problem is hard: ill-posedness

For almost every $\mathcal{D}$ above, many different $x$ produce the same $y$. Downsampling by 4 throws away 15 of every 16 pixels; blur suppresses high frequencies to near zero; haze with $t \approx 0$ erases distant objects. The inverse problem is **ill-posed**, and we need a **prior** on what clean images look like. In the Bayesian (MAP) form,

$$\hat{x} = \arg\min_{x} \; \tfrac{1}{2}\|y - \mathcal{D}(x)\|_2^2 + \lambda\, \Phi(x),$$

where the first term (data fidelity) says "$\hat{x}$ must explain the observation", $\Phi$ is a regularizer encoding the prior (small $\Phi$ for natural-looking images), and $\lambda > 0$ balances the two. The whole history of the field is a history of better priors $\Phi$.

## Before deep learning: the era of hand-crafted priors

Before deep learning, each task had its own carefully reasoned prior, and success depended on having enough physical or statistical knowledge of the problem. Denoising used local smoothness, non-local self-similarity and sparsity; deblurring used Wiener and constrained least-squares filters plus heavy-tailed gradient priors; dehazing used the dark channel prior [25]; super-resolution used interpolation and example-based dictionaries; low-light enhancement used histogram equalization and Retinex decompositions. IP 05 covers the core of this toolbox

(noise models, mean/median/adaptive filters, inverse, Wiener and CLS filtering) and ends with a short history of the move to learned denoisers. The view in [1] is that once deep learning surpassed what these models could do, the decisive factor stopped being a strong theoretical prior for each task and became a strong **learning mechanism and feature extractor**. The degradation model did not disappear: it moved from the solver into the **data generator**.

## The deep learning formulation

**Plain version.** Make lots of (damaged, clean) pairs, and train a network to map damaged to clean. Most of the hard work is in making the damage realistic.

### Supervised training on pairs

Given a dataset of clean images $\{x_i\}$ and a degradation sampler, we build pairs $(y_i, x_i)$ with $y_i = \mathcal{D}_i(x_i) + n_i$, and fit

$$\theta^\star = \arg\min_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(y_i),\, x_i),$$

where $f_\theta$ is the restoration network with parameters $\theta$, $N$ the number of training pairs and $\ell$ a loss (L1, L2, perceptual, adversarial, ...; see the losses section). Training is done on random crops ("patches", e.g. 64×64 to 256×256) with flips and rotations, because convolutional and windowed-attention networks are translation-equivariant and full-resolution images do not fit in GPU memory.

A useful fact explains a lot of what follows: with the L2 loss, the optimal network outputs the **posterior mean** $f^\star(y) = \mathbb{E}[x \mid y]$. When many sharp images are consistent with $y$, their average is blurry. This is why MSE-trained super-resolution looks smooth, and why GANs and diffusion models were brought in.

### Synthetic degradation pipelines

Pairs come from three sources. (1) **Synthetic**: degrade clean images with a known model (bicubic downsampling, Gaussian noise, Gaussian blur, JPEG). Cheap and unlimited, but limited by how realistic the model is. (2) **Captured**: photograph the same scene twice, e.g. a short and a long exposure, or a burst averaged into a clean reference (as in the SIDD real-noise dataset [22]), or averaging high-speed video frames to create real motion blur (GoPro [44]). Realistic but expensive and specific to one camera. (3) **Unpaired / self-supervised**: learn from degraded images alone, using GANs, cycle consistency or noise statistics. Most strong models today train on synthetic pairs from a rich degradation pipeline and evaluate on both synthetic and captured data.

A typical "classic" pipeline is blur → downsample → noise → JPEG, mirroring the order in which a camera and an image-sharing site would damage a photo:

```
import numpy as np, cv2, torch, torch.nn.functional as F
from skimage import data, img_as_float32

def gaussian_kernel(sigma, size=21):
    ax = torch.arange(size) - size // 2
    g = torch.exp(-ax**2 / (2 * sigma**2))
    k = torch.outer(g, g)
    return k / k.sum()

def degrade(x, scale=4, sigma_blur=1.6, sigma_noise=0.03, jpeg_q=40, rng=None):
    """x: float tensor (C,H,W) in [0,1]. Returns the low-quality image y = D(x) + n."""
    rng = rng or np.random.default_rng(0)
    C = x.shape[0]
    k = gaussian_kernel(sigma_blur).to(x)[None, None].repeat(C, 1, 1, 1)
    y = F.conv2d(F.pad(x[None], (10,) * 4, mode="reflect"), k, groups=C)   # 1) blur  h * x
    y = y[..., ::scale, ::scale]                                           # 2) downsample
     ↪  (.)↓s
    y = y + sigma_noise * torch.from_numpy(rng.standard_normal(y.shape).astype(np.float32))
     ↪  # 3) noise n
    y = y.clamp(0, 1)[0]
    img = (y.permute(1, 2, 0).numpy() * 255).round().astype(np.uint8)      # 4) JPEG round
     ↪  trip
    ok, buf = cv2.imencode(".jpg", img[..., ::-1], [cv2.IMWRITE_JPEG_QUALITY, jpeg_q])
    img = cv2.imdecode(buf, cv2.IMREAD_COLOR)[..., ::-1]
    return torch.from_numpy(img.copy()).permute(2, 0, 1).float() / 255

x = torch.from_numpy(img_as_float32(data.astronaut())).permute(2, 0, 1)    # (3, 512, 512)
y = degrade(x)
print(x.shape, "->", y.shape)        # torch.Size([3, 512, 512]) -> torch.Size([3, 128, 128])
```

In training, every parameter (kernel width and shape, noise level and type, JPEG quality, even the order) is **sampled at random per image**, so the network sees a distribution of degradations instead of a single one.

### Real-world and blind degradations

A model trained on one fixed $\mathcal{D}$ (say, bicubic ×4) breaks on real photos, whose degradation is unknown, spatially varying and a mix of several effects. This is the **blind** (or real-world) restoration problem. Two broad strategies exist:

- **Make the synthetic distribution wide enough to cover reality.** Real-ESRGAN [4] applies the classic pipeline *twice* in sequence ("high-order" degradation, with randomized blur kernels, resizing, noise and JPEG in each round) and adds sinc filters to simulate ringing and overshoot. Trained on purely synthetic data, it generalizes much better to real low-quality images than bicubic-trained models.
- **Bring in a strong generative prior.** Pretrained text-to-image diffusion models already know what natural images look like. DiffBIR [16], SUPIR [17] and OSEDiff [18] condition such a model on the degraded input, so that the prior "fills in" plausible detail. This is the main

recent trend in real-world SR and is discussed below.

A third, complementary strategy is to **estimate the degradation** (noise level, kernel, degradation type) and feed it to the network, as FFDNet's noise-level map [21] and the all-in-one models do. In blind super-resolution the main unknown is the blur kernel $k$ in $y = (x * k) \downarrow_s + n$, and a whole family of methods gives the kernel to the network, estimates and corrects it in a loop, or estimates it from the test image alone [87]–[91]. The result to remember: explicit kernel methods are accurate and interpretable when the degradation really is "blur, downsample, noise", but a wrong kernel propagates into the output. That is why most real-world models, such as BSRGAN [92], Real-ESRGAN [4] and the diffusion-prior methods, learn the degradation **implicitly** from a randomized pipeline instead.

*Online only: Blind super-resolution: conditioning on and estimating the kernel — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/notes/blind-super-resolution-conditioning-on-and-estimating-the-kernel/*

*Online only: Plug-and-play priors and half-quadratic splitting — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/notes/plug-and-play-priors-and-half-quadratic-splitting/*

## Super-resolution

**Plain version.** Make a small image bigger *and* sharper, i.e. predict the missing high-resolution pixels rather than just stretching the existing ones.

This section is an overview; DLCV 09A ·Non-Blind Image Super-Resolution treats known-degradation SR in depth: ambiguity, classical baselines, network design decisions, perceptual and diffusion SR, and evaluation. Blind and real-world SR has its own chapter.

### Problem setup

Single-image super-resolution (SISR) estimates $x \in \mathbb{R}^{sH \times sW \times C}$ from $y \in \mathbb{R}^{H \times W \times C}$ under $y = (x * k) \downarrow_s + n$. The traditional answer is **interpolation**: nearest-neighbor, bilinear and bicubic. Bicubic, which fits a cubic polynomial through a $4 \times 4$ neighborhood, is the best known and is still used everywhere, both as a baseline and as the standard way to create benchmark low-resolution inputs. Interpolation only redistributes existing information; it cannot recreate the high frequencies that $\downarrow_s$ removed.

### How a network gets bigger: three upsampling strategies

Because the output is larger than the input, every SR network must decide *where* the resolution increases.

**a. Pre-upsampling (interpolate first).** SRCNN [6] bicubically enlarges the input to the target size, then applies three convolution layers (patch extraction, non-linear mapping, reconstruction).

Simple and intuitive, but every layer runs at high resolution, which is slow, and the network must clean up interpolation artifacts.

**b. Transposed convolution (deconvolution).** FSRCNN [7] works at low resolution and enlarges only at the end with a learned transposed convolution, giving a large speed-up over SRCNN. Transposed convolutions with a stride that does not divide the kernel size overlap unevenly and tend to produce **checkerboard artifacts** [9], so they are now rare as the final SR upsampler, although they remain common elsewhere in neural networks (for example in decoders of segmentation and generative models).

**c. Sub-pixel convolution (pixel shuffle).** ESPCN [8] computes $Cs^2$ channels at low resolution and rearranges them into an image $s$ times larger:

$$\mathrm{PS}(T)_{c,\,sh+i,\,sw+j} = T_{cs^2+si+j,\,h,\,w}, \qquad 0 \le i,j < s,$$

where $T$ is the low-resolution feature tensor with $Cs^2$ channels, $(h, w)$ a low-resolution position, $(i, j)$ the sub-pixel offset and $c$ the output channel. All learning happens at low resolution, and the rearrangement itself has no parameters. It is the dominant choice in modern SR networks (`torch.nn.PixelShuffle`).

### Milestones and current families

**Plain version.** SR networks first got deeper, then learned to draw realistic texture, and most recently borrowed transformers and large pretrained generative models.

The main steps: **SRCNN** [6] was the first end-to-end CNN for SR; **EDSR** [10] and **RCAN** [11] made SR networks very deep (EDSR without batch normalization, RCAN with channel attention); **SRGAN** [12] and **ESRGAN** [13] added adversarial and perceptual losses, giving sharp, realistic texture instead of the blurry MSE average. Today's families are transformers such as **SwinIR** [14] and **HAT** [15], state-space models such as MambaIR [49], and **diffusion priors** (DiffBIR [16], SUPIR [17] and the one-step OSEDiff [18]), which produce far more convincing textures on real photos but can also invent detail that was not in the scene. Models are usually trained on **DIV2K** [19] and compared by PSNR and SSIM on the luminance channel; real-world SR, which has no ground truth, relies on perceptual and no-reference metrics (see Evaluation). Once outputs looked much better than interpolation (around 2018), how to *judge* SR became a central question (see the perception–distortion section).

*Online only: Super-resolution milestones, families and benchmarks — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/notes/super-resolution-milestones-families-and-benchmarks/*

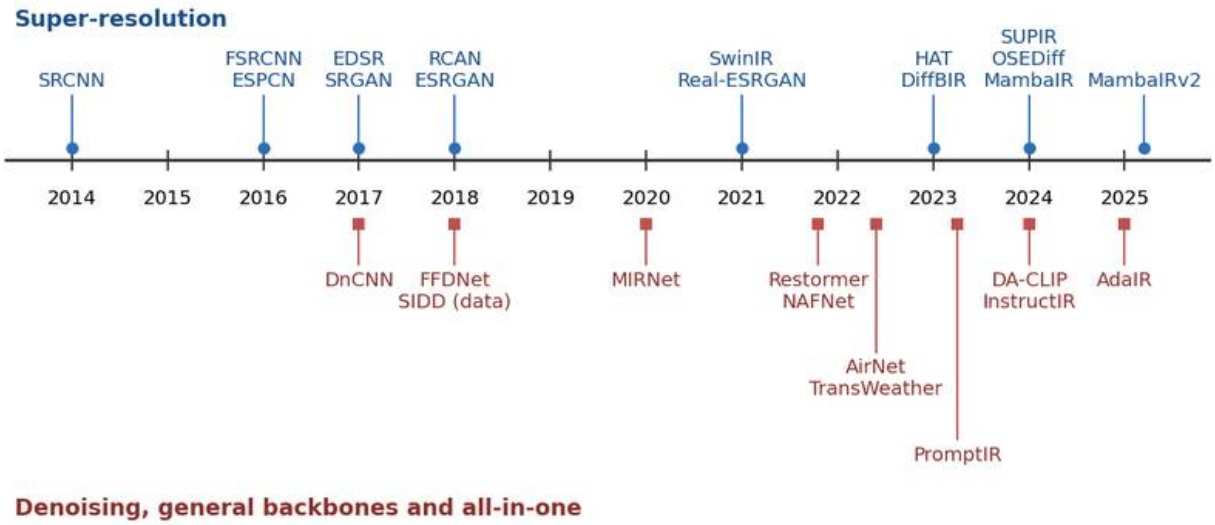



*Figure 3 — Milestones in super-resolution (top) and in denoising, general backbones and all-in-one restoration (bottom), placed by publication year (positions within a year are only for legibility).*

## Denoising

**Plain version.** Remove the random grain from a photo without wiping out the real texture. Noise is the simplest degradation, so denoising became the testbed where most deep restoration ideas appeared first.

### Problem setup

$y = x + n$. In the synthetic setting $n \sim \mathcal{N}(0, \sigma^2 I)$ (additive white Gaussian noise, AWGN, with standard deviation $\sigma$, usually quoted on a 0–255 scale such as $\sigma = 25$). Real camera noise is different: it is signal-dependent (photon shot noise grows with brightness), spatially correlated after demosaicking, and reshaped by the camera's processing pipeline. The Poisson–Gaussian model of such noise, written out for dark images, is in DLCV 09C.

### Milestones

**DnCNN** [20] made two choices that became standard. It uses **residual learning**: instead of predicting $x$, the network $\mathcal{R}_\theta$ predicts the noise, and the output is

$$\hat{x} = y - \mathcal{R}_\theta(y),$$

where $\mathcal{R}_\theta(y) \approx n$. The residual (noise) is a "simpler" function to learn than the clean image, which made deep plain CNNs with batch normalization train well. A single DnCNN can also be

trained for a range of noise levels (blind Gaussian denoising), and the same network handles SR and JPEG deblocking.

**FFDNet** [21] feeds a **noise-level map** $M$ (same size as the image, value $\sigma$ at each pixel) to the network along with a downsampled version of the input. Changing $M$ at test time trades noise removal against detail preservation, and a non-uniform $M$ handles spatially varying noise. It is fast enough to run on a CPU.

**Real noise.** The **SIDD** dataset [22] provides real smartphone noisy images with clean references obtained by careful capture and processing of image bursts. Models trained only on AWGN do poorly on it, which pushed the field toward realistic noise synthesis and captured training data.

**General backbones.** MIRNet [30], **Restormer** [23] and **NAFNet** [24] are general restoration architectures that reported state-of-the-art results on SIDD as well as on deblurring and other tasks. They are covered in the backbones section.

### Code: a tiny residual denoiser

The snippet trains a five-layer DnCNN-style network on random $32 \times 32$ patches cut from four `skimage` images, with fresh Gaussian noise every step, then tests on an image it has never seen. On a laptop CPU it takes well under a minute (we measured about one minute on a heavily loaded machine).

```
import time, numpy as np, torch, torch.nn as nn
from skimage import data, img_as_float32
from skimage.color import rgb2gray

torch.manual_seed(0); torch.set_num_threads(2)
rng = np.random.default_rng(0)
sigma = 25 / 255                                    # noise level (on a [0,1] scale)
train_imgs = [img_as_float32(im) for im in (data.camera(), data.coins(), data.moon(),
 ↪  data.page())]
test_img = img_as_float32(rgb2gray(data.astronaut()))   # held out

def random_patches(n=32, p=32):
    batch = []
    for _ in range(n):
        im = train_imgs[rng.integers(len(train_imgs))]
        i, j = rng.integers(0, im.shape[0] - p), rng.integers(0, im.shape[1] - p)
        batch.append(im[i:i + p, j:j + p])
    return torch.from_numpy(np.stack(batch))[:, None]   # (n, 1, p, p)

class TinyDnCNN(nn.Module):
    """conv-ReLU, (depth-2) x conv-BN-ReLU, conv. Predicts the NOISE, not the image."""
    def __init__(self, depth=5, ch=32):
        super().__init__()
        layers = [nn.Conv2d(1, ch, 3, padding=1), nn.ReLU(inplace=True)]
        for _ in range(depth - 2):
            layers += [nn.Conv2d(ch, ch, 3, padding=1), nn.BatchNorm2d(ch),
             ↪  nn.ReLU(inplace=True)]
        layers += [nn.Conv2d(ch, 1, 3, padding=1)]
```

```
        self.body = nn.Sequential(*layers)
    def forward(self, y):
        return y - self.body(y)                          # x_hat = y - R(y): residual learning

net = TinyDnCNN()
opt = torch.optim.Adam(net.parameters(), lr=2e-3)
t0 = time.time()
for step in range(300):
    x = random_patches()
    y = x + sigma * torch.randn_like(x)                  # synthetic pairs, made on the fly
    loss = nn.functional.l1_loss(net(y), x)
    opt.zero_grad(); loss.backward(); opt.step()
print(f"trained in {time.time() - t0:.0f} s, last L1 = {loss.item():.4f}")

net.eval()
with torch.no_grad():
    x = torch.from_numpy(test_img)[None, None]
    y = x + sigma * torch.randn_like(x)
    x_hat = net(y).clamp(0, 1)
mse = lambda a, b: torch.mean((a - b) ** 2).item()
print(f"PSNR noisy {10*np.log10(1/mse(y.clamp(0,1), x)):.2f} dB -> denoised
 ↪ {10*np.log10(1/mse(x_hat, x)):.2f} dB")
# e.g. PSNR noisy 20.74 dB -> denoised 28.64 dB
```

Even this toy model, trained for 300 small steps on four images, gains about 8 dB on an unseen photo and beats a hand-tuned Gaussian filter (Figure 4). Drag the slider to compare the noisy input with the tiny network's output.



*Noisy* ($\sigma = 25$)



*Tiny residual CNN*

Slider comparing a grainy grayscale astronaut face with the same crop after denoising by a small CNN, which is clean but slightly smoothed.

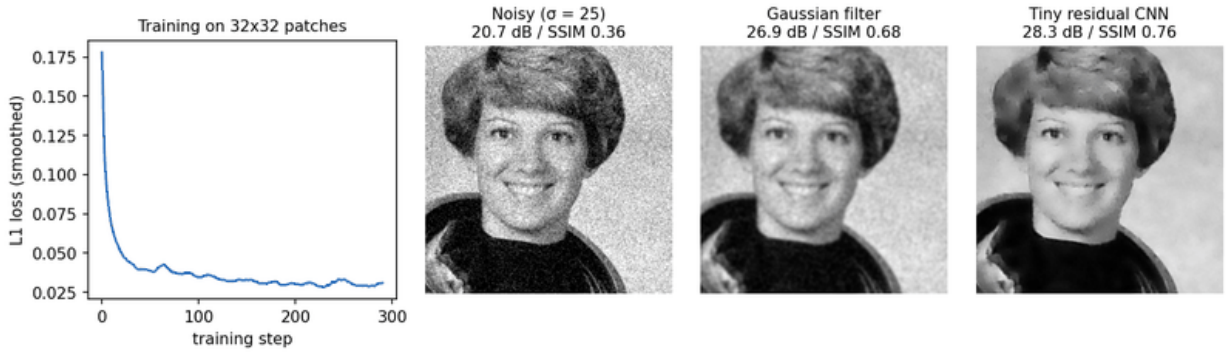



*Figure 4 — The tiny residual denoiser. PSNR and SSIM are computed on the full 512 × 512 test image. The CNN removes more noise than a Gaussian filter while keeping edges sharper, but it also flattens fine texture, a typical MSE/L1 behavior. Values vary slightly between runs and machines.*

## Dehazing

**Plain version.** Dehazing aims to recover contrast and color obscured by atmospheric scattering, especially in distant regions.

### The classical recipe and the dark channel prior

Classical methods estimate the unknowns of $I = Jt + A(1 - t)$ (the transmission map $t$ and the airlight $A$), then invert the model. The hard part is finding a reliable relationship that pins those unknowns down. The most famous one is He et al.'s **dark channel prior** [25]. In haze-free outdoor images, most local patches contain some pixel that is very dark in at least one color channel (shadows, dark or colorful objects):

$$J^{\text{dark}}(p) = \min_{c\in\{r,g,b\}} \Big( \min_{q\in\Omega(p)} J^c(q) \Big) \approx 0,$$

where $\Omega(p)$ is a small patch around $p$ and $J^c$ is color channel $c$. Applying the same min–min operation to the haze model and using $J^{\text{dark}} \approx 0$ gives a transmission estimate

$$\tilde{t}(p) = 1 - \omega \min_c \Big( \min_{q\in\Omega(p)} \frac{I^c(q)}{A^c} \Big),$$

where $\omega < 1$ (0.95 in the paper) keeps a little haze for depth perception. Then $\hat{J} = (I - A)/\max(\tilde{t}, t_0) + A$, with a small $t_0$ to avoid dividing by zero. The prior fails on large bright regions similar to the airlight (white walls, sky), and the coarse patch-based $t$ must be refined to follow edges.

### Deep dehazing: two schools

The post [1] divides deep dehazing into two schools, and the split still describes the field well.

**1. Physics-guided ("apply the formula").** Learn the unknown parameters, then invert the model. **DehazeNet** [26] uses a CNN (with Maxout units and a bounded ReLU) to regress the transmission $t$ from hazy patches. **AOD-Net** [27] simplifies the scattering model further by folding $t$ and $A$ into a single function $K$:

$$J(p) = K(p)\, I(p) - K(p) + b, \qquad K(p) = \frac{\frac{1}{t(p)}(I(p) - A) + (A - b)}{I(p) - 1},$$

where $b$ is a constant bias. A light-weight CNN predicts $K$, so the whole network is end-to-end, and it can be attached in front of a detector and trained jointly.

**2. Strong general architectures ("big learner").** Skip the physics and learn $I \mapsto J$ directly with a powerful network. **GridDehazeNet** [28] uses an attention-based multi-scale grid; **FFA-Net** [29] stacks channel- and pixel-attention blocks with adaptive feature fusion. These designs transfer well to other degradations, which is exactly the route taken by general restoration networks such as MIRNet [30]. Transformer dehazers followed: **DehazeFormer** [31] adapts the Swin Transformer to dehazing with a modified normalization layer, activation and spatial aggregation.

### Datasets

- **RESIDE** [32]: a large benchmark with synthetic indoor and outdoor hazy images (generated from depth with the scattering model) plus real hazy images for task-driven evaluation. Its SOTS test set is the most quoted dehazing benchmark.
- **O-HAZE** [33] and **Dense-Haze** [34]: real hazy/haze-free pairs captured with professional haze machines, outdoor and with dense haze respectively. They are small but expose the synthetic-to-real gap.
- **KITTI** [35] is *not* a dehazing dataset. It is an autonomous-driving benchmark (stereo, optical flow, odometry, 3-D detection, tracking) recorded around Karlsruhe. Some dehazing papers synthesize fog on its images (it has depth from LiDAR and stereo) or use it to measure how dehazing affects downstream detection.

## Desnowing

**Plain version.** Remove falling snow. It looks like a cousin of deraining, but it also behaves like dehazing.

Snow is harder than it looks. Snowflakes vary hugely in size and transparency, from opaque white blobs to semi-transparent streaks, and heavy snow also creates a **veiling effect** much like haze. So a desnowing model has to handle both the flakes and the fog-like veil. **DesnowNet** [36] introduced the **Snow100K** dataset (synthetic snow over real images, plus real snowy images) and a network that estimates both the translucency of snow and the residual it leaves. **JSTASR** [37] explicitly models snow size and transparency and adds a veiling-effect removal step. Desnowing is now usually studied inside "adverse weather" all-in-one models such as TransWeather [52].

## Deraining

**Plain version.** Remove rain from photos. "Rain" means two very different things: thin streaks falling through the air, and drops sitting on the lens or windshield.

### Rain streaks versus raindrops

- **Rain streaks** are thin, bright, oriented lines overlaid on the scene, modeled as $I = B + R$. Distant streaks accumulate into a haze-like veil. Methods learn to separate the two layers. An early influential deep method jointly detects and removes rain [38] and comes with widely used synthetic datasets of light and heavy rain.
- **Raindrops** on a glass surface act as tiny lenses: they hide and refract the background, are often out of focus, and have no consistent orientation. The attentive GAN of Qian et al. [39] uses an attention map (where are the drops?) to guide both the generator and the discriminator, and introduced a paired raindrop dataset captured through glass with and without water drops.

### Toward real scenes

Most deraining datasets are synthetic, and models trained on synthetic streaks often fail on real rain. The **NTIRE 2025 Day and Night Raindrop Removal for Dual-Focused Images** challenge [40] reflects the shift to real data: it uses the real-captured **Raindrop Clarity** dataset, covering daytime and nighttime scenes and both raindrop-focused and background-focused shots (14,139 training, 240 validation and 731 test images). The focus setting matters: when the camera focuses on the drops, the background is blurred too, so raindrop removal becomes partly deblurring.

## Low-light enhancement (brief)

**Plain version.** Make a dark photo look properly exposed, with visible details and natural colors, without blowing up the noise.

Training compares dark images with normally exposed references of the same scene, for example the **LOL** dataset of paired low/normal-light images introduced with **RetinexNet** [41]. Classical methods rely on contrast adjustment (histogram equalization) and Retinex-based models (and, less often, an inverted atmospheric scattering model). Deep methods follow two lines: Retinex-inspired networks that decompose $S = R \cdot L$ and adjust $L$ (RetinexNet [41]), and general well-designed networks. **Retinexformer** [42] (ICCV 2023), which its authors describe as the first Transformer-based low-light enhancement method, combines both: a one-stage Retinex formulation with illumination-guided attention. Low-light enhancement has enough of its own (zero-reference methods, curves, noise, color, video) to deserve a separate chapter: see DLCV 09C.

## Image fusion (brief)

**Plain version.** Combine several images of the same scene, each good at something different, into one image that has the best of each.

Three common settings appear in the post [1]: **infrared and visible** fusion (thermal targets from IR, textures and color from the visible camera), **multi-exposure** fusion (detail from under- and over-exposed shots, an HDR-like result) and **multi-focus** fusion (an all-in-focus image from shots focused at different depths). Unlike the other tasks, there is usually no ground-truth fused image, so deep fusion relies on unsupervised losses that keep gradients and intensities from the sources, or on GANs. The survey by Zhang et al. [43] organizes deep fusion methods by these scenarios and by learning strategy.

## Deblurring

**Plain version.** Recover a sharp image from one smeared by camera shake, object motion or defocus.

### Problem setup

Deblurring is the direct instance of $g = h * f + n$. If the kernel $h$ is known it is **non-blind** deblurring (the setting of the classical Wiener and CLS filters in DIP 5, and of learned non-blind solvers); if $h$ is unknown it is **blind** deblurring. Real motion blur in dynamic scenes is worse: different objects move differently and depth varies, so there is no single global $h$.

### Blind deblurring: estimating the kernel

**Plain version.** Blind deblurring must infer both a sharp image and the blur. Alternating methods refine one estimate while holding the other fixed.

**The joint problem.** For a spatially uniform blur, blind deconvolution solves

$$(\hat{f}, \hat{h}) = \arg\min_{f,h} \|h * f - g\|_2^2 + \lambda_f \, \phi(f) + \lambda_h \, \psi(h) \quad \text{subject to} \quad h \geq 0, \;\; \textstyle\sum_{u,v} h(u,v) = 1,$$

where $g$ is the blurred image, $f$ the unknown sharp image, $h$ the unknown kernel (a small window, e.g. 15×15 to 75×75 pixels for camera shake), $\phi$ an image prior (typically a sparse gradient penalty, because natural images have mostly flat regions and a few strong edges), $\psi$ a kernel prior, and $\lambda_f, \lambda_h > 0$ their weights. The constraints say that a blur kernel only redistributes light: it is non-negative and preserves total brightness. Even so, the kernel can only be recovered up to a small shift, because moving $f$ one way and $h$ the other leaves $h * f$ unchanged.

**Why it is badly ill-posed.** Far more unknowns ($f$ and $h$) than equations, and many pairs explain $g$ exactly. Worse, the most obvious pair is wrong: $h = \delta$ (no blur) and $f = g$ fits the data perfectly. Levin et al. [74] showed that with the usual sparse gradient priors, the blurry image often has a *lower* prior cost than the sharp one, because blur shrinks gradients and $\phi$ rewards small gradients. So naive joint MAP estimation over $(f, h)$ tends to return the no-blur solution. Their analysis points to a better-posed target: estimate only the kernel, averaging over all the sharp images that could have produced $g$. A kernel has few unknowns compared with the many pixel observations, so it is much better determined than the full image; once $\hat{h}$ is known, a non-blind deconvolution recovers $f$.

**Classical remedies.** The methods that worked in practice either approximate that averaging over images (Fergus et al. [75], with variational Bayes), **predict and select sharp edges** so that the kernel is estimated only from reliable gradients (Cho and Lee [77] sharpen the current estimate with a shock filter, as the code below does; Xu and Jia [78] keep only informative edges), or change the prior so that the sharp image wins (Krishnan et al. [79], Pan et al. [80]). All of them share the loop below.

**The standard recipe.** Almost all classical blind deblurring methods follow the same loop. Starting from a delta kernel at the coarsest level of an image pyramid, alternate

$$f^{(t+1)} = \arg\min_{f} \|h^{(t)} * f - g\|_2^2 + \lambda_f \, \phi(f),$$

$$h^{(t+1)} = \arg\min_{h \geq 0,\; \sum h = 1} \|h * \nabla \tilde{f}^{(t+1)} - \nabla g\|_2^2 + \lambda_h \|h\|_2^2,$$

where $t$ is the iteration, $\nabla$ the image gradient, and $\tilde{f}$ the latent image after edge prediction or selection (sharpened, with weak gradients set to zero). The kernel step works in the gradient domain because convolution commutes with differentiation, $\nabla(h * f) = h * \nabla f$, and gradients are where the blur information sits; with a quadratic penalty it has a closed form in the Fourier domain, followed by projection onto the constraints. After a few iterations the kernel is upsampled to the next pyramid level (**coarse-to-fine**), which avoids poor local minima because a large blur is only a few pixels wide at coarse scales. Finally, with $\hat{h}$ fixed, a **non-blind deconvolution** produces the output image: Wiener or constrained least squares filtering as in IP 05, or a hyper-Laplacian or learned non-blind solver.

**Deep learning and the kernel.** Networks entered blind deblurring in three ways: they **predict the kernel** (or a deconvolution filter) and then deconvolve classically [81]–[83]; they **fit a kernel and an image to one photo** with untrained generator networks and no training data, as SelfDeblur [85] does; or they **skip the kernel** and map blurred to sharp images end to end, like the multi-scale network of Nah et al. [44], DeblurGAN [86], Restormer and NAFNet.

*Online only: Classical and learned blind-deblurring methods — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/notes/classical-and-learned-blind-deblurring-methods/*

*Online only: Deep Image Prior: the network as the prior — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/deep-image-prior-the-network-as-the-prior/*

Which is preferable? Kernel-based methods suit **spatially uniform** blur and give an **interpretable** kernel that can be checked and reused, but they are slower and break when the blur varies across the image. Kernel-free networks handle **non-uniform** blur from object motion and depth and are the current state of the art on dynamic-scene benchmarks such as GoPro, but they generalize only as far as their training blur and give no explicit kernel. The survey of Zhang et al. [69] summarizes this shift (the full comparison is in the same note).

**Code: a toy alternating blind deconvolution.** The snippet blurs a $256 \times 256$ `camera` image with a curved $15 \times 15$ camera-shake kernel, then alternates a closed-form image step (a CLS-like filter with a quadratic gradient penalty) and a kernel step (least squares on the strongest 2% of shock-filtered gradients, projected onto $h \geq 0$, $\sum h = 1$). It runs in about a second on a CPU.

```
import numpy as np
from numpy.fft import fft2, ifft2
from scipy.ndimage import convolve, gaussian_filter, laplace
from skimage import data, img_as_float, transform

rng = np.random.default_rng(0)
f_true = transform.resize(img_as_float(data.camera()), (256, 256), anti_aliasing=True)
K = 15                                              # kernel support K x K
h_true = np.zeros((K, K))
for t in np.linspace(0, 1, 40):                     # a curved "camera-shake" path
    h_true[round(3 + 8 * t), round(2 + 10 * t + 2 * np.sin(3 * t))] += 1
h_true = gaussian_filter(h_true, 0.5); h_true /= h_true.sum()
g = convolve(f_true, h_true, mode="wrap") + 0.005 * rng.standard_normal(f_true.shape)

def otf(h, shape):                                  # centered kernel -> transfer function
    p = np.zeros(shape); p[:h.shape[0], :h.shape[1]] = h
```

```
    return fft2(np.roll(p, (-(h.shape[0] // 2), -(h.shape[1] // 2)), (0, 1)))

G = fft2(g)
Dx, Dy = otf(np.array([[0, -1, 1]]), g.shape), otf(np.array([[0], [-1], [1]]), g.shape)

def image_step(h, lam):        # f = argmin ||h*f - g||^2 + lam ||grad f||^2  (closed form,
 ↪  CLS-like)
    H = otf(h, g.shape)
    return np.real(ifft2(np.conj(H) * G / (np.abs(H)**2 + lam * (np.abs(Dx)**2 +
     ↪  np.abs(Dy)**2))))

def shock(f, n=5, dt=0.5):     # edge prediction: a shock filter turns soft edges into steps
    f = gaussian_filter(f, 0.7)
    for _ in range(n):
        gy, gx = np.gradient(f)
        f = f - dt * np.sign(laplace(f)) * np.hypot(gx, gy)
    return f

def kernel_step(f, keep=0.02, beta=1e-3):   # least squares in the gradient domain, strong
 ↪  edges only
    num = den = 0
    for D in (Dx, Dy):
        fx = np.real(ifft2(D * fft2(f)))
        fx[np.abs(fx) < np.quantile(np.abs(fx), 1 - keep)] = 0
        Fx = fft2(fx)
        num, den = num + np.conj(Fx) * D * G, den + np.abs(Fx)**2
    h = np.roll(np.real(ifft2(num / (den + beta))), (K // 2, K // 2), (0, 1))[:K, :K]
    h = np.maximum(h, 0)                            # project onto h >= 0 ...
    return h / h.sum()                              # ... and sum(h) = 1

def best_shift(fn):            # blind deconvolution recovers h (and f) only up to a small
 ↪  translation
    return min(fn(dy, dx) for dy in range(-3, 4) for dx in range(-3, 4))
kernel_error = lambda h: best_shift(lambda dy, dx: np.linalg.norm(np.roll(h, (dy, dx), (0,
 ↪  1)) - h_true)) / np.linalg.norm(h_true)
psnr = lambda f: -best_shift(lambda dy, dx: 10 * np.log10(np.mean((np.clip(np.roll(f, (dy,
 ↪  dx), (0, 1)), 0, 1) - f_true)[8:-8, 8:-8] ** 2)))

h = np.zeros((K, K)); h[K // 2, K // 2] = 1
h = gaussian_filter(h, 1.0); h /= h.sum()           # start from an almost-delta kernel ("no
 ↪  blur")
print(f"kernel error at start: {kernel_error(h):.2f}")
for _ in range(30):                                 # alternate: image update, then kernel
 ↪  update
    h = kernel_step(shock(image_step(h, lam=0.02)))
print(f"kernel error after 30 alternations: {kernel_error(h):.2f}")
print(f"PSNR  blurry {psnr(g):.2f} dB | estimated kernel {psnr(image_step(h, 1e-2)):.2f} dB"
      f" | true kernel {psnr(image_step(h_true, 1e-2)):.2f} dB")
# kernel error at start: 1.19
# kernel error after 30 alternations: 0.57
# PSNR  blurry 22.59 dB | estimated kernel 27.11 dB | true kernel 29.67 dB
```

The relative kernel error roughly halves, and the final non-blind step with the estimated kernel recovers most of the gap between the blurry input and the true-kernel result. Be honest about

what this toy is: the blur is synthetic, uniform and circular (so the FFT model is exact), the noise is low, and the parameters (`lam`, `keep`, the number of shock iterations) were tuned on this one image. There is no image pyramid, so larger kernels would fail, and with real photos you would also need boundary handling, saturated-pixel handling and robust losses. Both errors are measured after the best small shift, because of the translation ambiguity explained above.

### Data and milestones

Nah et al. [44] (CVPR 2017) addressed this with the **GoPro** dataset: they recorded sharp video with a high-speed camera and averaged consecutive frames to produce realistically blurred images, keeping the middle frame as the sharp target. Their network is **multi-scale**: it restores a coarse version first, then refines at finer scales, like the coarse-to-fine schemes of classical blind deblurring, but without estimating a kernel at all. GoPro is still the most common deblurring benchmark.

Today the most common baselines are the general backbones **Restormer** [23] and **NAFNet** [24]. NAFNet's striking finding is that the usual nonlinear activations (ReLU, GELU, Sigmoid) are not needed: they can be replaced by a simple element-wise multiplication of two feature halves ("SimpleGate") or removed, giving a "nonlinear activation free" network that is both simpler and stronger on GoPro and SIDD. Multiplying features is still nonlinear; the name concerns conventional activation functions. The deblurring survey by Zhang et al. [69] covers the wider landscape (blind and non-blind, video, defocus, losses).

## Inpainting (brief)

**Plain version.** Fill in missing or damaged regions (scratches, removed objects, holes) with plausible content.

Inpainting is the restoration task closest to image **generation**: inside a large hole there is nothing to restore, only something to invent that matches the surroundings. The degradation is a mask, $y = m \odot x$, where $m$ is 0 in the hole and 1 elsewhere. **Masked autoencoders** (MAE) [45] can be seen as inpainting used as a pre-training task: hide 75% of image patches and train a ViT to reconstruct them, which turns out to teach strong visual representations. For practical inpainting, **LaMa** [46] uses fast Fourier convolutions with an image-wide receptive field and training on large masks to fill big holes at high resolution; today, diffusion-based inpainting is the default for large, semantic edits.

## Matting (brief)

**Plain version.** Cut out the foreground (a person, their hair) with soft, partially transparent edges so it can be pasted on a new background.

Matting assumes each pixel is a blend of foreground and background,

$$I = \alpha F + (1 - \alpha)B,$$

where $\alpha \in [0, 1]$ is the opacity (alpha matte), $F$ the foreground color and $B$ the background color. With seven unknowns per pixel ($\alpha$ and two RGB colors) and three observations, the problem is badly underdetermined, so classical methods ask the user for a **trimap** (definite foreground, definite background, unknown band). It can be seen as a soft variant of segmentation. **Background Matting** [47] replaces the trimap with an extra photo of the background without the subject, and trains networks first on synthetic composites and then on real images with an adversarial loss that judges how natural the new composites look.

## General backbones

**Plain version.** Across all these tasks, a handful of network designs keep winning. They are general "image-to-image engines" that you can train on any of the degradations above.

| Backbone | Core idea | Why it suits restoration |
|---|---|---|
| U-Net [48] | Encoder–decoder with skip connections at every scale | Large receptive field from the downsampling path, fine detail from the skips |
| SwinIR [14] | Residual groups of shifted-window self-attention blocks at full resolution | Content-adaptive weights; strong for SR, denoising, JPEG |
| Restormer [23] | U-Net of transformer blocks; attention computed across **channels** (transposed attention) plus gated feed-forward | Cost linear in the number of pixels, so it scales to large images |
| NAFNet [24] | U-Net of simple conv blocks with LayerNorm, SimpleGate and simplified channel attention; no conventional activation functions; gates remain nonlinear | Very strong accuracy per FLOP; the default "simple baseline" |
| MambaIR / MambaIRv2 [49], [50] | Selective state-space (Mamba) scans with local enhancement and channel attention; v2 adds "attentive" non-causal scanning | Global receptive field at linear cost |

**Design reasons that recur.** (1) **Global residual**: predict $x - y$ (or $x$ minus an upsampled $y$) so the network learns only the missing part. (2) **No batch normalization** in most SR/restoration nets since EDSR [10]: BN statistics differ between training patches and full test images and hurt color/range fidelity. (3) **Large receptive field** at manageable cost: multi-scale U-Nets, windowed or channel attention, state-space scans. Restoration needs context (to tell a texture from noise) *and* full-resolution detail. (4) **Attention or gating** for content-adaptive processing: channel attention (RCAN), gating (NAFNet, Restormer). The post's "Extension" list predicted new architectures from ViT, flow, GAN and diffusion; by 2026 the state-space model joined CNNs and transformers as a third backbone option [49], [50].

## All-in-one and universal restoration

**Plain version.** Instead of training one model for rain, another for haze and another for noise, train *one* model that figures out what is wrong with the image and fixes it.

### Motivation

The post's "Extension" section lists all-in-one restoration as one of the hottest topics of 2023, starting around 2021. The motivation is practical: in a deployed system we do not know in advance which degradation (or mixture) a given image has, and weather can change or combine at any moment. A bank of task-specific models needs a reliable classifier in front and wastes memory; a single model that adapts is more convenient. The difficulty is **task interference**: different degradations need different, sometimes opposite operations (denoising smooths, deblurring sharpens), so naive multi-task training underperforms specialists.

### Representative methods

- **AirNet** [51] (CVPR 2022): a contrastive degradation encoder learns from paired restoration data and contrastive crop pairs *how* the image is degraded, without a user-supplied task label at inference and guides the restorer; its noise + rain + haze setting became the standard three-task benchmark.
- **TransWeather** [52] (CVPR 2022): one transformer for rain, fog and snow, with learnable weather-type queries.
- **PromptIR** [53] (NeurIPS 2023): lightweight prompt blocks inject degradation-specific information into a Restormer-style decoder.
- **AdaIR** [54] (ICLR 2025): different degradations affect different frequency bands, so it adapts low- and high-frequency features separately.
- **DA-CLIP** [55] (ICLR 2024): an adapted CLIP model supplies a degradation embedding that guides the restorer.
- **InstructIR** [56] (ECCV 2024): a natural-language instruction ("remove the rain") conditions the network.

*Online only: All-in-one methods in more detail — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/notes/all-in-one-methods-in-more-detail/*

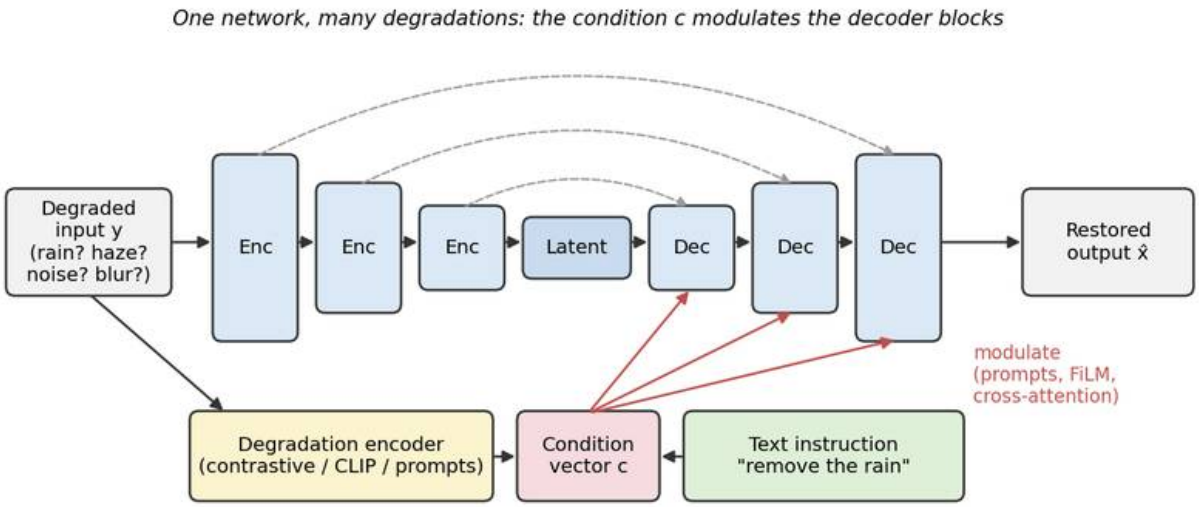



*Figure 5 — The common template behind all-in-one restoration: an encoder–decoder backbone plus a degradation-aware condition c, inferred from the image (AirNet, PromptIR, AdaIR, DA-CLIP) or given as text (InstructIR), that modulates the decoder.*

The post's original motivation is still the core; what changed is *who decides* the degradation type. It moved from fixed task labels to learned prompts, to vision–language embeddings and finally to human instructions, often backed by large pretrained models. The all-in-one survey [72] discusses this in detail.

**Generative and tool-based extensions.** AutoDIR [119] uses automatic quality assessment, latent diffusion and structural correction. PromptFix [122] combines user instructions with visual auxiliary descriptions and high-frequency preservation. AgenticIR [120] and 4KAgent [121] instead arrange and evaluate pretrained tools; their planning loop and repeated candidates have a different cost from one forward pass through an all-in-one backbone.

*Online only: Restormer: global channel relationships without pixel-wise attention — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/restormer-global-channel-relationships-without-pixel-wise-attention/*

*Online only: AirNet: infer the degradation before conditioning restoration — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/airnet-infer-the-degradation-before-conditioning-restoration/; TransWeather: learned weather queries for a shared restoration model — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/transweather-learned-weather-queries-for-a-shared-restoration-model/; PromptIR: input-conditioned feature prompts, without language instructions — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/promptir-input-conditioned-feature-prompts-without-language-instructions/; AdaIR: adaptively separate and exchange frequency information — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/adair-adaptively-separate-and-exchange-frequency-information/*

*Online only: DA-CLIP: separate content guidance from degradation guidance — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/da-clip-separate-content-guidance-*

*from-degradation-guidance/; InstructIR: route image features with a human instruction — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/instructir-route-image-features-with-a-human-instruction/; AutoDIR: diagnose degradation, guide diffusion, then correct structure — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/autodir-diagnose-degradation-guide-diffusion-then-correct-structure/; PromptFix: combine instructions, visual diagnosis and high-frequency preservation — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/promptfix-combine-instructions-visual-diagnosis-and-high-frequency-preservation/*

*Online only: AgenticIR: plan, inspect and revise a sequence of restoration tools — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/agenticir-plan-inspect-and-revise-a-sequence-of-restoration-tools/; 4KAgent: select restoration experts for a configured fidelity goal — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/4kagent-select-restoration-experts-for-a-configured-fidelity-goal/*

## Loss functions

**Plain version.** The loss decides what "good" means. Pixel losses reward being close to the truth on average; perceptual and adversarial losses reward looking real; frequency losses reward getting fine detail.

**Pixel losses.** With $\hat{x} = f_\theta(y)$,

$$\ell_2 = \frac{1}{P}\sum_p (\hat{x}_p - x_p)^2, \qquad \ell_1 = \frac{1}{P}\sum_p |\hat{x}_p - x_p|, \qquad \ell_\varepsilon = \frac{1}{P}\sum_p \sqrt{(\hat{x}_p - x_p)^2 + \varepsilon^2},$$

where $P$ is the number of pixel values and $\varepsilon$ a small constant that makes the last one a smooth, differentiable version of L1. L2 directly optimizes PSNR but penalizes large errors heavily and converges to the posterior mean. Zhao et al. [57] showed that the loss alone changes results noticeably with the architecture fixed, and that L1 and perceptually motivated losses (including combinations with SSIM) often beat L2 even on L2-related metrics. L1 is the default in most modern SR and restoration papers.

**Perceptual (feature) loss.** Compare images in the feature space of a pretrained classification network $\phi$ (usually VGG) [58]:

$$\ell_{\text{feat}} = \frac{1}{C_j H_j W_j}\|\phi_j(\hat{x}) - \phi_j(x)\|_2^2,$$

where $\phi_j$ is the feature map of layer $j$ with $C_j$ channels and size $H_j \times W_j$. It tolerates small misalignments and rewards matching edges and textures rather than exact values.

**Adversarial loss.** A discriminator $D_\psi$ learns to tell restored from real images and the restorer learns to fool it, e.g. $\ell_{\text{adv}} = -\log D_\psi(\hat{x})$. This pushes outputs onto the manifold of natural images (SRGAN [12], ESRGAN [13], Real-ESRGAN [4]) at the cost of possible hallucinated texture. In practice GAN-based restorers use a weighted sum such as $\ell = \ell_1 + \lambda_p \ell_{\text{feat}} + \lambda_a \ell_{\text{adv}}$ with weights $\lambda_p, \lambda_a$.

**Frequency losses.** Networks tend to fit low frequencies first and under-fit fine detail. Adding a loss on the Fourier transform $\mathcal{F}$, such as $\|\mathcal{F}(\hat{x}) - \mathcal{F}(x)\|_1$ on complex values or on amplitude, directly penalizes missing high frequencies. The **focal frequency loss** [59] goes further and adaptively up-weights the frequency components that are currently hardest to reconstruct. Frequency terms are now common in deblurring and all-in-one models.

## Evaluation metrics

**Plain version.** "Is the restored image close to the truth?" and "Does it look like a real photo?" are different questions, answered by different metrics, and you cannot maximize both at once.

### Full-reference metrics

**PSNR** (peak signal-to-noise ratio):

$$\mathrm{PSNR}(x, \hat{x}) = 10 \log_{10} \frac{\mathrm{MAX}^2}{\mathrm{MSE}(x, \hat{x})},$$

where MAX is the largest possible pixel value (1 or 255) and MSE is the mean squared error. It is measured in dB; +1 dB is a 21% reduction in MSE.

**SSIM** (structural similarity) [60] compares local means, variances and covariance:

$$\mathrm{SSIM}(a, b) = \frac{(2\mu_a\mu_b + C_1)(2\sigma_{ab} + C_2)}{(\mu_a^2 + \mu_b^2 + C_1)(\sigma_a^2 + \sigma_b^2 + C_2)},$$

computed over local windows $a$, $b$ of the two images and averaged, where $\mu$ are local means, $\sigma^2$ local variances, $\sigma_{ab}$ the local covariance and $C_1, C_2$ small constants for stability. It correlates better with perceived structure than PSNR but still prefers smooth results.

**LPIPS** [61] measures distance between deep-network features (calibrated on human similarity judgments). Lower is better, and it rewards realistic textures that PSNR penalizes.

```
import numpy as np
from scipy import ndimage as ndi
from skimage import data, img_as_float32
from skimage.metrics import peak_signal_noise_ratio, structural_similarity

x = img_as_float32(data.camera())
y = np.clip(x + np.random.default_rng(0).normal(0, 0.1, x.shape), 0, 1).astype(np.float32)
x_hat = ndi.gaussian_filter(y, 1.0)

def psnr(ref, est, peak=1.0):
    mse = np.mean((ref.astype(np.float64) - est) ** 2)
    return 10 * np.log10(peak ** 2 / mse)

print(f"PSNR  ours {psnr(x, x_hat):.2f} dB | skimage {peak_signal_noise_ratio(x, x_hat,
 ↪  data_range=1):.2f} dB")
```

```
print(f"SSIM  {structural_similarity(x, x_hat, data_range=1):.3f}")
# PSNR  ours 27.20 dB | skimage 27.20 dB
# SSIM  0.639
```

Always report the conventions (RGB or Y channel, border crop, data range, 8-bit rounding): differences in these alone can shift published PSNR by tenths of a dB.

### No-reference metrics

**Plain version.** A no-reference metric estimates quality from the restored image alone. It is useful when no clean reference exists, but cannot establish that recovered details are correct.

### Full-reference, reduced-reference and no-reference

Image quality assessment (IQA) methods are grouped by how much they know about the pristine image $x$:

- **Full-reference (FR)**: the whole $x$ is available, and the metric compares $\hat{x}$ with it pixel by pixel or feature by feature (PSNR, SSIM [60], LPIPS [61]). This is what synthetic benchmarks use.
- **Reduced-reference (RR)**: only a few summary features of $x$ are available (for example statistics sent alongside a compressed video), and the metric compares the same features computed on $\hat{x}$.
- **No-reference (NR, blind)**: only $\hat{x}$ is available. The metric must carry its own model of what "good" looks like.

Restoration needs NR metrics for three reasons. (1) **Real test images have no ground truth**: real-world SR, deraining and dehazing are evaluated on photos where nobody has the clean version. (2) **The reference itself can be imperfect or only one of many valid answers**: a GAN or diffusion output can look excellent yet have low PSNR because its texture differs from the reference's (the perception–distortion trade-off below). (3) **NR scores can steer systems at run time**, for example to choose among outputs or parameters when no reference exists. The price: an NR metric is a *model of human opinion*, and every model has blind spots.

### How NR metrics are validated: correlation with human opinion

**Plain version.** We trust an NR metric only if it ranks images the way people do. So researchers collect many human ratings on many images and check how well the metric's scores agree with them.

**Mean opinion score.** In a subjective study, each image $i$ is rated by several observers, and the average rating is its **mean opinion score**, $\mathrm{MOS}_i = \frac{1}{R}\sum_{r=1}^{R} s_{ir}$, where $s_{ir}$ is the score of rater $r$ and $R$ the number of raters. Some datasets publish a **differential** score (DMOS), based on the

difference between the ratings of the distorted image and its reference; for DMOS, lower usually means better.

**SRCC** (Spearman rank-order correlation) measures whether the metric puts the images in the same *order* as people:

$$\mathrm{SRCC} = 1 - \frac{6\sum_{i=1}^{n} d_i^2}{n\,(n^2-1)},$$

where $n$ is the number of test images and $d_i$ the difference between the rank of image $i$ by predicted score and its rank by MOS (this short form assumes no tied ranks; with ties one computes the Pearson correlation of the ranks). SRCC is unchanged by any monotonic transformation of the scores, so it measures **prediction monotonicity**.

**PLCC** (Pearson linear correlation) measures how *linearly* the scores agree:

$$\mathrm{PLCC} = \frac{\sum_{i=1}^{n}(q_i-\bar{q})(m_i-\bar{m})}{\sqrt{\sum_{i=1}^{n}(q_i-\bar{q})^2}\,\sqrt{\sum_{i=1}^{n}(m_i-\bar{m})^2}},$$

where $q_i$ is the metric's score for image $i$, $m_i$ its MOS, and $\bar{q}$, $\bar{m}$ their means. Because metric scales are arbitrary, it is common practice to first pass $q_i$ through a fitted monotonic (usually logistic) function and compute PLCC on the mapped scores, so PLCC measures **prediction accuracy** after calibration. Both lie in $[-1, 1]$. For "lower is better" metrics such as NIQE the raw correlation with MOS is negative, and papers report its absolute value. A metric is typically trained on part of a dataset and tested on the rest, and its cross-dataset correlation is the more honest test of generalization.

**Human-rated datasets.** NR metrics are trained and tested on image-quality datasets with human scores. What matters for restoration is *where the distortions come from*: synthetic damage (LIVE [104], TID2013 [105], KADID-10k [106]), real camera photos (KonIQ-10k [107], SPAQ [108]) or the outputs of restoration algorithms themselves (PIPAL [109]). Only datasets of the last kind contain the images a restoration paper actually needs to score.

*Online only: Human-rated IQA datasets — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/notes/human-rated-iqa-datasets/*

### Family 1: natural-scene-statistics (NSS) models

**Plain version.** Clean photos share hidden statistical habits. If you remove local brightness and local contrast from a photo, the leftovers follow a bell-shaped histogram of a typical shape, and neighboring leftovers are only weakly related. Blur, noise and compression change that shape in characteristic ways. NSS metrics measure the shape and ask how unusual it is.

**MSCN coefficients.** For a grayscale image $I$, the **mean-subtracted contrast-normalized** (MSCN) coefficients are [94]

$$\hat{I}(i,j) = \frac{I(i,j)-\mu(i,j)}{\sigma(i,j)+C}, \qquad \mu(i,j) = \sum_{k,l} w_{k,l}\, I(i+k, j+l), \qquad \sigma(i,j) = \sqrt{\sum_{k,l} w_{k,l}\,\big(I(i+k,j+l)-\mu(i,j)\big)^2},$$

where $(i, j)$ is a pixel position, $w$ a small circularly symmetric Gaussian window whose weights sum to 1, $\mu$ and $\sigma$ the local weighted mean and standard deviation, and $C$ a small constant (1 for 8-bit images) that prevents division by zero in flat regions. This is a divisive normalization, loosely modeled on contrast gain control in early human vision.

**The generalized Gaussian fit.** The histogram of $\hat{I}$ is modeled by a zero-mean generalized Gaussian distribution (GGD),

$$f(t; \alpha, \sigma^2) = \frac{\alpha}{2\beta\,\Gamma(1/\alpha)} \exp\!\big(-(|t|/\beta)^{\alpha}\big), \qquad \beta = \sigma\sqrt{\frac{\Gamma(1/\alpha)}{\Gamma(3/\alpha)}},$$

where $t$ is an MSCN value, $\alpha > 0$ the **shape** ($\alpha = 2$ is a Gaussian, $\alpha = 1$ a Laplacian; smaller $\alpha$ means a peakier histogram with heavier tails, larger $\alpha$ a flatter top), $\sigma^2$ the variance, $\beta$ the corresponding scale and $\Gamma$ the gamma function. A simple way to estimate $\alpha$ is **moment matching**: the ratio $\mathbb{E}[t^2]/(\mathbb{E}|t|)^2 = \Gamma(1/\alpha)\,\Gamma(3/\alpha)/\Gamma(2/\alpha)^2$ depends only on $\alpha$, so one computes the left side from the data and searches for the $\alpha$ that matches it.

**BRISQUE** [94] (Blind/Referenceless Image Spatial Quality Evaluator) computes the GGD shape and variance of the MSCN coefficients, plus parameters of *asymmetric* GGDs fitted to products of neighboring MSCN coefficients in four orientations (horizontal, vertical and the two diagonals), at two image scales. A support-vector regressor trained on human scores of synthetically distorted images maps these features to a quality score. Lower is better. It is **opinion-aware**: it needs human scores for training.

**NIQE** [62] (Natural Image Quality Evaluator) is **opinion-unaware**: it never sees human scores or distorted images. It computes similar NSS features on sharp patches, fits a multivariate Gaussian (MVG) with mean $\nu_1$ and covariance $\Sigma_1$ to features from a corpus of pristine images, fits another MVG $(\nu_2, \Sigma_2)$ to the test image's patches, and reports

$$D(\nu_1, \nu_2, \Sigma_1, \Sigma_2) = \sqrt{(\nu_1 - \nu_2)^{\top}\Big(\frac{\Sigma_1 + \Sigma_2}{2}\Big)^{-1}(\nu_1 - \nu_2)},$$

a Mahalanobis-like distance between the two Gaussians ($^{\top}$ is the transpose). Lower is better; zero means "statistically like the pristine corpus".

Two relatives appear often in restoration papers. **PIQE** [96] applies MSCN statistics block by block, needs no opinion scores, and also gives a map of where the image looks distorted. The **Perceptual Index** (PI) of the 2018 PIRM super-resolution challenge [63] combines NIQE with a score learned by Ma et al. [95] from human ratings of super-resolved images; lower PI means better perceptual quality, and many perceptual SR papers still report it.

*Online only: PIQE and the Perceptual Index — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/notes/piqe-and-the-perceptual-index/*

**Code: MSCN statistics in practice.** The snippet computes MSCN coefficients, the GGD shape by moment matching, the MSCN variance and the correlation between horizontal neighbors (a simplified version of BRISQUE's pairwise features) on `camera` and on blurred, noisy and over-sharpened versions:

```
import numpy as np
from scipy.ndimage import gaussian_filter
from scipy.special import gamma
from skimage import data, img_as_float

def mscn(img, sigma=7/6, C=1.0):
    """Mean-subtracted contrast-normalized coefficients of a grayscale image in [0, 255]."""
    mu = gaussian_filter(img, sigma)
    sd = np.sqrt(np.abs(gaussian_filter(img * img, sigma) - mu * mu))
    return (img - mu) / (sd + C)

def ggd_shape(x):
    """Moment-matching estimate of the generalized Gaussian shape alpha (2 = Gaussian)."""
    alphas = np.arange(0.2, 10.0, 0.001)
    r_theory = gamma(1 / alphas) * gamma(3 / alphas) / gamma(2 / alphas) ** 2
    r_data = np.mean(x ** 2) / np.mean(np.abs(x)) ** 2
    return alphas[np.argmin((r_theory - r_data) ** 2)], np.mean(x ** 2)

rng = np.random.default_rng(0)
clean = img_as_float(data.camera()) * 255
versions = {
    "clean": clean,
    "blur (sigma=2)": gaussian_filter(clean, 2),
    "noise (sigma=20)": np.clip(clean + rng.normal(0, 20, clean.shape), 0, 255),
    "over-sharpened": np.clip(clean + 2.0 * (clean - gaussian_filter(clean, 2)), 0, 255),
}
for name, img in versions.items():
    m = mscn(img)
    alpha, var = ggd_shape(m)
    rho = np.mean(m[:, :-1] * m[:, 1:]) / var           # correlation of horizontal neighbors
    print(f"{name:17s} GGD shape {alpha:4.2f} | MSCN variance {var:4.2f} | neighbor corr.
     ↪ {rho:+.2f}")
# clean             GGD shape 1.56 | MSCN variance 0.28 | neighbor corr. +0.12
# blur (sigma=2)    GGD shape 0.87 | MSCN variance 0.04 | neighbor corr. +0.89
# noise (sigma=20)  GGD shape 3.12 | MSCN variance 0.68 | neighbor corr. -0.10
# over-sharpened    GGD shape 2.35 | MSCN variance 0.43 | neighbor corr. +0.09
```

Each distortion pushes the features in its own direction (Figure 6). **Blur** removes local contrast, so most MSCN values collapse toward zero: tiny variance, a very peaky histogram (small $\alpha$) and strongly correlated neighbors. **Noise** adds independent energy: larger variance, a flat-topped histogram ($\alpha$ above 2) and neighbor correlation near zero (slightly negative, because subtracting the local mean couples neighbors). **Over-sharpening** moves the features *in the same direction as noise*. That is the core weakness of NSS metrics for restoration: they cannot tell restored texture from invented high-frequency energy. Note also that the absolute numbers depend on content and on $C$ (the clean `camera` image is not a unit Gaussian), which is why BRISQUE needs a trained regressor and NIQE a pristine reference corpus rather than fixed thresholds.

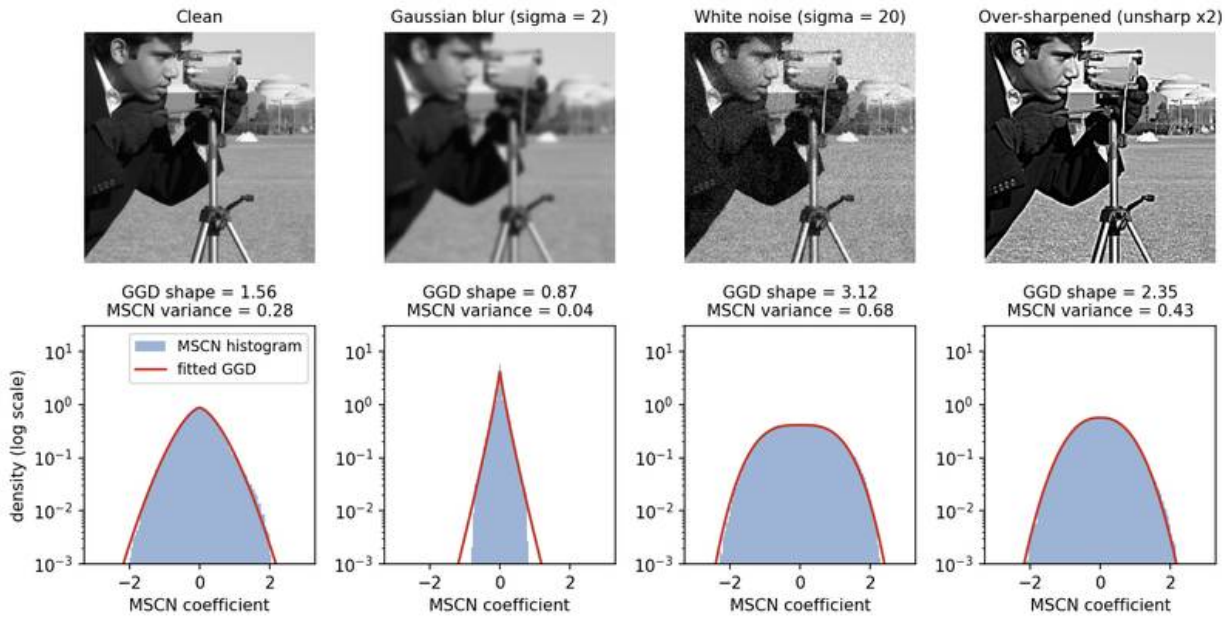



*Figure 6 — MSCN statistics, the basis of BRISQUE, NIQE and PIQE. Blur makes the histogram narrow and peaky; noise and over-sharpening both make it wider and flatter, so an NSS metric sees them as similar changes.*

### Family 2: learned deep models

**Plain version.** Instead of hand-made statistics, train a neural network to predict human ratings directly. Deep features capture content, semantics and complex artifacts far better, but the model only knows the kinds of images and opinions it was trained on.

Five widely used examples: **NIMA** [97] predicts the whole distribution of human ratings rather than one number; **HyperIQA** [98] adapts its quality rule to the image content, since the same blur is judged differently on a sky and on a face; **MUSIQ** [99] reads the image at its native resolution and aspect ratio, so no resizing changes the artifacts it judges; **MANIQA** [100] is a transformer that won the no-reference track of the NTIRE 2022 perceptual IQA challenge, whose data include restoration and GAN outputs; and **TOPIQ** [101] lets high-level semantics decide which low-level distortions matter.

All of these are **opinion-aware** and output "higher is better" scores, and each depends on its training dataset: a model trained on KonIQ-10k and the same architecture trained on PIPAL can rank restoration methods differently.

*Online only: How the learned NR assessors work — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/notes/how-the-learned-nr-assessors-work/*

### Family 3: vision-language and large multimodal models

**Plain version.** Models trained on huge collections of images and text already "know" what a good or bad photo looks like. You can ask them, either by comparing the image with text prompts or by literally asking a chatbot-style model to rate it.

**CLIP-IQA** [102] compares the CLIP embedding of the image with those of an opposite prompt pair such as "Good photo." and "Bad photo." and turns the two similarities into a score between 0 and 1 (higher is better). The basic version needs no opinion scores at all; other prompt pairs ("Sharp photo." / "Blurry photo.") probe single attributes, and the wording of the prompts changes the scores. **Q-Align** [103] fine-tunes a large multimodal model to answer the way human raters do, with a rating word from bad to excellent, and turns its probabilities for the five words into a score. Such models are strong across datasets but heavy, and they inherit the biases of their base model.

*Online only: CLIP-IQA and Q-Align scores — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/notes/clip-iqa-and-q-align-scores/*

### Distribution-level metrics

Some metrics score a *set* of outputs rather than one image. **FID** [64] compares the distribution of Inception features of the restored set with that of a set of real images; it was designed for generative models and is used for generative restoration such as diffusion SR, but it needs many images and says nothing about any single one. **IS** (Inception Score), mentioned in the post, measures the confidence and diversity of class predictions; it suits class-conditional generation rather than restoration and is rarely used there today.

### User studies and paired comparisons

**Plain version.** In the end, people decide what looks good. The two standard ways to ask them are to rate images on a scale or to choose the better of two. Such a study is also the yardstick every NR metric above is validated against, and it is the strongest evidence a new method can bring.

**Mean opinion score (MOS).** Each of $N$ raters gives an image a score $s_n$ on a fixed scale (often 1 to 5), and the MOS is their average, as defined for metric validation above. ITU-R BT.500 [111] standardizes such tests: viewing conditions, rating scales, single- and double-stimulus and comparison methods, and screening of unreliable raters.

**Paired comparison.** Absolute scales drift between raters; choosing the better of two images is easier and more consistent. The **Bradley–Terry** model [112] turns many such choices into one strength $\pi_i > 0$ per method $i$, fitted so that method $i$ is preferred over method $j$ with probability $\pi_i/(\pi_i + \pi_j)$. Sorting methods by their fitted strengths gives one consistent ranking, although each rater only ever compared two images at a time.

*Online only: Fitting the Bradley–Terry model — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/notes/fitting-the-bradleyterry-model/*

Good practice: hide method names, randomize the order and the left/right position, show the input for context, use enough images *and* raters, report confidence intervals or significance, and state the display and viewing conditions. A study with a handful of raters and hand-picked images is a demonstration, not evidence.

### Pitfalls

- **Metrics can be gamed.** A restorer that adds grain or sharpens edges can improve NSS scores, as the code above shows, without becoming more faithful. Any differentiable metric (and any learned IQA network) used as a training loss or for model selection can be optimized directly, and the network may learn to please the metric rather than people. If a method uses CLIP or an IQA network in its loss, evaluating it with the same model is not independent.
- **Domain mismatch.** Most opinion-aware metrics were trained on synthetic distortions (LIVE, TID2013, KADID-10k) or on camera photos (KonIQ-10k, SPAQ). Restoration outputs contain different artifacts: hallucinated textures, over-smoothing, GAN patterns. This mismatch was the explicit motivation for PIPAL [109], whose authors found that existing IQA methods could not fairly evaluate GAN-based restoration. For low light, the problems are exposure and color, which most NSS features normalize away; see the low-light-specific discussion in DLCV 09C.
- **Metrics disagree.** NSS, learned and vision-language metrics often rank the same methods differently, and all can disagree with people. Scores also depend on the implementation, checkpoint, color space, input size and whether the image is resized, so numbers copied from different papers are often not comparable.
- **"Higher NR quality" is not "more correct".** An NR metric cannot know whether a sharp face or line of text matches the real scene. A hallucinated license plate can score perfectly.

### Recommended practice

1. When a reference exists, lead with FR metrics (PSNR, SSIM, LPIPS) and add NR metrics as a complement; place your method on the perception–distortion plane (next subsection).
2. On real images without references, report **several NR metrics from different families** (for example NIQE or PI, one or two learned models such as MUSIQ, MANIQA or TOPIQ, and a vision-language one such as CLIP-IQA or Q-Align), and say which way each one points.
3. Use one shared implementation and state it, with versions and checkpoints. The open-source **IQA-PyTorch** toolbox (`pyiqa`) [110] reimplements most metrics in this section (NIQE, BRISQUE, PIQE, the Ma score and PI, NIMA, HyperIQA, MUSIQ, MANIQA, TOPIQ, CLIP-IQA, Q-Align, and FR metrics) behind one interface.
4. Do not tune on the judge: keep at least one metric that played no role in training or model selection.
5. Back important claims with a **user study** (mean opinion scores or paired comparisons, blind and randomized); the user-study subsection above describes MOS and Bradley–Terry paired-comparison protocols. When metrics disagree with each other or with people, say so.

### Summary table

The four families at a glance:

| Family | Examples | Needs human scores? | Better | Typical blind spot |
|---|---|---|---|---|
| NSS | NIQE, BRISQUE, PIQE, PI | NIQE and PIQE no; BRISQUE and PI (through Ma) yes | Lower | Rewards added grain and sharpening; normalizes away exposure and color |
| Learned deep | NIMA, HyperIQA, MUSIQ, MANIQA, TOPIQ | Yes | Higher | Depends on its training data; restoration artifacts differ from the training distortions |
| Vision-language | CLIP-IQA, Q-Align | CLIP-IQA (zero-shot) no; Q-Align yes | Higher | Prompt wording; biases of the base model; heavy |
| Distribution-level | FID | No (pretrained Inception features) | Lower | Needs many images; says nothing about a single one |

*Online only: Metric-by-metric summary table — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/notes/metric-by-metric-summary-table/*

### The perception–distortion trade-off

Blau and Michaeli [65] formalized the tension. Let $\Delta(x, \hat{x})$ be any distortion measure (MSE, $1 - $ SSIM, ...) and $d(p_X, p_{\hat{X}})$ a divergence between the distribution of real images and that of restored images (perceptual quality: small $d$ means outputs are statistically indistinguishable from real images). The **perception–distortion function** is

$$P(D) = \min_{p_{\hat{X}|Y}} d(p_X, p_{\hat{X}}) \quad \text{subject to} \quad \mathbb{E}[\Delta(X, \hat{X})] \leq D,$$

where the minimum is over all (possibly random) restoration methods. They proved that $P(D)$ is non-increasing and (for divergences convex in their second argument) convex: there is an unattainable region near the origin. A method cannot be both minimal-distortion and perfectly realistic. In particular, the MMSE estimator $\mathbb{E}[x \mid y]$ has the lowest distortion but a blurry, unrealistic output distribution; to look real you must accept higher MSE. GANs (and now diffusion models) are principled ways to move along the bound.

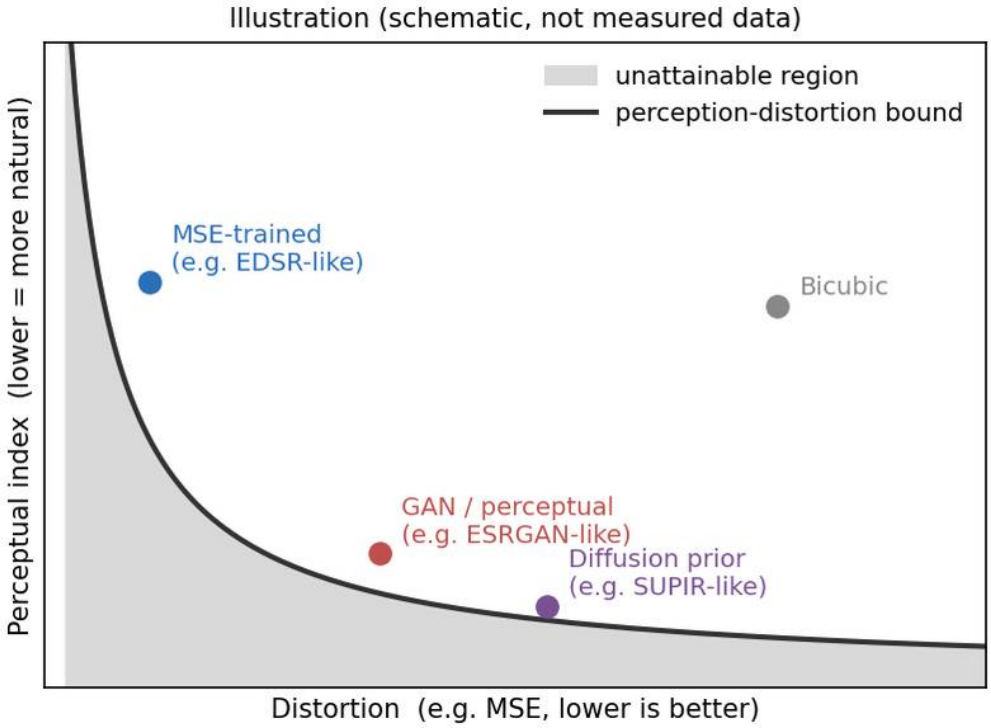



*Figure 7 — Schematic of the perception–distortion trade-off* [65] (*not measured data*). *The bound is convex and decreasing; methods trade fidelity* (*left*) *against realism* (*bottom*).

This is the update in the post's metric section: since diffusion models can generate plausible but unfaithful detail, the gap between fidelity (PSNR/SSIM) and perceptual quality has become central. The **NTIRE 2025** ×4 SR challenge therefore ran two tracks: a restoration track ranked by PSNR and a perceptual track ranked by a perceptual score [3], whereas the 2024 edition ranked by PSNR [2]. Practical advice: report both a distortion and a perceptual metric, and say where on the trade-off your method aims.

## NTIRE challenges

**Plain version.** Each year a CVPR workshop runs public competitions on fixed datasets and publishes a report describing the winning ideas: the easiest way to see what currently works.

The NTIRE challenge reports are a good way to follow restoration now that per-task leaderboards are gone. Each report describes the task, data, evaluation protocol and the methods of top teams. Examples cited in this tutorial:

- **NTIRE 2017 SR** [19]: introduced DIV2K; EDSR [10] won.
- **NTIRE 2024** ×4 **SR** [2]: bicubic ×4 on DIV2K, ranked by PSNR.
- **NTIRE 2025** ×4 **SR** [3]: separate restoration (PSNR) and perceptual tracks.
- **NTIRE 2025 raindrop removal** [40]: real day and night raindrop images with dual focus.

- **NTIRE 2024 and 2025 low-light enhancement**: the two editions of the low-light challenge are summarized in DLCV 09C.

Two patterns appear across recent reports: winning entries are usually ensembles or large hybrids of the backbones above (HAT-, Restormer-, NAFNet- or Mamba-style, sometimes with diffusion priors) trained with heavy data and compute, and challenges increasingly use real captured data and perceptual tracks.

## Modern view

The post closes with an extension list (medical images, multimodal, unsupervised, new architectures, all-in-one, low-level plus high-level, real-time, underwater). Most of these are now active areas. Here we summarize the survey literature task by task, then the 2023–2026 state of the art and the open problems.

### Survey papers

- **Super-resolution** (Wang, Chen and Hoi [66]): describes SR models as choices of upsampling position and operator, network design, losses and training, and flagged unsupervised, real-world SR as the main open direction.
- **Denoising** (Tian et al. [67]; Elad, Kawar and Vaksman [68]): the realism of the noise model is often the bottleneck; denoisers are now building blocks for other inverse problems and for diffusion models, and the frontier moved toward sampling from the posterior.
- **Deblurring** (Zhang et al. [69]): dynamic-scene deblurring moved from kernel estimation to end-to-end multi-scale networks trained on captured pairs such as GoPro.
- **Dehazing** (Gui et al. [70]) and **deraining** (Yang et al. [71]): physics-guided and physics-free designs coexist, and realistic degradation models and real-world evaluation matter more than further architecture tweaks on synthetic data.
- **All-in-one** (Jiang et al. [72]): a taxonomy by architecture, learning paradigm and core innovation; inconsistent task sets complicate comparisons, and generalization to unseen and mixed degradations is open.
- **Diffusion restoration** (Li et al. [73]): the best perceptual quality, but costly and with fidelity and hallucination concerns; few-step distillation and better evaluation are the key directions.

*Online only: Survey papers in detail — https://blog.shyandram.dev/en/learn/deep-learning/image-restoration/papers/survey-papers-in-detail/*

### State of the art, 2023–2026

1. **All-in-one and blind restoration became the mainstream setting**, not a niche topic. Methods moved from learned degradation representations (AirNet) to prompts (PromptIR), frequency-aware adaptation (AdaIR) and language guidance (DA-CLIP, InstructIR).

2. **Pretrained diffusion models as generative priors** transformed real-world SR and blind restoration (DiffBIR, SUPIR), and one-step variants (OSEDiff) cut the inference cost from tens or hundreds of steps to one.
3. **State-space models** (MambaIR, MambaIRv2) joined CNNs and transformers as backbone options with global receptive fields at linear cost.
4. **Evaluation caught up with generation**: challenges separate fidelity from perceptual tracks [3], and real captured datasets (SIDD, O-HAZE, Dense-Haze, Raindrop Clarity) are standard alongside synthetic benchmarks.

### Open problems

- **Real-world generalization.** Synthetic pipelines still miss real camera processing, spatially varying blur and mixed weather; models degrade on unseen combinations.
- **Fidelity versus hallucination.** Generative priors can invent text, faces or textures. For medical, forensic and scientific use this is unacceptable, and we lack good metrics for "faithful but sharp".
- **Evaluation.** PSNR/SSIM reward blur; no-reference metrics can be gamed; human studies are costly. Agreement between metrics and human preference for generative restoration is still weak.
- **Efficiency.** High-resolution, real-time (and video) restoration on phones and cameras remains a strong constraint, for transformers and diffusion models especially.
- **Task interference and scaling in all-in-one models**, and how to add a new degradation without retraining.
- **Low-level plus high-level.** Restoration that helps downstream detection or segmentation, rather than human viewers, is a different objective (task-driven evaluation, as in RESIDE [32]).

### A perspective from the original post

The post ends with two observations that still hold. First, because each task is simple to state, the race for state of the art is largely won by special architectures, which means a lot of compute and complicated experiments; but these tasks are very practical and easy to apply, so there is much room in **applications** and in **combining tasks** (exactly where all-in-one restoration went). Second, compared with high-level tasks the learning curve is gentle: a beginner can easily picture what the model is doing (a damaged image goes in, a cleaner one comes out), which makes low-level vision a good door into deep learning research.

## Key takeaways

- State the degradation assumptions before choosing a restorer or interpreting its output.
- Training pairs and degradation synthesis define what the network learns to undo.
- Match resolution paths, receptive fields and normalization to the restoration task.

- Conditioning can help one model handle multiple degradations; test unknown and mixed cases separately.
- Pixel accuracy, perceptual realism and input consistency answer different questions; report more than one.
- Blind restoration may estimate, condition on or implicitly handle degradation; joint estimation can favor degenerate solutions [74].
- Use task-appropriate datasets and treat no-reference scores as fallible quality estimates, not proof of recovered detail.

## Exercises

1. **Degradation order.** Using the `degrade` function, compare (a) blur → downsample → noise → JPEG with (b) noise → blur → downsample → JPEG at the same parameters. Visually and with the noise standard deviation measured on a flat patch, explain why the order matters for what the network has to learn.

   **Hint**

   In (b), the blur and downsampling filter the noise: its standard deviation drops and it becomes spatially correlated (smooth blotches rather than per-pixel grain). In (a), the noise stays white at the low resolution. A network trained only on (a) will under-remove correlated noise at test time. This is one reason Real-ESRGAN randomizes and repeats the stages.

2. **Pixel shuffle by hand.** A feature tensor has shape $(Cs^2, H, W) = (4, 2, 2)$ with $s = 2, C = 1$, and channel $k$ is filled with the constant $k$. Write the $4 \times 4$ output of pixel shuffle using the formula in the SR section, then check with `torch.nn.PixelShuffle(2)`.

   **Hint**

   Output pixel $(2h + i, 2w + j)$ takes channel $2i + j$. So every $2 \times 2$ block is $\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$, and the $4 \times 4$ output repeats that block four times. Each output block gathers one value from each of the four channels at the same low-resolution position.

3. **Residual or direct?** Modify `TinyDnCNN.forward` to return `self.body(y)` (predict the clean image directly) and train both versions with the same seed and steps. Compare PSNR and the loss curves. Then repeat with $\sigma = 5/255$. When does residual learning help most?

   **Hint**

   With few training steps, the residual version usually converges faster and reaches a higher PSNR, because at initialization it already outputs roughly $y$ (good PSNR) and only needs to learn a small correction. At low noise the direct version struggles most, since it must learn a near-identity mapping through several ReLU layers.

4. **Dark channel by numbers.** A hazy pixel has $I = (0.75, 0.78, 0.80)$, the darkest value of $I^c/A^c$ over its patch and channels is 0.70, and $A = (0.9, 0.9, 0.9)$. With $\omega = 0.95$ and $t_0 = 0.1$, compute $\tilde{t}$ and the recovered $\hat{J}$ for this pixel.

**Hint**

$\tilde{t} = 1 - 0.95 \times 0.70 = 0.335$. Then $\hat{J}^c = (I^c - 0.9)/0.335 + 0.9$, giving about $(0.452, 0.542, 0.601)$. The output is darker and more saturated than the input, as expected when the veil is removed.

5. **Perception versus distortion.** Take the tiny denoiser's output and add a small amount of fresh Gaussian noise ($\sigma = 3/255$) to it. Compute PSNR and SSIM before and after. The image may look *more* natural (less plastic) although both metrics drop. Relate this to the perception–distortion function $P(D)$.

**Hint**

The MMSE-like output lies at low distortion but far from the natural-image distribution (too smooth). Adding grain moves the output distribution closer to real photos (lower $d$) at the price of higher MSE: a move along, or toward, the convex bound, not past it.

6. **Design an all-in-one experiment.** You must build a single model for noise ($\sigma \in [0, 50]$), rain streaks and haze. Describe the training data mix, how you would condition the network (no labels at test time), the baselines you would compare against, and how you would detect task interference.

**Hint**

Mix datasets per task with balanced sampling; add a degradation encoder (contrastive as in AirNet, or prompts as in PromptIR) trained jointly. Compare with three single-task models of the same backbone and size, and with a version without conditioning. Interference shows up as the all-in-one model falling behind the single-task model on one task while matching it on others; also test on mixed degradations (rain + haze) unseen in training.

7. **Kernel estimation by hand.** In the blind-deconvolution snippet, (a) remove the `shock(...)` call and set `keep=1.0` (use all gradients) and compare the kernel error and final PSNR; (b) start from a $3 \times 3$ box kernel instead of the near-delta; (c) replace `h_true` by a straight 25-pixel motion path in a $31 \times 31$ window. Explain each result with the discussion of Levin et al. [74] and the coarse-to-fine strategy.

**Hint**

(a) With all gradients, the many weak gradients of the smoothed latent image dominate the least-squares fit, and the estimate no longer improves (in our run the kernel error grew instead of shrinking and the final PSNR stayed at the blurry input's level). Selecting a few strong, well-predicted edges is what makes the kernel step informative, which is the practical answer to the "no-blur" bias of naive joint MAP. (b) The loop is a non-convex alternation, so the start matters; a different start

can converge to a different local minimum. (c) A large kernel at full resolution is hard to find from a delta start: in our run the error dropped much less and most of the true-kernel PSNR was not recovered. Estimating the kernel on a downsampled image first (where it is only a few pixels long) and upsampling the result is what the image pyramid is for.

8. **Validating an NR metric.** Five restored images have mean opinion scores MOS $= (4.1, 3.2, 2.5, 1.8, 3.9)$ and NIQE scores $(3.4, 4.2, 5.1, 6.3, 3.0)$. (a) Compute SRCC by hand and interpret its sign. (b) Would SRCC change if you reported $\log(\text{NIQE})$ instead? Would PLCC? (c) Run the MSCN snippet on an over-sharpened version of a *denoised* image and explain why an NSS metric might rate it better than the plain denoised output.

**Hint**

(a) Ranking from smallest to largest, the MOS ranks of images 1–5 are $(5, 3, 2, 1, 4)$ and the NIQE ranks are $(2, 3, 4, 5, 1)$. The rank differences are $(3, 0, -2, -4, 3)$, so $\sum d_i^2 = 38$ and SRCC $= 1 - 6 \cdot 38/(5 \cdot 24) = -0.9$. The sign is negative because lower NIQE means better quality; one reports $|\text{SRCC}| = 0.9$. Only images 1 and 5 are swapped. (b) SRCC depends only on ranks, so any monotonic transform such as log leaves it unchanged; PLCC changes, which is why PLCC is usually computed after a fitted monotonic mapping. (c) Denoising leaves a smooth image whose MSCN statistics resemble mild blur (low variance, peaky histogram). Sharpening restores high-frequency energy and pushes the statistics back toward those of natural images, whether or not the added detail is real.

## References


1. Shyandram, "Low-Level Vision Task-Image Restoration 簡介," Medium post (in Traditional Chinese), 2023; updated 2026. link
2. Z. Chen, Z. Wu, E. Zamfir, K. Zhang, Y. Zhang, R. Timofte et al., "NTIRE 2024 Challenge on Image Super-Resolution (×4): Methods and Results," in *Proc. IEEE/CVF CVPR Workshops*, 2024. arXiv
3. Z. Chen, K. Liu, J. Gong, J. Wang, L. Sun et al., "NTIRE 2025 Challenge on Image Super-Resolution (×4): Methods and Results," in *Proc. IEEE/CVF CVPR Workshops*, 2025. arXiv
4. X. Wang, L. Xie, C. Dong and Y. Shan, "Real-ESRGAN: Training Real-World Blind Super-Resolution with Pure Synthetic Data," in *Proc. IEEE/CVF ICCV Workshops*, 2021. arXiv
5. E. H. Land and J. J. McCann, "Lightness and Retinex Theory," *Journal of the Optical Society of America*, vol. 61, no. 1, pp. 1–11, 1971. doi
6. C. Dong, C. C. Loy, K. He and X. Tang, "Image Super-Resolution Using Deep Convolutional Networks," *IEEE TPAMI*, 2016 (conference version ECCV 2014). arXiv
7. C. Dong, C. C. Loy and X. Tang, "Accelerating the Super-Resolution Convolutional Neural Network," in *Proc. ECCV*, 2016. arXiv
8. W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert and Z. Wang, "Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network," in *Proc. IEEE CVPR*, 2016. arXiv
9. A. Odena, V. Dumoulin and C. Olah, "Deconvolution and Checkerboard Artifacts," *Distill*, 2016. link

10. B. Lim, S. Son, H. Kim, S. Nah and K. M. Lee, "Enhanced Deep Residual Networks for Single Image Super-Resolution," in *Proc. IEEE CVPR Workshops*, 2017. arXiv
11. Y. Zhang, K. Li, K. Li, L. Wang, B. Zhong and Y. Fu, "Image Super-Resolution Using Very Deep Residual Channel Attention Networks," in *Proc. ECCV*, 2018. arXiv
12. C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang and W. Shi, "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network," in *Proc. IEEE CVPR*, 2017. arXiv
13. X. Wang, K. Yu, S. Wu, J. Gu, Y. Liu, C. Dong, C. C. Loy, Y. Qiao and X. Tang, "ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks," in *Proc. ECCV Workshops*, 2018. arXiv
14. J. Liang, J. Cao, G. Sun, K. Zhang, L. Van Gool and R. Timofte, "SwinIR: Image Restoration Using Swin Transformer," in *Proc. IEEE/CVF ICCV Workshops*, 2021. arXiv
15. X. Chen, X. Wang, J. Zhou, Y. Qiao and C. Dong, "Activating More Pixels in Image Super-Resolution Transformer," in *Proc. IEEE/CVF CVPR*, 2023. arXiv
16. X. Lin, J. He, Z. Chen, Z. Lyu, B. Dai, F. Yu, W. Ouyang, Y. Qiao and C. Dong, "DiffBIR: Towards Blind Image Restoration with Generative Diffusion Prior," arXiv:2308.15070, 2023. arXiv
17. F. Yu, J. Gu, Z. Li, J. Hu, X. Kong, X. Wang, J. He, Y. Qiao and C. Dong, "Scaling Up to Excellence: Practicing Model Scaling for Photo-Realistic Image Restoration In the Wild," in *Proc. IEEE/CVF CVPR*, 2024. arXiv
18. R. Wu, L. Sun, Z. Ma and L. Zhang, "One-Step Effective Diffusion Network for Real-World Image Super-Resolution," in *Proc. NeurIPS*, 2024. arXiv
19. E. Agustsson and R. Timofte, "NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study," in *Proc. IEEE CVPR Workshops*, 2017. doi
20. K. Zhang, W. Zuo, Y. Chen, D. Meng and L. Zhang, "Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising," *IEEE Transactions on Image Processing*, vol. 26, no. 7, pp. 3142–3155, 2017. arXiv
21. K. Zhang, W. Zuo and L. Zhang, "FFDNet: Toward a Fast and Flexible Solution for CNN-Based Image Denoising," *IEEE Transactions on Image Processing*, 2018. arXiv
22. A. Abdelhamed, S. Lin and M. S. Brown, "A High-Quality Denoising Dataset for Smartphone Cameras," in *Proc. IEEE/CVF CVPR*, 2018. doi
23. S. W. Zamir, A. Arora, S. Khan, M. Hayat, F. S. Khan and M.-H. Yang, "Restormer: Efficient Transformer for High-Resolution Image Restoration," in *Proc. IEEE/CVF CVPR*, 2022. arXiv
24. L. Chen, X. Chu, X. Zhang and J. Sun, "Simple Baselines for Image Restoration," in *Proc. ECCV*, 2022. arXiv
25. K. He, J. Sun and X. Tang, "Single Image Haze Removal Using Dark Channel Prior," in *Proc. IEEE CVPR*, 2009. doi
26. B. Cai, X. Xu, K. Jia, C. Qing and D. Tao, "DehazeNet: An End-to-End System for Single Image Haze Removal," *IEEE Transactions on Image Processing*, 2016. arXiv
27. B. Li, X. Peng, Z. Wang, J. Xu and D. Feng, "AOD-Net: All-in-One Dehazing Network," in *Proc. IEEE ICCV*, 2017. doi ·arXiv
28. X. Liu, Y. Ma, Z. Shi and J. Chen, "GridDehazeNet: Attention-Based Multi-Scale Network for Image Dehazing," in *Proc. IEEE/CVF ICCV*, 2019. arXiv
29. X. Qin, Z. Wang, Y. Bai, X. Xie and H. Jia, "FFA-Net: Feature Fusion Attention Network for Single Image Dehazing," in *Proc. AAAI*, 2020. arXiv

30. S. W. Zamir, A. Arora, S. Khan, M. Hayat, F. S. Khan, M.-H. Yang and L. Shao, "Learning Enriched Features for Real Image Restoration and Enhancement," in *Proc. ECCV*, 2020. arXiv
31. Y. Song, Z. He, H. Qian and X. Du, "Vision Transformers for Single Image Dehazing," *IEEE Transactions on Image Processing*, vol. 32, 2023. arXiv
32. B. Li, W. Ren, D. Fu, D. Tao, D. Feng, W. Zeng and Z. Wang, "Benchmarking Single-Image Dehazing and Beyond," *IEEE Transactions on Image Processing*, 2019. arXiv
33. C. O. Ancuti, C. Ancuti, R. Timofte and C. De Vleeschouwer, "O-HAZE: A Dehazing Benchmark with Real Hazy and Haze-Free Outdoor Images," in *Proc. IEEE/CVF CVPR Workshops*, 2018. doi
34. C. O. Ancuti, C. Ancuti, M. Sbert and R. Timofte, "Dense-Haze: A Benchmark for Image Dehazing with Dense-Haze and Haze-Free Images," in *Proc. IEEE ICIP*, 2019. doi
35. A. Geiger, P. Lenz and R. Urtasun, "Are We Ready for Autonomous Driving? The KITTI Vision Benchmark Suite," in *Proc. IEEE CVPR*, 2012. project page
36. Y.-F. Liu, D.-W. Jaw, S.-C. Huang and J.-N. Hwang, "DesnowNet: Context-Aware Deep Network for Snow Removal," *IEEE Transactions on Image Processing*, 2018. arXiv
37. W.-T. Chen, H.-Y. Fang, J.-J. Ding, C.-C. Tsai and S.-Y. Kuo, "JSTASR: Joint Size and Transparency-Aware Snow Removal Algorithm Based on Modified Partial Convolution and Veiling Effect Removal," in *Proc. ECCV*, 2020. doi
38. W. Yang, R. T. Tan, J. Feng, J. Liu, Z. Guo and S. Yan, "Deep Joint Rain Detection and Removal from a Single Image," in *Proc. IEEE CVPR*, 2017. arXiv
39. R. Qian, R. T. Tan, W. Yang, J. Su and J. Liu, "Attentive Generative Adversarial Network for Raindrop Removal from a Single Image," in *Proc. IEEE/CVF CVPR*, 2018. arXiv
40. X. Li, Y. Jin, X. Jin, Z. Wu, B. Li, Y. Wang et al., "NTIRE 2025 Challenge on Day and Night Raindrop Removal for Dual-Focused Images: Methods and Results," in *Proc. IEEE/CVF CVPR Workshops*, 2025. arXiv
41. C. Wei, W. Wang, W. Yang and J. Liu, "Deep Retinex Decomposition for Low-Light Enhancement," in *Proc. BMVC*, 2018. arXiv
42. Y. Cai, H. Bian, J. Lin, H. Wang, R. Timofte and Y. Zhang, "Retinexformer: One-stage Retinex-based Transformer for Low-light Image Enhancement," in *Proc. IEEE/CVF ICCV*, 2023. arXiv
43. H. Zhang, H. Xu, X. Tian, J. Jiang and J. Ma, "Image Fusion Meets Deep Learning: A Survey and Perspective," *Information Fusion*, vol. 76, pp. 323–336, 2021. doi
44. S. Nah, T. H. Kim and K. M. Lee, "Deep Multi-scale Convolutional Neural Network for Dynamic Scene Deblurring," in *Proc. IEEE CVPR*, 2017. doi ·arXiv
45. K. He, X. Chen, S. Xie, Y. Li, P. Dollár and R. Girshick, "Masked Autoencoders Are Scalable Vision Learners," in *Proc. IEEE/CVF CVPR*, 2022. arXiv
46. R. Suvorov, E. Logacheva, A. Mashikhin, A. Remizova, A. Ashukha, A. Silvestrov, N. Kong, H. Goka, K. Park and V. Lempitsky, "Resolution-robust Large Mask Inpainting with Fourier Convolutions," in *Proc. IEEE/CVF WACV*, 2022. arXiv
47. S. Sengupta, V. Jayaram, B. Curless, S. Seitz and I. Kemelmacher-Shlizerman, "Background Matting: The World is Your Green Screen," in *Proc. IEEE/CVF CVPR*, 2020. arXiv
48. O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Proc. MICCAI*, 2015. arXiv
49. H. Guo, J. Li, T. Dai, Z. Ouyang, X. Ren and S.-T. Xia, "MambaIR: A Simple Baseline for Image Restoration with State-Space Model," in *Proc. ECCV*, 2024. arXiv

50. H. Guo, Y. Guo, Y. Zha, Y. Zhang, W. Li, T. Dai, S.-T. Xia and Y. Li, "MambaIRv2: Attentive State Space Restoration," in *Proc. IEEE/CVF CVPR*, 2025. arXiv
51. B. Li, X. Liu, P. Hu, Z. Wu, J. Lv and X. Peng, "All-In-One Image Restoration for Unknown Corruption," in *Proc. IEEE/CVF CVPR*, 2022. code & paper
52. J. M. J. Valanarasu, R. Yasarla and V. M. Patel, "TransWeather: Transformer-based Restoration of Images Degraded by Adverse Weather Conditions," in *Proc. IEEE/CVF CVPR*, 2022. arXiv
53. V. Potlapalli, S. W. Zamir, S. Khan and F. S. Khan, "PromptIR: Prompting for All-in-One Image Restoration," in *Proc. NeurIPS*, 2023. arXiv
54. Y. Cui, S. W. Zamir, S. Khan, A. Knoll, M. Shah and F. S. Khan, "AdaIR: Adaptive All-in-One Image Restoration via Frequency Mining and Modulation," in *Proc. ICLR*, 2025. arXiv
55. Z. Luo, F. K. Gustafsson, Z. Zhao, J. Sjölund and T. B. Schön, "Controlling Vision-Language Models for Multi-Task Image Restoration," in *Proc. ICLR*, 2024. arXiv
56. M. V. Conde, G. Geigle and R. Timofte, "InstructIR: High-Quality Image Restoration Following Human Instructions," in *Proc. ECCV*, 2024. arXiv
57. H. Zhao, O. Gallo, I. Frosio and J. Kautz, "Loss Functions for Image Restoration with Neural Networks," *IEEE Transactions on Computational Imaging*, 2017 (arXiv title: "Loss Functions for Neural Networks for Image Processing"). arXiv
58. J. Johnson, A. Alahi and L. Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution," in *Proc. ECCV*, 2016. doi ·arXiv
59. L. Jiang, B. Dai, W. Wu and C. C. Loy, "Focal Frequency Loss for Image Reconstruction and Synthesis," in *Proc. IEEE/CVF ICCV*, 2021. arXiv
60. Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. doi
61. R. Zhang, P. Isola, A. A. Efros, E. Shechtman and O. Wang, "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric," in *Proc. IEEE/CVF CVPR*, 2018. arXiv
62. A. Mittal, R. Soundararajan and A. C. Bovik, "Making a 'Completely Blind' Image Quality Analyzer," *IEEE Signal Processing Letters*, 2013. doi
63. Y. Blau, R. Mechrez, R. Timofte, T. Michaeli and L. Zelnik-Manor, "The 2018 PIRM Challenge on Perceptual Image Super-resolution," in *Proc. ECCV Workshops*, 2018. arXiv
64. M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler and S. Hochreiter, "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium," in *Proc. NeurIPS*, 2017. arXiv
65. Y. Blau and T. Michaeli, "The Perception-Distortion Tradeoff," in *Proc. IEEE/CVF CVPR*, 2018. arXiv
66. Z. Wang, J. Chen and S. C. H. Hoi, "Deep Learning for Image Super-Resolution: A Survey," *IEEE TPAMI*, 2021. arXiv
67. C. Tian, L. Fei, W. Zheng, Y. Xu, W. Zuo and C.-W. Lin, "Deep Learning on Image Denoising: An Overview," *Neural Networks*, vol. 131, pp. 251–275, 2020. arXiv
68. M. Elad, B. Kawar and G. Vaksman, "Image Denoising: The Deep Learning Revolution and Beyond — A Survey Paper," *SIAM Journal on Imaging Sciences*, vol. 16, no. 3, pp. 1594–1654, 2023. doi ·arXiv
69. K. Zhang, W. Ren, W. Luo, W.-S. Lai, B. Stenger, M.-H. Yang and H. Li, "Deep Image Deblurring: A Survey," *International Journal of Computer Vision*, 2022. arXiv
70. J. Gui, X. Cong, Y. Cao, W. Ren, J. Zhang, J. Zhang, J. Cao and D. Tao, "A Comprehensive Survey

and Taxonomy on Single Image Dehazing Based on Deep Learning," *ACM Computing Surveys*, 2023. doi ·arXiv
71. W. Yang, R. T. Tan, S. Wang, Y. Fang and J. Liu, "Single Image Deraining: From Model-Based to Data-Driven and Beyond," *IEEE TPAMI*, 2021. doi ·arXiv
72. J. Jiang, Z. Zuo, G. Wu, K. Jiang and X. Liu, "A Survey on All-in-One Image Restoration: Taxonomy, Evaluation and Future Trends," *IEEE TPAMI*, 2025. arXiv
73. X. Li, Y. Ren, X. Jin, C. Lan, X. Wang, W. Zeng, X. Wang and Z. Chen, "Diffusion Models for Image Restoration and Enhancement: A Comprehensive Survey," *International Journal of Computer Vision*, 2025. arXiv
74. A. Levin, Y. Weiss, F. Durand and W. T. Freeman, "Understanding and Evaluating Blind Deconvolution Algorithms," in *Proc. IEEE CVPR*, 2009, pp. 1964–1971. doi
75. R. Fergus, B. Singh, A. Hertzmann, S. T. Roweis and W. T. Freeman, "Removing Camera Shake from a Single Photograph," *ACM Transactions on Graphics* (Proc. SIGGRAPH), vol. 25, no. 3, pp. 787–794, 2006. doi
76. Q. Shan, J. Jia and A. Agarwala, "High-quality Motion Deblurring from a Single Image," *ACM Transactions on Graphics* (Proc. SIGGRAPH), vol. 27, no. 3, 2008. doi
77. S. Cho and S. Lee, "Fast Motion Deblurring," *ACM Transactions on Graphics* (Proc. SIGGRAPH Asia), vol. 28, no. 5, 2009. doi
78. L. Xu and J. Jia, "Two-Phase Kernel Estimation for Robust Motion Deblurring," in *Proc. ECCV*, 2010, pp. 157–170. doi
79. D. Krishnan, T. Tay and R. Fergus, "Blind Deconvolution Using a Normalized Sparsity Measure," in *Proc. IEEE CVPR*, 2011, pp. 233–240. doi
80. J. Pan, D. Sun, H. Pfister and M.-H. Yang, "Blind Image Deblurring Using Dark Channel Prior," in *Proc. IEEE CVPR*, 2016, pp. 1628–1636. doi
81. J. Sun, W. Cao, Z. Xu and J. Ponce, "Learning a Convolutional Neural Network for Non-uniform Motion Blur Removal," in *Proc. IEEE CVPR*, 2015. arXiv
82. C. J. Schuler, M. Hirsch, S. Harmeling and B. Schölkopf, "Learning to Deblur," *IEEE TPAMI*, vol. 38, no. 7, pp. 1439–1451, 2016. doi ·arXiv
83. A. Chakrabarti, "A Neural Approach to Blind Motion Deblurring," in *Proc. ECCV*, 2016. arXiv
84. D. Ulyanov, A. Vedaldi and V. Lempitsky, "Deep Image Prior," in *Proc. IEEE/CVF CVPR*, 2018, pp. 9446–9454. doi ·arXiv
85. D. Ren, K. Zhang, Q. Wang, Q. Hu and W. Zuo, "Neural Blind Deconvolution Using Deep Priors," in *Proc. IEEE/CVF CVPR*, 2020. arXiv
86. O. Kupyn, V. Budzan, M. Mykhailych, D. Mishkin and J. Matas, "DeblurGAN: Blind Motion Deblurring Using Conditional Adversarial Networks," in *Proc. IEEE/CVF CVPR*, 2018. arXiv
87. K. Zhang, W. Zuo and L. Zhang, "Learning a Single Convolutional Super-Resolution Network for Multiple Degradations," in *Proc. IEEE/CVF CVPR*, 2018. arXiv
88. J. Gu, H. Lu, W. Zuo and C. Dong, "Blind Super-Resolution With Iterative Kernel Correction," in *Proc. IEEE/CVF CVPR*, 2019. arXiv
89. S. Bell-Kligler, A. Shocher and M. Irani, "Blind Super-Resolution Kernel Estimation using an Internal-GAN," in *Proc. NeurIPS*, 2019. link ·arXiv
90. Z. Luo, Y. Huang, S. Li, L. Wang and T. Tan, "Unfolding the Alternating Optimization for Blind Super Resolution," in *Proc. NeurIPS*, 2020. arXiv
91. K. Zhang, L. Van Gool and R. Timofte, "Deep Unfolding Network for Image Super-Resolution,"

in *Proc. IEEE/CVF CVPR*, 2020. arXiv
92. K. Zhang, J. Liang, L. Van Gool and R. Timofte, "Designing a Practical Degradation Model for Deep Blind Image Super-Resolution," in *Proc. IEEE/CVF ICCV*, 2021. arXiv
93. A. Liu, Y. Liu, J. Gu, Y. Qiao and C. Dong, "Blind Image Super-Resolution: A Survey and Beyond," *IEEE TPAMI*, 2022 (early access). doi ·arXiv
94. A. Mittal, A. K. Moorthy and A. C. Bovik, "No-Reference Image Quality Assessment in the Spatial Domain," *IEEE Transactions on Image Processing*, vol. 21, no. 12, pp. 4695–4708, 2012. doi
95. C. Ma, C.-Y. Yang, X. Yang and M.-H. Yang, "Learning a No-Reference Quality Metric for Single-Image Super-Resolution," *Computer Vision and Image Understanding*, vol. 158, pp. 1–16, 2017. doi ·arXiv
96. N. Venkatanath, D. Praneeth, M. C. Bh, S. S. Channappayya and S. S. Medasani, "Blind Image Quality Evaluation Using Perception Based Features," in *Proc. Twenty First National Conference on Communications* (*NCC*), 2015, pp. 1–6. doi ·MATLAB documentation
97. H. Talebi and P. Milanfar, "NIMA: Neural Image Assessment," *IEEE Transactions on Image Processing*, vol. 27, no. 8, pp. 3998–4011, 2018. doi ·arXiv
98. S. Su, Q. Yan, Y. Zhu, C. Zhang, X. Ge, J. Sun and Y. Zhang, "Blindly Assess Image Quality in the Wild Guided by a Self-Adaptive Hyper Network," in *Proc. IEEE/CVF CVPR*, 2020, pp. 3664–3673. doi
99. J. Ke, Q. Wang, Y. Wang, P. Milanfar and F. Yang, "MUSIQ: Multi-scale Image Quality Transformer," in *Proc. IEEE/CVF ICCV*, 2021, pp. 5128–5137. doi ·arXiv
100. S. Yang, T. Wu, S. Shi, S. Lao, Y. Gong, M. Cao, J. Wang and Y. Yang, "MANIQA: Multi-dimension Attention Network for No-Reference Image Quality Assessment," in *Proc. IEEE/CVF CVPR Workshops*, 2022, pp. 1190–1199. doi ·arXiv
101. C. Chen, J. Mo, J. Hou, H. Wu, L. Liao, W. Sun, Q. Yan and W. Lin, "TOPIQ: A Top-Down Approach From Semantics to Distortions for Image Quality Assessment," *IEEE Transactions on Image Processing*, vol. 33, pp. 2404–2418, 2024. doi ·arXiv
102. J. Wang, K. C. K. Chan and C. C. Loy, "Exploring CLIP for Assessing the Look and Feel of Images," in *Proc. AAAI Conference on Artificial Intelligence*, 2023. arXiv
103. H. Wu, Z. Zhang, W. Zhang, C. Chen, L. Liao, C. Li, Y. Gao, A. Wang, E. Zhang, W. Sun, Q. Yan, X. Min, G. Zhai and W. Lin, "Q-Align: Teaching LMMs for Visual Scoring via Discrete Text-Defined Levels," in *Proc. ICML*, PMLR 235, pp. 54015–54029, 2024. link ·arXiv
104. H. R. Sheikh, M. F. Sabir and A. C. Bovik, "A Statistical Evaluation of Recent Full Reference Image Quality Assessment Algorithms," *IEEE Transactions on Image Processing*, vol. 15, no. 11, pp. 3440–3451, 2006 (LIVE Image Quality Assessment Database). doi ·database
105. N. Ponomarenko, L. Jin, O. Ieremeiev, V. Lukin, K. Egiazarian, J. Astola, B. Vozel, K. Chehdi, M. Carli, F. Battisti and C.-C. J. Kuo, "Image Database TID2013: Peculiarities, Results and Perspectives," *Signal Processing: Image Communication*, vol. 30, pp. 57–77, 2015. doi ·database
106. H. Lin, V. Hosu and D. Saupe, "KADID-10k: A Large-scale Artificially Distorted IQA Database," in *Proc. Eleventh International Conference on Quality of Multimedia Experience* (*QoMEX*), 2019. doi ·database
107. V. Hosu, H. Lin, T. Sziranyi and D. Saupe, "KonIQ-10k: An Ecologically Valid Database for Deep Learning of Blind Image Quality Assessment," *IEEE Transactions on Image Processing*, vol. 29, pp. 4041–4056, 2020. arXiv

108. Y. Fang, H. Zhu, Y. Zeng, K. Ma and Z. Wang, "Perceptual Quality Assessment of Smartphone Photography," in *Proc. IEEE/CVF CVPR*, 2020, pp. 3674–3683. doi
109. J. Gu, H. Cai, H. Chen, X. Ye, J. Ren and C. Dong, "PIPAL: A Large-Scale Image Quality Assessment Dataset for Perceptual Image Restoration," in *Proc. ECCV*, 2020. arXiv
110. C. Chen and J. Mo, "IQA-PyTorch: PyTorch Toolbox for Image Quality Assessment," software (pip package `pyiqa`), 2022. GitHub
111. ITU-R, "Recommendation BT.500-15: Methodologies for the Subjective Assessment of the Quality of Television Images," 2023. ITU
112. R. A. Bradley and M. E. Terry, "Rank Analysis of Incomplete Block Designs: The Method of Paired Comparisons," *Biometrika*, vol. 39, no. 3–4, pp. 324–345, 1952. doi
113. D. Geman and C. Yang, "Nonlinear Image Recovery with Half-Quadratic Regularization," *IEEE Transactions on Image Processing*, vol. 4, no. 7, pp. 932–946, 1995. doi
114. S. V. Venkatakrishnan, C. A. Bouman and B. Wohlberg, "Plug-and-Play Priors for Model Based Reconstruction," in *Proc. IEEE Global Conference on Signal and Information Processing* (*GlobalSIP*), 2013, pp. 945–948. doi
115. K. Zhang, W. Zuo, S. Gu and L. Zhang, "Learning Deep CNN Denoiser Prior for Image Restoration," in *Proc. IEEE/CVF CVPR*, 2017, pp. 2808–2817. doi ·arXiv
116. A. Chambolle, "An Algorithm for Total Variation Minimization and Applications," *Journal of Mathematical Imaging and Vision*, vol. 20, no. 1–2, pp. 89–97, 2004. doi
117. A. Buades, B. Coll and J.-M. Morel, "A Non-Local Algorithm for Image Denoising," in *Proc. IEEE CVPR*, vol. 2, 2005, pp. 60–65. doi
118. D. Ulyanov, A. Vedaldi and V. Lempitsky, "Deep Image Prior," *International Journal of Computer Vision*, vol. 128, no. 7, pp. 1867–1888, 2020. doi ·arXiv
119. Y. Jiang, Z. Zhang, T. Xue and J. Gu, "AutoDIR: Automatic All-in-One Image Restoration with Latent Diffusion," in *Proc. ECCV, 2024*. paper
120. K. Zhu, J. Gu, Z. You, Y. Qiao and C. Dong, "An Intelligent Agentic System for Complex Image Restoration Problems," in *Proc. ICLR, 2025*. paper
121. Y. Zuo, Q. Zheng, M. Wu, X. Jiang, R. Li, J. Wang, Y. Zhang, G. Mai, L. V. Wang, J. Zou, X. Wang, M.-H. Yang and Z. Tu, "4KAgent: Agentic Any Image to 4K Super-Resolution," in *Proc. NeurIPS, 2025*. paper
122. Y. Yu, Z. Zeng, H. Hua, J. Fu and J. Luo, "PromptFix: You Prompt and We Fix the Photo," in *Proc. NeurIPS, 2024*. paper

# DLCV 09A • Non-Blind Image Super-Resolution

Non-blind single-image super-resolution: why it is ill-posed, why pixel losses give blur, the design decisions behind SRCNN, ESPCN, EDSR, RCAN, SwinIR and HAT, perceptual and diffusion SR, and how to evaluate the results honestly.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/super-resolution/

## What you'll learn

- Explain why super-resolution is ambiguous even with known degradation.
- Distinguish MSE's conditional mean from L1's coordinate-wise median.
- Compare upsampling, residual learning, normalization and attention choices.
- Interpret the fidelity–realism tradeoff of perceptual and generative objectives.
- Evaluate with explicit color, border and degradation protocols, and inspect artifacts.

## The big picture

Super-resolution makes a small image bigger *and* sharper. Stretching the pixels is easy; the hard part is filling in detail that the small image never recorded. A modern SR network fills it in from what it learned about natural images, and that is both its strength and its risk.

> **Explain like I'm five**
>
> Imagine a friend describes a picture to you over the phone: "a woman with short brown hair, smiling, in front of a light wall." You can draw a good picture from that, but you cannot draw *her* exact freckles, because the description did not include them. Many different drawings fit the same description. A careful artist draws soft, safe shapes where they are unsure (the result looks blurry). A bold artist draws crisp freckles anyway (the result looks real, but the freckles may be in the wrong place). Super-resolution networks are artists of one kind or the other. This chapter shows how they are built, how to tell them apart, and how to check their work.

Why it matters: phones zoom digitally, games and video are rendered small and upscaled, and satellites and microscopes are limited by their optics. SR is also where many low-level vision ideas were first tested, from residual learning to diffusion priors.

**Scope.** This chapter is about **non-blind** single-image SR: the way the low-resolution image was made is known and fixed, usually bicubic downscaling. The network designs were developed and compared in this setting, so it is the clearest place to learn them. Blind and real-world SR, where the degradation is unknown, is in DLCV 09A.1; video SR in DLCV 09A.3; GAN and diffusion priors for restoration in general in DLCV 09B; the generic losses and metrics in DLCV 09. You should know basic interpolation (IP 02) and aliasing (IP 04).

**The running example.** Most code and figures use one case: the astronaut image from `skimage.data` (512 × 512), shrunk by 4 with a known blur and block averaging, then enlarged back. All code uses NumPy, SciPy, scikit-image, scikit-learn and OpenCV and runs on a CPU in seconds.

### The problem: one input, many answers

**Question.** What exactly is lost when an image is shrunk, and what does a network trained to minimize pixel error do about it? **Short answer:** many high-resolution images shrink to the same small image, and squared error targets their mean, which can blur disagreeing edges and textures.

### Setup and notation

Single-image super-resolution (SISR) estimates a high-resolution (HR) image $x \in \mathbb{R}^{sH \times sW \times C}$ from one low-resolution (LR) image $y \in \mathbb{R}^{H \times W \times C}$, where $s$ is the **scale factor** $(2, 3, 4, \ldots)$, $H \times W$ the LR size and $C$ the number of color channels. The standard forward model, introduced in DLCV 09, is

$$y = (x * k) \downarrow_s + n,$$

where $k$ is a blur kernel (the camera's optics and pixel integration, or a deliberate anti-aliasing filter), $*$ is 2-D convolution applied to each channel, $\downarrow_s$ keeps one sample out of every $s \times s$ block, and $n$ is noise. In **non-blind** SR, $k$, $s$ and the noise level are known at inference. They can vary if the method accepts them as conditions. This chapter mainly studies fixed benchmark degradations, where the training and test operators match; a separate network is often trained for each $s$. Combining several shifted LR images of the same scene, the older multi-frame form of the problem [1], is the subject of DLCV 09A.3.

### Why the problem is ill-posed

Count the numbers. Each LR pixel summarizes $s^2$ HR pixels, so the HR image has $s^2$ times more unknowns than there are measurements. At $s = 4$, there are 16 HR values per LR measurement in the overall dimension count. Even a full-row-rank, noise-free linear degradation therefore leaves at least 15 unconstrained HR dimensions per measurement. Blur can mix neighbouring blocks, so these are degrees of freedom of the whole operator, not 15 independently invisible pixels in each block. In linear-algebra terms, the degradation operator $\mathcal{D}(x) = (x * k) \downarrow_s$ has a large **null space**: any HR pattern $z$ with $\mathcal{D}(z) = 0$ can be added to a solution without changing $y$. Fine textures, the exact position of an edge and small text live largely in that null space.

So SR needs a **prior** $p(x)$, a model of which HR images are plausible. With Bayes' rule, every HR candidate gets a posterior probability

$$p(x \mid y) \propto p(y \mid x)\, p(x),$$

where $p(y \mid x)$ is the likelihood (how well $\mathcal{D}(x)$ explains $y$ given the noise) and $p(x)$ the prior. Classical methods write the prior by hand; deep SR learns it from (LR, HR) pairs.

### The posterior-mean blur

Most SR networks are trained by minimizing a pixel loss over pairs. With the squared error, the best possible network outputs the **posterior mean**

$$\hat{x}_{\mathrm{MMSE}}(y) = \mathbb{E}[x \mid y] = \int x\, p(x \mid y)\, dx,$$

the minimum mean-squared-error (MMSE) estimate. With the L1 loss, an optimal deterministic predictor is a coordinate-wise posterior median instead. That is a different statistic, and it need not equal or resemble the mean for every posterior. When the posterior contains many sharp candidates that differ in the exact position of an edge or the phase of a texture, their average is a smooth compromise that is *not itself* a plausible image. This is the blur you see in MSE-trained SR, and more training data does not remove it: it is the correct answer to the question the loss asks.

A one-dimensional toy makes this concrete. An HR signal of 64 samples is a single step edge at an unknown position. We average blocks of 4 samples and add noise with standard deviation 0.2, then compute the exact posterior over the 32 possible edge positions, its mean, and one random sample from it.

```
import numpy as np

rng = np.random.default_rng(0)
n, s, sigma = 64, 4, 0.2                # HR length, scale factor, LR noise level
cand = np.arange(16, 48)                # possible edge positions t (the prior: uniform)
X = (np.arange(n)[None, :] >= cand[:, None]).astype(float)    # every HR step edge x_t
DX = X.reshape(len(cand), n // s, s).mean(axis=2)              # their LR versions D(x_t)

err_mmse, err_samp, jump_mmse, jump_samp = [], [], [], []
for _ in range(2000):
    t = rng.integers(len(cand))
    y = DX[t] + sigma * rng.standard_normal(n // s)               # observed LR signal
    logp = -((y - DX) ** 2).sum(axis=1) / (2 * sigma**2)          # log posterior over t
    post = np.exp(logp - logp.max()); post /= post.sum()
    x_mmse = post @ X                                             # posterior mean E[x | y]
    x_samp = X[rng.choice(len(cand), p=post)]                     # one posterior sample
    err_mmse.append(np.mean((x_mmse - X[t]) ** 2)); err_samp.append(np.mean((x_samp - X[t])
     ↪  ** 2))
    jump_mmse.append(np.diff(x_mmse).max()); jump_samp.append(np.diff(x_samp).max())

print(f"MSE   posterior mean {np.mean(err_mmse):.4f} | posterior sample
 ↪  {np.mean(err_samp):.4f} "
      f"| ratio {np.mean(err_samp) / np.mean(err_mmse):.2f}")
print(f"largest one-pixel jump (a real edge has 1.00): mean {np.mean(jump_mmse):.2f} | sample
 ↪  {np.mean(jump_samp):.2f}")
```

```
MSE   posterior mean 0.0072 | posterior sample 0.0143 | ratio 1.97
largest one-pixel jump (a real edge has 1.00): mean 0.48 | sample 1.00
```

The posterior mean has half the error of a posterior sample, but its edges are ramps: on average its largest one-pixel jump is 0.48, while every real edge (and every sample) jumps by 1.00. The

sample looks exactly like a real edge, only sometimes in the wrong place. The ratio of about 2 is not a coincidence. Blau and Michaeli proved that an estimator whose outputs have exactly the distribution of real images (for example, a posterior sampler) never needs more than twice the MSE of the MMSE estimator, which is 3 dB of PSNR [2]. Figure 1 shows one example and the whole range between the two estimators.

*Online only: Why a posterior sample costs twice the MMSE — https://blog.shyandram.dev/en/learn/deep-learning/super-resolution/notes/why-a-posterior-sample-costs-twice-the-mmse/*

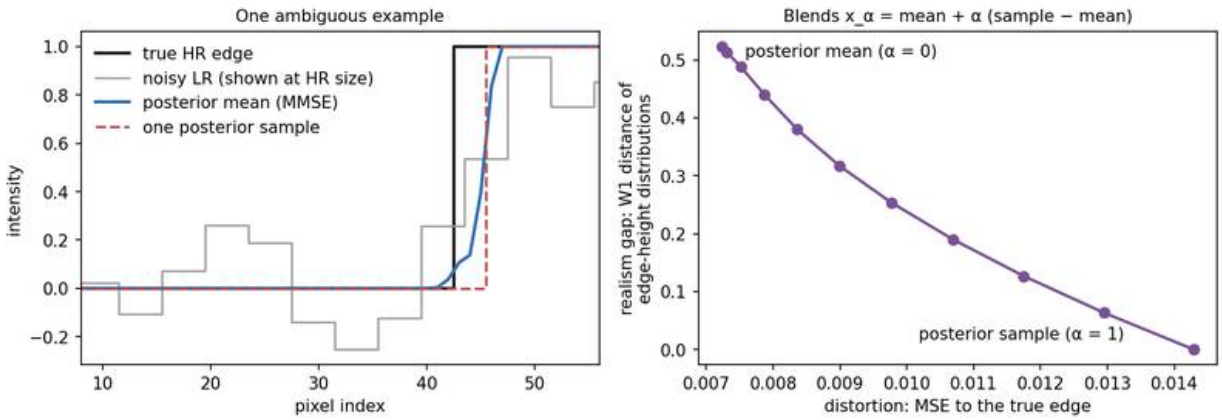



*Figure 1 — The posterior-mean blur on a 1-D step-edge toy (×4, noise $\sigma = 0.2$). Left: one ambiguous example. Right: blending the posterior mean with a posterior sample trades distortion (MSE) for realism (distance between the distribution of edge heights in the outputs and in real edges). No blend is best on both axes.*

This toy is the thread of the chapter: MSE-trained networks target the posterior mean, while L1-trained networks target coordinate-wise medians. Both choose a central prediction rather than describe all possible answers. Perceptual and adversarial losses favour plausible-looking detail; stochastic diffusion models can generate different candidates. Evaluation must distinguish fidelity, realism and uncertainty.

### What the low-resolution image has lost: the frequency view

**Plain version.** Downsampling can alias fine patterns and weaken surviving detail. SR must both restore weakened structure and infer information the input no longer determines.

The LR grid can only represent frequencies up to the LR **Nyquist limit**, $1/(2s)$ cycles per HR pixel (0.125 at $s = 4$), where $s$ is the scale factor. Higher HR frequencies either fold onto lower ones (**aliasing**, IP 04) or, if the blur $k$ removed them first, vanish. So content above the limit must be unfolded or invented with a prior, and content below it, weakened by the blur, must be restored (Figure 2).

*Online only: Aliasing in the LR spectrum — https://blog.shyandram.dev/en/learn/deep-learning/super-resolution/notes/aliasing-in-the-lr-spectrum/*

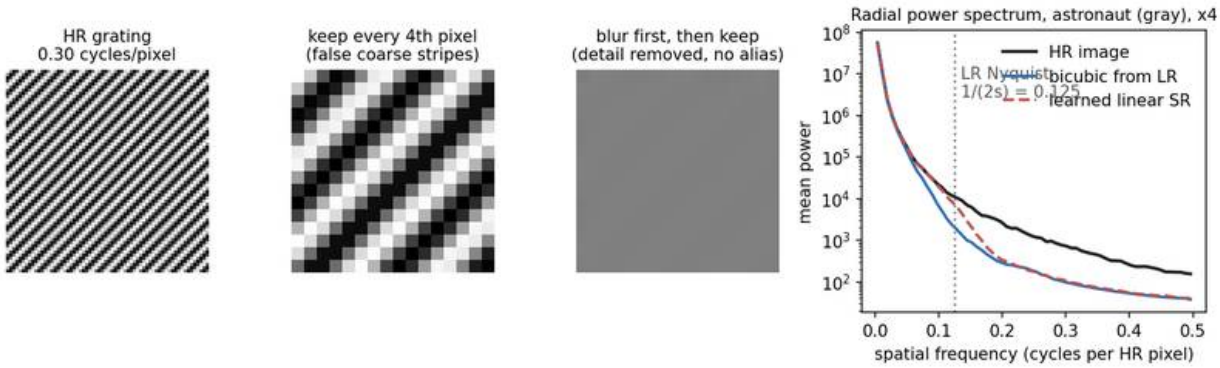



*Figure 2 — Left: a grating at 0.30 cycles per pixel decimated by 4 without a prefilter turns into false coarse stripes (each axis component of about 0.21 cycles per pixel becomes about −0.15 cycles per LR pixel); blurring first removes it instead. Right: radial power spectra at ×4. Interpolation cannot determine the missing true HR spectrum from these samples. Non-ideal interpolation kernels may still produce spectral replicas or high-frequency artifacts; the blur also weakens frequencies below the LR Nyquist limit (dotted line). The learned linear model of the classical-baselines section recovers much of the band below the limit and little above it.*

### The benchmark setting: a known degradation

**Question.** How are training and test pairs made in non-blind SR, and how do we measure the result? **Short answer:** shrink HR images with one fixed, known operator, and compare the enlarged result with the original by PSNR on the luminance channel.

Most SR benchmarks create LR images with MATLAB's `imresize` at its default settings, which use bicubic interpolation and, when shrinking, an anti-aliasing filter that is on by default [3], [4]. There is no added noise, no camera blur and no compression. This "bicubic" setting is clean and reproducible, which made years of results comparable. A model trained on it has learned to invert exactly that one operator.

Our running example uses a different but equally *known* operator, written out step by step: a Gaussian blur with $\sigma = 1.2$ HR pixels (the lens), the average of each $4 \times 4$ block (one larger sensor pixel), a little Gaussian noise ($\sigma = 0.01$) and rounding to 8 bits. The snippet enlarges the result with four standard interpolators and reports PSNR as SR papers do, on the luminance (Y) channel with a border of $s$ pixels removed (see Evaluating SR).

```
import numpy as np, cv2
from scipy import ndimage as ndi
from skimage import data, img_as_float64

def degrade(hr, s=4, blur=1.2, noise=0.01, rng=None):
    """hr: (H, W, 3) in [0, 1], H and W divisible by s. Returns the LR image y."""
    rng = rng or np.random.default_rng(0)
    x = ndi.gaussian_filter(hr, sigma=(blur, blur, 0))          # 1) lens blur
    H, W, C = x.shape
```

```
    x = x.reshape(H // s, s, W // s, s, C).mean(axis=(1, 3))      # 2) s x s pixel
     ↪  integration + decimation
    x = x + noise * rng.standard_normal(x.shape)                  # 3) sensor noise
    return np.round(np.clip(x, 0, 1) * 255) / 255                 # 4) 8-bit quantization

def y_channel(rgb):
    """BT.601 luma on the [16, 235] scale (the MATLAB rgb2ycbcr convention)."""
    return 16 + rgb @ np.array([65.481, 128.553, 24.966])

def psnr(a, b, shave, peak):
    if shave:
        a, b = a[shave:-shave, shave:-shave], b[shave:-shave, shave:-shave]
    return 10 * np.log10(peak**2 / np.mean((a - b) ** 2))

hr = img_as_float64(data.astronaut())                             # (512, 512, 3)
lr = degrade(hr)                                                  # (128, 128, 3)
print("LR shape:", lr.shape)
for name, flag in [("nearest", cv2.INTER_NEAREST), ("bilinear", cv2.INTER_LINEAR),
                   ("bicubic", cv2.INTER_CUBIC), ("lanczos", cv2.INTER_LANCZOS4)]:
    sr = cv2.resize(lr, (512, 512), interpolation=flag)
    sr = np.clip(sr, 0, 1)
    y_hr, y_sr = y_channel(hr), y_channel(sr)
    print(f"{name:8s}  Y, shave 4: {psnr(y_hr, y_sr, 4, 255):.2f} dB | "
          f"Y, no shave: {psnr(y_hr, y_sr, 0, 255):.2f} dB | RGB, shave 4: {psnr(hr, sr, 4,
           ↪  1):.2f} dB")
```

```
LR shape: (128, 128, 3)
nearest   Y, shave 4: 24.62 dB | Y, no shave: 24.66 dB | RGB, shave 4: 23.06 dB
bilinear  Y, shave 4: 25.35 dB | Y, no shave: 25.40 dB | RGB, shave 4: 23.83 dB
bicubic   Y, shave 4: 26.36 dB | Y, no shave: 26.40 dB | RGB, shave 4: 24.81 dB
lanczos   Y, shave 4: 26.50 dB | Y, no shave: 26.54 dB | RGB, shave 4: 24.94 dB
```

The ranking is the same under every protocol, but Y and RGB differ by about 1.5 dB, so only compare numbers measured the same way.

Real photos go through more than this; Figure 3 adds JPEG compression to the chain. Non-blind SR fixes the whole chain in advance. When it is unknown or varies from photo to photo, the problem becomes blind or real-world SR (DLCV 09A.1).




*Figure 3 — A blur → downsample → noise → JPEG chain at ×4. The non-blind setting of this chapter fixes the chain and knows it; DLCV 09A.1 treats the case where it is unknown. The LR panels are enlarged with*

*nearest-neighbor interpolation only for display.*

### Classical baselines

**Question.** What can be done without a deep network, and why is it not enough? **Short answer:** interpolation spreads the existing samples with a fixed filter; learned linear filters add some lost detail back; neither can model the variety of natural images.

#### Interpolation: a fixed linear filter

Interpolation places the LR samples on the HR grid and fills the gaps with a weighted sum of nearby samples. In one dimension,

$$\hat{x}(t) = \sum_{u} y[u]\, \phi\left( \frac{t + \frac{1}{2}}{s} - \frac{1}{2} - u \right),$$

where $t$ is an HR pixel index, $u$ an LR pixel index, $\phi$ the **interpolation kernel** and the half-pixel terms align the centers of LR and HR pixels (the convention OpenCV uses). In 2-D the kernel is applied along rows and then columns. IP 02 derives the common kernels:

- **Nearest neighbor** and **linear**: a box of width 1 (blocky) and a triangle of width 2 (soft).
- **Cubic convolution** (Keys) [5]: a piecewise cubic of width 4 with a free parameter $a$; Keys showed that $a = -\frac{1}{2}$ gives the best approximation order in that family. Its small negative lobes sharpen edges slightly. OpenCV's `INTER_CUBIC` uses $a = -0.75$ instead (Exercise 1).
- **Lanczos**: $\phi(t) = \mathrm{sinc}(t)\,\mathrm{sinc}(t/a)$ for $|t| < a$ and 0 otherwise, with $\mathrm{sinc}(t) = \sin(\pi t)/(\pi t)$ and $a$ the number of lobes kept on each side [6]; OpenCV's `INTER_LANCZOS4` uses $a = 4$.

Wider kernels approximate the ideal sinc better (Figure 4), which is why Lanczos scored highest above (26.50 dB versus 26.36 dB for bicubic). These fixed-kernel interpolators apply the same linear reconstruction rule to an edge and a flat region (away from boundary effects). They cannot identify the missing true HR frequencies from the LR samples alone. Finite kernels can nevertheless introduce spectral replicas and ringing; an output containing high frequencies is not proof that the original detail was recovered (Figure 5).

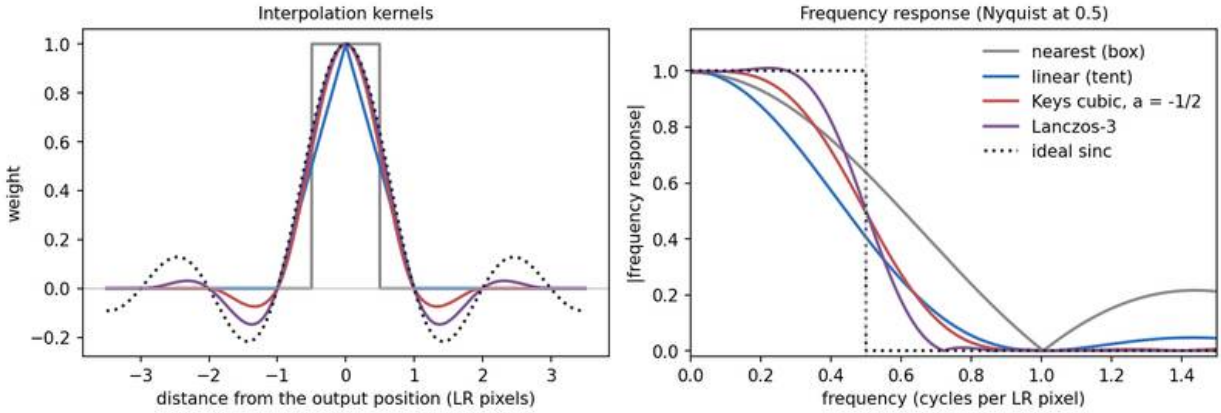



*Figure 4 — Interpolation kernels (left) and the magnitude of their frequency responses (right). A good interpolator passes frequencies below the LR Nyquist limit (0.5 cycles per LR pixel) and blocks the spectral copies above it.*




*Figure 5 — ×4 interpolation of the running example (blur $\sigma = 1.2$, noise $\sigma = 0.01$). Interpolation redistributes the information in the LR image but cannot restore what the blur and decimation removed.*

### Learning from examples: from patch pairs to linear filters

**Plain version.** Learn patch mappings from paired examples, then use precomputed linear corrections to restore similar patches efficiently.

Example-based methods took (LR patch, HR patch) pairs from training images [7] or from the input itself, because small patches recur across scales of one image [8]; sparse coding compressed the pairs into two matched dictionaries [9]. Anchored neighborhood regression (ANR) and A+ replaced the search by precomputed ridge regressors [10], [11], and RAISR learned one filter per bucket of gradient angle, strength and coherence [12]. The common structure is

$$\hat{r} = W_{c(p)}^{\top}\tilde{p}, \qquad W_c = \arg\min_{W} \sum_{i \in c} \|W^{\top}\tilde{p}_i - r_i\|_2^2 + \lambda \|W\|_F^2 = (P_c^{\top}P_c + \lambda I)^{-1} P_c^{\top} R_c,$$

where $p$ is an LR patch, $\tilde{p}$ its feature vector, $c(p)$ the class (atom or hash bucket) it belongs to, $r$ the HR detail to predict, $(\tilde{p}_i, r_i)$ the training pairs of class $c$ stacked as the rows of $P_c$ and $R_c$, $\lambda$ the ridge weight and $\|\cdot\|_F$ the Frobenius norm.

*Online only: Example-based SR and sparse coding in detail — https://blog.shyandram.dev/en/learn/deep-learning/super-resolution/notes/example-based-sr-and-sparse-coding-in-detail/*

The snippet implements the simplest version, one class, on the running example in grayscale. For every LR pixel it takes the $7 \times 7$ LR patch around it (mean removed, plus a bias) and predicts the $4 \times 4$ block of HR **residual** (HR image minus its bicubic upsampling). Predicting $s^2$ values per LR pixel and arranging them into an $s \times s$ block is what a sub-pixel convolution does (Decision 2), so this is a one-layer, linear version of the networks that follow. We train it on eleven other `skimage` images, once for "sharp" LR inputs (block averaging only) and once for "blurry" ones (blur $\sigma = 1.2$, then averaging), and test both on the astronaut.

```
import numpy as np, cv2
from numpy.lib.stride_tricks import sliding_window_view
from scipy import ndimage as ndi
from skimage import data, img_as_float64
from skimage.color import rgb2gray

s, p = 4, 7                                  # scale factor, LR patch size

def degrade(x, blur):                        # blur (optional) -> s x s average
    x = ndi.gaussian_filter(x, blur) if blur else x
    H, W = x.shape[0] // s * s, x.shape[1] // s * s
    return x[:H, :W].reshape(H // s, s, W // s, s).mean(axis=(1, 3))

def bicubic(lr):
    return cv2.resize(lr, None, fx=s, fy=s, interpolation=cv2.INTER_CUBIC)

def features(lr):                            # one p x p patch per LR pixel, mean removed,
 ↪  plus a bias
    f = sliding_window_view(np.pad(lr, p // 2, mode="reflect"), (p, p)).reshape(lr.size, p *
     ↪  p)
    return np.c_[f - f.mean(axis=1, keepdims=True), np.ones(lr.size)]

def to_blocks(r):                            # HR residual -> one s*s vector per LR pixel
 ↪  (space-to-depth)
    h, w = r.shape[0] // s, r.shape[1] // s
    return r.reshape(h, s, w, s).transpose(0, 2, 1, 3).reshape(h * w, s * s)

def from_blocks(b, h, w):                    # inverse: depth-to-space (pixel shuffle)
    return b.reshape(h, w, s, s).transpose(0, 2, 1, 3).reshape(h * s, w * s)

def train(images, blur, lam=1e-3):
    A, B = [], []
    for im in images:
        im = im[: im.shape[0] // s * s, : im.shape[1] // s * s]
        lr = degrade(im, blur)
        A.append(features(lr)); B.append(to_blocks(im - bicubic(lr)))
    A, B = np.concatenate(A), np.concatenate(B)
    return np.linalg.solve(A.T @ A + lam * np.eye(A.shape[1]), A.T @ B)   # ridge: (p*p+1,
     ↪  s*s)

def super_resolve(lr, Wt):
    return bicubic(lr) + from_blocks(features(lr) @ Wt, *lr.shape)
```

```
def psnr(a, b):
    a, b = a[s:-s, s:-s], b[s:-s, s:-s]          # shave s border pixels
    return 10 * np.log10(1.0 / np.mean((a - np.clip(b, 0, 1)) ** 2))

train_imgs = [img_as_float64(im) for im in (data.camera(), data.moon(), data.coins(),
 ↪  data.page(),
              data.text(), data.brick(), data.grass(), data.gravel(),
              rgb2gray(data.coffee()), rgb2gray(data.chelsea()), rgb2gray(data.rocket()))]
models = {"trained sharp": train(train_imgs, 0.0), "trained blurry": train(train_imgs, 1.2)}
test = img_as_float64(rgb2gray(data.astronaut()))           # never seen in training
print("weights per model:", models["trained sharp"].shape)
for name, blur in [("sharp", 0.0), ("blurry", 1.2)]:
    lr = degrade(test, blur)
    row = "  ".join(f"{m} {psnr(test, super_resolve(lr, W)):.2f}" for m, W in models.items())
    print(f"test {name:6s} | bicubic {psnr(test, bicubic(lr)):.2f} | {row}  (dB)")
```

```
weights per model: (50, 16)
test sharp  | bicubic 25.61 | trained sharp 26.05  trained blurry 25.43  (dB)
test blurry | bicubic 24.98 | trained sharp 25.66  trained blurry 26.26  (dB)
```

Each model has only $50 \times 16 = 800$ weights, fitted in closed form in about a second. With the matching degradation it gains 0.44 dB (sharp inputs) and 1.28 dB (blurry inputs) over bicubic on an unseen image. The off-diagonal entries show the price of a wrong assumption: the blurry-trained model **over-sharpens** sharp inputs and falls below bicubic (25.43 dB versus 25.61 dB), and the sharp-trained model under-corrects blurry ones (25.66 dB versus 26.26 dB). This is the core limitation of non-blind SR: a model is only as good as its degradation assumption. Efrat et al. found the same for classical methods: an accurate blur model matters more than a sophisticated image prior [13]. Figure 6 shows the crops and the 16 learned filters, one per sub-pixel offset.



*Bicubic*



*Linear SR (matched)*

Slider comparing a blurry bicubic ×4 upsampling of the astronaut's face with the sharper output of the learned linear regressor on the same crop.

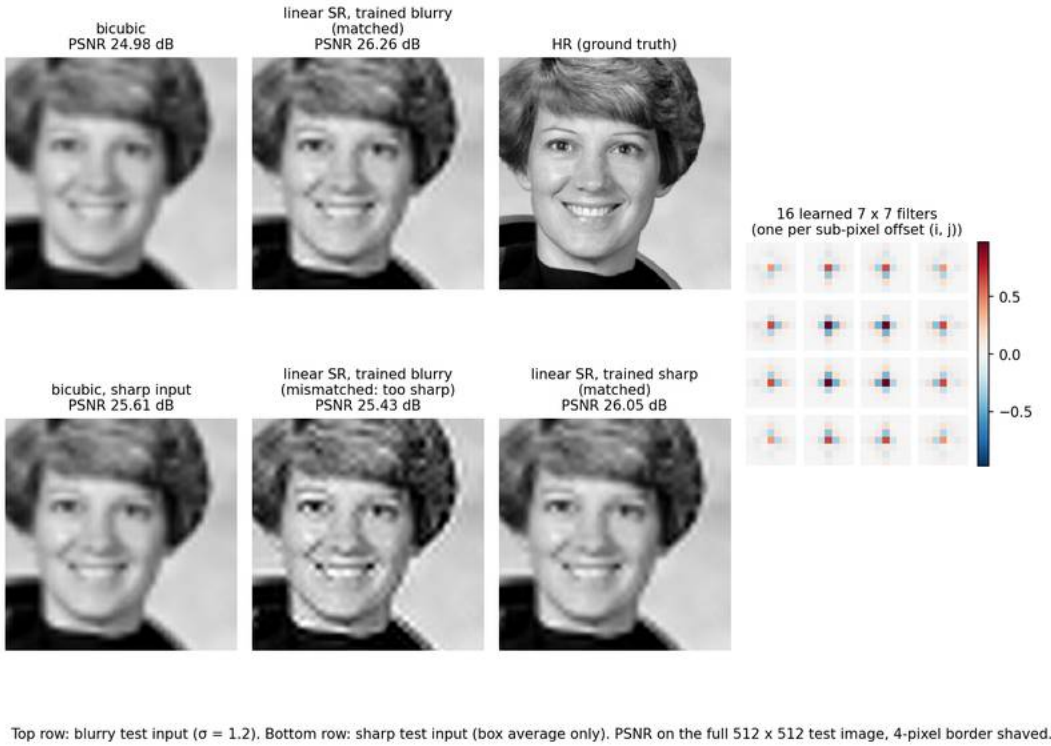



*Figure 6 — A learned linear regressor (one 7 × 7 filter per sub-pixel offset, predicting the residual over bicubic). Matched training and test degradations help; a mismatch (bottom middle) over-sharpens and scores below bicubic. PSNR on the full grayscale test image with a 4-pixel border shaved.*

### Deep SR networks: a story of design decisions

**Question.** A deep SR network is a much richer, non-linear version of the regressor above. How did the field get from a three-layer CNN to today's transformers, and why does each piece exist? **Short answer:** through six decisions, each made to fix a specific problem of the previous design, and each with a cost.

Table 1 is the map of this section; each row is explained below in the same order.

| Decision | Landmark model | Problem it solved | What it cost |
|---|---|---|---|
| 1. Learn the mapping end to end, after bicubic upsampling | SRCNN [14] | hand-designed patch pipelines | every layer runs at HR size |
| 1. Upsample at the end instead | FSRCNN [15], ESPCN [16] | the $s^2$ cost of working at HR size | one upsampling tail per scale |

| Decision | Landmark model | Problem it solved | What it cost |
|---|---|---|---|
| 2. Upsample with sub-pixel convolution (pixel shuffle) | ESPCN [16] | the cost of a learned upsampler | checkerboard risk at initialization |
| 3. Go deep and predict only the residual | VDSR [17] | slow, unstable training of deep networks | deeper bodies still cost computation and activation memory |
| 4. Remove batch normalization, scale residuals | EDSR [18] | BN's memory and its normalized feature range | needs residual scaling to train wide models |
| 5. Reweight channels (channel attention), residual in residual | RCAN [19] | all channels treated equally; training 400+ layers | depth makes inference slow |
| 6. Let pixels attend to content far away | SwinIR [20], HAT [21] | fixed, local convolution weights | memory and runtime of attention |

These networks use pixel losses: squared error targets a conditional mean, while L1 targets a coordinate-wise conditional median. Their deterministic outputs do not represent the entire conditional distribution. Architectural improvements can improve a predictor without changing this statistical objective; the next section changes the loss instead.

### Decision 1: where to upsample

**Plain version.** The output is $s$ times larger than the input in each direction, so somewhere the network must grow the image. If it grows first, every later layer works on $s^2$ times more pixels. If it grows last, every layer works on the small image.

SRCNN, the first end-to-end CNN for SR, enlarges the input with bicubic interpolation and then applies three convolution layers that its authors read as patch extraction, non-linear mapping and reconstruction, the same three steps as sparse-coding SR [14]. This **pre-upsampling** design is simple, and one network can serve several scales because the interpolation already did the resizing (VDSR trained one network for ×2, ×3 and ×4 this way [17]). Its cost follows from a count. A $k \times k$ convolution with $C_{\text{in}}$ input and $C_{\text{out}}$ output channels on an $h \times w$ map needs

$$\text{MAC} = h\, w\, C_{\text{in}}\, C_{\text{out}}\, k^2$$

multiply-accumulates, where $h \times w$ is the spatial size of the feature map. The count is proportional to $hw$, so running the same layer at HR size costs exactly $s^2$ times more.

```
def conv_macs(h, w, c_in, c_out, k=3):
    """Multiply-accumulates of one k x k convolution on an h x w feature map."""
    return h * w * c_in * c_out * k * k

s, H, W, C = 4, 128, 128, 64                    # x4 SR of a 128 x 128 input, 64 feature
 ↪  channels
body_pre = conv_macs(s * H, s * W, C, C)        # one body layer at HR size (pre-upsampling)
body_post = conv_macs(H, W, C, C)               # the same layer at LR size (post-upsampling)
tail = conv_macs(H, W, C, 3 * s * s)            # sub-pixel tail: 64 -> 3*s*s channels, then
 ↪  pixel shuffle
```

```
print(f"one body layer, pre-upsampling : {body_pre / 1e9:.2f} GMAC")
print(f"one body layer, post-upsampling: {body_post / 1e9:.2f} GMAC (ratio {body_pre /
 ↪  body_post:.0f})")
print(f"sub-pixel tail, 64 -> 48 channels at LR size: {tail / 1e9:.2f} GMAC")
```

```
one body layer, pre-upsampling : 9.66 GMAC
one body layer, post-upsampling: 0.60 GMAC (ratio 16)
sub-pixel tail, 64 -> 48 channels at LR size: 0.45 GMAC
```

For our 128 × 128 example at ×4, one 64-channel layer costs 9.66 GMAC at HR size and 0.60 GMAC at LR size, and networks have tens of such layers. **Post-upsampling** therefore keeps the body at LR size and enlarges only at the end. FSRCNN did this with a learned transposed convolution (and a thinner mapping stage) and reported a speed-up of more than 40 times over the larger SRCNN-Ex in its C++ Intel i7 CPU comparison [15]. The price is that the tail is tied to one scale factor, so multi-scale models keep one tail per scale; EDSR's multi-scale variant MDSR shares one body across ×2, ×3 and ×4 [18]. Almost every later network uses post-upsampling (Figure 7).

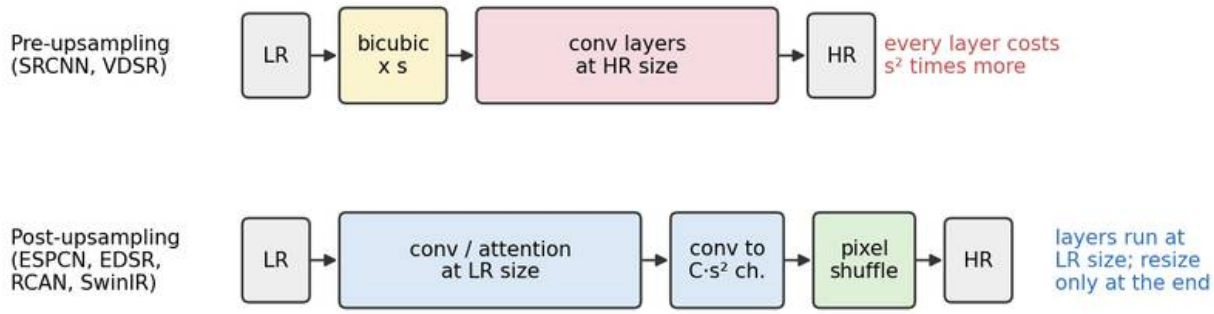



*Figure 7 — Pre-upsampling (SRCNN, VDSR) versus post-upsampling with sub-pixel convolution (ESPCN and most later networks). FSRCNN also upsamples at the end, but with a transposed convolution.*

### Decision 2: how to upsample: transposed or sub-pixel convolution

**Plain version.** A learned upsampler must produce $s^2$ output pixels for every input pixel. A transposed convolution spreads each input pixel over the output with a learned stamp. A sub-pixel convolution computes the $s^2$ values as $s^2$ separate channels at low resolution and then simply rearranges them into an $s \times s$ block.

A **transposed convolution** with stride $s$ inserts $s-1$ zeros between neighbouring LR samples and then convolves; equivalently, every LR pixel adds a scaled copy of the kernel to the output, with copies $s$ pixels apart. ESPCN observed that each output pixel then uses only some of the kernel's taps, and the pattern of used taps depends only on the **sub-pixel offset** (the output position

modulo $s$). There are exactly $s^2$ patterns, so the operation equals $s^2$ ordinary convolutions at LR size, one per offset, with interleaved outputs [16]. ESPCN implements this directly: the last convolution produces $Cs^2$ channels at LR resolution, and a fixed rearrangement, **pixel shuffle** (or depth-to-space), turns them into $C$ channels at $s$ times the resolution. The rearrangement has no parameters, and EDSR, RCAN, SwinIR and HAT all use it [18], [19], [20], [21].

The precise version is an index rule. Let $T$ be the LR feature tensor of shape $(Cs^2, H, W)$ and $X = \mathrm{PS}(T)$ the output of shape $(C, sH, sW)$. Then

$$X[c,\ sh+i,\ sw+j] = T[c\,s^2 + s\,i + j,\ h,\ w], \qquad 0 \le c < C,\ \ 0 \le h < H,\ \ 0 \le w < W,\ \ 0 \le i,j < s,$$

where $c$ is the output channel, $(h, w)$ an LR position and $(i, j)$ the sub-pixel offset inside the $s \times s$ output block that belongs to that LR position. In words: the $s^2$ channels that belong to output channel $c$ are read in row-major order of $(i, j)$ and spread over one $s \times s$ block. This is the ordering of PyTorch's `PixelShuffle`, whose implementation reshapes the channel axis into $(C, s, s)$ and permutes it next to the height and width axes [22]. The inverse, **space-to-depth** (pixel unshuffle), reads each $s \times s$ block back into $s^2$ channels; efficient networks use it at the input to work at a lower resolution. Figure 8 draws the rule for $s = 2$.

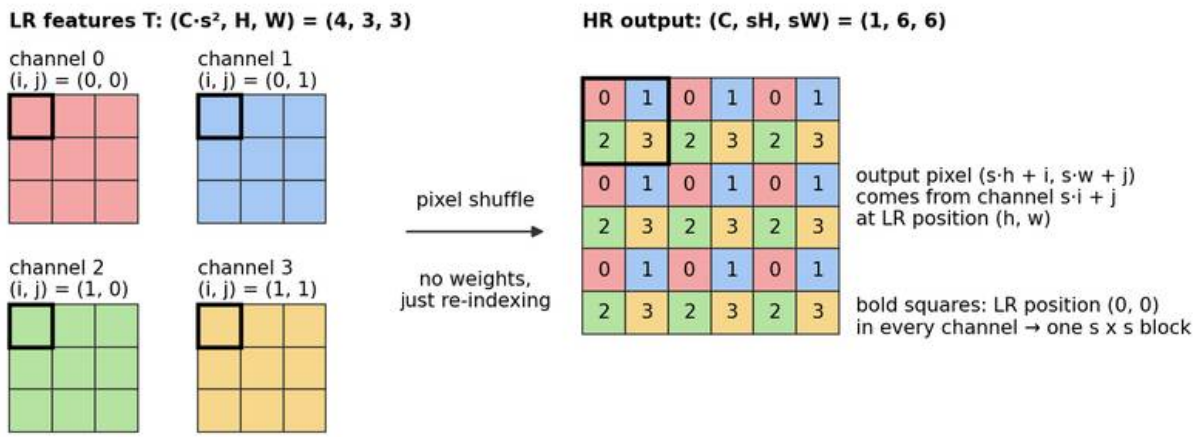



*Figure 8 — Pixel shuffle for $s = 2$ and $C = 1$. The four values at one LR position, one from each channel, become one $2 \times 2$ output block.*

The NumPy version writes pixel shuffle and its inverse as one reshape, one transpose and one reshape each, checks the index rule element by element, and checks that the two operations are exact inverses.

```
import numpy as np

def pixel_shuffle(t, s):
    """Depth-to-space: (C*s*s, H, W) -> (C, H*s, W*s)."""
    Cs2, H, W = t.shape
    C = Cs2 // (s * s)
    return t.reshape(C, s, s, H, W).transpose(0, 3, 1, 4, 2).reshape(C, H * s, W * s)
```

```
def pixel_unshuffle(x, s):
    """Space-to-depth: (C, H*s, W*s) -> (C*s*s, H, W)."""
    C, Hs, Ws = x.shape
    H, W = Hs // s, Ws // s
    return x.reshape(C, H, s, W, s).transpose(0, 2, 4, 1, 3).reshape(C * s * s, H, W)

rng = np.random.default_rng(0)
s, C, H, W = 3, 2, 4, 5
t = rng.standard_normal((C * s * s, H, W))
x = pixel_shuffle(t, s)
print(x.shape)                                                    # (2, 12, 15)

# The index rule: output (c, s*h + i, s*w + j) comes from channel c*s*s + s*i + j at (h, w).
ok = all(x[c, s*h + i, s*w + j] == t[c*s*s + s*i + j, h, w]
         for c in range(C) for h in range(H) for w in range(W)
         for i in range(s) for j in range(s))
print("index rule holds:", ok)
print("unshuffle(shuffle(t)) == t:", np.array_equal(pixel_unshuffle(x, s), t))
print("shuffle(unshuffle(x)) == x:", np.array_equal(pixel_shuffle(pixel_unshuffle(x, s), s),
 ↪  x))
```

```
(2, 12, 15)
index rule holds: True
unshuffle(shuffle(t)) == t: True
shuffle(unshuffle(x)) == x: True
```

So, when the kernel size is a multiple of $s$, the two upsamplers represent the same operations; they differ in cost and behaviour. Pixel shuffle is a permutation of array entries: exactly invertible, with the inverse permutation as its gradient, and nothing is computed at HR size. Its one weakness is shared with the transposed convolution: each sub-pixel offset has its own filter, so a freshly initialized layer responds unequally at the $s^2$ offsets and a flat input comes out as a **checkerboard**. Aitken et al. initialize the $s^2$ filters as copies of one filter, so the layer starts as a convolution followed by nearest-neighbor upsampling, which removes the pattern [23]; DLCV 01B works through the cause with code. Our linear regressor is a one-layer sub-pixel convolution: its 16 filters in Figure 6 are the 16 offsets.

### Decision 3: how deep, and why predict a residual

**Plain version.** Deeper networks capture wider context but are harder to optimize. Residual learning predicts a correction to a simple upsampled input.

VDSR made the network much deeper than SRCNN (20 weight layers of $3 \times 3$ filters) and made that depth trainable with two choices [17]. First, **global residual learning**: the network $f_\theta$ predicts only the correction to the interpolated input,

$$\hat{x} = U(y) + f_\theta(U(y)), \qquad \mathcal{L}(\theta) = \tfrac{1}{2}\|x - U(y) - f_\theta(U(y))\|_2^2,$$

where $U$ is bicubic upsampling, $f_\theta$ takes the upsampled HR-sized image, $\hat{x}$ is the SR output and $\mathcal{L}$ is the training loss for one pair. Second, a very high learning rate kept stable by **adjustable gradient clipping** [17].

*Online only: Adjustable gradient clipping — https://blog.shyandram.dev/en/learn/deep-learning/super-resolution/notes/adjustable-gradient-clipping/*

Why does the residual help? The interpolated image already contains the low frequencies, so the target $x - U(y)$ is small and centred at zero. The snippet measures this on the running example.

```
import numpy as np, cv2
from scipy import ndimage as ndi
from skimage import data, img_as_float64
from skimage.color import rgb2gray

s = 4
hr = img_as_float64(rgb2gray(data.astronaut()))                        # (512, 512), values
 ↪  in [0, 1]
lr = ndi.gaussian_filter(hr, 1.2).reshape(128, s, 128, s).mean(axis=(1, 3))   # the running
 ↪  example
up = cv2.resize(lr, None, fx=s, fy=s, interpolation=cv2.INTER_CUBIC)   # bicubic upsampling
r = hr - up                                                            # what a residual
 ↪  network predicts
print(f"HR image: mean {hr.mean():.3f}  std {hr.std():.3f}")
print(f"residual: mean {r.mean():+.3f}  std {r.std():.3f}  |r| < 0.05 at {np.mean(np.abs(r) <
 ↪  0.05):.0%} of pixels")
```

```
HR image: mean 0.442  std 0.296
residual: mean -0.000  std 0.056  |r| < 0.05 at 82% of pixels
```

The residual has about one fifth of the image's standard deviation and is below 0.05 at 82% of the pixels. A network whose output starts near zero is thus already close to a sensible answer (bicubic) and can spend its capacity on edges and textures. Post-upsampling networks keep the idea in two forms: **local** residual blocks inside the body (DLCV 01B compares them with ResNet's), and a **long skip** from the first features to the last. A matching-shape skip addition is cheap relative to the convolutional body, and later networks in Table 1 retain residual paths; increasing the body depth itself is not free.

### Decision 4: remove batch normalization

**Plain version.** Batch normalization (BN) rescales every feature channel to a standard range, which helps classifiers. An SR network must reproduce exact intensities, so this throws away something it needs, and EDSR uses the memory saved by removing BN for a wider network.

A BN layer normalizes each channel with the mean and variance of the current mini-batch during training, then applies a learned scale and shift; at test time it uses running averages. SRResNet, the generator of SRGAN, used residual blocks with BN, as classifiers did [24]. EDSR removed them,

for two reasons [18]: BN normalizes the features and so takes away the network's **range flexibility**, and BN layers use as much memory as the convolutions before them (without BN, training used about 40% less memory than SRResNet). The savings went into width: 32 residual blocks with 256 channels in the full EDSR model. At that width training became numerically unstable, which EDSR fixed with **residual scaling**: each residual branch is multiplied by 0.1 before it is added. EDSR also switched to the L1 loss (it converged better), initialized its ×3 and ×4 models from the ×2 model, and won the NTIRE 2017 SR challenge [18], [25]. ESRGAN later added a third reason: when training and test statistics differ, BN tends to introduce artifacts, especially in deep GAN-trained networks [26].

The cost: the network loses BN's help with optimization and needs residual scaling and careful initialization instead. And at deployment, BN-free networks have activations with a much larger dynamic range, which makes them harder to quantize (DLCV 01C). DLCV 01B discusses when to leave normalization out in general.

### Decision 5: reweight channels (RCAN)

**Plain version.** Some feature channels carry smooth, low-frequency content that the input already shows; others carry the fine detail SR is really about. Channel attention lets the network turn each channel up or down depending on the image, instead of treating all channels the same.

RCAN made two changes [19]. Its **channel attention** summarizes each channel by its average, passes the summary through a small bottleneck, and uses the result to rescale the channel. For a feature tensor with channels $x_1, \dots, x_C$ of size $H \times W$,

$$z_c = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} x_c(i,j), \qquad \mathbf{s} = f(W_U\, \delta(W_D\, \mathbf{z})), \qquad \hat{x}_c = s_c\, x_c,$$

where $x_c(i,j)$ is the value of channel $c$ at position $(i,j)$, $\mathbf{z} = (z_1, \dots, z_C)$ the vector of channel averages, $W_D$ a 1×1 convolution that reduces $C$ channels to $C/r$, $\delta$ the ReLU, $W_U$ a 1×1 convolution that expands them back to $C$, $f$ the sigmoid, $s_c \in (0,1)$ the weight of channel $c$, and $\hat{x}_c$ the rescaled channel. RCAN uses $C = 64$ and reduction ratio $r = 16$, so the bottleneck has 4 channels; the extra parameters are tiny. The same mechanism appears in classification networks as squeeze-and-excitation (DLCV 01B).

Its **residual-in-residual** structure makes great depth trainable: 10 residual groups of 20 channel-attention blocks, each group with a short skip connection, all inside one long skip connection, for over 400 layers [19]. The authors' argument is that the abundant low-frequency information of the input can bypass the body through these skips, so the main network can focus on high frequencies. The cost is latency: 400 layers run one after another.

### Decision 6: let pixels attend to content far away (SwinIR, HAT)

**Plain version.** A convolution applies the same weights everywhere; self-attention computes its weights from the content, so a pixel on one window frame can borrow detail from a similar frame

elsewhere. Full attention over an image is too expensive, so SR transformers attend within small windows that shift between layers.

Within a window of $M \times M$ tokens (here, one token per LR pixel), self-attention computes

$$\text{Attention}(Q, K, V) = \text{SoftMax}(QK^\top / \sqrt{d} + B)V,$$

where $Q$, $K$ and $V$ are the query, key and value matrices obtained by linear maps of the tokens in the window ($M^2$ rows each), $d$ is the dimension of a query or key, and $B$ is a learned bias that depends only on the relative position of two tokens [21]. Each output is a weighted average of the values in the window, weighted by how well its query matches each key (ML 05 explains attention and shifted windows in general). Windows are what make this affordable. For a map of $h \times w$ tokens with $C$ channels, the Swin Transformer paper counts

$$\Omega(\text{MSA}) = 4hwC^2 + 2(hw)^2C, \qquad \Omega(\text{W-MSA}) = 4hwC^2 + 2M^2hwC,$$

where MSA is global multi-head self-attention, W-MSA attention within $M \times M$ windows, the first term the linear projections and the second the attention itself [27]. Global attention grows with the square of the number of tokens; windowed attention grows linearly. With SwinIR's width $C = 180$ and window $M = 8$ on our $128 \times 128$ LR map, the formulas give about 98.8 GMAC for global attention and 2.50 GMAC for windows in one layer (our arithmetic; Exercise 6).

SwinIR builds an SR network around this block: a $3 \times 3$ convolution for shallow features, residual Swin Transformer blocks (several Swin layers plus a convolution, wrapped in a residual connection; 6 blocks of 6 layers, windows of 8 and 180 channels in its standard SR setting), and a pixel-shuffle reconstruction, trained with the L1 loss [20]. HAT asked how much of the input SwinIR really uses. With an attribution method for SR (LAM), its authors found that SwinIR uses a limited spatial range, and designed HAT to "activate more pixels" [21]: a channel-attention branch in parallel with window attention, windows of 16, an **overlapping cross-attention** block whose keys and values come from larger, overlapping windows, and pre-training on ImageNet for the same SR task before fine-tuning on DIV2K and Flickr2K. Later backbones continue the search for cheap long-range context: DAT alternates spatial and channel self-attention [28], and MambaIR replaces attention with state-space scans that reach the whole image at linear cost [29].

The cost is memory and runtime, which grow with the window size; large images are processed in tiles, which can leave seams. And Table 1's main point still holds: trained with L1, these networks owe their sharper outputs to better use of context, not to sampling.

**The common training recipe.** These networks are trained on small random crops from DIV2K or DIV2K plus Flickr2K (EDSR used $48 \times 48$ LR patches), with random flips and 90° rotations, the L1 loss and Adam. A "+" after a name (EDSR+) means **geometric self-ensemble**: the outputs for the 8 flipped and rotated versions of the input are transformed back and averaged [18]. The general losses are in DLCV 09.

### Perceptual and adversarial SR

**Question.** If a pixel loss favours a central estimate rather than one sharp candidate, can a different loss produce realistic detail? **Short answer:** yes, by rewarding outputs that look like real photos

(perceptual and adversarial losses), at the price of a lower PSNR and occasional invented detail.

### Perceptual losses and GANs

Johnson, Alahi and Fei-Fei trained SR networks with a **perceptual loss**, the distance between features of a pretrained classifier (VGG [30]) for the output and for the target, and obtained sharper edges and textures than with a pixel loss, at a lower PSNR [31]. SRGAN added an **adversarial loss**: a discriminator learns to tell SR outputs from real HR images, and the SR network (SRResNet, 16 residual blocks) is trained to fool it, which pushes the solution toward the natural image manifold; in a mean-opinion-score test, raters clearly preferred SRGAN's outputs [24]. ESRGAN improved each part: residual-in-residual dense blocks without BN (Decision 4), a relativistic discriminator that judges whether a real image looks *more* realistic than a generated one, and a perceptual loss on VGG features taken before the activation; a variant won region 3 (the most perceptual one) of the PIRM 2018 challenge [26], [32]. A typical generator loss is

$$\mathcal{L} = \lambda_1 \, \|\hat{x} - x\|_1 + \lambda_p \, \|\phi(\hat{x}) - \phi(x)\|_2^2 + \lambda_a \, \mathcal{L}_{\text{adv}}(\hat{x}),$$

where $\hat{x}$ is the SR output, $x$ the HR target, $\phi$ a fixed layer of a pretrained network, $\mathcal{L}_{\text{adv}}$ the adversarial term computed with the discriminator, and $\lambda_1, \lambda_p, \lambda_a \geq 0$ weights chosen by hand. GANs are explained in ML 07.

ESRGAN also introduced a simple knob, **network interpolation**: blend the weights of a PSNR-oriented network and its GAN-fine-tuned copy, $\theta = (1 - \alpha)\, \theta_{\text{PSNR}} + \alpha \, \theta_{\text{GAN}}$ with $0 \leq \alpha \leq 1$, to move between smooth and textured outputs without retraining [26].

*Online only: ESRGAN's changes in detail — https://blog.shyandram.dev/en/learn/deep-learning/super-resolution/notes/esrgans-changes-in-detail/*

### The perception–distortion trade-off

Blau and Michaeli defined **distortion** as the expected dissimilarity between $\hat{x}$ and $x$ (MSE, for example) and **perceptual quality** as the distance between the distribution of outputs and the distribution of real images, and formalized the frontier between the two objectives [2]. As the permitted distortion shrinks, the best achievable perceptual quality cannot improve; with an ambiguous posterior, the low-distortion MMSE solution can have an unrealistic output distribution. The conflict is not inevitable in a noiseless, invertible problem where the input identifies the HR image exactly. The formal statement is in DLCV 09. Our toy traces such a curve (Figure 1, right): moving from the posterior mean toward a posterior sample raises MSE from 0.0072 to 0.0143 and closes the realism gap. ESRGAN's network interpolation is an empirical way to explore operating points. Blending nonlinear network parameters is not the same operation as interpolating the toy's output images, and is not guaranteed to trace the theoretical optimum. Two practical consequences follow:

- A method should state *where* on the trade-off it aims. Comparing a GAN to an MSE network by PSNR alone, or by a perceptual metric alone, says nothing about which is better.

- Benchmarks must measure both axes. The PIRM 2018 challenge did this by splitting the perception–distortion plane into three regions by RMSE (at most 11.5, 12.5 and 16) and ranking methods within each region by a perceptual index built from two no-reference metrics, NIQE [33] and the SR-specific metric of Ma et al. [34], [32].

### Diffusion SR for a known degradation

**Question.** Can a model represent several plausible HR answers rather than just a central prediction? **Short answer:** conditional diffusion learns an approximation to the conditional image distribution and can sample different candidates, at the cost of iterative inference. A trained sampler does not certify exact Bayesian posterior sampling or exact data consistency.

A diffusion model learns to turn noise into realistic images step by step; conditioned on the LR image, it generates an HR image that fits the input, and different random seeds give different plausible answers. SR3 is the representative method for the non-blind setting, with LR images made by bicubic downscaling [35]. Its U-Net [36] receives the noisy HR estimate concatenated, along the channel axis, with the bicubic upsampling of the LR image, and learns to remove noise at many levels. Inference starts from pure Gaussian noise and refines it, in at most 100 steps in the paper's setting [35].

SR3's own comparisons show the trade-off of the previous section. On ×8 face SR ($16 \times 16$ to $128 \times 128$), human raters asked to pick the real photo chose SR3's output close to 50% of the time, while GAN baselines reached at most 34%. On ×4 ImageNet SR, a regression model with the same architecture scored higher PSNR and SSIM, while SR3 scored better on sample-quality metrics; the authors note that PSNR and SSIM penalize any synthetic detail that is not perfectly aligned [35]. SR3 also checked **consistency**: its outputs, downsampled again, matched the LR inputs better than the GAN baselines' outputs did [35]. DLCV 09B explains how such a conditional diffusion restorer is trained and sampled, and ML 08 the diffusion mechanics.

Diffusion can generate plausible detail that does not align with the paired target, lowering PSNR even when viewers prefer it; lower PSNR is not a universal guarantee for every model or setting. Generated details can be wrong (see Failure modes), and iterative sampling costs many network evaluations. Methods built on large pretrained text-to-image models, with sampling cut to a few steps or one, were developed for real-world photos and are covered in DLCV 09A.1.

### When the degradation is unknown: blind and real-world SR

**Plain version.** Everything so far assumed that we know how the small image was made. A real photo comes with its own unknown blur, noise and compression, and a model trained on bicubic pairs misjudges it, as the mismatched linear model above did. Blind and real-world SR either trains on a very wide range of synthetic degradations, estimates the degradation first, or learns from real captured pairs, and it evaluates results without ground truth. These methods, their datasets, and the diffusion-prior restorers used for real photos have their own chapter, DLCV 09A.1 ·Blind and Real-World Image Super-Resolution.

### Arbitrary-scale SR

**Plain version.** The networks above are trained for one fixed factor such as ×4. Arbitrary-scale networks treat the image as a continuous function, so you can ask for any output size, including non-integer factors like ×2.7.

Meta-SR replaced the fixed upsampling tail with a **meta-upscale module** that predicts the weights of the upscaling filters from the scale factor and the relative position of each output pixel [37]. LIIF went further and learned a **local implicit image function** [38]:

$$\hat{I}(x_q) = f_\theta(z^*,\ x_q - v^*),$$

where $x_q$ is a continuous query coordinate in the image plane, $z^*$ the latent feature vector (from a CNN encoder applied to the LR image) whose grid position $v^*$ is nearest to $x_q$, and $f_\theta$ a small MLP that outputs the RGB value at $x_q$. Because $x_q$ is continuous, the same network renders any output size. Trained on scales between ×1 and ×4, it was shown producing outputs up to ×30 [38]. The price is one MLP evaluation per output pixel, and queries far outside the training scales rely on learned assumptions rather than additional measured detail.

*Online only: LIIF's three refinements — https://blog.shyandram.dev/en/learn/deep-learning/super-resolution/notes/liifs-three-refinements/*

### Efficiency

**Plain version.** Many SR uses (phones, video calls, game rendering) need results in milliseconds on small chips. Efficient SR asks how small and fast a network can be while staying above a quality threshold.

Post-upsampling with pixel shuffle (Decisions 1 and 2) is the first rule: almost all computation happens at LR size. Beyond that, efficient networks reuse features cheaply; IMDN, for example, splits features step by step into a "distilled" part that is kept and a part that is processed further, then fuses them with a contrast-aware channel attention [39]. Parameters, FLOPs and runtime measure different costs, so the NTIRE efficient SR challenges score all three. In 2024 and 2025 the task was ×4 SR, and entries had to reach a PSNR of at least 26.90 dB on the validation set and 26.99 dB on the test set while minimizing runtime, parameters and FLOPs [40], [41]. Compressing an existing network (pruning, quantization, distillation) is the subject of DLCV 01C.

### Evaluating SR

**Question.** How do you tell whether one SR result is better than another? **Short answer:** for fidelity, follow the community's fixed PSNR/SSIM recipe exactly, because small details change the numbers; for realism, add perceptual metrics and human judgments, and never compare the two kinds by one number. The generic metrics are explained in DLCV 09; this section covers what is specific to SR.

### Y-channel PSNR and SSIM with border shaving

The classical protocol, inherited from the MATLAB code of early papers, is:

1. Convert both the HR image and the SR output to YCbCr and keep the luma channel

$$Y = 16 + 65.481\,R + 128.553\,G + 24.966\,B,$$

where $R, G, B \in [0, 1]$ are the (gamma-encoded) color values and $Y$ lies in $[16, 235]$. These are the BT.601 coefficients as implemented by MATLAB's `rgb2ycbcr` (and by `skimage.color.rgb2ycbcr`) [42], [43]. 2. Remove a border of pixels from both images before comparing ("shaving"), so that the effects of padding at the image border do not dominate. 3. Compute PSNR with peak value 255 and SSIM [44] on what remains. Definitions are in DLCV 09.

The details vary even within one paper. EDSR reports its benchmark results on Y with a border equal to the scale factor, but its ablation study on DIV2K uses all RGB channels and ignores $6 + s$ border pixels [18]; the NTIRE 2024 and 2025 ×4 challenges compute on Y with a 4-pixel border [45], [46]. Wang et al. note that the color space and channel alone can change results by up to 4 dB in one reported case [4]. Part of that is arithmetic: the coefficients above sum to 219, so $Y = 16 + 219\,\ell$ with $\ell \in [0, 1]$, and PSNR with peak 255 on $Y$ adds $20 \log_{10}(255/219) \approx 1.32$ dB to PSNR on $\ell$ with peak 1 (Exercise 2). Rounding, clipping and the resizer also shift results slightly, so fair comparisons run every method through one evaluation script.

### The standard test sets

SR papers report results on a small set of public test sets, each with a different character [4]:

| Test set | Images | Content | Source |
|---|---|---|---|
| Set5 | 5 | baby, bird, butterfly, head, woman | [47] |
| Set14 | 14 | people, animals, flowers, comic, slides | [48] |
| BSD100 (B100) | 100 | natural scenes, the test images of the Berkeley segmentation dataset | [49] |
| Urban100 | 100 | buildings and city scenes with regular structures | [50] |
| Manga109 | 109 | Japanese comic volumes (line art and screen tones) | [51], [52] |
| DIV2K validation | 100 | diverse 2K-resolution photos | [53] |

Set5 and Set14 are tiny, so a few hundredths of a dB on them mean little. The sets reward different abilities: Urban100 is full of repeated structures, and HAT's larger attention window helped most there [21].

### Why PSNR and perceptual rankings disagree

Three reasons, all seen in this chapter. First, PSNR is maximized by the blurry average of all plausible answers, so a method that commits to one sharp answer loses PSNR even when it looks better (Figure 1; SR3 versus its regression baseline). Second, a texture with the right statistics but shifted by half a period has a large pixel error. Third, learned perceptual metrics such as LPIPS [54] and no-reference metrics such as NIQE [33] reward natural-looking texture and can be fooled by over-sharpening, while PSNR rewards smoothness. The NTIRE 2025 ×4 challenge ran a restoration track ranked by PSNR and a perceptual track ranked by one combined score of six full-reference and no-reference quality metrics, including LPIPS and NIQE [46]. The advice from DLCV 09 applies: report a fidelity metric, a perceptual metric and, for generative SR, a user study.

## Datasets and benchmarks

**Plain version.** A few datasets and a yearly competition define most of what "state of the art" means in non-blind SR.

- **DIV2K** [53] contains 1,000 diverse 2K-resolution images, split 800/100/100 for training, validation and testing [46]. It was introduced for the NTIRE 2017 challenge and is the standard training set. It is often combined with **Flickr2K**, which the NTIRE 2025 report describes as 2,650 2K images [46]; the combination is called DF2K, and HAT, for example, fine-tunes on it after pre-training on ImageNet [21].
- **NTIRE 2017** [25], the first SR challenge of the series, had a bicubic track and a track with unknown downscaling operators (learnable from the provided training pairs), each at ×2, ×3 and ×4.
- **NTIRE 2024 and 2025 ×4 SR** [45], [46] use bicubic ×4 on DIV2K; the 2025 edition added the perceptual track described above.
- **NTIRE efficient SR** [40], [41] ranks ×4 models by runtime, parameters and FLOPs above a fixed PSNR.

The real-world datasets and challenges, which have no clean bicubic pairs, are described in DLCV 09A.1.

## Failure modes

**Plain version.** An SR model can fail by being too cautious (blurry) or too bold (invented detail). The bold failures are more dangerous, because the result looks convincing.

- **Over-smoothing.** A central prediction can suppress ambiguous detail, giving waxy skin, flat grass or soft hair. MSE targets a mean; L1 targets a median. Neither statistical objective guarantees blur on every input, but neither asks the network to represent all plausible sharp alternatives (Figure 1).

- **Hallucinated text and faces.** At large scales a generative model can turn illegible text into legible but *wrong* characters, or a few pixels of a face into a sharp face that is not the person photographed; for forensic, medical or scientific images this is unacceptable. Ren et al. argue that fidelity and no-reference metrics both capture such hallucinations poorly, and propose scoring them with a multimodal large language model [55].
- **Texture artifacts.** GAN-based models can add repetitive, noise-like textures; ESRGAN was designed partly to reduce such artifacts of SRGAN [26]. Our mismatched linear model shows halos and exaggerated edges in miniature (Figure 6).
- **Color shifts.** Diffusion-based SR can shift global colors away from the input; Moser et al. list it among the open problems of diffusion SR [56].
- **Degradation mismatch.** A non-blind model applied to images made differently under- or over-corrects (Figure 6). Testing on more than one degradation exposes this; DLCV 09A.1 covers the remedies.
- **Tiling seams.** Large images are often processed in patches; non-overlapping patches can leave visible borders, as the SwinIR authors note for patch-based processing [20].

A useful sanity check for any SR output is **consistency with the input**: degrade the output with the assumed model and compare it with $y$. A large difference means the model changed content that the LR image did measure.

## Modern view

### Survey papers

Two surveys cover this chapter's material. Wang, Chen and Hoi split deep SR into supervised, unsupervised and domain-specific SR, and describe supervised models along independent axes: where to upsample (pre-, post-, progressive, iterative up-and-down), with which operator, which network design (residual, recursive, dense, attention and others), which losses and which training strategies [4]. Their takeaway matches Table 1: most models are combinations of choices along these axes, so read a new model as "which choices did it change?". The survey also tabulates the test sets; it predates transformers and diffusion models. Moser et al. cover diffusion models for SR: the theory first, then methods organized by input domain (pixel, latent, wavelet), conditioning, guidance, corruption spaces and zero-shot use of pretrained models [56]. Their open issues (high compute, poor comparability across papers, limited explainability, color shifts) are a good checklist for reading any diffusion SR paper. The blind and real-world surveys are discussed in DLCV 09A.1.

### State of the art, 2023–2026

1. **Fidelity-oriented SR** uses attention and state-space backbones (HAT, DAT, MambaIR [21], [28], [29]), large-scale pre-training, and is benchmarked each year by the NTIRE ×4 restoration track [45], [46].

2. **Evaluation is catching up with generation**: perceptual SR has its own challenge track with a combined perceptual score, so realism-oriented methods are no longer ranked by PSNR [46], and newer work explicitly evaluates hallucination [55].
3. **Efficiency** has its own benchmark with fixed quality thresholds [40], [41].

### Open problems

- **Faithful generative SR.** How to get diffusion-level realism without inventing content, and how to show a user which details are measured and which are guessed.
- **Evaluation.** PSNR rewards blur, perceptual metrics can be gamed, and hallucination metrics are new.
- **Benchmark saturation.** Gains of a few hundredths of a dB on small bicubic test sets say little about real use; the bicubic setting itself is far from real cameras (DLCV 09A.1).
- **Cost.** Attention and diffusion models are far heavier than the networks of the efficient SR challenges.
- **Arbitrary and extreme scales.** Behaviour far outside the training scales, and at factors where almost everything is generated, is poorly understood.

### Key takeaways

- Known downsampling still loses information; a prior determines unresolved detail.
- MSE favors a conditional mean and L1 a coordinate-wise median; either can smooth uncertain structures.
- Non-blind results depend on the assumed degradation matching the input.
- Late upsampling, residual paths and feature mixing change cost and detail recovery; compare them under the same recipe.
- The perception–distortion frontier does not forbid improving both metrics from a weak baseline; learned generators have no exact-sampling guarantee.
- State color channels and border shaving, check low-resolution consistency, and inspect text, faces, textures and seams.

### Exercises

1. **Which cubic does OpenCV use?** Make a 1-D signal of 16 zeros with a single 1 in the middle, upsample it by 4 with `cv2.resize(..., interpolation=cv2.INTER_CUBIC)`, and compare the result with the Keys kernel evaluated at the half-pixel-aligned positions $(t+\frac{1}{2})/4-\frac{1}{2}-8$ for $a=-0.5$ and $a=-0.75$. Then plot both kernels' frequency responses and compare their values at the LR Nyquist frequency.

   **Hint**

   The output of a resize of a unit impulse is the kernel itself, sampled at the output positions. In our run the maximum absolute difference was 0.0 for $a=-0.75$ and about 0.037 for $a=-0.5$. A more negative

$a$ gives deeper negative lobes, a response that stays near 1 a little longer and then overshoots slightly: a sharper but more ringing interpolator. Repeat with `INTER_LANCZOS4` and the Lanczos kernel with $a = 4$; the match is close but not exact (we measured a maximum difference of about 0.0016), because the library normalizes the weights.

2. **Protocol arithmetic.** (a) Show that $Y$-PSNR with peak 255 equals the PSNR of $\ell = (Y-16)/219$ with peak 1 plus $20\log_{10}(255/219)$. (b) In the benchmark-setting snippet, change the shave from 4 to 0 and to 16 pixels and explain the size of the change. (c) Why is shaving necessary for a network with zero padding but nearly irrelevant for `cv2.resize`?

**Hint**

(a) $Y_{\text{hr}} - Y_{\text{sr}} = 219(\ell_{\text{hr}} - \ell_{\text{sr}})$, so the MSE scales by $219^2$ and PSNR becomes $10\log_{10}(255^2/(219^2\,\text{MSE}_\ell)) = \text{PSNR}_\ell + 20\log_{10}(255/219) \approx \text{PSNR}_\ell + 1.32$ dB. (b) Shaving changes the result by only a few hundredths of a dB here (24.66 versus 24.62 dB for nearest neighbor), because OpenCV extrapolates borders reasonably and the image edges are smooth. (c) Zero padding makes a convolutional network see black outside the image, so the outputs near the border are systematically worse; shaving removes a region whose quality depends on the padding choice rather than on the method.

3. **A channel-ordering bug.** Suppose one library orders the $Cs^2$ channels as $k = cs^2 + si + j$ (our formula) and another as $k = (si + j)C + c$. You load weights trained with the first ordering into the second. For $C = 3$ and $s = 2$, which of the 12 channels end up in the right place? What does the output look like?

**Hint**

Solve $4c + q = 3q + c$ with $q = 2i + j$: $3c = 2q$, so only $(c, q) = (0, 0)$ and $(2, 3)$ coincide, 2 channels out of 12 (we checked this by enumerating all 12). The other values are moved to the wrong color channel and the wrong sub-pixel offset, so the output shows a fine color mosaic in every $2 \times 2$ block even though the coarse image is recognizable. For $C = 1$ the two orderings agree, which is why such bugs often appear only when moving from grayscale to color models.

4. **Make the linear model better.** Starting from the regressor snippet, (a) change the patch size $p$ to 3, 5, 9 and 11; (b) split LR pixels into classes by the angle of the local gradient (as RAISR does) and fit one regressor per class; (c) add noise with $\sigma = 0.01$ to the LR training images but not to the test images, and the other way round. Report PSNR for each change and explain.

**Hint**

(a) On the blurry test input we measured 25.94 dB for $p = 3$, 26.18 dB for $p = 5$ and 26.26 dB for $p = 7, 9$, 11: beyond $7 \times 7$ a linear model gains nothing. (b) Four angle classes (computed from the gradient of the slightly smoothed LR image) gave 26.33 dB and eight gave 26.32 dB, a small gain: classes let the filter follow edge direction, which is what makes RAISR and A+ better than one global filter. (c) Training on noisy inputs teaches the model to smooth, so it under-sharpens clean inputs; training on clean inputs teaches it to amplify fine detail, so it amplifies noise. This is the noise version of the blur mismatch in Figure 6.

5. **Posterior mean by hand.** Two HR pixels $a, b \in \{0, 1\}$ are independent and equally likely to be 0 or 1. The LR pixel is $y = (a+b)/2$ with no noise. (a) For each value of $y$, give the posterior mean of $(a, b)$ and a posterior sample. (b) Compute the expected per-pixel MSE of both estimators. (c) Which estimator produces outputs that look like the HR data?

**Hint**

(a) $y = 0$ and $y = 1$ determine $(a, b)$ exactly. For $y = 0.5$ (probability $1/2$) the posterior is $(0, 1)$ or $(1, 0)$ with equal probability, so the mean is $(0.5, 0.5)$ and a sample is one of the two. (b) Only $y = 0.5$ contributes. The mean has squared error 0.25 per pixel there, so its MSE is $0.5 \times 0.25 = 0.125$. A sample is right half the time and wrong by 1 on both pixels otherwise, so its MSE is $0.5{\times}0.5{\times}1 = 0.25$: exactly twice as large. (c) Only the sample: real pixels are 0 or 1, and the value 0.5 never occurs in the HR data.

6. **The price of global attention.** Using the two formulas of Decision 6, compute $\Omega(\text{MSA})$ and $\Omega(\text{W-MSA})$ for $C = 180$, $M = 8$ and LR maps of $64 \times 64$ and $128 \times 128$ tokens. How does the ratio between them change when the image side doubles, and why? What window size $M$ would make windowed attention as expensive as global attention?

**Hint**

We obtained 6.6 versus 0.63 GMAC (ratio 10.5) at $64 \times 64$ and 98.8 versus 2.50 GMAC (ratio 39.5) at $128{\times}128$. Doubling the side multiplies $hw$ by 4: the projection term $4hwC^2$ and the windowed attention term grow by 4, but the global attention term $2(hw)^2C$ grows by 16. The two attention terms are equal when $M^2 = hw$, that is when one window covers the whole image.

## References


1. S. C. Park, M. K. Park and M. G. Kang, "Super-Resolution Image Reconstruction: A Technical Overview," *IEEE Signal Processing Magazine*, vol. 20, no. 3, pp. 21–36, 2003. doi
2. Y. Blau and T. Michaeli, "The Perception-Distortion Tradeoff," in *Proc. IEEE/CVF CVPR*, 2018, pp. 6228–6237. doi ·arXiv
3. MathWorks, "imresize: Resize image," MATLAB documentation (default method bicubic; antialiasing on when shrinking). link
4. Z. Wang, J. Chen and S. C. H. Hoi, "Deep Learning for Image Super-Resolution: A Survey," *IEEE TPAMI*, vol. 43, no. 10, pp. 3365–3387, 2021. doi ·arXiv
5. R. Keys, "Cubic Convolution Interpolation for Digital Image Processing," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 6, pp. 1153–1160, 1981. doi
6. C. E. Duchon, "Lanczos Filtering in One and Two Dimensions," *Journal of Applied Meteorology*, vol. 18, no. 8, pp. 1016–1022, 1979. doi
7. W. T. Freeman, T. R. Jones and E. C. Pasztor, "Example-Based Super-Resolution," *IEEE Computer Graphics and Applications*, vol. 22, no. 2, pp. 56–65, 2002. doi
8. D. Glasner, S. Bagon and M. Irani, "Super-Resolution from a Single Image," in *Proc. IEEE ICCV*, 2009, pp. 349–356. doi

9. J. Yang, J. Wright, T. S. Huang and Y. Ma, “Image Super-Resolution via Sparse Representation,” *IEEE Transactions on Image Processing*, vol. 19, no. 11, pp. 2861–2873, 2010. doi
10. R. Timofte, V. De Smet and L. Van Gool, “Anchored Neighborhood Regression for Fast Example-Based Super-Resolution,” in *Proc. IEEE ICCV*, 2013, pp. 1920–1927. doi ·CVF
11. R. Timofte, V. De Smet and L. Van Gool, “A+: Adjusted Anchored Neighborhood Regression for Fast Super-Resolution,” in *Computer Vision – ACCV 2014*, Lecture Notes in Computer Science, Springer, 2015, pp. 111–126. doi
12. Y. Romano, J. Isidoro and P. Milanfar, “RAISR: Rapid and Accurate Image Super Resolution,” *IEEE Transactions on Computational Imaging*, vol. 3, no. 1, pp. 110–125, 2017. doi ·arXiv
13. N. Efrat, D. Glasner, A. Apartsin, B. Nadler and A. Levin, “Accurate Blur Models vs. Image Priors in Single Image Super-Resolution,” in *Proc. IEEE ICCV*, 2013, pp. 2832–2839. doi ·CVF
14. C. Dong, C. C. Loy, K. He and X. Tang, “Image Super-Resolution Using Deep Convolutional Networks,” *IEEE TPAMI*, vol. 38, no. 2, pp. 295–307, 2016. doi ·arXiv
15. C. Dong, C. C. Loy and X. Tang, “Accelerating the Super-Resolution Convolutional Neural Network,” in *Proc. ECCV*, 2016, pp. 391–407. doi ·arXiv
16. W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert and Z. Wang, “Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network,” in *Proc. IEEE CVPR*, 2016, pp. 1874–1883. doi ·arXiv
17. J. Kim, J. K. Lee and K. M. Lee, “Accurate Image Super-Resolution Using Very Deep Convolutional Networks,” in *Proc. IEEE CVPR*, 2016, pp. 1646–1654. doi ·arXiv
18. B. Lim, S. Son, H. Kim, S. Nah and K. M. Lee, “Enhanced Deep Residual Networks for Single Image Super-Resolution,” in *Proc. IEEE CVPR Workshops*, 2017, pp. 1132–1140. doi ·arXiv
19. Y. Zhang, K. Li, K. Li, L. Wang, B. Zhong and Y. Fu, “Image Super-Resolution Using Very Deep Residual Channel Attention Networks,” in *Proc. ECCV*, 2018, pp. 294–310. doi ·arXiv
20. J. Liang, J. Cao, G. Sun, K. Zhang, L. Van Gool and R. Timofte, “SwinIR: Image Restoration Using Swin Transformer,” in *Proc. IEEE/CVF ICCV Workshops*, 2021, pp. 1833–1844. doi ·arXiv
21. X. Chen, X. Wang, J. Zhou, Y. Qiao and C. Dong, “Activating More Pixels in Image Super-Resolution Transformer,” in *Proc. IEEE/CVF CVPR*, 2023, pp. 22367–22377. doi ·arXiv
22. PyTorch contributors, “PixelShuffle.cpp” (implementation of `torch.nn.PixelShuffle` and `torch.nn.PixelUnshuffle`), PyTorch source code. GitHub
23. A. Aitken, C. Ledig, L. Theis, J. Caballero, Z. Wang and W. Shi, “Checkerboard Artifact Free Sub-Pixel Convolution: A Note on Sub-Pixel Convolution, Resize Convolution and Convolution Resize,” arXiv:1707.02937, 2017. arXiv
24. C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang and W. Shi, “Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network,” in *Proc. IEEE CVPR*, 2017, pp. 105–114. doi ·arXiv
25. R. Timofte, E. Agustsson, L. Van Gool, M.-H. Yang, L. Zhang et al., “NTIRE 2017 Challenge on Single Image Super-Resolution: Methods and Results,” in *Proc. IEEE CVPR Workshops*, 2017, pp. 1110–1121. doi ·CVF
26. X. Wang, K. Yu, S. Wu, J. Gu, Y. Liu, C. Dong, Y. Qiao and C. C. Loy, “ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks,” in *Proc. ECCV Workshops*, 2018 (Lecture Notes in Computer Science, 2019), pp. 63–79. doi ·arXiv
27. Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin and B. Guo, “Swin Transformer: Hierarchical Vision Transformer using Shifted Windows,” in *Proc. IEEE/CVF ICCV*, 2021, pp. 9992–

10002. doi ·arXiv
28. Z. Chen, Y. Zhang, J. Gu, L. Kong, X. Yang and F. Yu, "Dual Aggregation Transformer for Image Super-Resolution," in *Proc. IEEE/CVF ICCV*, 2023, pp. 12278–12287. doi ·arXiv
29. H. Guo, J. Li, T. Dai, Z. Ouyang, X. Ren and S.-T. Xia, "MambaIR: A Simple Baseline for Image Restoration with State-Space Model," in *Proc. ECCV*, 2024, pp. 222–241. doi ·arXiv
30. K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *Proc. ICLR*, 2015. arXiv
31. J. Johnson, A. Alahi and L. Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution," in *Proc. ECCV*, 2016, pp. 694–711. doi ·arXiv
32. Y. Blau, R. Mechrez, R. Timofte, T. Michaeli and L. Zelnik-Manor, "The 2018 PIRM Challenge on Perceptual Image Super-Resolution," in *Proc. ECCV Workshops*, 2018. arXiv
33. A. Mittal, R. Soundararajan and A. C. Bovik, "Making a "Completely Blind" Image Quality Analyzer," *IEEE Signal Processing Letters*, vol. 20, no. 3, pp. 209–212, 2013. doi
34. C. Ma, C.-Y. Yang, X. Yang and M.-H. Yang, "Learning a No-Reference Quality Metric for Single-Image Super-Resolution," *Computer Vision and Image Understanding*, vol. 158, pp. 1–16, 2017. doi ·arXiv
35. C. Saharia, J. Ho, W. Chan, T. Salimans, D. J. Fleet and M. Norouzi, "Image Super-Resolution via Iterative Refinement," *IEEE TPAMI*, 2022. doi ·arXiv
36. O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Proc. MICCAI*, 2015, pp. 234–241. doi ·arXiv
37. X. Hu, H. Mu, X. Zhang, Z. Wang, T. Tan and J. Sun, "Meta-SR: A Magnification-Arbitrary Network for Super-Resolution," in *Proc. IEEE/CVF CVPR*, 2019, pp. 1575–1584. doi ·arXiv
38. Y. Chen, S. Liu and X. Wang, "Learning Continuous Image Representation with Local Implicit Image Function," in *Proc. IEEE/CVF CVPR*, 2021, pp. 8624–8634. doi ·arXiv
39. Z. Hui, X. Gao, Y. Yang and X. Wang, "Lightweight Image Super-Resolution with Information Multi-distillation Network," in *Proc. ACM International Conference on Multimedia*, 2019, pp. 2024–2032. doi ·arXiv
40. B. Ren, Y. Li, N. Mehta, R. Timofte et al., "The Ninth NTIRE 2024 Efficient Super-Resolution Challenge Report," in *Proc. IEEE/CVF CVPR Workshops*, 2024. CVF ·arXiv
41. B. Ren, H. Guo, L. Sun, Z. Wu, R. Timofte, Y. Li et al., "The Tenth NTIRE 2025 Efficient Super-Resolution Challenge Report," in *Proc. IEEE/CVF CVPR Workshops*, 2025. CVF ·arXiv
42. ITU-R, "Recommendation BT.601: Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide Screen 16:9 Aspect Ratios." ITU
43. MathWorks, "rgb2ycbcr: Convert RGB color values to YCbCr color space," Image Processing Toolbox documentation. link
44. Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. doi
45. Z. Chen, Z. Wu, E. Zamfir, K. Zhang, Y. Zhang, R. Timofte et al., "NTIRE 2024 Challenge on Image Super-Resolution (×4): Methods and Results," in *Proc. IEEE/CVF CVPR Workshops*, 2024, pp. 6108–6132. doi ·arXiv
46. Z. Chen, K. Liu, J. Gong, J. Wang, L. Sun, Z. Wu, R. Timofte, Y. Zhang et al., "NTIRE 2025 Challenge on Image Super-Resolution (×4): Methods and Results," in *Proc. IEEE/CVF CVPR Workshops*, 2025, pp. 1525–1535. doi ·arXiv

47. M. Bevilacqua, A. Roumy, C. Guillemot and M.-L. Alberi Morel, “Low-Complexity Single-Image Super-Resolution based on Nonnegative Neighbor Embedding,” in *Proc. BMVC*, 2012, pp. 135.1–135.10. doi
48. R. Zeyde, M. Elad and M. Protter, “On Single Image Scale-Up Using Sparse-Representations,” in *Curves and Surfaces*, Lecture Notes in Computer Science, Springer, 2012, pp. 711–730. doi
49. D. Martin, C. Fowlkes, D. Tal and J. Malik, “A Database of Human Segmented Natural Images and Its Application to Evaluating Segmentation Algorithms and Measuring Ecological Statistics,” in *Proc. IEEE ICCV*, 2001, vol. 2, pp. 416–423. doi
50. J.-B. Huang, A. Singh and N. Ahuja, “Single Image Super-Resolution from Transformed Self-Exemplars,” in *Proc. IEEE CVPR*, 2015, pp. 5197–5206. doi ·CVF
51. A. Fujimoto, T. Ogawa, K. Yamamoto, Y. Matsui, T. Yamasaki and K. Aizawa, “Manga109 Dataset and Creation of Metadata,” in *Proc. 1st International Workshop on coMics ANalysis, Processing and Understanding* (*MANPU*), 2016, pp. 1–5. doi
52. Y. Matsui, K. Ito, Y. Aramaki, A. Fujimoto, T. Ogawa, T. Yamasaki and K. Aizawa, “Sketch-Based Manga Retrieval Using Manga109 Dataset,” *Multimedia Tools and Applications*, vol. 76, no. 20, pp. 21811–21838, 2017. doi
53. E. Agustsson and R. Timofte, “NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study,” in *Proc. IEEE CVPR Workshops*, 2017, pp. 1122–1131. doi ·CVF
54. R. Zhang, P. Isola, A. A. Efros, E. Shechtman and O. Wang, “The Unreasonable Effectiveness of Deep Features as a Perceptual Metric,” in *Proc. IEEE/CVF CVPR*, 2018, pp. 586–595. doi ·arXiv
55. W. Ren, R. Goyal, Z. Hu, T. T. Aumentado-Armstrong, I. Mohomed and A. Levinshtein, “Hallucination Score: Towards Mitigating Hallucinations in Generative Image Super-Resolution,” arXiv:2507.14367, 2025. arXiv
56. B. B. Moser, A. S. Shanbhag, F. Raue, S. Frolov, S. Palacio and A. Dengel, “Diffusion Models, Image Super-Resolution, and Everything: A Survey,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 36, no. 7, pp. 11793–11813, 2025. doi ·arXiv
57. A. Shocher, N. Cohen and M. Irani, “ ‘Zero-Shot’ Super-Resolution Using Deep Internal Learning,” in *Proc. IEEE/CVF CVPR*, 2018, pp. 3118–3126. doi ·arXiv

# DLCV 09A.1 • Blind and Real-World Image Super-Resolution

How super-resolution copes with damage nobody measured: estimating or conditioning on the blur kernel, learning degradation embeddings, synthesizing realistic training data, capturing real pairs, and borrowing generative priors, with honest evaluation when there is no ground truth.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/

## What you'll learn

- Distinguish unknown degradation parameters from an uncertain real-world degradation family.
- Compare explicit degradation estimates with implicit representations.
- Understand kernel mismatch, internal image statistics and synthetic training pipelines.
- Trace how generative priors use low-resolution and semantic conditions.
- Evaluate fidelity and perceived quality when a clean reference is unavailable.

## The big picture

DLCV 09A treats super-resolution with a known degradation: we shrink HR images in a fixed way, train a network to undo exactly that, and measure the result against the original. Real photos do not come with a recipe. Their blur depends on the lens, the focus and a shaking hand; their noise depends on the sensor and the light; they were sharpened, resized and compressed by software we never saw. Blind and real-world SR is about making SR work anyway.

> **Explain like I'm five**
>
> Imagine a friend sends you a smudged, shrunken photo and asks you to make it big and sharp. If you knew *exactly* how it got smudged (wet thumb, cheap printer, coffee spill), you could undo that one thing carefully. But you were not there. So you can (1) guess what happened and undo your guess, (2) practise on thousands of photos you smudged yourself in every way you can think of, so that you get good at fixing any smudge, or (3) ask a talented painter who knows what photos usually look like to repaint the missing parts. This chapter is about those three strategies, and about how to check the work when you have no clean copy to compare with.

Why it matters: almost every SR model people actually use (phone zoom, old-photo restoration, upscaling of web images and video frames) faces unknown degradations. A model that is excellent on a bicubic benchmark can be useless on a real JPEG, and a model that looks excellent on a real JPEG can be inventing details. Knowing which family a method belongs to tells you what it assumes, what it can repair, and how it will fail.

**Where this chapter sits.** The degradation model, the perception–distortion trade-off and the standard benchmarks are introduced in DLCV 09A; generic diffusion-based restoration (posterior sampling, latent diffusion, hallucination) is in DLCV 09B; no-reference quality metrics are defined in DLCV 09. We link to them instead of repeating them. One-step and few-step diffusion

SR has its own chapter, DLCV 09A.2. Selected key papers have detailed introductions in the Paper guides; the main text explains the families and their assumptions. All code uses NumPy, SciPy, scikit-image and OpenCV, runs on a CPU in seconds, and uses images from `skimage.data`.

### The problem: damage nobody measured

**Plain version.** With unknown degradation, one observation must support estimates of both the sharp image and its damage process.

**The degradation model, briefly** The classical model, introduced in DLCV 09A, writes the low-resolution (LR) image $y$ as

$$y = (x \circledast k) \downarrow_s + n,$$

where $x$ is the high-resolution (HR) image, $k$ a blur kernel (lens, defocus, motion, pixel integration or an anti-aliasing filter), $\circledast$ 2-D convolution applied per color channel, $\downarrow_s$ downsampling by the scale factor $s$ (keeping one pixel per $s \times s$ block, or a bicubic resize), and $n$ additive noise. Real photos add steps that this model leaves out, most importantly JPEG compression $C_q$ at quality $q$, whose position can vary in a processing chain, and camera processing (demosaicking, denoising, sharpening, tone curves) that makes the noise signal-dependent and spatially varying. A more complete but still simplified chain is $y = C_q((x \circledast k) \downarrow_s + n)$.

**What "blind" and "real-world" mean** The words are used loosely, so it helps to fix them:

- **Non-blind SR**: $k$, $s$ and the noise level are known at test time, either fixed by the task or given to the method. The bicubic benchmark is the special case where they are the same for every image.
- **Blind SR**: at least one of them, usually $k$ and the noise level, is unknown and differs from image to image. Most blind-SR papers still assume the classical model with a Gaussian-like kernel and Gaussian noise; only the parameters are unknown.
- **Real-world SR** (often written Real-ISR): the input is a real photo whose degradation is not only unknown but also *not of the assumed form*: compression, processing, spatially varying blur and noise. There is usually no ground-truth HR image at all.

The distinction matters because methods that are excellent at blind SR in the narrow sense (the right family, unknown parameters) can fail completely in the real-world sense (the wrong family).

**Why bicubic-trained models fail on real photos** **Plain version.** A network trained on one shrinking recipe learns to undo that recipe and nothing else. Feed it a different recipe and it applies the wrong correction with full confidence.

DLCV 09A shows the effect with a tiny linear model: trained on sharp inputs and tested on blurry ones it under-corrects; trained on blurry inputs and tested on sharp ones it over-sharpens. Two classical observations explain why this matters so much. Efrat et al. showed that an accurate blur

model matters more for SR than a sophisticated image prior [1]. And a model trained without noise treats sensor noise and JPEG blocks as fine detail and amplifies them. The SRMD paper states the sign of the blur error precisely: if the assumed kernel is smoother (wider) than the true one, the result shows ringing; if it is sharper (narrower), the result is over-smoothed [2]. We reproduce both cases in the explicit-modelling section below.

### A map of the field

**Plain version.** Ask whether degradation is represented explicitly or learned implicitly, and whether the evidence comes from external data or the input itself.

Liu et al. organize blind SR along exactly these two axes: explicit modelling (the classical kernel-and-noise model or a variant, with or without kernel estimation) versus implicit modelling (a degradation learned from data distributions), and external datasets versus a single input image; they note that the fourth cell, implicit modelling from a single image, was empty at the time [3]. Chen et al. group real-world SR into four families: degradation-modelling-based, image-pairs-based (captured data), domain-translation-based and self-learning-based (the test image itself) [4]. Both surveys predate the diffusion era, so the last row of the table adds the generative-prior family that has expanded since 2023 [5].

| Family | What is estimated or learned | Representative methods | Strengths | Typical failure modes |
|---|---|---|---|---|
| Explicit, kernel given (non-blind conditioning) | Nothing at test time; the network is conditioned on a supplied kernel and noise level | SRMD [2], UDVD [6], USRNet [7] | One model for many known degradations; interpretable controls | Needs the kernel from somewhere; a wrong one gives ringing or blur |
| Explicit, kernel estimated with external data | A kernel code, then the SR image, often in alternation | IKC [8], DAN [9], DCLS [10] | Accurate when the degradation really is Gaussian-like blur plus noise | Breaks on JPEG, camera processing and spatially varying blur |
| Explicit, single image (internal statistics) | The image's own kernel, from recurrence of patches across scales | Michaeli–Irani [11], KernelGAN [12], ZSSR [13] | No training data; adapts to each image | Slow (training per image); weak prior; struggles with noise |
| Degradation representation learning | An abstract embedding that tells degradations apart | DASR [14], KDSR [15] | No kernel labels; one fast forward pass | Embedding only covers the training degradation family |
| Implicit, synthetic degradation pipelines | Nothing explicit; one network trained on a very wide random degradation distribution | BSRGAN [16], Real-ESRGAN [17], Liang et al.'s DASR [18] | Works on many real photos with one model | Over-smoothing or GAN artifacts; out-of-distribution damage is amplified |
| Implicit, captured real pairs | The mapping of one specific camera pair | RealSR [19], DRealSR [20], City100 [21] | Truly realistic training and test data | Small, camera-specific, alignment errors |

| Family | What is estimated or learned | Representative methods | Strengths | Typical failure modes |
|---|---|---|---|---|
| Implicit, unpaired domain translation | A degradation generator or a real-to-clean translator learned with GANs | CinCGAN [22], Bulat et al. [23], FSSR [24], Ji et al. [25] | Learns from real LR images without HR partners | Unstable GAN training, fake textures |
| Generative priors (2023–2026) | A pretrained text-to-image diffusion prior steered by the LR image | StableSR [26], DiffBIR [27], PASD [28], SeeSR [29], SUPIR [30], ResShift [31] | Realistic texture on severely degraded photos | Hallucinated content, low PSNR, slow sampling, color shifts |

Two warnings about the map. First, the boundaries are porous: DAN is explicit but learns its estimator from external data; BSRGAN is "implicit" yet its pipeline is a hand-written explicit model, used only for training. Second, the name **DASR** belongs to two different papers: Wang et al.'s degradation-aware SR with contrastive embeddings [14], and Liang et al.'s efficient degradation-adaptive network [18]. Always check which one a comparison table means.

### Explicit degradation modelling

**Plain version.** Keep the classical recipe $y = (x \circledast k) \downarrow_s + n$ and treat the kernel and the noise level as knobs. Either someone turns the knobs for the network, or the network learns to read the knob settings off the image.

**Telling the network the kernel** One early approach is a single network that can serve many *known* degradations. **SRMD** reduces a $p \times p$ blur kernel to a short vector with principal component analysis (PCA), appends the noise level, and "stretches" each number into a constant map of the LR image's size; these maps are concatenated with the LR image as extra input channels, so one plain CNN learns $\hat{x} = F(y, k, \sigma)$ instead of $F(y)$ [2]. Because the maps can vary across the image, the same network also handles spatially varying blur and noise. **UDVD** replaces concatenation by dynamic convolutions whose filters depend on the degradation, to cope with variation both across and within images [6]. **USRNet** unfolds a classical maximum a posteriori (MAP) solver into a network: a data step that uses the given kernel, scale and noise level in closed form alternates with a learned denoising prior step, so one model serves arbitrary kernels and scale factors [7]. The plug-and-play splitting behind it is explained in DLCV 09.

What to remember: these models are *non-blind*. They move the problem from "learn to undo one degradation" to "be told the degradation", which is only useful if a good estimate exists.

*Online only: SRMD: one network for many known degradations (CVPR 2018) — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/papers/srmd-one-network-for-many-known-degradations-cvpr-2018/*

**Estimating and correcting the kernel** **Plain version.** A wrong kernel leaves a recognizable signature in the output: too narrow gives blur, too wide gives ringing. So look at the output, read the signature, and fix the kernel.

**IKC** (iterative kernel correction) predicts an initial kernel code from the LR image, super-resolves with a kernel-conditioned network, and then repeatedly applies a corrector network that looks at the current SR result and the current kernel estimate and outputs a correction; its kernel-conditioned SR network, SFTMD, injects the kernel through spatial feature transform layers at every residual block instead of only at the input [8], [32]. **DAN** (deep alternating network) folds the alternation into one end-to-end network: an *Estimator* predicts the kernel from the LR image *and* the current SR image, a *Restorer* super-resolves with the current kernel, and the two alternate for a fixed number of rounds with shared weights; because the Restorer is trained on estimated rather than true kernels, it learns to tolerate the Estimator's errors [9]. **DCLS** reformulates the degradation so that the kernel estimation and a least-squares deblurring step happen in LR feature space [10].

In the notation of the classical model, the alternation solves

$$k^{(i+1)} = \arg\min_k \|y - (x^{(i)} \circledast k)\downarrow_s\|_1, \qquad x^{(i+1)} = \arg\min_x \|y - (x \circledast k^{(i+1)})\downarrow_s\|_1 + \phi(x),$$

where $i$ counts rounds, $x^{(i)}$ and $k^{(i)}$ are the current image and kernel estimates, and $\phi$ is an image prior; DAN writes it this way and replaces both minimizations by learned modules [9].

*Online only: IKC: iterative kernel correction (CVPR 2019) — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/papers/ikc-iterative-kernel-correction-cvpr-2019/; DAN: unfolding the alternating optimization (NeurIPS 2020) — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/papers/dan-unfolding-the-alternating-optimization-neurips-2020/*

**Estimating the kernel from the image itself** **Plain version.** Internal kernel estimation uses cross-scale patch recurrence: seek a downsampling kernel that preserves similar patch statistics across scales.

Michaeli and Irani turned this observation into a nonparametric kernel estimator: the correct SR kernel is the downscaling kernel that maximizes the similarity of patches across scales of the LR image [11]. **KernelGAN** learns that downscaler with an image-specific GAN trained only on the test image: a deep *linear* generator downscales the image by 2, a fully convolutional discriminator tries to tell its $7 \times 7$ patches from patches of the original, and the kernel is read off the trained generator and handed to a non-blind SR method such as ZSSR or SRMD [12]. **ZSSR** itself is the internal-learning counterpart for the SR step: it trains a small CNN at test time on LR–lower-resolution pairs made from the input image alone [13]. The price is time: KernelGAN reports about a minute per image on a GPU, and the per-image training of ZSSR adds more.

*Online only: KernelGAN: kernel estimation with an internal GAN (NeurIPS 2019) — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/papers/kernelgan-kernel-estimation-with-an-internal-gan-neurips-2019/*

*Online only: Why the kernel is hard to identify — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/notes/why-the-kernel-is-hard-to-identify/*

**What goes wrong: kernel mismatch** **Plain version.** Give a non-blind solver the wrong kernel and it does exactly what you asked: it undoes the wrong blur. Too little assumed blur leaves the image soft; too much assumed blur sharpens edges that were already sharp until they ring.

The snippet makes this concrete with the simplest possible non-blind ×2 SR: it solves

$$\hat{x}_{\sigma_a} = \arg\min_x \|(x \circledast g_{\sigma_a}) \downarrow_2 - y\|_2^2 + \lambda \|\nabla x\|_2^2$$

with conjugate gradients, where $g_{\sigma_a}$ is a Gaussian kernel of the *assumed* width $\sigma_a$, $y$ was produced with a *true* width $\sigma$, $\nabla$ is the image gradient (implemented through the Laplacian in the normal equations), and $\lambda = 2 \times 10^{-3}$ weights the smoothness prior.

```
import numpy as np
from scipy import ndimage as ndi
from scipy.sparse.linalg import LinearOperator, cg
from skimage import data, img_as_float64

s = 2                                                  # scale factor
x = img_as_float64(data.camera())[128:384, 128:384]    # HR test crop (256 x 256)
H, W = x.shape

def blur(z, sigma):                                    # circular Gaussian blur
 ↪  (self-adjoint)
    return ndi.gaussian_filter(z, sigma, mode="wrap")

def A(z, sigma):                                       # degradation: blur, then keep every
 ↪  s-th pixel
    return blur(z, sigma)[::s, ::s]

def At(r, sigma):                                      # its adjoint: zero-fill upsampling,
 ↪  then blur
    up = np.zeros((H, W)); up[::s, ::s] = r
    return blur(up, sigma)

def lap(z):                                            # discrete Laplacian (smoothness
 ↪  prior)
    return ndi.laplace(z, mode="wrap")

def nonblind_sr(y, sigma, lam=2e-3):
    """Solve min_x ||A_sigma x - y||^2 + lam ||grad x||^2 with conjugate gradients."""
    op = LinearOperator((H * W, H * W), dtype=float,
                        matvec=lambda v: (At(A(v.reshape(H, W), sigma), sigma) - lam *
                         ↪  lap(v.reshape(H, W))).ravel())
    sol, _ = cg(op, At(y, sigma).ravel(), maxiter=200)
    return sol.reshape(H, W)

def psnr(a, b):
    return 10 * np.log10(1.0 / np.mean((a - np.clip(b, 0, 1)) ** 2))
```

```
rng = np.random.default_rng(0)
assumed = [0.6, 1.0, 1.6, 2.2, 2.8]
print("true \\ assumed " + "".join(f"{a:7.1f}" for a in assumed))
for true in [1.0, 1.6, 2.2]:
    y = A(x, true) + 0.003 * rng.standard_normal((H // s, W // s))   # LR observation
    row = [psnr(x, nonblind_sr(y, a)) for a in assumed]
    print(f"sigma = {true:.1f}      " + "".join(f"{p:7.2f}" for p in row))
```

```
true \ assumed     0.6    1.0    1.6    2.2    2.8
sigma = 1.0      26.80  28.43  24.59  19.84  16.91
sigma = 1.6      24.22  25.30  27.23  24.13  19.83
sigma = 2.2      22.45  22.92  24.11  25.51  23.43
```

Every row peaks on the diagonal, where the assumed width equals the true one (our own toy numbers, in dB). The losses are **asymmetric**: assuming too wide a kernel is punished much harder than assuming too narrow a kernel (for true $\sigma = 1.0$, assuming 2.8 costs 11.5 dB, assuming 0.6 costs 1.6 dB), because over-deblurring amplifies noise and creates ringing, while under-deblurring only leaves blur. Figure 1 shows the three cases for true $\sigma = 1.6$. This is the signature IKC learns to read [8].

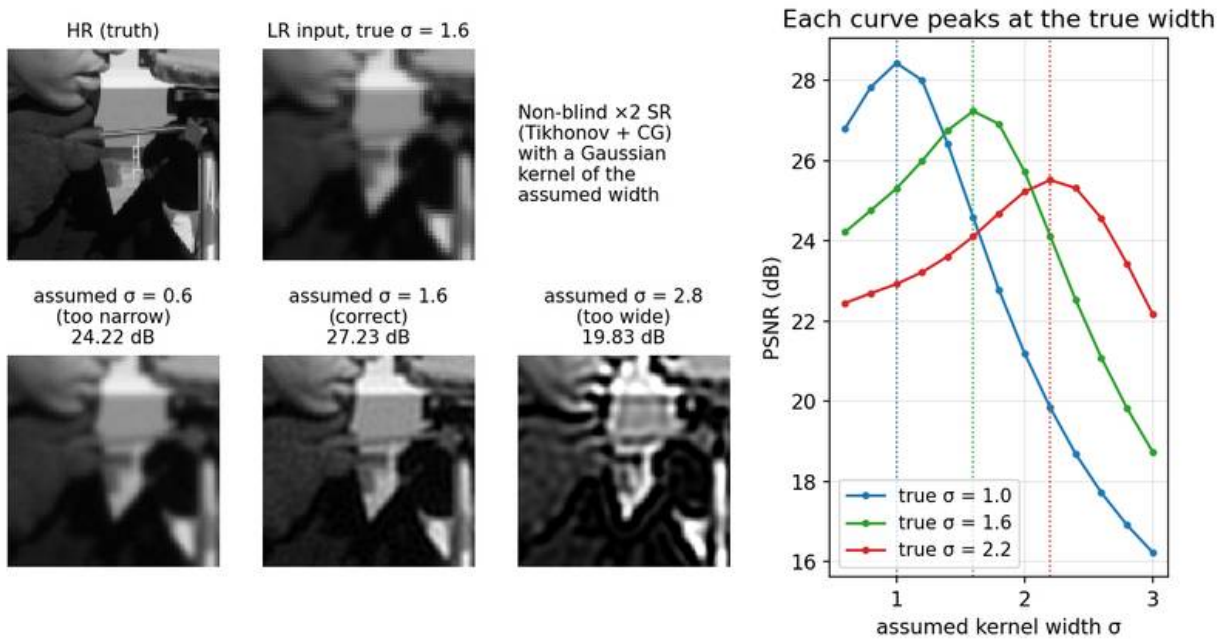



*Figure 1 — Kernel mismatch in non-blind ×2 SR (our toy). Assuming a narrower kernel than the true one leaves the result soft; assuming a wider one over-sharpens and rings. PSNR on the full 256 × 256 crop.*

### Degradation representation learning

**Plain version.** You do not need to know the kernel's exact shape to fix it; you need to know *which kind of damage* this image has, well enough to tell it apart from other kinds. So learn a compact "fingerprint" of the damage, without ever being told the true kernel.

**DASR** (Wang et al.) learns such a fingerprint with contrastive learning [14]. Its assumption is that the degradation is the same everywhere in one image and differs between images. Two patches from the same LR image are therefore a *positive* pair, and patches from other images are *negatives*. An encoder maps each patch to a vector, and an InfoNCE loss with a queue of negatives (as in MoCo [33]) pulls positives together and pushes negatives apart; the resulting representation then predicts the weights of depth-wise convolutions and channel-wise scaling factors inside the SR network. KDSR keeps the "implicit representation" idea but trains a teacher that sees the HR–LR pair and distils its representation into a student that sees only the LR image, which lets the representation cover combinations of blur, noise and JPEG for which no explicit label exists [15]. The other DASR, by Liang et al., goes the opposite way: a tiny regression network predicts explicit degradation parameters and uses them to mix several convolutional experts into one efficient SR network [18].

What to remember: a degradation embedding replaces "estimate the kernel, pixel by pixel" with "recognize the kind of degradation", which avoids explicit kernel labels but still depends on the degradation diversity of the training data and does not guarantee robustness to an unseen family.

*Online only: DASR: contrastive degradation representations (CVPR 2021) — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/papers/dasr-contrastive-degradation-representations-cvpr-2021/*

### Implicit modelling by degradation synthesis

**Plain version.** Stop trying to describe one image's damage. Instead, write a damage generator that is wild enough to produce almost any damage a real photo could have, use it to make millions of training pairs, and train one network on all of them.

**BSRGAN: shuffle the steps** **BSRGAN** argues that blur, downsampling and noise are the key factors, but that real images do not apply them in one fixed order [16]. Its practical degradation model widens each factor (two blurs, one isotropic and one anisotropic Gaussian; several downsamplers including nearest, bilinear, bicubic and a down-up resize; Gaussian noise, JPEG noise and processed camera sensor noise generated with a reverse-forward camera pipeline) and applies them in a **randomly shuffled order**, followed by a final JPEG compression. The bicubic and classical degradation models become special cases. The same ESRGAN-style network [34] is then trained on these pairs, first with an L1 loss (BSRNet) and then with added perceptual and adversarial losses (BSRGAN).

**Real-ESRGAN: repeat the steps** **Real-ESRGAN** keeps the classical order but applies it **twice**, a "second-order" degradation, arguing that a real web image was typically degraded by a camera,

then by editing, then by one or more rounds of compression and resizing [17]. Each round draws blur (Gaussian, generalized Gaussian, plateau-shaped or sinc kernels), a random resize, Gaussian or Poisson noise and JPEG compression with fresh parameters; a final resize and **sinc filter** are treated as a pair whose order relative to JPEG is randomly swapped. Its finite sinc kernels approximate an ideal low-pass filter and can produce ringing and overshoot, which real images acquire from sharpening and compression; training on it teaches the network to remove such halos instead of amplifying them. Because the degradation space is so large, Real-ESRGAN also replaces ESRGAN's discriminator with a U-Net-shaped one [35] that judges every pixel, stabilized with spectral normalization [36].

*Online only: The high-order degradation pipeline, step by step — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/notes/the-high-order-degradation-pipeline-step-by-step/ ; Sinc filters and ringing — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/notes/sinc-filters-and-ringing/*

*Online only: BSRGAN: a practical degradation model (ICCV 2021) — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/papers/bsrgan-a-practical-degradation-model-iccv-2021/ ; Real-ESRGAN: high-order degradation with pure synthetic data (ICCVW 2021) — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/papers/real-esrgan-high-order-degradation-with-pure-synthetic-data-iccvw-2021/*

**How training pairs are produced** The snippet implements a compact second-order pipeline in the spirit of Real-ESRGAN: two rounds of blur, random resize, noise and JPEG, then swapping JPEG with a final resize-and-sinc pair. The blur sizes, the noise ranges of the first round and the JPEG qualities follow the paper [17]. This teaching version simplifies the resize distribution, shot-noise model and kernel family; it sometimes skips the whole second round, whereas the official pipeline skips only its blur. It is not a reproduction of the training recipe.

```
import numpy as np, cv2
from scipy import ndimage as ndi
from scipy.special import j1
from skimage import data, img_as_float32

def gaussian_kernel(rng, size, smax):
    """Random isotropic or anisotropic Gaussian: k(c) ~ exp(-0.5 c^T S^-1 c), normalized."""
    s1, s2 = rng.uniform(0.2, smax, 2)
    if rng.random() < 0.5:
        s2 = s1                                                    # isotropic half of the
        ↪  time
    th = rng.uniform(0, np.pi)
    R = np.array([[np.cos(th), -np.sin(th)], [np.sin(th), np.cos(th)]])
    S = R @ np.diag([s1**2, s2**2]) @ R.T
    ax = np.arange(size) - size // 2
    C = np.stack(np.meshgrid(ax, ax, indexing="ij"), -1)
    k = np.exp(-0.5 * np.einsum("...i,ij,...j", C, np.linalg.inv(S), C))
    return k / k.sum(), f"gauss({s1:.1f},{s2:.1f})"

def sinc_kernel(size, wc):
    """2-D sinc low-pass: k(r) = wc J1(wc r) / (2 pi r), with k(0) = wc^2 / (4 pi)."""
```

```
    ax = np.arange(size) - size // 2
    r = np.hypot(*np.meshgrid(ax, ax, indexing="ij"))
    with np.errstate(invalid="ignore", divide="ignore"):
        k = np.where(r == 0, wc**2 / (4 * np.pi), wc * j1(wc * r) / (2 * np.pi * r))
    return k / k.sum()

def jpeg(img, q):
    ok, buf = cv2.imencode(".jpg", (np.clip(img, 0, 1) * 255).round().astype(np.uint8)[...,
     ↪  ::-1],
                           [cv2.IMWRITE_JPEG_QUALITY, int(q)])
    return cv2.imdecode(buf, cv2.IMREAD_COLOR)[..., ::-1].astype(np.float32) / 255

def one_round(img, rng, log, smax, nmax):
    """blur -> random resize -> noise -> JPEG: one 'classical' degradation."""
    size = int(rng.choice(np.arange(7, 22, 2)))
    if rng.random() < 0.1:
        k, name = sinc_kernel(size, rng.uniform(np.pi / 3, np.pi)), "sinc"
    else:
        k, name = gaussian_kernel(rng, size, smax)
    img = np.stack([ndi.convolve(img[..., c], k, mode="reflect") for c in range(3)], -1)
    f = rng.uniform(0.3, 1.2)                                       # resize factor
     ↪  (illustrative range)
    interp = rng.choice([cv2.INTER_AREA, cv2.INTER_LINEAR, cv2.INTER_CUBIC])
    img = cv2.resize(img, None, fx=f, fy=f, interpolation=interp)
    if rng.random() < 0.5:                                          # Gaussian noise, sigma
     ↪  in 1..nmax (8-bit units)
        sig = rng.uniform(1, nmax) / 255
        shape = img.shape[:2] + ((1,) if rng.random() < 0.4 else (3,))   # gray or color
         ↪  noise
        img = img + sig * rng.standard_normal(shape); noise = f"gauss{sig * 255:.0f}"
    else:                                                           # shot noise with a
     ↪  random photon count
        peak = 10 ** rng.uniform(1.5, 3)
        img = rng.poisson(np.clip(img, 0, 1) * peak) / peak; noise = f"poisson{peak:.0f}"
    q = rng.uniform(30, 95)
    img = jpeg(img, q)
    log.append(f"{name} x{f:.2f} {noise} q{q:.0f}")
    return img

def degrade(hr, rng, scale=4):
    """Second-order pipeline in the spirit of Real-ESRGAN (parameter ranges partly our
     ↪  own)."""
    log = []
    img = one_round(hr, rng, log, smax=3.0, nmax=30)
    if rng.random() < 0.8:
        img = one_round(img, rng, log, smax=1.5, nmax=25)          # second, milder round
    h, w = hr.shape[0] // scale, hr.shape[1] // scale
    def resize_sinc(z):
        z = cv2.resize(z, (w, h), interpolation=cv2.INTER_AREA)
        k = sinc_kernel(15, rng.uniform(np.pi / 3, np.pi))  # same kernel for all color
         ↪  channels
        return np.stack([ndi.convolve(z[..., c], k, mode="reflect") for c in range(3)], -1)
    final = [lambda z: jpeg(z, rng.uniform(30, 95)), resize_sinc]
    if rng.random() < 0.5:
        final = final[::-1]                                         # swap JPEG with the
         ↪  resize-and-sinc pair
```

```
    for op in final:
        img = op(img)
    return np.clip(img, 0, 1), log

hr = img_as_float32(data.astronaut())                                  # (512, 512, 3)
for seed in range(3):
    lr, log = degrade(hr, np.random.default_rng(seed))
    print(f"seed {seed}: LR {lr.shape} | " + " ; ".join(log))
```

```
seed 0: LR (128, 128, 3) | gauss(0.3,0.2) x0.85 poisson207 q64 ; gauss(0.9,0.9) x0.84 gauss5
 ↪  q90
seed 1: LR (128, 128, 3) | gauss(0.6,0.6) x1.04 gauss17 q71 ; gauss(1.4,0.6) x0.40 poisson42
 ↪  q40
seed 2: LR (128, 128, 3) | gauss(2.5,0.5) x0.47 gauss9 q67 ; gauss(0.3,1.1) x1.04 gauss19 q50
```

Each line is one training input: two rounds with different kernels (`gauss(σ1,σ2)` lists the two axis widths), resize factors, noise types and JPEG qualities. In training, the HR patch is paired with its LR version and a fresh degradation is drawn every time, so the network repeatedly encounters newly sampled damage. Figure 2 shows seven such inputs from one face; they differ much more than the bicubic inputs of DLCV 09A would. Real-ESRGAN generates its pairs on the GPU and keeps a pool of 180 recent pairs from which batches are drawn, because a batch produced in one call would otherwise share one resize factor [17].




*Figure 2 — Seven ×4 training inputs drawn from one HR image by the second-order pipeline of the snippet (seeds 0 to 6). One network has to restore all of them.*

**Later refinements** Synthetic degradation is an important data source for the real-world SR methods discussed here. StableSR, SeeSR, SUPIR and ResShift all train on the Real-ESRGAN pipeline [26], [29], [30], [31]. Two lessons came later. **Range can matter more than complexity**: DiffBIR trains the restoration stage that produces its diffusion model's conditions on a classical first-order model with *wide* parameter ranges, and reports better no-reference scores than with the Real-ESRGAN pipeline, which it describes as complex but with smaller ranges [27]. **GAN artifacts need their own treatment**: LDL computes an artifact map from local residual statistics and uses it to penalize artifact regions during GAN training, improving both fidelity and perceptual quality [37].

### Real paired and unpaired data

**Plain version.** Instead of imitating camera damage, photograph it. Either capture the same scene at two zoom levels to get true LR–HR pairs, or, if pairs are impossible, learn from unpaired collections of real low-quality and high-quality images.

**Captured pairs** **RealSR** photographs each scene with a DSLR at several focal lengths; the long-focal-length shot is the HR image and the short one, after registration and color correction, is the LR image [19]. **DRealSR** extends the capture to five DSLR cameras for more diverse degradations [20]. **City100** collects pairs for both a DSLR and a smartphone by changing the focal length or the shooting distance, to study the trade-off between resolution and field of view [21]. The DPED photos, taken simultaneously with phones and a DSLR, are often used as real low-quality test inputs without ground truth [38]. Captured pairs offer reference images for measuring fidelity on real degradations, but they are small, specific to the cameras used, and slightly misaligned, and a model trained on one camera pair does not cover a phone image from the internet.

**Unpaired learning and domain translation** When no pairs exist, GANs can translate between domains. **CinCGAN** first maps a noisy, blurry real LR image to a clean LR domain with a cycle-consistent GAN, super-resolves it with a pretrained network and then fine-tunes both stages end to end [22]. **Bulat et al.** go the other way: a High-to-Low GAN learns, from unpaired HR and real LR face images, how to *degrade* HR images realistically, and its outputs train a Low-to-High network with paired supervision [23]. **Lugmayr et al.** learn to put natural sensor characteristics back into bicubically downscaled images, again to synthesize realistic pairs [39]; **frequency separation** (DSGAN and FSSR) applies the adversarial loss only to high frequencies, because downsampling preserves the low ones [24]. **Ji et al.** estimate real blur kernels and extract real noise patches from the test-domain images and inject both into the classical model, which won the NTIRE 2020 real-world SR challenge [25], [40]. Liu et al. note that this data-distribution family is data-hungry and that GAN-based training often produces severe artifacts and fake textures [3].

### Real-world SR with generative priors

**Plain version.** The networks above learn what photos look like from a few thousand training images. Large text-to-image diffusion models supply priors learned from broad image collections. Since about 2023, several real-world SR methods keep such a model (almost) frozen and add a small network that shows it the LR image, so that it "paints in" detail that fits.

**Why a generative prior** When the input is badly damaged, most of the HR detail is simply not in the LR image, and an L2 regression network targets a conditional mean, which can average incompatible details; other losses target different statistics. A generative model instead returns one plausible answer. DLCV 09B explains why this costs PSNR, how diffusion sampling works (diffusion in one page), and why a strong prior can invent content (the hallucination problem). Here we concentrate on what is specific to SR: how each method feeds the LR image to the prior, what extra condition it uses, and what knob it offers between fidelity and realism.

All methods in this section train on synthetic pairs from the Real-ESRGAN pipeline or a similar one, and most build on Stable Diffusion, a latent diffusion model: an autoencoder compresses a $512 \times 512$ image into a $64 \times 64$ latent (a factor of 8 per side), and the denoising network works in that latent space [41]. Many adapt ControlNet, which attaches a trainable copy of the denoiser's encoder to a frozen model through zero-initialized convolutions [42].

#### What each method conditions on

| Method | Prior | How the LR image enters | Extra condition | Fidelity knob | Steps (as reported) |
|---|---|---|---|---|---|
| StableSR [26] | Stable Diffusion 2.1, frozen | Time-aware encoder + spatial feature transform (SFT) layers in every residual block | none (null prompt) | Controllable feature wrapping weight $w$ in the decoder | 200 (DDPM) |
| DiffBIR [27] | Stable Diffusion 2.1, frozen | First cleaned by a restoration network, then the *cleaned* image enters an IRControlNet | negative prompt | Region-adaptive restoration guidance scale | 50 |
| PASD [28] | Stable Diffusion | Degradation-removal module and pixel-aware cross-attention | high-level semantic features and captions | Adjustable noise schedule | 20 |
| SeeSR [29] | Stable Diffusion 2, frozen | ControlNet on the LR latent; LR latent also mixed into the starting noise | Degradation-aware tags (hard prompts) and tagger embeddings (soft prompts) | — | 50 |

| Method | Prior | How the LR image enters | Extra condition | Fidelity knob | Steps (as reported) |
|---|---|---|---|---|---|
| SUPIR [30] | SDXL, frozen | Degradation-robust encoder, trimmed ControlNet with ZeroSFT connectors | Captions from a multimodal LLM; negative-quality prompts | Restoration-guided sampling $\tau_r$ | 100 (EDM) |
| ResShift [31] | Trained from scratch (latent space of a VQGAN) | The chain *starts* at the LR image and shifts the residual | none | Noise schedule $\kappa$, $p$ | 15 |

Step counts are the defaults reported in each paper; the SeeSR supplementary material times them side by side on one V100 GPU for a 128 → 512 task: 18.70 s for StableSR (200 steps), 5.85 s for DiffBIR (50), 6.07 s for PASD (20), 7.24 s for SeeSR (50) and 1.12 s for ResShift (15), against 0.09 s for Real-ESRGAN [29].

Three design ideas recur:

- **Conditioning strength must vary with the noise level.** StableSR adds a time embedding to its condition encoder and finds that the encoder supplies its strongest guidance around a signal-to-noise ratio of about $5 \times 10^{-2}$, where image content is formed [26].
- **Do not show the prior the raw degradation.** DiffBIR observes that a diffusion model conditioned directly on noisy or compressed inputs turns the artifacts into content, so it removes the degradation first with an ordinary restoration network and only then regenerates detail [27]. SUPIR fine-tunes its autoencoder's encoder to be robust to degradations for the same reason [30].
- **Semantics help where pixels are gone.** PASD and SeeSR extract text from the LR image and feed it to the prior; SeeSR trains its tagger to give the same tags for degraded and clean images, so that a blurry ship is not "recognized" and then painted as a bird [28], [29]. SUPIR uses a multimodal large language model to caption the input and lets users steer restoration with text [30].

*Online only: StableSR: a time-aware encoder for a frozen diffusion prior (IJCV 2024) — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/papers/stablesr-a-time-aware-encoder-for-a-frozen-diffusion-prior-ijcv-2024/; DiffBIR: remove the degradation, then regenerate (ECCV 2024) — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/papers/diffbir-remove-the-degradation-then-regenerate-eccv-2024/; SeeSR: degradation-aware semantic prompts (CVPR 2024) — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/papers/seesr-degradation-aware-semantic-prompts-cvpr-2024/; SUPIR: scaling up the prior, the data and the text (CVPR 2024) — https://blog.shyandram.dev/en/learn/deep-learning/blind-real-world-super-resolution/papers/supir-scaling-up-the-prior-the-data-and-the-text-cvpr-2024/*

**Diffusion transformers and newer priors** **Plain version.** The denoiser inside the prior is changing from a U-Net to a transformer, and SR methods follow.

DreamClear builds a restorer on a diffusion transformer (DiT), curates one million synthetic high-quality training images with a generative pipeline, and mixes restoration experts with token-wise degradation priors [43]. DiT4SR feeds the LR latent into the attention of a large DiT as its own stream, so that the LR guidance evolves with the diffusion process instead of being injected once, plus a cross-stream convolution for local detail [44]. The trend is the same as in text-to-image generation: bigger and better priors, with the conditioning moved deeper inside the model.

**Fidelity, hallucination and the knob** **Plain version.** A stronger prior paints more convincingly, and paints more that was not there. Methods therefore use controls between "stay close to the input" and "look real".

The knobs differ in form but not in purpose: StableSR's $w \in [0,1]$ blends encoder features of the LR image into the decoder [26]; DiffBIR's guidance scale $s$ pulls each intermediate estimate toward the cleaned image, more strongly in flat regions [27]; SUPIR's $\tau_r$ interpolates each step's prediction toward the LR latent with a weight that decays during sampling [30]; ResShift's $p$ controls how fast the residual is shifted [31]. These controls change the balance of reference fidelity and generated texture; individual metric curves need not be monotonic. StableSR's coefficient sweep even improves several scores together for some settings [26]. Two hallucination patterns are specific to SR: **semantic errors**, where a degraded object is reconstructed as a different object (SeeSR's examples of a ship turned into a bird or a fish), and **texture invention in flat regions**, where the random starting noise of a sampler is turned into detail on sky or walls (the reason for SeeSR's LR-embedding start and for SUPIR's restoration-guided sampling) [29], [30]. Small text and small faces remain difficult for all of them; StableSR, SeeSR and OSEDiff each name these as limitations [26], [29], [45].

**Fewer steps** **Plain version.** Fifty or two hundred network evaluations per image is too slow for a phone or for video. For SR, the LR image already fixes most of the content, so a diffusion model does not need to start from pure noise, and a few steps, or one, can be enough.

ResShift's 15-step chain is the bridge: it starts from the LR image instead of noise [31]. Distilled and one-step methods go further: SinSR distils ResShift into a single step [46], and OSEDiff fine-tunes Stable Diffusion with low-rank adapters to map the LR latent to the HR latent in one step, regularized by variational score distillation [45]. These one-step and few-step methods, and how their speed is measured, have their own chapter: DLCV 09A.2.

### Evaluation in the blind and real-world setting

**Plain version.** On real photos there is no ground truth, so you cannot compute an error. You can test on synthetic damage where the truth is known, score real outputs with a model of human judgement, or ask people. Each answer is partial, and rankings can disagree.

**Full-reference metrics on synthetic test sets** Blind-SR papers in the explicit family use fixed synthetic protocols so that kernels are known: **Gaussian8** (eight isotropic Gaussian kernels per

scale, applied before bicubic downsampling, defined in IKC [8]) and **DIV2KRK** (the 100 DIV2K validation images [47], each with its own random anisotropic, noise-perturbed kernel, defined in KernelGAN [12]). Real-world methods use synthetic test sets made with their own training pipeline, for example 3,000 DIV2K-validation crops degraded with the Real-ESRGAN pipeline in StableSR, SeeSR and OSEDiff [26], [29], [45], or BSRGAN's DIV2K4D with four degradation types [16]. On these sets PSNR and SSIM [48] measure distortion, and LPIPS [49] and DISTS [50] measure perceptual distance to the reference. A caveat: a test set made with a method's own training pipeline favors that method.

For captured pairs (RealSR, DRealSR) the same metrics apply, but misalignment between the two zoom levels lowers every method's PSNR and rewards blur; KernelGAN gives the same reason for not using the NTIRE 2018 blind-SR data [12].

**No-reference metrics** **Plain version.** A no-reference (NR) metric predicts how good an image looks without seeing the original. It is a learned or hand-made model of human opinion, and it can be fooled.

The common NR metrics in real-world SR papers are **NIQE** (distance of an image's natural-scene statistics from those of pristine images; lower is better) [51], **MUSIQ** (a multi-scale transformer trained on human ratings; higher is better) [52], **MANIQA** (an attention-based model that won an NTIRE 2022 IQA challenge track; higher is better) [53] and **CLIPIQA** (similarity of the image's CLIP embedding to the prompts "Good photo." and "Bad photo."; higher is better) [54]. Their definitions and validation are in DLCV 09. FID, a distance between feature distributions of a set of outputs and a set of real images, is also reported on paired sets [55].

The snippet shows in miniature why such scores must be read with care. It compares a blurry output, an unsharp-masked output and a blurry output with added fine noise, and scores each with PSNR and with a deliberately crude NR "crispness" score, the mean gradient magnitude. Real NR metrics are far better than this, but they share its basic weakness: they score the look of an image, not its correctness.

```
import numpy as np
from scipy import ndimage as ndi
from skimage import data, img_as_float64
from skimage.color import rgb2gray

hr = img_as_float64(rgb2gray(data.astronaut()))
rng = np.random.default_rng(0)

def psnr(a, b):
    return 10 * np.log10(1.0 / np.mean((a - np.clip(b, 0, 1)) ** 2))

def crispness(img):
    """A crude no-reference score: mean gradient magnitude (higher = 'sharper looking')."""
    return np.mean(np.hypot(ndi.sobel(img, 0), ndi.sobel(img, 1)))

smooth = ndi.gaussian_filter(hr, 1.5)                              # a cautious, blurry
 ↪  restoration
candidates = {
```

```
    "smooth (posterior-mean-like)": smooth,
    "unsharp-masked x2": smooth + 2.0 * (smooth - ndi.gaussian_filter(smooth, 2)),
    "smooth + fake fine texture": smooth + 0.04 * rng.standard_normal(hr.shape),
}
for name, img in candidates.items():
    img = np.clip(img, 0, 1)
    print(f"{name:28s} PSNR {psnr(hr, img):6.2f} dB | crispness {crispness(img):.3f}")
print(f"{'ground truth':28s} PSNR    -     | crispness {crispness(hr):.3f}")
```

```
smooth (posterior-mean-like) PSNR  26.82 dB | crispness 0.170
unsharp-masked x2            PSNR  28.42 dB | crispness 0.259
smooth + fake fine texture   PSNR  24.48 dB | crispness 0.262
ground truth                 PSNR    -     | crispness 0.246
```

Both "sharpened" outputs score *crisper than the ground truth* (our toy numbers), and the one with fake texture is the crispest of all while having the lowest PSNR. PSNR, in turn, prefers the unsharp-masked output only because sharpening partly undid the blur; it would also prefer a slightly blurry correct image to a sharp image with shifted texture. Figure 3 shows the crops.

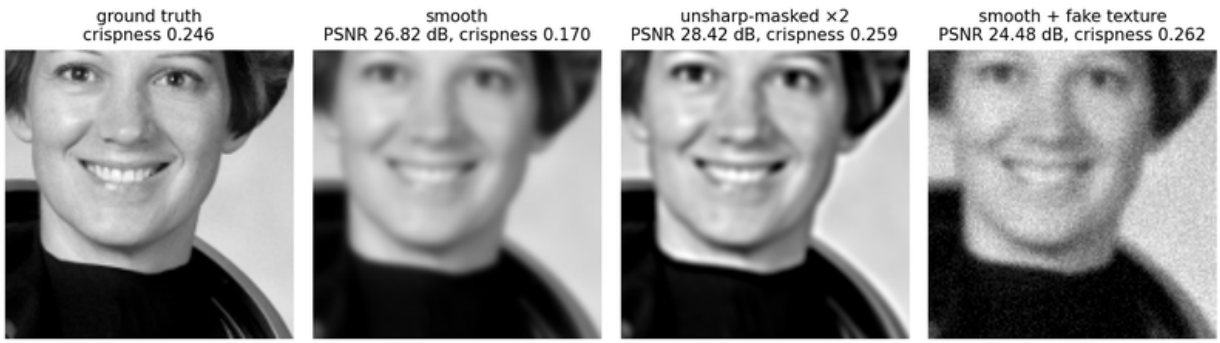



*Figure 3 — A crude no-reference score rewards sharpening and fake texture above the ground truth, while PSNR penalizes the fake texture. Real NR metrics are more sophisticated, but the same tension appears in published comparisons.*

**Why rankings disagree** The published tables show the same tension at full scale:

- **NR metrics versus visual quality.** On BSRGAN's 20 real images (RealSRSet), BSRGAN has the worst NIQE of the methods it compares (5.60, against 3.99 to 4.95 for the others), although its authors find its results visually better; they conclude that these NR metrics "do not always match perceptual visual quality" and should be updated for new degradation types [16].
- **Fidelity versus realism.** In StableSR's comparison on DIV2K validation crops, StableSR has the best FID, CLIPIQA and MUSIQ but a lower PSNR (23.26 dB) than several GAN methods (up to 24.62 dB) [26]. In SUPIR's synthetic tests SUPIR leads every NR metric and trails in PSNR, SSIM and LPIPS, and its authors show cases where the metrics prefer a visibly worse result [30].

- **NR metrics versus each other.** In OSEDiff's tables, multi-step methods (SeeSR, PASD) score better on NIQE, MUSIQ and MANIQA, while OSEDiff is better on LPIPS, DISTS, FID and CLIP-IQA [45]. Which method "wins" depends on the metric chosen.
- **Prompts move NR scores.** StableSR shows that classifier-free guidance with negative prompts raises CLIPIQA and MUSIQ, and that too strong guidance over-sharpens [26]. A score that a sampling trick can raise is not an independent judge.

Rule of thumb: report at least one full-reference metric on a synthetic or captured set, two NR metrics from different families on real images, and a user study for headline claims; state the test resolution and crop protocol, because methods trained at 512 × 512 behave differently on small crops (StableSR reports that SUPIR does poorly on its small crops but better at large resolutions [26]).

**User studies and challenges** User studies measure perceived quality or preference, not whether an invented detail belongs to the scene. For example, StableSR collected 1,400 votes from 35 participants on 40 real images [26]; SeeSR measured how often 20 participants confused its output with the ground truth (a confusion rate of 38.6%) and how often they chose it as the best real-world result (57.1%) [29]. Good practice (hidden method names, randomized order, enough images and raters) is described in DLCV 09.

The challenges trace the field's shift from known to unknown degradations:

- **NTIRE 2018** SR had one bicubic track and three tracks with realistic unknown downgrading operators that simulated a camera pipeline, learnable from provided training pairs [56].
- **NTIRE 2019** Real Image SR used captured pairs, introducing RealSR (DSLR images at two focal lengths) [57].
- **AIM 2019** and **NTIRE 2020** Real-World SR provided no pairs at all, only source-domain LR images (and in one track unpaired high-quality targets), and ranked entries by perceptual quality, in 2020 through a human study [58], [40].
- **AIM 2020** Real Image SR used captured pairs at ×2, ×3 and ×4 and ranked by PSNR and SSIM [59].
- **NTIRE 2025** ×4 SR returned to bicubic inputs but added a perceptual track ranked by a perceptual score next to the PSNR-ranked restoration track [60].

### Failure modes

**Plain version.** Real-world SR fails in two directions: it misreads the damage (and so removes too little or too much), or it trusts its prior too much (and so invents).

- **Out-of-distribution degradations.** A synthesis-trained model has no guarantee of handling damage outside its training pipeline; Real-ESRGAN lists unknown, out-of-distribution degradations among its limitations and notes that it may even amplify them [17].
- **Kernel misestimation.** In explicit methods a wrong kernel gives the blur or ringing of Figure 1; spatially varying blur breaks the assumption of a single kernel.

- **GAN textures and artifacts.** Real-ESRGAN reports twisted lines from aliasing and unpleasant GAN artifacts [17]; BSRGAN reports "bubble" artifacts in texture regions [16]; LDL was designed to suppress such artifacts [37].
- **Semantic hallucination.** Diffusion-prior methods can reconstruct the wrong object or invent texture in flat regions; degraded small text and faces are the hardest cases [29], [26].
- **Color shifts.** Diffusion models can shift colors; StableSR corrects them by matching each output channel's mean and variance to the input, or by replacing the output's low-frequency wavelet band with the input's [26].
- **Tiling seams and resolution effects.** Models trained on $512 \times 512$ crops need tiling for large images; StableSR shows block inconsistency without its aggregation sampling, and performance depends on the test resolution [26].
- **Metric gaming.** Sharpening, added grain or prompt tricks can raise NR scores without improving fidelity (Figure 3).

A cheap sanity check works for every method: degrade the SR output back to the input size with a plausible model and compare it with the input. Large differences in flat regions or on text mean that the method changed content the input did measure. As DLCV 09B explains, passing this check is a useful diagnostic but not a proof of correctness: the true degradation is itself unknown.

### Modern view

**Survey papers** **Plain version.** Two surveys define the vocabulary of this chapter, one for blind SR and one for real-world SR; a diffusion-SR survey and a broad recent SR survey extend them.

- **Liu et al., "Blind Image Super-Resolution: A Survey and Beyond" (TPAMI)** [3]. *Covers*: blind SR methods, datasets and earlier challenges, with a comparison on synthetic and real images. *Taxonomy*: explicit modelling with external data (image-specific adaptation without kernel estimation, such as SRMD, UDVD and USRNet, and with kernel estimation, such as IKC and DAN), explicit modelling with a single image (internal statistics: KernelGAN, ZSSR), and implicit modelling with external data (data-distribution learning with GANs: CinCGAN, FSSR and others). *Takeaways*: the taxonomy reveals a gap, implicit modelling from a single image; data-distribution methods are data-hungry and produce GAN artifacts; fair comparison is hard because methods assume different degradations.
- **Chen et al., "Real-World Single Image Super-Resolution: A Brief Review" (Information Fusion 2022)** [4]. *Covers*: real-world datasets (RealSR, City100, DRealSR and others) and metrics, with comparisons of quality and speed. *Taxonomy*: degradation-modelling-based (IKC, DAN, KernelGAN), image-pairs-based, domain-translation-based (CinCGAN) and self-learning-based (ZSSR). *Takeaway*: results on synthetic data overestimate how well methods super-resolve real images.
- **Moser et al., "Diffusion Models, Image Super-Resolution, and Everything" (TNNLS 2025)** [5]. *Covers*: diffusion foundations and SR methods by input domain, conditioning, guidance and corruption space. *Takeaways*: realism has improved greatly; compute, comparability across papers and color shifts remain open.
- **Zhang et al., "Deep-Learning-Empowered Super Resolution: A Comprehensive Survey and**

**Future Prospects" (Proceedings of the IEEE 2025)** [61]. *Covers*: single-image, video, stereo and light-field SR, with a taxonomy by backbone (CNN, transformer, GAN, diffusion), datasets, evaluation protocols and complexity. *Use*: a wide index into the literature, including the real-world and diffusion methods of this chapter.

### State of the art, 2023–2026

1. **Generative priors form an important family of real-world SR methods.** StableSR, DiffBIR, PASD, SeeSR and SUPIR attach a frozen text-to-image model through encoders, ControlNet variants and SFT-style connectors [26], [27], [28], [29], [30]; DiT-based restorers such as DreamClear and DiT4SR follow the move of generators to transformers [43], [44].
2. **Semantics enter as conditions**: degradation-aware tags, captions from multimodal models, and text control [29], [30].
3. **Synthetic pipelines are still the data source**: almost every method trains on BSRGAN- or Real-ESRGAN-style degradations, with the training images scaled up (SUPIR reports 20 million images with text; DreamClear one million curated synthetic images) [30], [43].
4. **Speed is an important engineering objective**: from 200 steps to 15 (ResShift) and to one step (SinSR, OSEDiff and later methods; see DLCV 09A.2) [31], [46], [45].
5. **Explicit blind SR continues in the classical setting** with better estimators and representations (DCLS, KDSR), benchmarked on Gaussian8 and DIV2KRK [10], [15].

### Open problems

- **Faithfulness.** No metric reliably separates "sharp and correct" from "sharp and invented"; recent proposals score hallucination directly with multimodal models [62].
- **Degradations beyond the pipeline.** Spatially varying blur, camera processing, repeated social-media compression and video codecs are only partly covered by synthetic pipelines.
- **Evaluation without ground truth.** NR metrics disagree with each other and with people, can be raised by sampling tricks, and were not designed for diffusion outputs [16], [26].
- **Cost.** Large priors (billions of parameters) and multi-step sampling remain far from phone budgets.
- **Implicit modelling from a single image**, was a gap highlighted by Liu et al.'s survey; it is a useful historical distinction, not evidence that no later method addresses it [3].

## Key takeaways

- State what is unknown: degradation parameters, the degradation family or both.
- Classify methods by explicit/implicit degradation handling and external/internal evidence.
- Kernel mismatch can cause blur or ringing; evaluate the estimator and the reconstruction together.
- Synthetic degradation broadens training coverage but cannot guarantee performance on unseen damage.
- Generative priors can supply realistic texture while inventing unsupported detail.

- Combine reference-based tests, several no-reference metric families and user studies; expect different rankings.

## Exercises

1. **The asymmetry of kernel errors.** In the kernel-mismatch snippet, (a) set the noise to zero and repeat the table; (b) increase $\lambda$ to $10^{-2}$. How does the penalty for assuming too wide a kernel change in each case, and why?

   **Hint**

   Too wide an assumed kernel asks the solver to amplify frequencies that the true kernel did not attenuate, so it boosts noise and creates overshoot. Without noise the loss on the right of each curve shrinks (only ringing remains); a larger $\lambda$ damps the amplification and flattens both sides of the curve, at the cost of a lower peak. Print the full rows and compare the drop from the diagonal entry to the entry two steps to the right with the drop two steps to the left.

2. **Shuffle versus repeat.** Modify `degrade` in the pipeline snippet so that it can also run in a BSRGAN-like mode: build a list of operations (isotropic blur, anisotropic blur, downsampling by 4, noise, JPEG), shuffle it with `rng.permutation`, apply it, and finish with JPEG. Compare the two modes by the average PSNR of the bicubic upsampling of 50 random LR images against the HR image. Which mode produces harder inputs on average?

   **Hint**

   Downsampling placed *before* blur or noise means the blur acts on LR pixels and the noise is not smoothed by the resize, so shuffled sequences include some very hard cases. Report the mean and the spread over the 50 images; the spread tells you more about the coverage of the degradation space than the mean.

3. **Reading a no-reference metric.** Extend the evaluation snippet with two more candidates: the ground truth with mild JPEG compression (quality 50) and the ground truth shifted by one pixel. Rank all candidates by PSNR and by crispness. Which candidate shows best that PSNR punishes things a viewer would never notice?

   **Hint**

   A one-pixel shift can look very similar to the ground truth but has a much lower PSNR; JPEG at quality 50 reduces crispness slightly and adds block edges that may even raise it. This is why captured pairs, which are never perfectly aligned, reward blurrier methods.

4. **Choosing a method.** For each scenario, choose a family from the map and justify it in two sentences: (a) restoring scanned 1990s family photos for a print album; (b) enlarging microscope images for cell counting; (c) a phone's 2× digital zoom that must run in 30 ms; (d) super-resolving frames from one known security camera model for which you can capture calibration pairs.

**Hint**

(a) A generative-prior method with a moderate fidelity knob: unknown, mixed damage and a goal of pleasant images. (b) Not a generative prior: counts must not be invented; use a non-blind or captured-pair model with a measured point spread function. (c) Start by measuring a compact synthesis-trained regression or GAN network. Compare a one-step diffusion model only after including its encoder, decoder and memory costs in the target-device budget. (d) Captured pairs for that camera, possibly combined with an explicit kernel measured from calibration targets.

5. **The KernelGAN ×4 kernel.** KernelGAN estimates a ×2 kernel $k_2$ and derives the ×4 kernel as $k_4 = k_2 \circledast k_2^{\uparrow}$, where $k_2^{\uparrow}$ is $k_2$ with zeros inserted between its taps. Using two 1-D Gaussians, verify numerically that downscaling by 2 with $k_2$ twice equals downscaling by 4 once with $k_4$, and find the width of $k_4$ when $k_2$ has width $\sigma$.

**Hint**

Downscaling twice means $((y \circledast k_2)\downarrow_2 \circledast k_2)\downarrow_2$. Moving the second convolution in front of the first decimation dilates it by 2. For Gaussians the widths add in quadrature: $\sqrt{\sigma^2 + (2\sigma)^2} = \sqrt{5}\,\sigma$. Test with a random 1-D signal of length 256 and `np.convolve`.

## References


1. N. Efrat, D. Glasner, A. Apartsin, B. Nadler and A. Levin, "Accurate Blur Models vs. Image Priors in Single Image Super-Resolution," in *Proc. IEEE ICCV*, 2013, pp. 2832–2839. doi ·CVF
2. K. Zhang, W. Zuo and L. Zhang, "Learning a Single Convolutional Super-Resolution Network for Multiple Degradations," *CVPR*, 2018. paper
3. A. Liu, Y. Liu, J. Gu, Y. Qiao, C. Dong, "Blind Image Super-Resolution: A Survey and Beyond," *IEEE TPAMI* (*early access*), 2022. source
4. Honggang Chen, Xiaohai He, Linbo Qing, Yuanyuan Wu, Chao Ren, Ray E. Sheriff, Ce Zhu, "Real-world single image super-resolution: A brief review," *Information Fusion*, 2022. source
5. B. B. Moser, A. S. Shanbhag, F. Raue, S. Frolov, S. Palacio and A. Dengel, "Diffusion Models, Image Super-Resolution, and Everything: A Survey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 36, no. 7, pp. 11793–11813, 2025. doi ·arXiv
6. Y. Xu, S. R. Tseng, Y. Tseng, H. Kuo, Y. Tsai, "Unified Dynamic Convolutional Network for Super-Resolution with Variational Degradations," *CVPR*, 2020. source
7. K. Zhang, L. Van Gool, R. Timofte, "Deep Unfolding Network for Image Super-Resolution," *CVPR*, 2020. source
8. J. Gu, H. Lu, W. Zuo and C. Dong, "Blind Super-Resolution With Iterative Kernel Correction," *CVPR*, 2019. paper
9. Z. Luo, Y. Huang, S. Li, L. Wang and T. Tan, "Unfolding the Alternating Optimization for Blind Super Resolution," *NeurIPS*, 2020. paper
10. Z. Luo, H. Huang, L. Yu, Y. Li, H. Fan and S. Liu, "Deep Constrained Least Squares for Blind Image Super-Resolution," *CVPR*, 2022. paper
11. T. Michaeli, M. Irani, "Nonparametric Blind Super-resolution," *Proceedings of the IEEE International Conference on Computer Vision*, 2013. source

12. S. Bell-Kligler, A. Shocher and M. Irani, “Blind Super-Resolution Kernel Estimation using an Internal-GAN,” *NeurIPS*, 2019. paper
13. A. Shocher, N. Cohen and M. Irani, “ ‘Zero-Shot’ Super-Resolution Using Deep Internal Learning,” in *Proc. IEEE/CVF CVPR*, 2018, pp. 3118–3126. doi ·arXiv
14. L. Wang, Y. Wang, X. Dong, Q. Xu, J. Yang, W. An and Y. Guo, “Unsupervised Degradation Representation Learning for Blind Super-Resolution,” *CVPR*, 2021. paper
15. B. Xia, Y. Zhang, Y. Wang, Y. Tian, W. Yang, R. Timofte and L. Van Gool, “Knowledge Distillation Based Degradation Estimation for Blind Super-Resolution,” *ICLR*, 2023. paper
16. K. Zhang, J. Liang, L. Van Gool, R. Timofte, “Designing a Practical Degradation Model for Deep Blind Image Super-Resolution,” *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021. source
17. X. Wang, L. Xie, C. Dong, Y. Shan, “Real-ESRGAN: Training Real-World Blind Super-Resolution With Pure Synthetic Data,” *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021. source
18. J. Liang, H. Zeng, L. Zhang, “Efficient and Degradation-Adaptive Network for Real-World Image Super-Resolution,” *ECCV*, 2022. source
19. J. Cai, H. Zeng, H. Yong, Z. Cao, L. Zhang, “Toward Real-World Single Image Super-Resolution: A New Benchmark and a New Model,” *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019. source
20. P. Wei, Z. Xie, H. Lu, Z. Zhan, Q. Ye, W. Zuo, L. Lin, “Component Divide-and-Conquer for Real-World Image Super-Resolution,” *ECCV*, 2020. source
21. C. Chen, Z. Xiong, X. Tian, Z. Zha, F. Wu, “Camera Lens Super-Resolution,” *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019. source
22. Y. Yuan, S. Liu, J. Zhang, Y. Zhang, C. Dong, L. Lin, “Unsupervised Image Super-Resolution using Cycle-in-Cycle Generative Adversarial Networks,” *CVPR Workshops*, 2018. source
23. A. Bulat, J. Yang, G. Tzimiropoulos, “To learn image super-resolution, use a GAN to learn how to do image degradation first,” *ECCV*, 2018. source
24. M. Fritsche, S. Gu, R. Timofte, “Frequency Separation for Real-World Super-Resolution,” *ICCV Workshops*, 2019. source
25. X. Ji, Y. Cao, Y. Tai, C. Wang, J. Li, F. Huang, “Real-World Super-Resolution via Kernel Estimation and Noise Injection,” *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020. source
26. J. Wang, Z. Yue, S. Zhou, K. C. K. Chan, C. C. Loy, “Exploiting Diffusion Prior for Real-World Image Super-Resolution,” *IJCV*, 2024. source
27. X. Lin, J. He, Z. Chen, et al., “DiffBIR: Towards Blind Image Restoration with Generative Diffusion Prior,” *ECCV*, 2024. source
28. T. Yang, R. Wu, P. Ren, X. Xie, L. Zhang, “Pixel-Aware Stable Diffusion for Realistic Image Super-resolution and Personalized Stylization,” *ECCV*, 2024. source
29. R. Wu, T. Yang, L. Sun, Z. Zhang, S. Li, L. Zhang, “SeeSR: Towards Semantics-Aware Real-World Image Super-Resolution,” *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024. source ·supplementary experiments
30. F. Yu, J. Gu, Z. Li, et al., “Scaling Up to Excellence: Practicing Model Scaling for Photo-Realistic Image Restoration In the Wild,” *CVPR*, 2024. source
31. Z. Yue, J. Wang, C. C. Loy, “ResShift: Efficient Diffusion Model for Image Super-resolution by

Residual Shifting," *NeurIPS*, 2023. source
32. X. Wang, K. Yu, C. Dong, C. C. Loy, "Recovering Realistic Texture in Image Super-Resolution by Deep Spatial Feature Transform," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018. source
33. K. He, H. Fan, Y. Wu, S. Xie, R. Girshick, "Momentum Contrast for Unsupervised Visual Representation Learning," *CVPR*, 2020. source
34. X. Wang, K. Yu, S. Wu, J. Gu, Y. Liu, C. Dong, Y. Qiao and C. C. Loy, "ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks," in *Proc. ECCV Workshops*, 2018 (Lecture Notes in Computer Science, 2019), pp. 63–79. doi ·arXiv
35. O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Proc. MICCAI*, 2015, pp. 234–241. doi ·arXiv
36. T. Miyato, T. Kataoka, M. Koyama, Y. Yoshida, "Spectral Normalization for Generative Adversarial Networks," *ICLR*, 2018. source
37. J. Liang, H. Zeng, L. Zhang, "Details or Artifacts: A Locally Discriminative Learning Approach to Realistic Image Super-Resolution," *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022. source
38. A. Ignatov, N. Kobyshev, R. Timofte, K. Vanhoey, L. Van Gool, "DSLR-Quality Photos on Mobile Devices with Deep Convolutional Networks," *ICCV*, 2017. source
39. A. Lugmayr, M. Danelljan, R. Timofte, "Unsupervised Learning for Real-World Super-Resolution," *arXiv preprint*, 2019. source
40. A. Lugmayr, M. Danelljan, R. Timofte, et al., "NTIRE 2020 Challenge on Real-World Image Super-Resolution: Methods and Results," *CVPR Workshops*, 2020. source
41. R. Rombach, A. Blattmann, D. Lorenz, P. Esser, B. Ommer, "High-Resolution Image Synthesis with Latent Diffusion Models," *CVPR*, 2022. source
42. L. Zhang, A. Rao, M. Agrawala, "Adding Conditional Control to Text-to-Image Diffusion Models," *ICCV*, 2023. source
43. Y. Ai, X. Zhou, H. Huang, X. Han, Z. Chen, Q. You, H. Yang, "DreamClear: High-Capacity Real-World Image Restoration with Privacy-Safe Dataset Curation," *NeurIPS*, 2024. source
44. Z. Duan, J. Zhang, X. Jin, et al., "DiT4SR: Taming Diffusion Transformer for Real-World Image Super-Resolution," *ICCV*, 2025. source
45. R. Wu, L. Sun, Z. Ma, L. Zhang, "One-Step Effective Diffusion Network for Real-World Image Super-Resolution," *NeurIPS*, 2024. source
46. Y. Wang, W. Yang, X. Chen, et al., "SinSR: Diffusion-Based Image Super-Resolution in a Single Step," *CVPR*, 2024. source
47. E. Agustsson and R. Timofte, "NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study," in *Proc. IEEE CVPR Workshops*, 2017, pp. 1122–1131. doi ·CVF
48. Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. doi
49. R. Zhang, P. Isola, A. A. Efros, E. Shechtman and O. Wang, "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric," in *Proc. IEEE/CVF CVPR*, 2018, pp. 586–595. doi ·arXiv
50. K. Ding, K. Ma, S. Wang, E. P. Simoncelli, "Image Quality Assessment: Unifying Structure and Texture Similarity," *IEEE TPAMI*, 2022. source
51. A. Mittal, R. Soundararajan and A. C. Bovik, "Making a "Completely Blind" Image Quality

Analyzer," *IEEE Signal Processing Letters*, vol. 20, no. 3, pp. 209–212, 2013. doi
52. J. Ke, Q. Wang, Y. Wang, P. Milanfar, F. Yang, "MUSIQ: Multi-scale Image Quality Transformer," *ICCV*, 2021. source
53. S. Yang, T. Wu, S. Shi, S. Lao, Y. Gong, M. Cao, J. Wang, Y. Yang, "MANIQA: Multi-dimension Attention Network for No-Reference Image Quality Assessment," *CVPR Workshops*, 2022. source
54. J. Wang, K. C. K. Chan, C. C. Loy, "Exploring CLIP for Assessing the Look and Feel of Images," *AAAI*, 2023. source
55. M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, S. Hochreiter, "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium," *NeurIPS*, 2017. source
56. R. Timofte, S. Gu, J. Wu, L. Van Gool, "NTIRE 2018 Challenge on Single Image Super-Resolution: Methods and Results," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018. source
57. J. Cai, S. Gu, R. Timofte, L. Zhang, "NTIRE 2019 Challenge on Real Image Super-Resolution: Methods and Results," *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2019. source
58. A. Lugmayr, M. Danelljan, R. Timofte, et al., "AIM 2019 Challenge on Real-World Image Super-Resolution: Methods and Results," *ICCV Workshops*, 2019. source
59. P. Wei, H. Lu, R. Timofte, et al., "AIM 2020 Challenge on Real Image Super-Resolution: Methods and Results," *ECCV Workshops*, 2020. source
60. Z. Chen, K. Liu, J. Gong, J. Wang, L. Sun, Z. Wu, R. Timofte, Y. Zhang et al., "NTIRE 2025 Challenge on Image Super-Resolution (×4): Methods and Results," in *Proc. IEEE/CVF CVPR Workshops*, 2025, pp. 1525–1535. doi ·arXiv
61. Le Zhang, Ao Li, Qibin Hou, Ce Zhu, Yonina C. Eldar, "Deep-Learning-Empowered Super Resolution: A Comprehensive Survey and Future Prospects," *Proceedings of the IEEE*, 2025. source
62. W. Ren, R. Goyal, Z. Hu, T. T. Aumentado-Armstrong, I. Mohomed and A. Levinshtein, "Hallucination Score: Towards Mitigating Hallucinations in Generative Image Super-Resolution," arXiv:2507.14367, 2025. arXiv
63. Real-ESRGAN contributors, "Real-ESRGAN model and ×4-plus training configuration," official source, accessed 2026-11-02. model ·configuration
64. D. P. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization," *ICLR*, 2015. source
65. M. Bevilacqua, A. Roumy, C. Guillemot and M.-L. Alberi Morel, "Low-Complexity Single-Image Super-Resolution based on Nonnegative Neighbor Embedding," in *Proc. BMVC*, 2012, pp. 135.1–135.10. doi
66. R. Zeyde, M. Elad and M. Protter, "On Single Image Scale-Up Using Sparse-Representations," in *Curves and Surfaces*, Lecture Notes in Computer Science, Springer, 2012, pp. 711–730. doi
67. D. Martin, C. Fowlkes, D. Tal and J. Malik, "A Database of Human Segmented Natural Images and Its Application to Evaluating Segmentation Algorithms and Measuring Ecological Statistics," in *Proc. IEEE ICCV*, 2001, vol. 2, pp. 416–423. doi
68. J.-B. Huang, A. Singh and N. Ahuja, "Single Image Super-Resolution from Transformed Self-Exemplars," in *Proc. IEEE CVPR*, 2015, pp. 5197–5206. doi ·CVF
69. J. Kim, J. K. Lee and K. M. Lee, "Accurate Image Super-Resolution Using Very Deep Convolutional Networks," in *Proc. IEEE CVPR*, 2016, pp. 1646–1654. doi ·arXiv
70. C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J.

Totz, Z. Wang and W. Shi, “Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network,” in *Proc. IEEE CVPR*, 2017, pp. 105–114. doi ·arXiv
71. R. Timofte, E. Agustsson, L. Van Gool, M.-H. Yang, L. Zhang et al., “NTIRE 2017 Challenge on Single Image Super-Resolution: Methods and Results,” in *Proc. IEEE CVPR Workshops*, 2017, pp. 1110–1121. doi ·CVF
72. Y. Matsui, K. Ito, Y. Aramaki, A. Fujimoto, T. Ogawa, T. Yamasaki and K. Aizawa, “Sketch-Based Manga Retrieval Using Manga109 Dataset,” *Multimedia Tools and Applications*, vol. 76, no. 20, pp. 21811–21838, 2017. doi
73. KernelGAN contributors, “Adversarial losses and kernel constraints,” official source, accessed 2026-11-02. loss definitions ·alternating updates
74. X. Mao, Q. Li, H. Xie, R. Y. K. Lau, Z. Wang, S. P. Smolley, “Least Squares Generative Adversarial Networks,” *ICCV*, 2017. source
75. DASR contributors, “MoCo logits and contrastive cross-entropy implementation,” official DASR source, commit `51524af8021c94a502ee002e135867a1b75dc7d0`, accessed 2026-11-02. encoder ·loss
76. Y. Zhang, K. Li, K. Li, L. Wang, B. Zhong and Y. Fu, “Image Super-Resolution Using Very Deep Residual Channel Attention Networks,” in *Proc. ECCV*, 2018, pp. 294–310. doi ·arXiv
77. K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” in *Proc. ICLR*, 2015. arXiv
78. Y. Zhang, X. Huang, J. Ma, et al., “Recognize Anything: A Strong Image Tagging Model,” *CVPR Workshops*, 2024. source
79. E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, “LoRA: Low-Rank Adaptation of Large Language Models,” *ICLR*, 2022. source
80. D. Podell, Z. English, K. Lacey, A. Blattmann, T. Dockhorn, J. Müller, J. Penna, R. Rombach, “SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis,” *ICLR*, 2024. source
81. H. Liu, C. Li, Q. Wu, Y. J. Lee, “Visual Instruction Tuning,” *NeurIPS*, 2023. source
82. T. Karras, M. Aittala, T. Aila, S. Laine, “Elucidating the Design Space of Diffusion-Based Generative Models,” *NeurIPS*, 2022. source

# DLCV 09A.2 • One-Step Diffusion Super-Resolution

Understand LR-latent starts, residual-shift distillation, score regularization and adjustable one-step SR, with explicit limits on fidelity, speed and invented detail.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/one-step-diffusion-super-resolution/

## What you'll learn

- Distinguish observation-conditioned restoration from generation starting at noise.
- Compare restoration-chain distillation with adaptation of a pretrained prior.
- Trace latent starts, LoRA, prompts and score regularization.
- Separate degradation adaptation, inversion and fidelity/detail controls.
- Evaluate missing-detail ambiguity and complete deployment cost.

## The big picture

> **Explain like I'm five**
>
> Making a picture from nothing is like painting a whole scene from a story. Enlarging a small photograph is like painting on a sheet that already shows where the objects are. You can save time by starting from that sheet. But the small sheet does not reveal every leaf or letter, so a fast painter can still add the wrong detail. Working faster does not make the guess more certain.

A real-world SR input already contains useful low-frequency structure, layout and some semantic information. Starting from its latent representation can avoid spending many sampling steps rebuilding that structure. The network still has to remove damage and supply missing detail. It can do so through a learned one-step map, a distilled image-conditioned flow, or a pretrained prior adapted with reconstruction and distribution objectives.

Read DLCV 09A: Non-Blind Image Super-Resolution for the inverse problem and DLCV 09A.1: Blind and Real-World SR for degradation synthesis, datasets and evaluation. DLCV 12B: One-Step Generation explains endpoint distillation, consistency and score-based distribution matching. This chapter applies those ideas to a **particular observation**, where factual fidelity matters in addition to plausible appearance.

For a first reading, follow the family table, the measurement-ambiguity experiment and the OSEDiff overview. The optional paper pages contain full training procedures and controlled comparisons. The main lesson is not that one method is best for every image: the useful choice depends on damage, acceptable invention and the complete deployment budget.

## What a one-step SR model must do

**Plain version.** The input tells the model where to begin, but not every detail of where to end. A useful restorer must use that evidence while recognizing that the unobserved detail is ambiguous.

Write the observation as $y = \mathcal{D}(x) + n$, where $x$ is an unknown high-resolution image, $y$ the observed low-resolution image, $\mathcal{D}$ a degradation operator, and $n$ noise. In real-world SR, $\mathcal{D}$ is usually unknown and may include nonlinear processing. A trained conditional generator returns $\hat{x} = G_\theta(y)$ or $G_\theta(y, z)$ if it also receives random input $z$.

A deterministic $G_\theta(y)$ can give repeatable output and still invent details. A stochastic generator can express several plausible reconstructions without knowing which one was photographed. Neither property alone establishes fidelity. The observation constrains some aspects of the image, while the learned prior supplies preferences for the rest.

**An exact ambiguity experiment** Consider ×2 block averaging, $A(x)$, which replaces each 2 × 2 block by its mean. A checkerboard with values $+a, -a, -a, +a$ sums to zero inside that block. Adding or subtracting it changes HR texture without changing the observed LR image. The texture lies in the null space of $A$.

The following synthetic experiment builds a smooth LR pattern and two opposite HR textures. It compares a conditional mean with two plausible modes. We deliberately know the measurement model here; real-world photos make consistency checking harder because that model must itself be estimated.

```
import numpy as np

row, col = np.mgrid[:32, :32]
lr = 0.5 + 0.12*np.sin(2*np.pi*col/16) + 0.08*np.cos(2*np.pi*row/13)
base = lr.repeat(2, 0).repeat(2, 1)
i, j = np.indices(base.shape)
texture = 0.12*(-1.)**(i+j)  # zero mean in every 2x2 block
truth = base + texture
candidates = {'conditional mean': base, 'matching mode': base+texture, 'opposite mode':
 ↪  base-texture}
def measure(x):
    return x.reshape(32,2,32,2).mean((1,3))
for name, x in candidates.items():
    mse = np.mean((x-truth)**2)
    discrepancy = np.max(np.abs(measure(x)-lr))
    print(f'{name:16s} MSE {mse:.4f} | LR discrepancy {discrepancy:.1e}')
```

```
conditional mean MSE 0.0144 | LR discrepancy 0.0e+00
matching mode    MSE 0.0000 | LR discrepancy 5.6e-17
opposite mode    MSE 0.0576 | LR discrepancy 5.6e-17
```

These are our calculated outputs. All candidates reproduce the LR observation up to floating-point rounding. The sharp opposite mode nevertheless has four times the conditional mean's squared error against this chosen ground truth. If the two hidden texture signs are equally likely, the conditional mean minimizes expected squared error, while a randomly sampled plausible mode is not guaranteed to match the particular original. No amount of inference acceleration identifies a bit the observation never measured.

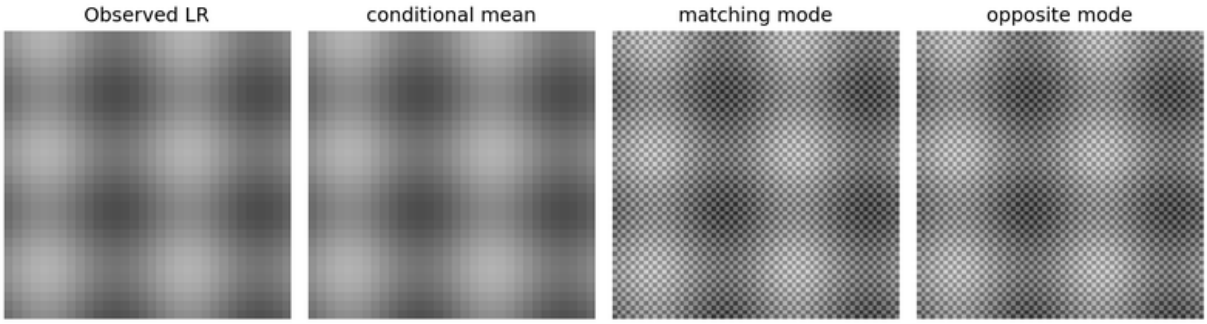



*Figure 1 — Measurement consistency is weaker than correct hidden detail. A deterministic one-step network can choose the wrong sharp texture just as a multi-step sampler can.*

**What is transferred from inference to training?** Multi-step restoration repeatedly calls a denoising network. A one-step model must learn a larger image-conditioned transformation, so adaptation, teacher sampling, discriminators or score regularizers can make training more demanding. Report both sides of the trade: fewer deployment calls may require substantial training machinery.

A pretrained text-to-image denoiser has not automatically learned that an LR image latent is a legitimate high-noise input. Feeding it such a latent without adaptation can fail. Methods such as OSEDiff explicitly train that image-to-image use [1]. The diffusion network is then a useful pretrained parameterization of a conditional restorer, rather than proof that the input follows its original forward-noise distribution.

### A map of short-chain and one-step SR

**Plain version.** Some methods shorten a restoration-specific chain; some adapt a large pretrained prior; others learn the starting state or expose separate controls. Identify which component changes before comparing names.

| Family | Starting point | Main training signal | Representative method | What to check |
|---|---|---|---|---|
| Residual-shift chain | LR-centered noisy state | Image-conditioned diffusion reconstruction | ResShift [2] | Noise schedule, number of steps, training domain |
| Conditional endpoint distillation | LR plus an initial random perturbation | Teacher endpoints and a learned inverse/GT cycle | SinSR [3] | Teacher procedure and student's actual one-step mapping |
| Pretrained-prior adaptation | LR image latent | Paired reconstruction plus latent score regularization | OSEDiff [1] | Encoder/decoder adaptation, prompt cost, auxiliary score tracking |

| Family | Starting point | Main training signal | Representative method | What to check |
|---|---|---|---|---|
| Degradation-aware adaptation | LR latent with estimated degradation | Reconstruction, perceptual and adversarial losses | S3Diff [4] | Degradation estimator and guidance evaluation count |
| Adversarial distillation | Image-conditioned short sampler | Teacher and discriminator | AddSR [5] | Training version and inference budget |
| Target-aware score training | LR-conditioned student | Paired target scores and distribution-aware training states | TSD-SR [6] | How target HR information enters training |
| Learned inversion/ start | Predicted image-conditioned noisy state | Loss after a pretrained reverse process | InvSR [7] | Starting-state predictor and total number of calls |
| Separate enhancement controls | LR latent | Pixel and semantic LoRA objectives | PiSA-SR [8] | Which weights are varied and whether fidelity changes monotonically |

The categories overlap. The table is a functional map, not an exclusive taxonomy. A pretrained one-step generator, a learned start and an adversarial objective can occur in the same system. Some methods support several inference budgets; a four-step score must retain its four-step label.

The larger field is organized in the blind-SR and diffusion-SR surveys [9], [10]. Their degradation assumptions remain important here. Fast inference does not make a model trained on one synthetic degradation distribution robust to every camera, codec or historical photograph.

### A short chain centered on the observation

**Plain version.** Shift the high-resolution image toward the low-resolution observation rather than pure noise, so reversal starts near observed structure.

ResShift defines a residual $e = y - x_0$ between the enlarged LR image $y$ and HR target $x_0$ [2]. A convenient expression for its forward marginal is

$$q(x_t \mid x_0, y) = \mathcal{N}\big(x_0 + \eta_t(y - x_0), \kappa^2 \eta_t I\big)\,.$$

Here $\eta_t$ increases from near zero toward one, $\kappa$ scales noise, and $I$ is the identity covariance. Image dimensions match because $y$ has been enlarged, or both images have been encoded into matching latent grids. Near the final forward time, the state is approximately LR-centered Gaussian noise; reversing the chain restores detail.

The network predicts the HR target from a noisy state, the observation and time. Training therefore learns a restoration-conditioned process. The published latent implementation uses an image autoencoder and a short default chain. This is different from freezing a large text-to-image prior and adding a controller. Its capacity, training images and representation differ, so score differences with a large pretrained prior do not isolate the chain length alone.

SinSR constructs a deterministic version of the ResShift reverse process, then distills its endpoint into one student prediction [3]. It also trains an inverse mapping and uses ground-truth reconstructions through that inverse. The student is not merely ResShift evaluated once with an unchanged prediction target.

A deterministic teacher mapping fixes the output associated with each LR-centered random start. It makes endpoint regression well defined, while the random start can still select different results. The ground-truth consistency term gives the student a paired-data signal beyond reproducing the teacher's endpoint. Its use of the same network in inverse and restoration directions is a training design; inference only needs the restoration direction.

*Online only: ResShift: a short restoration-conditioned chain (NeurIPS 2023) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-diffusion-super-resolution/papers/resshift-a-short-restoration-conditioned-chain-neurips-2023/; SinSR: distilling a deterministic image-conditioned map (CVPR 2024) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-diffusion-super-resolution/papers/sinsr-distilling-a-deterministic-image-conditioned-map-cvpr-2024/*

### Starting from an LR latent: OSEDiff

**Plain version.** OSEDiff converts an input-derived representation to a high-resolution latent in one network call; a separate training regularizer encourages natural outputs.

**The deployed path** The model has an adapted image encoder $E_\theta$, an adapted diffusion denoiser $\epsilon_\theta$, a frozen image decoder $D$, and a prompt extractor $Y$ [1]. With input enlarged to the working resolution, write

$$z_L = E_\theta(y), \quad c = Y(y), \quad \hat{z}_H = \frac{z_L - \beta_T \epsilon_\theta(z_L, T, c)}{\alpha_T}, \quad \hat{x}_H = D(\hat{z}_H).$$

Here $z_L$ is the LR-derived latent, $c$ text conditioning, $T$ the selected fixed diffusion timestep, and $\alpha_T, \beta_T$ signal/noise coefficients in the paper's notation. The formula does **not** add fresh Gaussian noise to $z_L$. It is a learned image-to-image transformation at a fixed model timestep. The denoiser output is adapted for this use; the degraded latent is not assumed to be a genuine sample of the original high-noise forward process.

LoRA adds low-rank trainable updates to the pretrained encoder and denoiser [11]. Freezing the decoder keeps the representation's output interpretation tied to the pretrained latent space. A small trainable parameter count describes adaptation size, not the total parameters or memory required to execute the pretrained network.

**Paired fidelity and a learned prior** OSEDiff combines pixel MSE and LPIPS reconstruction with variational score distillation (VSD) [12], [13]. The first two terms compare outputs with paired HR training targets. The score regularizer compares the distribution of generated latents with a pretrained natural-image prior, using a fixed pretrained denoiser and a second denoiser adapted to current generated samples.

This is the restoration counterpart of the distribution-matching idea in DLCV 12B. It does not directly calculate the unknown real-world degradation likelihood. The paired synthetic examples teach how to preserve content under sampled damage; the prior regularizer teaches natural appearance. Both may be wrong for a particular out-of-distribution input.

**Why regularize in latent space?** A pixel-space prior-gradient calculation can repeatedly decode a generated latent and encode the resulting image again. OSEDiff approximates that round trip by operating directly on the predicted latent. This reduces repeated encoding/decoding on the regularization path. It relies on the pretrained autoencoder's representation being sufficiently consistent; it is an approximation, not an exact identity for arbitrary latents.

The auxiliary regularizer networks are training-time components. Deployment still includes image encoding, prompt extraction, one denoiser pass and image decoding. A paper's timing must state which of these were measured. A change from a lightweight tag extractor to a large captioning model can erase much of the practical speed benefit even when the denoiser still runs once.

*Online only: OSEDiff: LR-latent start and variational score regularization (NeurIPS 2024) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-diffusion-super-resolution/papers/osediff-lr-latent-start-and-variational-score-regularization-neurips-2024/*

### Making adaptation depend on the damage

**Plain version.** One fixed correction is not equally suitable for slight blur and heavy noise. S3Diff changes the low-rank adaptation according to an estimated degradation description [4].

For a weight matrix $W \in \mathbb{R}^{m\times n}$, ordinary LoRA gives $W' = W + AB$, with $A \in \mathbb{R}^{m\times r}$ and $B \in \mathbb{R}^{r\times n}$ for a small rank $r$. S3Diff introduces a degradation-conditioned middle matrix:

$$W_i' = W_i + A_i C_i(d) B_i.$$

Here $i$ identifies a network block, $d$ contains estimated blur/noise information, and $C_i(d) \in \mathbb{R}^{r\times r}$ modulates the low-rank update. A shared mapping uses degradation embeddings and block identifiers to produce block-specific corrections. Different images can therefore receive different effective updates.

The degradation vector is an estimate, not complete knowledge of the camera pipeline. If it is wrong or fails to describe compression and spatial variation, the adaptive correction can be wrong too. The method is also different from a single static LoRA that can be merged once for all inputs: an input-dependent correction still requires input-dependent computation.

S3Diff uses generic positive/negative quality prompts and online negative targets rather than requiring a semantic captioner for every image. The training mixture teaches the meaning of poor-quality output. Classifier-free guidance can then combine differently conditioned predictions. If that configuration makes two denoiser predictions, count both when reporting NFE even though there is only one state-update step.

*Online only: S3Diff: degradation-guided low-rank adaptation — https://blog.shyandram.dev/en/learn/deep-learning/one-step-diffusion-super-resolution/papers/s3diff-degradation-guided-low-rank-adaptation/*

### Other ways to improve a short restoration map

**Plain version.** Once the chain is short, errors in conditions, targets and starting states matter more. Recent methods address different parts of that problem instead of only reducing the step count further.

**Adversarial restoration distillation** AddSR applies adversarial diffusion distillation to image-conditioned restoration [5], [14]. A pretrained teacher supplies restoration knowledge, while a discriminator pushes outputs toward realistic images. The method combines conditioning and distillation rather than simply applying a text-to-image one-step generator to the enlarged input.

An adversarial objective can add convincing detail without proving that the detail is correct. Check the teacher, restoration conditioning, number of inference steps and training version. A method that supports a few steps should not have its best few-step result relabeled as one-step performance.

**Target-aware scores and training states** TSD-SR observes that a pretrained teacher can provide unreliable directions on poor student outputs, especially early in training [6]. It introduces a target-score construction that uses the paired HR training latent, and a distribution-aware sampling module that produces training states closer to the adapted diffusion trajectory.

The paired target is available during training, not at deployment. Comparing the teacher's predictions on a generated latent and its paired target supplies an additional direction beyond comparing teacher and current-student scores. This is a change to the regularization target, not direct evidence that all teacher-score errors have been eliminated. The sampling module also costs training work, even if it is discarded from a one-step inference pipeline.

**Learning where the reverse process starts** InvSR learns an image-conditioned noise/start predictor and runs a pretrained reverse process from the predicted state [7]. At the starting point, the predicted "noise" need not be independent standard Gaussian noise; it can be correlated with the observed image to compensate for replacing an unknown clean image with its LR observation.

This is not the same as taking an arbitrary off-the-shelf inversion of a real photograph. The starting-state model is trained for restoration, with losses evaluated after the chosen reverse process. It can support different selected step budgets, but a learned starting-state predictor adds a network call. One reverse denoising step therefore need not equal one total model evaluation.

**Separating pixel and semantic enhancement** PiSA-SR trains two low-rank update groups: one for pixel reconstruction and another for semantic/perceptual enhancement [8]. The first is trained

before the second; the semantic stage holds the learned pixel update fixed. The default inference uses one combined prediction. Its adjustable configuration computes pixel- and semantic-enhanced outputs separately and uses two diffusion evaluations; separate weights then alter the residual contributions.

That adjustable setting must be counted separately from the default one-step setting. These controls change a learned model's behavior. They do not physically separate every pixel into a "factual" and "invented" channel, and metric responses need not be monotonic. PiSA-SR uses a classifier-score-distillation objective rather than maintaining the same generated-distribution score model as OSEDiff. The change reduces some training machinery while changing the statistical objective; it is not an algebraically identical shortcut to the same VSD loss.

*Online only: TSD-SR: target-aware score regularization (CVPR 2025) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-diffusion-super-resolution/papers/tsd-sr-target-aware-score-regularization-cvpr-2025/; InvSR: a learned image-conditioned start (CVPR 2025) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-diffusion-super-resolution/papers/invsr-a-learned-image-conditioned-start-cvpr-2025/; PiSA-SR: separate pixel and semantic adaptation (CVPR 2025) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-diffusion-super-resolution/papers/pisa-sr-separate-pixel-and-semantic-adaptation-cvpr-2025/*

### Fidelity, realism and control

**Plain version.** Making an output agree with a natural-image prior does not necessarily make it agree with the observation. A fidelity term and a realism term can pull in different directions.

For an illustrative quadratic problem, let $a$ represent an error in a measured coordinate and $b$ an unmeasured texture coordinate. Suppose a prior prefers $(a, b) = (1, 1)$ while the observation constrains $a = 0$. Minimizing $a^2 + \lambda[(a-1)^2 + (b-1)^2]$ gives $a = \lambda/(1+\lambda)$ and $b = 1$ when $\lambda > 0$. Stronger prior weight changes a coordinate the observation actually constrains. A separate constraint $a = 0$ would protect that coordinate, but would still leave the texture's correctness unresolved.

Here $\lambda$ is a hypothetical objective weight, not a measured hyperparameter of any cited model. The example explains why "more realistic" is not synonymous with "more faithful." In a neural restorer the coordinates are not cleanly separated, the degradation is unknown, and pixel/perceptual losses are trained over a dataset rather than solved exactly for each input.

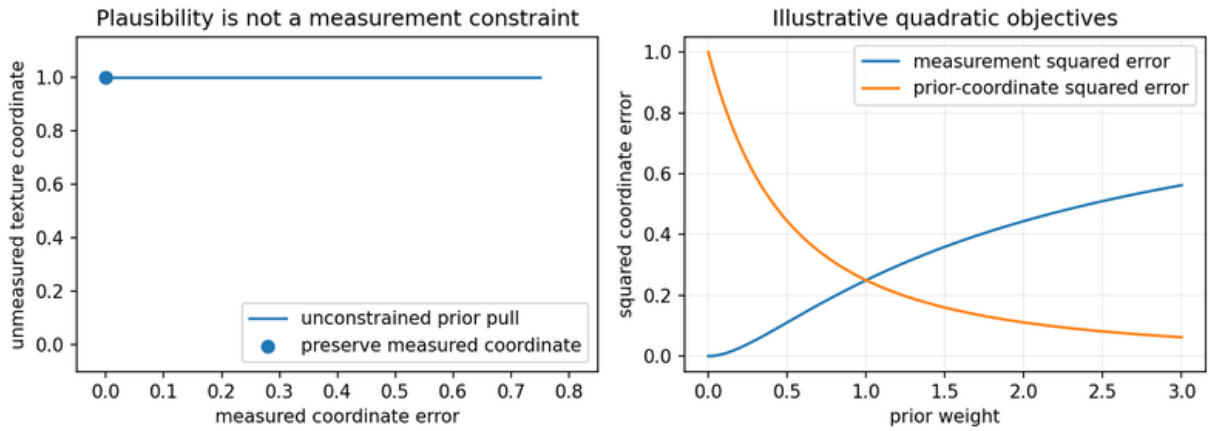



*Figure 2 — An illustrative quadratic objective separates measurement agreement from prior preference. It is not a benchmark or an exact model of a neural score-distillation loss.*

Useful controls include a prior-loss weight during training, a feature/residual strength at inference, a guidance scale, a degradation estimate or a selected starting noise level. They are not interchangeable. A training-loss weight usually requires retraining; an inference coefficient changes one fitted model; choosing another checkpoint may change both training data and objectives. State which operation produces a comparison.

For text, faces and repeated structure, inspect whether the model changes discrete content: a letter, identity cue, brick count or boundary. Repeatability helps compare settings but does not establish factual correctness. For evidence-sensitive use, retain the original and describe generated detail as a hypothesis rather than treating a sharper output as an additional measurement.

### Measuring speed and quality honestly

**Plain version.** Count everything a user waits for, and keep the quality protocol attached to the timing row. A smaller adaptation file or one diffusion update is not a complete speed measurement.

**The pipeline to time** A typical system performs resizing, image encoding, optional degradation estimation, optional prompt extraction, generation/denoising, decoding and postprocessing. Some systems also use tiling. Record input resolution **and output resolution**, device, precision, batch size, warm-up, synchronization and whether text features are cached.

For OSEDiff, its Table 2 reports 0.11 seconds on an A100 under a stated 512 × 512 input-size test; the prompt-extractor study separately reports costs [1]. Do not assume the same timing boundary as another paper's 128-to-512 benchmark. A rigorous reproduction should inspect the timing code and report prompt extraction separately if its inclusion is unclear. At high resolutions, tiled encoding and decoding can become substantial even when there is only one denoiser call.

Distinguish **trainable parameters**, **total model parameters**, **peak inference memory** and

**adaptation-file size**. LoRA can reduce the first and last without proportionally reducing the frozen backbone required for inference. Dynamic LoRA, prompt extraction and learned inversion add their own work. Even static adapters merged into a backbone do not make its matrix multiplications disappear.

**The quality protocol** Use full-reference fidelity and perceptual metrics on synthetic or captured pairs, and no-reference measurements plus inspection on unpaired real images [15], [16], [17]. PSNR and SSIM measure different aspects of reference agreement; LPIPS and DISTS compare perceptual representations [18], [12], [19]. NIQE, MUSIQ, MANIQA and CLIPIQA predict aspects of visual quality without an HR reference [20], [21], [22], [23]. They do not verify the original scene.

Keep channel, border crop, resizing, alignment and random seed conventions fixed. A method can improve a no-reference score by adding texture and worsen reference fidelity. On captured pairs, slight misregistration affects pixel metrics. On synthetic tests, matching a model's own training degradation may favor it. The detailed evaluation caveats remain those in DLCV 09A.1.

For stochastic models, report variability across seeds rather than choosing only an attractive output. For deterministic models, examine multiple image types and damages rather than interpreting a single repeatable output as robustness. Evaluate letters, small faces, flat regions and regular grids separately, because an average score can hide different failure mechanisms.

### Modern view

**Plain version.** Shortening inference has exposed several distinct learning problems: choose a useful start, preserve observed content, estimate a reliable prior direction and offer understandable controls.

Liu et al.'s blind-SR survey [9] distinguishes explicit degradation modeling, implicit data-distribution learning and single-image information. It helps identify what uncertainty an efficient model still assumes away. Moser et al.'s diffusion-SR survey [10] organizes diffusion restoration by domain, conditioning and generative process; it places short-chain designs within a larger set of fidelity and computation trade-offs. Fan et al.'s distillation survey [24] supplies a complementary objective-based view: endpoint fidelity, trajectories and adversarial learning are different supervision sources.

The 2023–2025 sequence is not a single replacement chain. ResShift changes the restoration process; SinSR distills its conditional endpoint; OSEDiff adapts a large pretrained prior with score regularization; S3Diff adapts to degradation; TSD-SR changes score targets and training states; InvSR learns the start; PiSA-SR separates enhancement controls [2], [3], [1], [4], [6], [7], [8]. Each addresses a different bottleneck, and their benchmark distributions and backbones differ.

The 2026 TUDSR work uses a twice-upsampling/diffusion design for higher-scale restoration [25]. It is relevant to how short diffusion components scale into a complete upsampling system, but multiple stages should be counted as multiple stages rather than advertised as one total operation. New architecture or scale settings also require renewed fidelity and cost comparisons.

Open problems include uncertain degradation, reliable text and identity preservation, score estimates on poor generated samples, generalization beyond synthetic processing, large-image tiling, and controls that users can understand without learning training terminology. A practical interface should expose meaningful visual choices and preserve the input for comparison. It should not imply that an "enhance detail" setting recovers information that was never measured.

### Key takeaways

- An observation-derived start saves work only when trained for the intended image-to-image use.
- One-step inference describes a budget; methods can use different training objectives.
- ResShift/SinSR compress a restoration process; OSEDiff adapts a pretrained prior.
- Latent encoding, prompts, decoder choices and auxiliary training each have separate roles and costs.
- Determinism and low-resolution consistency do not establish correct missing detail.
- Measure the whole pipeline and compare fidelity, preference, semantic correctness and variability.

### Exercises

1. **Null-space ambiguity.** Change the checkerboard amplitude in the experiment. Derive the squared errors of the conditional mean and opposite texture, and check that the LR measurements remain unchanged.

   **Hint**

   For amplitude $a$, each texture value has magnitude $a$, so the mean's MSE is $a^2$ and the opposite mode's MSE is $4a^2$. Each $2 \times 2$ texture sums to zero. Keep the amplitude small enough that clipping does not alter the measurement equation.

2. **A wrong consistency test.** Replace block averaging with a mildly blurred resize while keeping the same HR candidates. Do identical old block averages guarantee identical new LR images? Explain what this says about testing real-world consistency with an estimated degradation.

   **Hint**

   Null spaces depend on the operator. A texture annihilated by one downsampler need not be annihilated by another. Agreement under a guessed operator is evidence conditional on that operator, not proof about the unknown camera pipeline.

3. **Training-time and inference-time networks.** Draw the OSEDiff computation and mark the encoder, denoiser, decoder, prompt extractor, fixed regularizer and adapted regularizer. Which networks receive generator gradients, and which execute at deployment?

**Hint**

The generator's adapted encoder and denoiser are updated by reconstruction and prior signals. The decoder is fixed. The adapted regularizer has its own denoising objective. The two regularizers are discarded at deployment; the prompt extractor remains unless prompts are supplied or a different configuration is used.

4. **Count dynamic guidance.** A method predicts one updated state but evaluates positive and negative prompts separately, and also calls a degradation estimator and VAE encoder/decoder. What can be called one step, and what should a timing report count?

**Hint**

There is one state-update step but two prompt-conditioned denoiser evaluations. Include degradation estimation, encoding and decoding in end-to-end timing; report any batching convention. A cached prompt embedding reduces text-encoder work, not the two denoising predictions themselves.

5. **A meaningful control.** Design a comparison for a "more detail" control using flat sky, small text and repeated windows. State one fidelity check, one visual-quality check and one failure that an average no-reference score could miss.

**Hint**

Compare the same inputs and crops across settings. Check changed letters and window boundaries, inspect new sky texture, and record a quality metric without treating it as truth. A setting can improve an aggregate score while changing a discrete character or repeating a false pattern.

## References


1. R. Wu, L. Sun, Z. Ma, L. Zhang, "One-Step Effective Diffusion Network for Real-World Image Super-Resolution," *NeurIPS*, 2024. source
2. Z. Yue, J. Wang, C. C. Loy, "ResShift: Efficient Diffusion Model for Image Super-resolution by Residual Shifting," *NeurIPS*, 2023. source
3. Y. Wang, W. Yang, X. Chen, et al., "SinSR: Diffusion-Based Image Super-Resolution in a Single Step," *CVPR*, 2024. source ·supplementary ablations
4. A. Zhang, Z. Yue, R. Pei, W. Ren, X. Cao, "Degradation-Guided One-Step Image Super-Resolution with Diffusion Priors," *arXiv preprint*, 2024. paper
5. Y. Tai, R. Xie, C. Zhao, K. Zhang, Z. Zhang, J. Zhou, J. Yang, "AddSR: Accelerating diffusion-based blind super-resolution with adversarial diffusion distillation," *Pattern Recognition*, 2026. published version ·manuscript supplying the discussed tables
6. L. Dong, Q. Fan, Y. Guo, Z. Wang, Q. Zhang, J. Chen, Y. Luo, C. Zou, "TSD-SR: One-Step Diffusion with Target Score Distillation for Real-World Image Super-Resolution," *CVPR*, 2025. paper
7. Z. Yue, K. Liao, C. C. Loy, "Arbitrary-steps Image Super-resolution via Diffusion Inversion," *CVPR*, 2025. paper

8. L. Sun, R. Wu, Z. Ma, S. Liu, Q. Yi, L. Zhang, "Pixel-level and Semantic-level Adjustable Super-resolution: A Dual-LoRA Approach," *CVPR*, 2025. paper
9. A. Liu, Y. Liu, J. Gu, Y. Qiao, C. Dong, "Blind Image Super-Resolution: A Survey and Beyond," *IEEE TPAMI* (*early access*), 2022. source
10. B. B. Moser, A. S. Shanbhag, F. Raue, S. Frolov, S. Palacio and A. Dengel, "Diffusion Models, Image Super-Resolution, and Everything: A Survey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 36, no. 7, pp. 11793–11813, 2025. doi ·arXiv
11. E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, "LoRA: Low-Rank Adaptation of Large Language Models," *ICLR*, 2022. source
12. R. Zhang, P. Isola, A. A. Efros, E. Shechtman and O. Wang, "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric," in *Proc. IEEE/CVF CVPR*, 2018, pp. 586–595. doi ·arXiv
13. Z. Wang, C. Lu, Y. Wang, F. Bao, C. Li, H. Su, J. Zhu, "ProlificDreamer: High-Fidelity and Diverse Text-to-3D Generation with Variational Score Distillation," *NeurIPS*, 2023. paper
14. A. Sauer, D. Lorenz, A. Blattmann, R. Rombach, "Adversarial Diffusion Distillation," *ECCV*, 2024. paper
15. J. Cai, H. Zeng, H. Yong, Z. Cao, L. Zhang, "Toward Real-World Single Image Super-Resolution: A New Benchmark and a New Model," *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019. source
16. P. Wei, Z. Xie, H. Lu, Z. Zhan, Q. Ye, W. Zuo, L. Lin, "Component Divide-and-Conquer for Real-World Image Super-Resolution," *ECCV*, 2020. source
17. E. Agustsson and R. Timofte, "NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study," in *Proc. IEEE CVPR Workshops*, 2017, pp. 1122–1131. doi ·CVF
18. Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. doi
19. K. Ding, K. Ma, S. Wang, E. P. Simoncelli, "Image Quality Assessment: Unifying Structure and Texture Similarity," *IEEE TPAMI*, 2022. source
20. A. Mittal, R. Soundararajan and A. C. Bovik, "Making a "Completely Blind" Image Quality Analyzer," *IEEE Signal Processing Letters*, vol. 20, no. 3, pp. 209–212, 2013. doi
21. J. Ke, Q. Wang, Y. Wang, P. Milanfar, F. Yang, "MUSIQ: Multi-scale Image Quality Transformer," *ICCV*, 2021. source
22. S. Yang, T. Wu, S. Shi, S. Lao, Y. Gong, M. Cao, J. Wang, Y. Yang, "MANIQA: Multi-dimension Attention Network for No-Reference Image Quality Assessment," *CVPR Workshops*, 2022. source
23. J. Wang, K. C. K. Chan, C. C. Loy, "Exploring CLIP for Assessing the Look and Feel of Images," *AAAI*, 2023. source
24. X. Fan, Z. Wu, H. Wu, L. Cao, "A Survey on Pre-Trained Diffusion Model Distillations," *arXiv preprint, version 2*, 2025. paper
25. Z. Wu, Y. Dong, X. Wei, "TUDSR: Twice Upsampling-Diffusion for Higher Super-Resolution," *CVPR*, 2026. paper
26. P. Esser, R. Rombach, B. Ommer, "Taming Transformers for High-Resolution Image Synthesis," *CVPR*, 2021. paper
27. X. Wang, L. Xie, C. Dong, Y. Shan, "Real-ESRGAN: Training Real-World Blind Super-Resolution With Pure Synthetic Data," *Proceedings of the IEEE/CVF International Conference on Computer*

*Vision*, 2021. source
28. J. Deng, W. Dong, R. Socher, L. Li, Kai Li, Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” *CVPR*, 2009. doi
29. R. Rombach, A. Blattmann, D. Lorenz, P. Esser, B. Ommer, “High-Resolution Image Synthesis with Latent Diffusion Models,” *CVPR*, 2022. source
30. R. Wu, T. Yang, L. Sun, Z. Zhang, S. Li, L. Zhang, “SeeSR: Towards Semantics-Aware Real-World Image Super-Resolution,” *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024. source ·supplementary experiments
31. H. Liu, C. Li, Q. Wu, Y. J. Lee, “Visual Instruction Tuning,” *NeurIPS*, 2023. source
32. Y. Li, K. Zhang, J. Liang, et al., “LSDIR: A Large Scale Dataset for Image Restoration,” *CVPR Workshops*, 2023. paper
33. T. Karras, S. Laine, T. Aila, “A Style-Based Generator Architecture for Generative Adversarial Networks,” *CVPR*, 2019. paper
34. I. Loshchilov, F. Hutter, “Decoupled Weight Decay Regularization,” *ICLR*, 2019. paper
35. P. Esser, S. Kulal, A. Blattmann, et al., “Scaling Rectified Flow Transformers for High-Resolution Image Synthesis,” *ICML*, 2024. paper
36. J. Ho, T. Salimans, “Classifier-Free Diffusion Guidance,” *arXiv preprint*, 2022. paper
37. R. Timofte, E. Agustsson, L. Van Gool, M.-H. Yang, L. Zhang et al., “NTIRE 2017 Challenge on Single Image Super-Resolution: Methods and Results,” in *Proc. IEEE CVPR Workshops*, 2017, pp. 1110–1121. doi ·CVF

# DLCV 09A.3 • Video Super-Resolution: Using Time to Recover Detail

How video super-resolution uses several misaligned low-resolution frames to recover detail one frame cannot hold: the video degradation model, optical flow and warping, flow, deformable and attention alignment, sliding-window, recurrent and transformer designs, real-world and diffusion VSR, and how results are measured.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/video-super-resolution/

## What you'll learn

- Explain how neighboring frames add measurements and temporal constraints.
- Model motion, blur, downsampling, noise and compression across frames.
- Compare flow warping, deformable alignment and attention.
- Choose windowed, recurrent or bidirectional processing under latency and memory limits.
- Evaluate spatial fidelity, temporal stability and generative artifacts together.

## The big picture

A video is a stack of photos taken a fraction of a second apart. Each photo is small and a bit blurry, but the camera or the objects move a little between shots. Video super-resolution turns such a clip into a sharper, larger clip by combining what all the nearby frames saw.

> **Explain like I'm five**
>
> Imagine looking at a painting through a window screen. The wires hide some of the painting, so one look gives you only bits of it. Now move your head a tiny bit and look again: the wires hide *different* bits. After a few small head movements you have seen almost the whole painting, as long as you remember how far you moved each time. A VSR model does exactly this: it works out how far everything moved between frames (that is the "remember how far you moved" part) and fills in the hidden bits from the other frames. If it forgets the movement, the bits land in the wrong places and the picture gets smeared.

Why it matters: most of the pixels people look at are video. Streams drop to low resolution when bandwidth is short, old films and home videos were recorded at low resolution, and phones, surveillance cameras, satellites and microscopes record small, noisy image sequences. A good VSR model makes these sharper without the frame-to-frame shimmering of a frame-by-frame image model. VSR is also where ideas now common in video models were developed: flow-based feature alignment, deformable alignment, long-range recurrent propagation and video transformers.

This chapter builds on DLCV 09, which covers the degradation model, training on synthetic pairs, losses and the generic image metrics; we link to it instead of repeating it. Single-image super-resolution, the setting of one frame in and one frame out, is covered in DLCV 09A. Here we focus on what time adds.

### Why video is not just many images

**Plain version.** Independent frame upscaling misses neighboring measurements and does not enforce stable detail across time.

**Sub-pixel motion gives new information** Suppose a camera samples a scene on a coarse grid of pixels. In one frame, details that fall between the sampling points are lost (or worse, folded into wrong, low-frequency patterns, which is called **aliasing**). If the scene shifts by a quarter of a pixel in the next frame, the same grid now samples the scene at different places. Four frames shifted by $0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}$ of a pixel together sample the scene four times more densely than any one of them (Figure 1, in one dimension; in two dimensions, a factor $s$ needs $s^2$ distinct shifts). This is the basic reason multi-frame super-resolution can beat single-image super-resolution: the extra frames are new measurements, not copies [1].

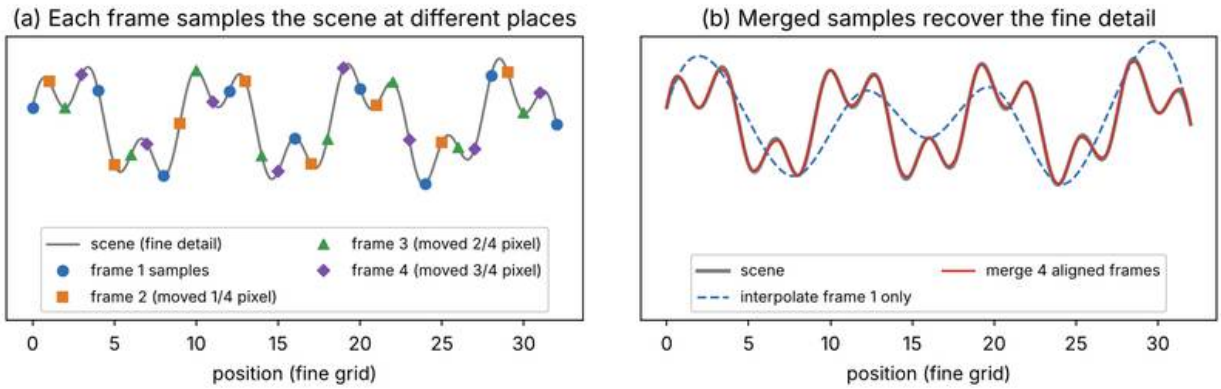



*Figure 1 — One-dimensional illustration. (a) Four frames sample the same scene at positions shifted by quarter-pixel steps. (b) Interpolating one frame misses the fine detail; merging the four aligned sample sets recovers it. Synthetic signal.*

Two conditions must hold. First, the motion must be **sub-pixel**: a shift by a whole number of low-resolution pixels samples the same places again and adds only noise averaging. Second, we must **know the motion** accurately enough to put each sample in the right place. Real motion is not a global shift: objects move differently, rotate, deform, appear and disappear. Estimating and compensating that motion is the core problem of VSR, and most of this chapter is about it.

**Temporal redundancy** Consecutive frames are also highly redundant: most of the scene is seen again. Independent noise averages out over repeated views, and a model can reuse work done for earlier frames. Video codecs exploit the same redundancy by predicting each frame from the previous one with motion vectors (see the motion-compensated prediction in IP 08).

**Temporal consistency and flicker** A single-image model, applied frame by frame, makes small independent decisions in every frame. Where it invents texture (grass, hair, water, noise-like

detail), the invented texture differs from frame to frame, and the video **flickers** or "boils". Such temporal artifacts are visible even when each frame looks fine on its own. VESPCN [2] and FRVSR [3] both list temporal consistency as a main motivation for processing frames jointly instead of independently. So a VSR model has two goals: each frame should be faithful and sharp, and consecutive frames should change only as the scene changes.

### The video degradation model

**Plain version.** Each low-resolution frame is the high-resolution frame at that moment, blurred, shrunk, made noisy and usually compressed. The frames differ because the scene moved.

**One equation per frame** Extending the super-resolution model of DLCV 09 to a clip $\{x_t\}_{t=1}^{T}$ of $T$ high-resolution (HR) frames gives

$$y_t = C((x_t * k) \downarrow_s + n_t), \qquad t = 1, \dots, T,$$

where $y_t$ is the low-resolution (LR) frame at time $t$, $k$ a blur kernel (camera optics, sensor integration, anti-aliasing filter), $*$ 2-D convolution, $\downarrow_s$ keeps every $s$-th pixel in each direction ($s$ is the scale factor, usually 4 in benchmarks), $n_t$ noise that is new in every frame, and $C$ lossy video compression (the identity in clean benchmarks). Motion enters because each $x_t$ is a moved version of its neighbors. Near frame $t$ we can write

$$x_{t+j} \approx \mathcal{W}_{t \to t+j}(x_t),$$

where $\mathcal{W}_{t \to t+j}$ is the motion (warping) from frame $t$ to frame $t + j$, $j$ is a small frame offset, and $\approx$ hides what motion cannot explain: occluded regions, objects entering the frame, lighting changes. Combining the two equations,

$$y_{t+j} \approx C((\mathcal{W}_{t \to t+j}(x_t) * k) \downarrow_s + n_{t+j}),$$

so every neighboring LR frame is a *different* measurement of the same unknown $x_t$. VSR estimates $x_t$ from $y_{t-K}, \dots, y_{t+K}$ (a window of $2K + 1$ frames) or from the whole clip.

**Multi-frame reconstruction as an inverse problem** If the motions, blur and scale were all known and there were no compression, $x_t$ could be found by least squares:

$$\hat{x}_t = \arg\min_x \frac{1}{2K+1} \sum_{j=-K}^{K} \| DB\, \mathcal{W}_{t \to t+j}(x) - y_{t+j} \|_2^2,$$

where $B$ is blurring with $k$, $D$ is the downsampling $\downarrow_s$, and the sum runs over the frames in the window. With one frame this problem is badly underdetermined ($s^2 = 16$ unknown HR pixels per LR pixel for $s = 4$). With many frames at different sub-pixel positions it becomes much better determined. Gradient descent on this objective repeatedly simulates the LR frames from the current estimate, compares them with the observed frames and spreads the error back onto the HR grid; this is the classical **iterative back-projection** idea introduced by Irani and Peleg [4], [1], which also inspired the RBPN architecture discussed later [5].

*Online only: Classical multi-frame super-resolution — https://blog.shyandram.dev/en/learn/deep-learning/video-super-resolution/notes/classical-multi-frame-super-resolution/*

**A toy experiment: what alignment buys** The snippet builds a 16-frame LR video from one `skimage` photo by random sub-pixel camera shifts, Gaussian blur, ×4 subsampling and noise. It then compares four reconstructions of the first frame: cubic interpolation of that frame alone; the average of all upsampled frames *without* alignment; the least-squares solver above using only the first frame; and the same solver using all 16 frames with the true shifts and with shifts **estimated from the LR frames** by sub-pixel phase correlation [6]. It runs in about ten seconds on a laptop CPU.

```
import numpy as np
from scipy import ndimage as ndi
from skimage import data, color
from skimage.registration import phase_cross_correlation

rng = np.random.default_rng(0)
s, T, blur, noise = 4, 16, 1.2, 0.01                        # scale, frames, blur sigma (HR
 ↪  px), noise std
hr = color.rgb2gray(data.astronaut())[64:320, 128:384]      # 256 x 256 ground-truth frame,
 ↪  values in [0, 1]
shifts = rng.uniform(0, s, size=(T, 2)); shifts[0] = 0      # sub-pixel camera motion of each
 ↪  frame (HR px)

def degrade(x, d):                                          # y_t = D B W_t x : shift, blur,
 ↪  keep every s-th pixel
    return ndi.gaussian_filter(ndi.shift(x, d, order=3, mode="reflect"), blur)[::s, ::s]
def degrade_T(r, d):                                        # adjoint: zero-fill, blur, shift
 ↪  back
    z = np.zeros((r.shape[0] * s, r.shape[1] * s)); z[::s, ::s] = r
    return ndi.shift(ndi.gaussian_filter(z, blur), -d, order=3, mode="reflect")
def upsample(y):                                            # cubic-spline interpolation on
 ↪  the HR grid
    u, v = np.mgrid[0:y.shape[0] * s, 0:y.shape[1] * s] / s
    return ndi.map_coordinates(y, [u, v], order=3, mode="nearest")
def psnr(x):
    c = (slice(8, -8), slice(8, -8))                        # skip a border, as benchmarks do
    return 10 * np.log10(1 / np.mean((np.clip(x, 0, 1)[c] - hr[c]) ** 2))

lr = np.stack([degrade(hr, d) + noise * rng.standard_normal((64, 64)) for d in shifts])
print("LR video:", lr.shape)

def reconstruct(frames, ds, iters=20):
    """Multi-frame least squares by gradient descent (iterative back-projection)."""
    x = np.mean([ndi.shift(upsample(f), -d, order=3, mode="reflect") for f, d in zip(frames,
     ↪  ds)], 0)
    for _ in range(iters):
        grad = np.mean([degrade_T(degrade(x, d) - f, d) for f, d in zip(frames, ds)], 0)
        x -= s * s * grad
    return x

# shifts estimated from the LR frames themselves (sub-pixel phase correlation), in HR pixels
```

```
est = np.array([-s * phase_cross_correlation(lr[0], f, upsample_factor=100)[0] for f in lr])
print(f"mean shift error: {np.abs(est - shifts).mean():.2f} HR px")

print(f"single frame, cubic interpolation {psnr(upsample(lr[0])):.2f} dB")
print(f"average of frames, no alignment   {psnr(np.mean([upsample(f) for f in lr], 0)):.2f}
 ↪  dB")
print(f"single frame, same solver         {psnr(reconstruct(lr[:1], shifts[:1])):.2f} dB")
print(f"{T} frames, true shifts            {psnr(reconstruct(lr, shifts)):.2f} dB")
print(f"{T} frames, estimated shifts       {psnr(reconstruct(lr, est)):.2f} dB")
# LR video: (16, 64, 64)
# mean shift error: 0.13 HR px
# single frame, cubic interpolation 25.50 dB
# average of frames, no alignment   20.99 dB
# single frame, same solver         25.81 dB
# 16 frames, true shifts            30.30 dB
# 16 frames, estimated shifts       29.33 dB
```

Three lessons from this toy run (all numbers are from our run, not from any paper). First, **averaging without alignment is worse than using one frame** (20.99 against 25.50 dB): misaligned frames smear the image, exactly the "forgot how far you moved" failure of the analogy. Second, the gain comes from the **frames, not the solver**: the same least-squares solver on one frame barely beats interpolation (25.81 dB), while 16 frames give 30.30 dB. Third, the motion can be **estimated from the LR frames themselves** with an average error of about an eighth of an HR pixel, which keeps most of the gain (29.33 dB). Figure 2 shows the crops and how PSNR grows with the number of frames.

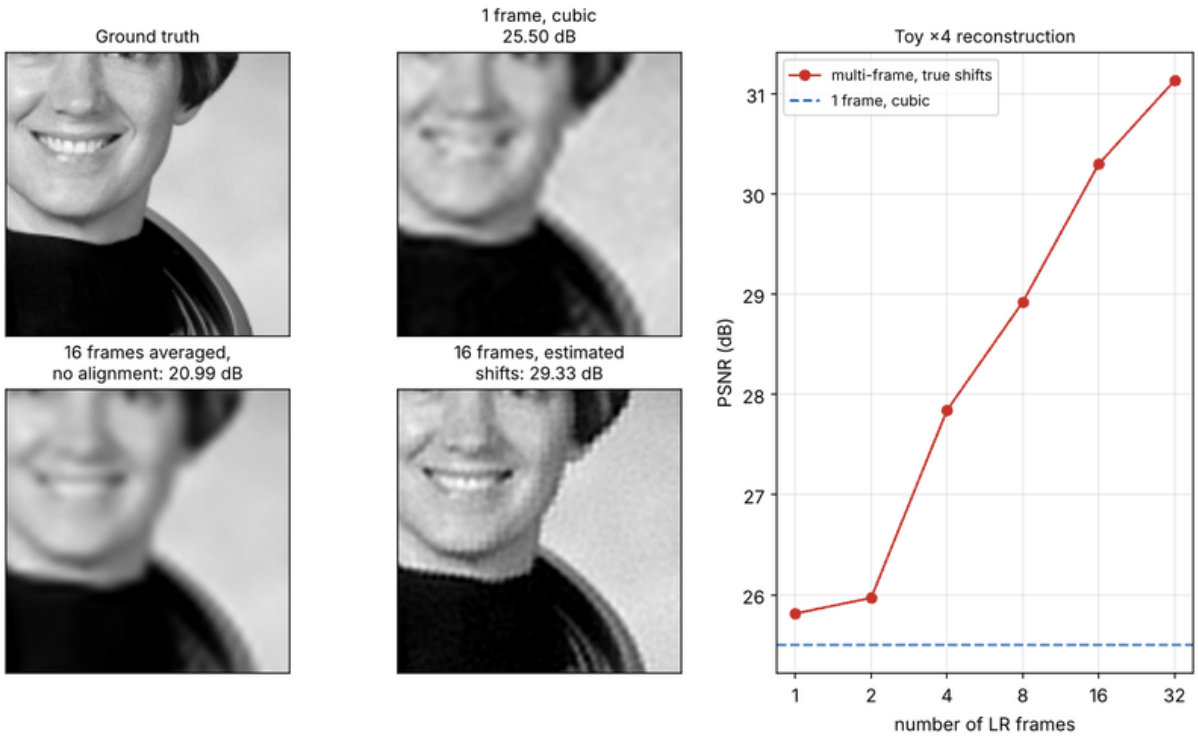


*Figure 2 — The toy multi-frame experiment. Left: crops of the ground truth and of three reconstructions (PSNR on the 256 × 256 frame without an 8-pixel border). Right: PSNR of the multi-frame solver with true shifts as more LR frames are used. Values from our run of the figure script.*

Drag the slider to compare single-frame interpolation with the 16-frame reconstruction on an enlarged crop.



*1 frame, cubic*



*16 frames, estimated shifts*

Slider comparing a blurry enlarged crop of the astronaut's smile from one frame with a sharper crop from sixteen aligned frames, where the teeth and the edge of the hair are clearer.

Real videos break the toy's assumptions (global shifts, known blur, no occlusion or compression). Deep VSR keeps the structure (estimate motion, align, fuse, reconstruct) and learns each part from data. Figure 3 shows the pipeline that almost every method follows in some form; BasicVSR [7] names its four components **propagation, alignment, aggregation and upsampling**.

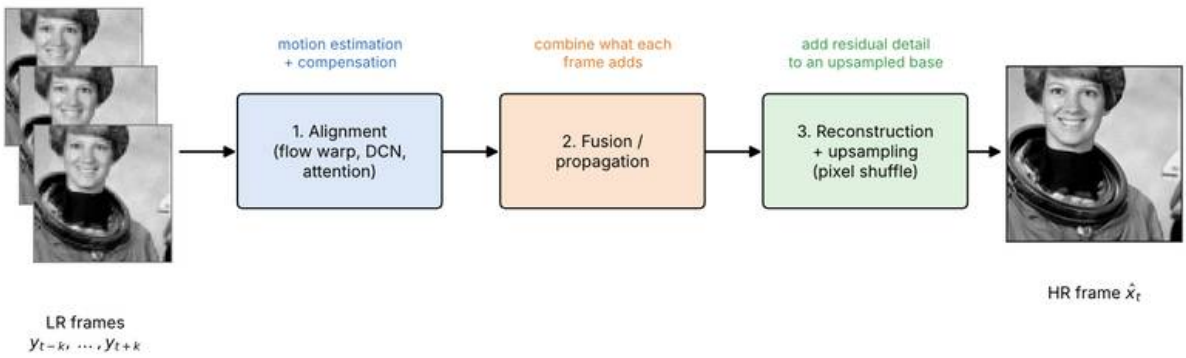



*Figure 3 — The generic VSR pipeline. Neighboring LR frames (or features) are aligned to the current frame,*

*fused or propagated over time, and reconstructed at the higher resolution, usually as a residual added to an upsampled base image.*

### Motion: optical flow and warping

**Plain version.** Optical flow estimates pixel correspondences between frames. Warping resamples a frame using those correspondences to align its content.

**Optical flow** The classical starting point is **brightness constancy**: a point keeps its brightness as it moves,

$$I(u + \delta u,\, v + \delta v,\, t + 1) = I(u, v, t),$$

where $I(u, v, t)$ is the intensity at column $u$, row $v$ and time $t$, and $(\delta u, \delta v)$ is the displacement of that point between frames $t$ and $t+1$. The field of displacements over all pixels is the **optical flow** $F(p) = (\delta u(p), \delta v(p))$ for each pixel $p$. One equation per pixel cannot determine two unknowns (the "aperture problem": along a straight edge only the motion across the edge is visible), so every method adds an assumption that neighboring pixels move similarly. Horn and Schunck [8] add a global smoothness penalty; Farnebäck's method [9], available in OpenCV, fits a quadratic polynomial to each neighborhood and estimates the displacement from how the polynomial moves.

Deep flow estimators learn this mapping from data. SPyNet [10] estimates large motions coarse to fine: at each level of an image pyramid it warps one image by the current flow and a small network predicts the remaining, small update; it is compact and is the flow module of BasicVSR [7]. RAFT [11] builds correlation volumes between all pairs of pixels and refines a flow field iteratively with a recurrent unit. BasicVSR, for example, starts from a pretrained SPyNet and fine-tunes it together with the rest of the model [7].

*Online only: Brightness constancy and the classical flow solvers — https://blog.shyandram.dev/en/learn/deep-learning/video-super-resolution/notes/brightness-constancy-and-the-classical-flow-solvers/*

**Backward warping with bilinear sampling** To align frame $t+1$ to frame $t$, we use the flow from $t$ to $t+1$ and **pull** values: for every pixel $p$ of the reference frame, look up where that point is in the other frame and read the value there,

$$\hat{I}_{t+1\to t}(p) = I_{t+1}(p + F_{t\to t+1}(p)),$$

where $\hat{I}_{t+1\to t}$ is frame $t+1$ warped into the coordinates of frame $t$ and $F_{t\to t+1}(p)$ is the displacement of pixel $p$. This is **backward warping**: every output pixel gets exactly one value, with no holes. (Forward warping, pushing each source pixel to its new place, leaves holes and collisions.) The position $q = p + F(p)$ is usually not on the pixel grid, so the value is interpolated **bilinearly** from the four surrounding pixels:

$$I(q) = (1-a)(1-b)\, I_{00} + a(1-b)\, I_{10} + (1-a)\, b\, I_{01} + a\, b\, I_{11},$$

where $q = (u_0 + a,\, v_0 + b)$ with integer $(u_0, v_0)$ and fractions $a, b \in [0, 1)$, and $I_{ij}$ is the value at $(u_0 + i,\, v_0 + j)$. Bilinear sampling is differentiable with respect to both the image values and the

flow, so a network can learn the flow end to end through the warp; this "bilinear sampler" is the core of spatial transformer networks [12] and of every flow-based VSR model.

The snippet below implements backward warping by hand, checks it against `cv2.remap`, builds two frames in which the background moves by $(1.5, -2)$ pixels and a textured square moves by $(6, 4)$ pixels, and warps the second frame onto the first with the true flow and with Farnebäck flow from OpenCV.

```
import numpy as np, cv2
from scipy import ndimage as ndi
from skimage import data, transform

def bilinear_warp(img, flow):
    """Backward warping: out(p) = img(p + flow(p)), bilinear sampling. flow[..., 0] = dx,
     ↪  flow[..., 1] = dy."""
    H, W = img.shape
    yy, xx = np.mgrid[0:H, 0:W].astype(np.float64)
    x = np.clip(xx + flow[..., 0], 0, W - 1); y = np.clip(yy + flow[..., 1], 0, H - 1)
    x0 = np.minimum(np.floor(x).astype(int), W - 2); y0 = np.minimum(np.floor(y).astype(int),
     ↪  H - 2)
    ax, ay = x - x0, y - y0                                  # fractional parts = bilinear
     ↪  weights
    return ((1 - ay) * ((1 - ax) * img[y0, x0] + ax * img[y0, x0 + 1])
            + ay * ((1 - ax) * img[y0 + 1, x0] + ax * img[y0 + 1, x0 + 1]))

# two frames: the background moves by (dx, dy) = (1.5, -2.0) px, a textured square moves by
 ↪  (6, 4) px
bg = transform.resize(data.camera() / 255.0, (256, 256), anti_aliasing=True)
obj = transform.resize(data.coins() / 255.0, (60, 60), anti_aliasing=True)
def frame(t):
    f = ndi.shift(bg, (-2.0 * t, 1.5 * t), order=3, mode="reflect")   # ndi.shift takes
     ↪  (rows, cols)
    r, c = 90 + 4 * t, 100 + 6 * t
    f[r:r + 60, c:c + 60] = obj
    return f
I0, I1 = frame(0).astype(np.float32), frame(1).astype(np.float32)

# true backward flow from frame 0 to frame 1: where is pixel p of frame 0 found in frame 1?
flow = np.zeros((256, 256, 2), np.float32); flow[..., 0], flow[..., 1] = 1.5, -2.0
flow[90:150, 100:160] = (6.0, 4.0)
mine = bilinear_warp(I1, flow)
H, W = I0.shape; yy, xx = np.mgrid[0:H, 0:W].astype(np.float32)
ocv = cv2.remap(I1, xx + flow[..., 0], yy + flow[..., 1], cv2.INTER_LINEAR,
 ↪  borderMode=cv2.BORDER_REPLICATE)
print(f"max |ours - cv2.remap| = {np.abs(mine - ocv)[5:-5, 5:-5].max():.4f}")

u8 = lambda a: (np.clip(a, 0, 1) * 255).round().astype(np.uint8)        # Farneback expects
 ↪  8-bit images
est = cv2.calcOpticalFlowFarneback(u8(I0), u8(I1), None, pyr_scale=0.5, levels=4, winsize=21,
                                   iterations=5, poly_n=7, poly_sigma=1.5, flags=0)
m0 = np.zeros((H, W), np.float32); m0[90:150, 100:160] = 1          # square in frame 0
m1 = np.zeros((H, W), np.float32); m1[94:154, 106:166] = 1          # square in frame 1
occl = (bilinear_warp(m1, flow) > 0) & (m0 == 0)                  # background of frame 0
 ↪  hidden in frame 1
```

```
inner = (slice(8, -8), slice(8, -8))
for name, img in [("no alignment", I1), ("warped, true flow", mine), ("warped, Farneback",
 ↪  bilinear_warp(I1, est))]:
    e = np.abs(img - I0)
    print(f"{name:18s} mean |error| {e[inner].mean():.4f}   on occluded pixels
     ↪  {e[occl].mean():.4f}")
print("Farneback median flow (dx, dy): square", np.median(est[100:140, 110:150], axis=(0,
 ↪  1)).round(2),
      " background", np.median(est[200:240, 20:60], axis=(0, 1)).round(2))
# max |ours - cv2.remap| = 0.0000
# no alignment       mean |error| 0.0562   on occluded pixels 0.2367
# warped, true flow  mean |error| 0.0089   on occluded pixels 0.2617
# warped, Farneback  mean |error| 0.0133   on occluded pixels 0.1733
# Farneback median flow (dx, dy): square [6. 4.]  background [ 1.42 -1.88]
```

Warping with the true flow cuts the mean difference between the frames by a factor of about six, and Farnebäck's estimate gets most of the way there (our run). But look at the **occluded pixels**: the strip of background that the square covers in frame $t+1$. There, even the *true* flow gives a large error, because the content simply is not in frame $t+1$; it is hidden behind the square. No flow can fix that. A VSR model must learn to trust the warped frame where it is consistent and to ignore it where it is not (Figure 4).

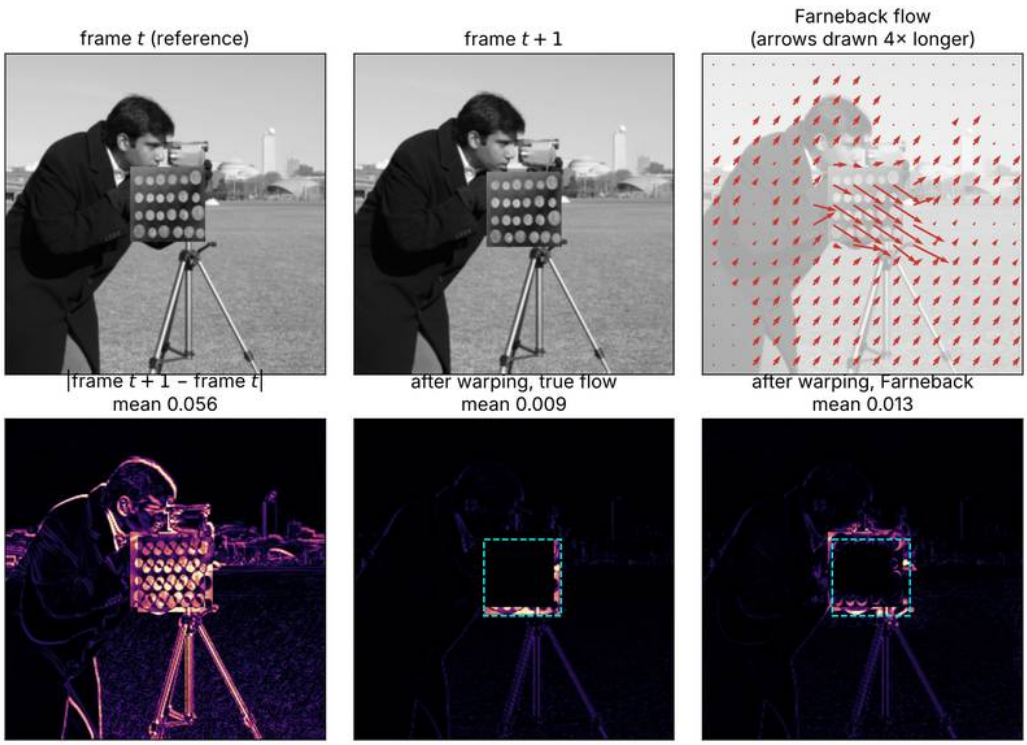



*Figure 4 — Backward warping. Warping frame t+1 onto frame t removes most of the difference, but not in the occluded strip (dashed box) where background visible in frame t is hidden in frame t+1. In our run the Farneback errors concentrate at the motion boundary. Mean absolute errors from our run.*

### Alignment strategies

**Plain version.** Alignment can use explicit flow, learned deformable sampling locations or attention to corresponding content.

**Explicit alignment: flow-based warping** The direct approach estimates a flow from each neighbor to the reference frame and warps the neighbor with it.

- **VESPCN** [2] (CVPR 2017) trains a fast multi-resolution spatial-transformer motion-compensation module end to end with a spatio-temporal version of the sub-pixel convolution (pixel shuffle) network ESPCN [13].
- **TOFlow** [14] (IJCV 2019) trains the flow network jointly with the restoration network, so the flow is **task-oriented**: it moves whatever makes the output best, even where that differs from physical motion. The paper also introduced Vimeo-90K.
- **Image versus feature warping.** Early methods warp the LR *images*. Inaccurate flow then produces blurry or wrong warped images, and the loss of detail cannot be undone later. Later methods compute flow on images but warp *feature maps*, so subsequent layers can correct small misalignments; the BasicVSR ablation found feature warping better than image warping [7].

Flow-based alignment is interpretable and works well for moderate motion. Its weaknesses are the cost of the flow network, errors at occlusions and motion boundaries, and the fact that one flow vector per pixel can point to only one place.

**Implicit alignment: deformable convolution** A normal $3 \times 3$ convolution reads its input on a fixed grid around each pixel. A **deformable convolution** [15] adds a learned offset to each of the sampling positions:

$$y(p) = \sum_{i=1}^{K} w_i \, x(p + p_i + \Delta p_i(p)),$$

where $y(p)$ is the output at pixel $p$, $K$ the number of kernel taps (9 for $3 \times 3$), $w_i$ the weight of tap $i$, $p_i$ its fixed grid position (for example $(-1, -1)$), $\Delta p_i(p)$ a learned, real-valued offset predicted by a small network for every pixel and tap, and $x(\cdot)$ is read with bilinear sampling. DCNv2 [16] also multiplies each tap by a learned modulation weight in $[0, 1]$, which lets the layer down-weight unreliable positions.

For alignment, the offsets are predicted from the features of the reference frame and a neighbor, and the deformable convolution is applied to the neighbor's features. **TDAN** [17] aligns each supporting frame to the reference this way at the feature level without computing optical flow. **EDVR** [18] applies deformable alignment in a **Pyramid, Cascading and Deformable (PCD)** module, coarse to fine over a feature pyramid to handle large motions, and fuses the aligned features with **temporal and spatial attention (TSA)**; it won all four tracks of the NTIRE 2019 video restoration challenges.

Chan et al. [19] showed that a deformable convolution can be decomposed into several spatial

warps (one per kernel tap, each with its own offset field) followed by a convolution. So deformable alignment is a generalization of flow warping with several flow fields instead of one; this **offset diversity** is why it aligns better. The same work, and BasicVSR++ [20], report that deformable alignment is unstable to train: offsets can grow very large ("offset overflow"). BasicVSR++ therefore predicts the deformable offsets as a **residual on top of optical flow** (flow-guided deformable alignment), combining the stability of flow with the diversity of deformable sampling.

*Online only: Deformable convolution as warping plus convolution — https://blog.shyandram.dev/en/learn/deep-learning/video-super-resolution/notes/deformable-convolution-as-warping-plus-convolution/*

**Alignment without an explicit motion step** Some models skip motion estimation altogether and let the network absorb motion. **DUF** [21] predicts, for every output pixel, a dynamic upsampling filter and a residual from a spatio-temporal neighborhood of the input frames, so motion is handled implicitly by the content-dependent filters. **PFNL** [22] fuses frames progressively and uses a non-local operation, which compares every position with every other, instead of motion estimation and compensation. 3-D convolutions over (time, height, width) are a third option. These methods avoid flow errors, but their effective search range is bounded by the receptive field, so large motions are harder.

**Attention and transformers** Attention lets each position compare itself with many candidate positions in other frames and take a weighted average of the best matches, which is a soft, learned version of motion search. **VRT** [23] divides the video into short clips and uses **temporal mutual self-attention (TMSA)**: mutual attention between two frames performs joint motion estimation, alignment and fusion, while self-attention extracts features; the clip partition shifts in alternate layers so information crosses clip borders, and a **parallel warping** module adds flow-based warping of neighboring features. VRT is applied to video SR, deblurring, denoising, frame interpolation and space-time SR. **RVRT** [24] uses **guided deformable attention**: starting from optical flow, it predicts several relevant locations in the previous clip and aggregates their features with attention weights.

### Architectures by temporal design

**Plain version.** Once frames can be aligned, the question is which frames each output gets to see, and how information travels along the video. There are four main answers, shown in Figure 5.

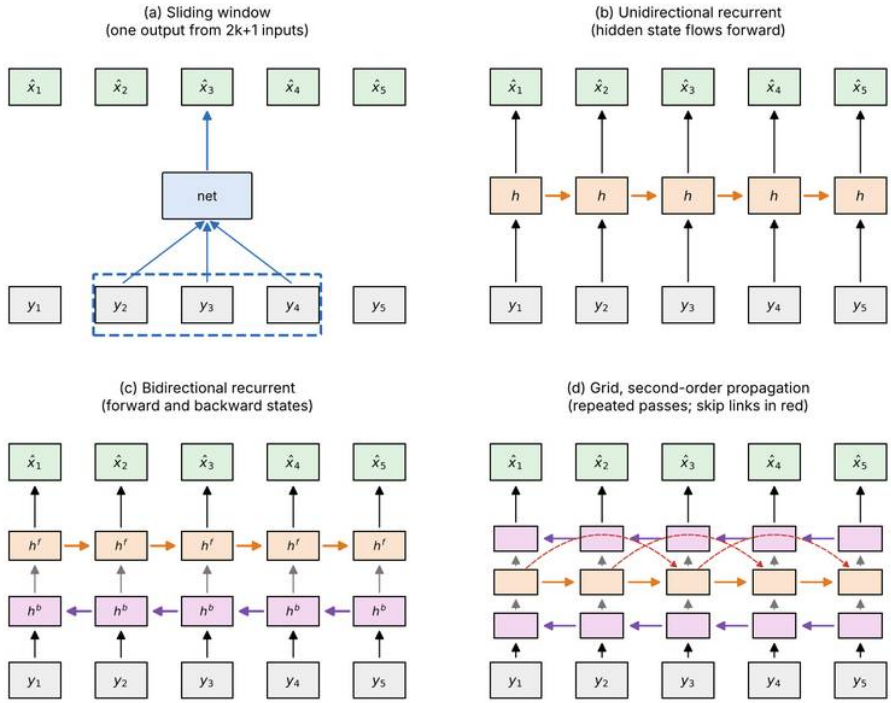



*Figure 5 — Temporal designs. (a) Sliding window: each output is computed from a fixed window of 2K + 1 frames. (b) Unidirectional recurrent: a hidden state carries information forward. (c) Bidirectional recurrent: separate forward and backward states; each output combines both at its time step. (d) Grid propagation with second-order connections (BasicVSR++): repeated backward and forward passes, with links that also skip one frame.*

**Sliding window** A **sliding-window** model takes $2K + 1$ consecutive LR frames and produces one HR frame, the center one, then moves the window by one frame. VESPCN, DUF, TDAN and EDVR are sliding-window models. It is simple and parallel, and errors cannot accumulate over time. FRVSR [3] points out two weaknesses: each input frame is processed and warped $2K + 1$ times, and each output is estimated independently, which limits temporal consistency. A fixed window also cannot use information from farther away; the BasicVSR authors measured that cutting the test sequences into shorter independent segments lowers PSNR, most strongly at segment ends [7].

**Recurrent** A **recurrent** model keeps a hidden state $h_t$ that summarizes the past and updates it frame by frame:

$$h_t = F(y_t,\ \mathcal{W}(h_{t-1}, F_{t\to t-1})), \qquad \hat{x}_t = U(h_t),$$

where $F$ is a network that combines the current LR frame $y_t$ with the previous state warped to frame $t$ by the flow $F_{t\to t-1}$ (the warp $\mathcal{W}$ is the backward warping of the previous section), and $U$

is the reconstruction and upsampling head. Because the state is aligned before reuse, information can travel over many frames at a fixed cost per frame. (For recurrent networks in general, see ML 04.) **FRVSR** [3] feeds the *previous HR output*, warped by flow and rearranged to LR size, into the network for the next frame, which encourages temporally consistent results and needs only one warp per step. **RBPN** [5] is a hybrid: within a window it integrates the neighboring frames one at a time with a recurrent encoder–decoder inspired by back-projection, feeding the estimated motion as an input instead of warping the frames.

A unidirectional recurrent model sees only the past. Early frames receive little information, and errors in the state (from a bad flow, an occlusion or a scene cut) are carried into later frames: **error accumulation**.

**Bidirectional recurrent** **BasicVSR** [7] runs two recurrent branches, one forward and one backward in time:

$$h_t^b = F_b(y_t, y_{t+1}, h_{t+1}^b), \qquad h_t^f = F_f(y_t, y_{t-1}, h_{t-1}^f), \qquad \hat{x}_t = U(h_t^f, h_t^b),$$

where $h_t^b$ and $h_t^f$ are the backward and forward features at frame $t$, $F_b$ and $F_f$ the two branches (each estimates flow with SPyNet between its two input frames, warps the propagated feature with it and refines it with residual blocks), and $U$ concatenates both features and upsamples them with pixel shuffle. Every output now sees the whole clip, past and future. The paper's ablations found a clear PSNR drop for a unidirectional variant at early frames and a larger drop without alignment, which supports its recipe: bidirectional propagation plus flow-based feature alignment is enough to be a strong baseline. Its extension **IconVSR** adds an *information-refill* module that re-extracts features from sparse keyframes to limit error accumulation at occlusions and boundaries, and *coupled propagation* that feeds the backward branch's output into the forward branch.

The price is latency: the backward branch needs the *future* frames, so the whole clip (or a long chunk) must be available before the first output. That suits offline processing and streaming with a buffer, not live video.

*Online only: BasicVSR and IconVSR — https://blog.shyandram.dev/en/learn/deep-learning/video-super-resolution/papers/basicvsr-and-iconvsr/*

**Grid and second-order propagation** **BasicVSR++** [20] makes propagation richer in two ways. **Grid propagation** passes features backward and forward several times in alternation, so each frame's features are refined repeatedly with information from the whole sequence. **Second-order connections** let the state at frame $t$ use frames $t-1$ *and* $t-2$ (a second-order Markov chain instead of first-order), so information can bypass one bad frame, which helps at occlusions and fine structures. With the flow-guided deformable alignment described above, BasicVSR++ reports a 0.82 dB PSNR gain over BasicVSR at a similar parameter count, and it won three tracks and placed second in a fourth in the NTIRE 2021 video super-resolution and compressed video enhancement challenges [20].

**Transformers and recurrent–transformer hybrids** VRT [23] processes all frames **in parallel** with long-range temporal attention at several scales, which is powerful but heavy. RVRT [24] describes the trade-off directly: parallel models fuse temporal information well but have a large model size and high memory use, while recurrent models share parameters across frames and are compact but are weaker at long-range modeling and harder to parallelize. RVRT processes neighboring frames as a clip in parallel *inside* a globally recurrent framework, using the inferred previous clip to estimate the next, with guided deformable attention for clip-to-clip alignment.

The table summarizes the representative methods of this chapter.

| Method | Venue | Temporal design | Alignment |
|---|---|---|---|
| VESPCN [2] | CVPR 2017 | sliding window | flow (spatial transformer), image level |
| FRVSR [3] | CVPR 2018 | recurrent (previous HR output) | flow, image level |
| DUF [21] | CVPR 2018 | sliding window | none (dynamic filters) |
| TOFlow [14] | IJCV 2019 | sliding window | task-oriented flow |
| EDVR [18] | CVPRW 2019 | sliding window | deformable (PCD) + attention fusion |
| RBPN [5] | CVPR 2019 | recurrent over a window | motion as input, back-projection |
| PFNL [22] | ICCV 2019 | sliding window (progressive fusion) | none (non-local) |
| TDAN [17] | CVPR 2020 | sliding window | deformable, feature level |
| BasicVSR / IconVSR [7] | CVPR 2021 | bidirectional recurrent | flow, feature level |
| BasicVSR++ [20] | CVPR 2022 | grid, second-order | flow-guided deformable |
| RVRT [24] | NeurIPS 2022 | clip-recurrent transformer | guided deformable attention |
| VRT [23] | TIP 2024 | parallel transformer | mutual attention + parallel warping |

### Real-world video and compression

**Plain version.** Benchmark LR videos are made by one clean, known recipe. Real videos are noisy, blurred in unknown ways and, above all, compressed. A model trained on the clean recipe often sharpens the compression artifacts instead of the scene.

**Compression is the main real-world degradation** Most video is stored and streamed with block-based codecs such as H.264/AVC [25], which predict each block from earlier frames with motion vectors and quantize the residual. The resulting artifacts (blocking, ringing, smeared texture, mosquito noise around edges) differ from frame to frame: frames coded without reference to others (intra frames) look different from predicted frames, and the quality changes with the bitrate. These artifacts are not independent noise and they move with the content, so a VSR model can mistake them for real detail and propagate them. **COMISR** [26] designs VSR for compressed input, with bidirectional recurrent warping, detail-preserving flow estimation and a Laplacian enhancement module that take compression properties such as the position of intra frames into account, and evaluates over a range of compression rates.

**Real-world VSR: RealBasicVSR** **RealBasicVSR** [27] (CVPR 2022) found that long-term propagation, which helps for mild degradations, can **exaggerate severe artifacts**: noise and compression artifacts are propagated and amplified along with the useful information. Its fix is an image **pre-cleaning** module applied to each frame before a BasicVSR network. It also introduced **VideoLQ**, a set of real low-quality videos without ground truth, evaluated with no-reference metrics and visual comparison.

*Online only: RealBasicVSR — https://blog.shyandram.dev/en/learn/deep-learning/video-super-resolution/papers/realbasicvsr/*

The degradation pipelines are the video versions of the blind pipelines in DLCV 09: random blur, resizing, noise and compression, with the extra choice of applying the same or different parameters to all frames of a clip.

**Diffusion-based VSR** Pretrained text-to-image diffusion models produce very realistic textures for real-world image SR (see DLCV 09). In video, their randomness is a problem: independent sampling per frame produces different textures in every frame, which is flicker at its worst. Recent methods therefore add temporal machinery to a pretrained generative model: temporal layers plus flow-guided propagation of latents across the sequence (Upscale-A-Video [28], CVPR 2024), a motion-guided loss that steers the sampling path (MGLD-VSR [29], ECCV 2024), a text-to-*video* model that already models motion (STAR [30], ICCV 2025), a diffusion transformer for videos of arbitrary length and resolution (SeedVR [31], CVPR 2025), and one-step distillation to cut the cost (DLoRAL [32], NeurIPS 2025).

*Online only: Diffusion-based VSR models — https://blog.shyandram.dev/en/learn/deep-learning/video-super-resolution/notes/diffusion-based-vsr-models/*

These models give the most convincing textures on real videos, but they inherit the image-level risk of **hallucinated detail** (plausible content that was never in the scene) and add a video-level risk: detail that is plausible in each frame but not consistent across frames. They are also far more expensive than CNN or transformer VSR.

### Training a VSR model

**Plain version.** Training is the same as for image restoration, except that each training example is a short clip, not a single image.

Pairs are made by degrading HR clips with the model above: the bicubic (BI) or blur-downsampling (BD) recipe for benchmarks (see Evaluation), or a randomized real-world pipeline. Each step samples a short clip and random crops at the same location in all frames; flips and rotations are applied identically to all frames, and the frame order can be reversed as temporal augmentation (BasicVSR reverses Vimeo-90K's 7-frame sequences this way [7]). Recurrent models are trained on longer clips (BasicVSR's REDS setting uses 15-frame inputs) so that propagation is exercised; RealBasicVSR's finding that longer sequences beat larger batches points the same way [27]. In BasicVSR the pretrained flow network and feature extractor are

frozen for the first 5,000 iterations, and the flow network is then trained with a smaller learning rate than the rest [7].

The loss is usually an L1-type pixel loss on every output frame (BasicVSR uses the smooth version $\rho(z) = \sqrt{z^2 + \varepsilon^2}$ with a small constant $\varepsilon$, where $z$ is the difference between an output and a ground-truth pixel [7]), with perceptual and adversarial terms for real-world models; DLCV 09 covers these. Some methods add a **temporal loss** that penalizes differences between consecutive outputs after motion compensation, such as the short-term warping loss of Lai et al. [33], which compares each output frame with the previous output warped by optical flow, masked where the warp is unreliable.

### Evaluation

**Plain version.** Score every frame against the ground truth, average the scores, and say exactly how the LR frames were made. Then check the video in motion, because per-frame scores cannot see flicker.

**Per-frame fidelity: Y-channel PSNR and SSIM** Most VSR papers report PSNR and SSIM [34] (defined in DLCV 09) on the **luminance (Y) channel**, computed for every frame and averaged over the frames of a clip and then over clips. The Y channel follows ITU-R BT.601 [35]: luma $E_Y = 0.299E_R + 0.587E_G + 0.114E_B$ from gamma-corrected RGB in $[0, 1]$, quantized on the 8-bit "studio" scale as $Y = 219E_Y + 16$. The convention is not universal: in the BasicVSR and BasicVSR++ comparison tables, all results are computed on Y except REDS4, which is computed on RGB [7], [20]. Mixing conventions changes numbers by several dB, as the snippet shows.

```
import numpy as np
from skimage import data, color

def rgb_to_y(rgb):
    """BT.601 luma on the 8-bit studio scale: Y = 16 + 219 * (0.299 R + 0.587 G + 0.114 B),
     ↪  RGB in [0, 1]."""
    return 16 + 65.481 * rgb[..., 0] + 128.553 * rgb[..., 1] + 24.966 * rgb[..., 2]

def psnr(a, b, peak=255.0):
    return 10 * np.log10(peak ** 2 / np.mean((a.astype(np.float64) - b) ** 2))

rng = np.random.default_rng(0)
gt = np.stack([np.roll(data.astronaut(), 2 * t, axis=1) for t in range(5)]) / 255.0   # 5
 ↪  ground-truth frames
out = np.clip(gt + rng.normal(0, 0.02, gt.shape) * np.array([1.0, 0.3, 1.5]), 0, 1)  #
 ↪  unequal error per channel

y_scores = [psnr(rgb_to_y(o), rgb_to_y(g)) for o, g in zip(out, gt)]
rgb_scores = [psnr(255 * o, 255 * g) for o, g in zip(out, gt)]
print("matches skimage rgb2ycbcr:", np.allclose(rgb_to_y(gt[0]), color.rgb2ycbcr(gt[0])[...,
 ↪  0]))
print(f"Y-channel PSNR, mean of frames: {np.mean(y_scores):.2f} dB")
print(f"RGB PSNR,       mean of frames: {np.mean(rgb_scores):.2f} dB")
```

```
# matches skimage rgb2ycbcr: True
# Y-channel PSNR, mean of frames: 43.74 dB
# RGB PSNR,       mean of frames: 33.94 dB
```

The same output scores about 10 dB higher on Y than on RGB here (our run), partly because Y is a weighted average of the three channels, so independent channel errors partly cancel, and partly because Y spans only 219 of the 255 levels while the peak in the formula stays 255. Always state the channel, the border crop and whether the frames were rounded to 8 bits.

**Degradation settings: BI and BD** Two LR recipes dominate the clean benchmarks, and results under them are not comparable:

- **BI (bicubic):** downsample each HR frame by ×4 with MATLAB's `imresize`, which blurs with a cubic filter and subsamples. The Vimeo-90K SR benchmark was built this way [14].
- **BD (blur-downsampling):** blur each HR frame with a Gaussian of standard deviation $\sigma = 1.6$, then keep every fourth pixel, as specified in BasicVSR [7].

BD includes a stronger, known blur, so a model must deblur as well as upsample. The BasicVSR tables report REDS4 under BI only, UDM10 under BD only, and Vimeo-90K-T and Vid4 under both [7]. When you read a comparison table, check the setting of each column first.

**Temporal consistency** Per-frame PSNR is blind to flicker: errors that change randomly from frame to frame can give the same average PSNR as stable errors. Three kinds of measures are used:

- **Warping error** [33]: warp output frame $t+1$ onto frame $t$ with optical flow and measure the masked difference,

$$E_{\text{warp}}(V_t, V_{t+1}) = \frac{1}{\sum_i M_t^{(i)}} \sum_{i=1}^{N} M_t^{(i)} \|V_t^{(i)} - \hat{V}_{t+1}^{(i)}\|_2^2,$$

where $V_t^{(i)}$ is pixel $i$ of output frame $t$, $\hat{V}_{t+1}$ is frame $t+1$ warped to frame $t$, $N$ the number of pixels and $M_t^{(i)} \in \{0,1\}$ a non-occlusion mask; the video score averages over all consecutive pairs. Its weakness: a blurry, detail-free video is very stable and scores well.

- **tOF and tLP** (TecoGAN [36]) compare the temporal behavior of the output with that of the ground truth instead of demanding zero change:

$$\text{tOF} = \|\text{OF}(b_{t-1}, b_t) - \text{OF}(g_{t-1}, g_t)\|_1, \qquad \text{tLP} = |\text{LP}(b_{t-1}, b_t) - \text{LP}(g_{t-1}, g_t)|,$$

where $b_t$ and $g_t$ are the output and ground-truth frames, OF is optical flow estimated with Farnebäck's method and LP is the LPIPS perceptual distance [37]. The TecoGAN authors introduced them because a plain warped-difference measure "can be easily deceived" by blurry results such as bicubic interpolation.

- **Temporal profiles**: one row of pixels followed through time and stacked into an image; speckle and streaks reveal flicker. VESPCN used them for qualitative comparison [2].

The snippet builds a toy scene with known motion and four "outputs": independent per-frame invented texture, the same outputs smoothed recursively along the motion, the same smoothing with the motion ignored, and a blurry output. Flicker is measured in the spirit of tOF: the temporal change of the output after motion compensation, minus that of the ground truth.

```
import numpy as np
from scipy import ndimage as ndi
from skimage import data, transform

rng = np.random.default_rng(0)
T, v = 12, (1, 2)                                        # 12 frames, the scene moves 1 px
 ↪  down and 2 px right per frame
base = transform.resize(data.camera() / 255.0, (256, 256), anti_aliasing=True)
gt = np.stack([np.roll(base, (v[0] * t, v[1] * t), axis=(0, 1)) for t in range(T)])
warp_prev = lambda f: np.roll(f, v, axis=(0, 1))          # exact motion compensation for
 ↪  this toy scene

def fake_detail():                                        # zero-mean "hallucinated texture",
 ↪  new every frame
    n = ndi.gaussian_filter(rng.standard_normal(base.shape), 0.8)
    return 0.03 * n / n.std()

per_frame = gt + np.stack([fake_detail() for _ in range(T)])  # each frame restored
 ↪  independently
smoothed, unaligned, blurry = per_frame.copy(), per_frame.copy(),
 ↪  np.stack([ndi.gaussian_filter(g, 1.5) for g in gt])
for t in range(1, T):
    smoothed[t] = 0.3 * per_frame[t] + 0.7 * warp_prev(smoothed[t - 1])   # recursive filter
     ↪  along the motion
    unaligned[t] = 0.3 * per_frame[t] + 0.7 * unaligned[t - 1]           # same filter,
     ↪  motion ignored

c = (slice(None), slice(16, -16), slice(16, -16))          # ignore borders (np.roll wraps
 ↪  around)
def psnr(out):
    return np.mean([10 * np.log10(1 / np.mean((o - g) ** 2)) for o, g in zip(out[c], gt[c])])
def flicker(out):
    """Mean |temporal change of output - temporal change of ground truth| after motion
     ↪  compensation."""
    d_out = out[1:] - np.stack([warp_prev(f) for f in out[:-1]])
    d_gt = gt[1:] - np.stack([warp_prev(f) for f in gt[:-1]])
    return np.abs(d_out - d_gt)[c].mean()

for name, out in [("per-frame", per_frame), ("aligned smoothing", smoothed),
                  ("unaligned smoothing", unaligned), ("blurry output", blurry)]:
    print(f"{name:20s} PSNR {psnr(out):5.2f} dB   flicker {flicker(out):.4f}")
# per-frame            PSNR 30.50 dB   flicker 0.0338
# aligned smoothing    PSNR 36.40 dB   flicker 0.0082
# unaligned smoothing  PSNR 21.74 dB   flicker 0.0223
# blurry output        PSNR 25.99 dB   flicker 0.0000
```

Read the four lines together (all from our toy run). Propagating information **along the motion** reduces flicker and, in this toy, also improves PSNR, because the invented texture is zero-mean noise that averages out; real hallucinated texture is not, so do not expect PSNR gains of this size. Propagating **without alignment** ghosts the moving scene: PSNR collapses and flicker stays high. And the **blurry output** has zero flicker but poor PSNR, which is the TecoGAN warning in miniature: always report a temporal measure *together with* a fidelity or perceptual measure. Figure 6 shows the same effects as motion-compensated temporal profiles.

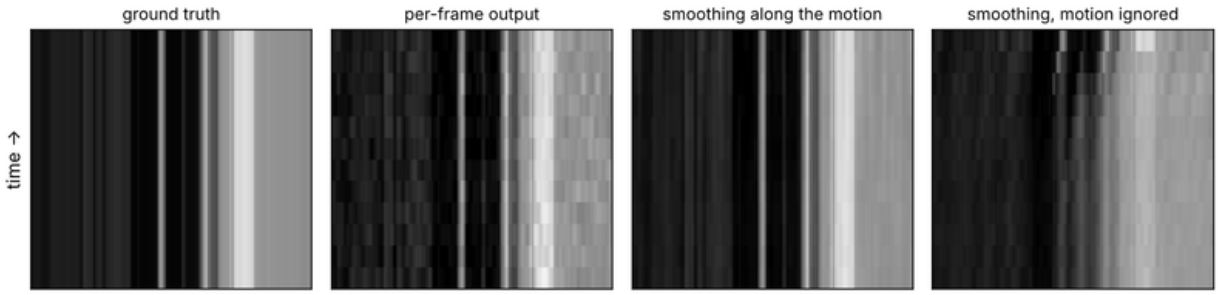



*Figure 6 — Motion-compensated temporal profiles of the toy scene: one scene row followed over 12 frames (time runs downward). Flicker appears as row-to-row speckle; unaligned smoothing smears the structure.*

### Datasets and test sets

| Dataset | What it is | Typical use |
|---|---|---|
| **Vimeo-90K** [14] | 89,800 video clips (720p or higher) from Vimeo; the SR benchmark uses 7-frame sequences at 448 × 256, with ×4 LR inputs of 112 × 64 made by MATLAB `imresize` | training; test split **Vimeo-90K-T** |
| **REDS** [38] | 300 sequences of 100 frames at 720 × 1280 (240 training, 30 validation, 30 test), captured for the NTIRE 2019 video challenges, with large and complex motion | training; **REDS4** test set |
| **REDS4** | four REDS training clips (000, 011, 015, 020) held out for testing, following EDVR [7] | test under BI |
| **Vid4** [39] | four short sequences from Liu and Sun's Bayesian adaptive VSR work | test under BI and BD |
| **UDM10** | ten test sequences released with PFNL [22] | test under BD |
| **VideoLQ** [27] | real-world low-quality videos without ground truth | real-world evaluation (no-reference metrics, visual) |

The Liu et al. survey notes that test videos evolved from Vid4 toward REDS as methods aimed at more complex motion and scene changes [40]; BasicVSR describes REDS4 as containing large and complicated motions [7]. Because Vimeo-90K-T sequences have only seven frames, long-range

propagation has little to work with there: BasicVSR reports slightly lower scores on Vimeo-90K-T than sliding-window methods such as EDVR and attributes this to the short sequences [7].

**Challenges** Challenges at the NTIRE (CVPR) and AIM (ECCV) workshops created datasets and track the state of the art. **NTIRE 2019** introduced REDS [38]; its video SR challenge [41] had a bicubic track and a track with realistic motion blur, and EDVR won all four video restoration tracks that year [18]. **NTIRE 2021** video SR [42] had a restoration-quality track and a lower-frame-rate track, which turns the problem into space-time SR (raising resolution and frame rate together); BasicVSR++ reports three first places and one second place across that challenge and the compressed video enhancement challenge [20]. **Efficiency challenges** judge quality per unit of compute: the Mobile AI & AIM 2022 report [43] asked for real-time ×4 VSR on a phone's neural processing unit, trained on REDS and judged on runtime and power, and the AIM 2024 challenge on efficient VSR for compressed content [44] targets streaming with ×4 upscaling from 540p to 4K and ×3 from 360p to 1080p.

### Efficiency and streaming

**Plain version.** A video model must run on every frame, often in real time, and often before the future frames exist. Those constraints shape the architecture as much as accuracy does.

- **Cost per frame.** A sliding window processes each input frame $2K + 1$ times unless features are cached; a recurrent model processes each frame once and carries a state, so its cost per frame does not grow with the length of the video [3].
- **Latency and causality.** Bidirectional and grid propagation need future frames, so the first output waits for the end of the chunk. Live video (calls, game streaming) allows only causal designs or short look-ahead buffers, at some cost in quality (the unidirectional ablation of BasicVSR [7]). Long videos are processed in chunks, and quality dips at chunk ends, as BasicVSR's segment experiment shows.
- **Memory.** Parallel transformers such as VRT attend across many frames at several scales; their model size and memory use are the drawback RVRT addresses with clip-level recurrence [24]. Diffusion VSR is costlier still, which motivates one-step models [32].
- **Deployment.** The efficiency challenges set the target for phones and streaming: few parameters and operations, real-time frame rates and low power, with quality judged against interpolation baselines rather than against the largest models [43], [44].

### Failure modes

**Plain version.** VSR fails where its main assumption fails: that neighboring frames show the same content, slightly moved.

- **Occlusion and disocclusion.** Content hidden in the neighbors (Figure 4) has no extra measurements; a model that trusts the warped features there produces ghosts or doubled edges. Modulation masks and attention weights let a model down-weight such regions [20].

- **Large and complex motion.** Fast, non-rigid motion and motion blur make alignment inaccurate; coarse-to-fine alignment (PCD in EDVR, the pyramid in SPyNet) helps but does not remove the problem. REDS was used to stress exactly this [18].
- **Error accumulation.** In recurrent models a wrong state is carried forward, especially after occlusions or scene cuts; IconVSR's keyframe refill and BasicVSR++'s second-order links are direct countermeasures [7], [20]. Long propagation can also amplify severe input artifacts, which RealBasicVSR counters with pre-cleaning [27].
- **Flicker.** Independent per-frame decisions, in single-image models or in diffusion sampling, cause temporal instability that per-frame metrics cannot see.
- **Hallucinated detail.** GAN and diffusion VSR can create plausible texture, text or faces that were never there, and in video the invented content may also drift. For surveillance, forensic, medical or scientific video, prefer fidelity-oriented models and state which kind of model produced an output.
- **Domain gap.** Models trained on BI or BD degradations do not account for compression and real noise [26], [27].

## Modern view

### Survey papers

- **Liu et al., "Video Super-Resolution Based on Deep Learning: A Comprehensive Survey" (Artificial Intelligence Review, 2022)** [40]. Reviews 37 deep VSR methods in seven subcategories by **how they use inter-frame information**: *with alignment* (motion estimation and compensation, deformable convolution) or *without alignment* (2-D and 3-D convolution, recurrent CNNs, non-local operations, others), and compares them on benchmarks. Takeaway: the field moved from aligned short windows toward a mix of designs and long sequences, with REDS replacing Vid4 as the demanding test set.
- **Tu et al., "Optical Flow for Video Super-Resolution: A Survey" (Artificial Intelligence Review, 2022)** [45]. Covers metrics and classical and deep flow-based VSR, with an analysis of representative deep methods. Takeaway: dense, sub-pixel flow is among the most common tools for motion compensation, so understanding flow is a large part of understanding VSR alignment.
- **Baniya et al., "A Survey of Deep Learning Video Super-Resolution" (IEEE Transactions on Emerging Topics in Computational Intelligence, 2024)** [46]. A component taxonomy (inputs, alignment, fusion, refinement and reconstruction, upsampling), plus losses, quality and efficiency evaluation, applications and trends such as online, real-world, resource-constrained and space-time VSR. Takeaway: design choices are often driven by benchmark gains rather than explained, and practical settings deserve more attention.

### State of the art, 2023–2026

1. **Propagation plus alignment is the core of fidelity-oriented VSR.** Bidirectional or grid recurrent propagation with flow-guided deformable alignment (BasicVSR++) and parallel or clip-

recurrent transformers (VRT, RVRT) are the reference designs of this chapter, trading accuracy against memory and latency.
2. **Real-world VSR moved to generative priors.** After CNN-based real-world models such as RealBasicVSR, diffusion-based VSR built on image diffusion models (Upscale-A-Video, MGLD-VSR), then on video diffusion models and diffusion transformers (STAR, SeedVR), with temporal consistency as the central technical problem.
3. **Cost reduction of generative VSR.** One-step diffusion (DLoRAL) follows the same path as one-step image SR.
4. **Efficiency and compression as first-class targets**, with challenge tracks for mobile hardware and compressed streaming content [43], [44].

### Open problems

- **Faithful versus plausible.** Measuring and controlling hallucination in video, where invented detail may also be temporally unstable, is unsolved.
- **Temporal evaluation.** Warping error can be gamed by blur, tOF and tLP depend on flow and feature estimates, there is no agreed perceptual measure of temporal quality, and user studies are costly.
- **Real degradations and codecs.** Synthetic pipelines still miss real camera processing and codec behavior; ground truth for real videos is rarely available.
- **Long videos, online processing and scene cuts.** Combining the quality of bidirectional and global models with causal, constant-memory processing remains difficult.
- **Occlusions and large motion.** Alignment is still the weak point under fast, non-rigid motion and disocclusion.
- **Cost of generative VSR**, which is still far above that of CNN and transformer VSR for long, high-resolution video.

## Key takeaways

- Sub-pixel motion can reveal extra scene measurements; independent frame upscaling discards them.
- Align corresponding content before combining it; occluded regions may have no valid match.
- Flow, deformable sampling and attention offer different correspondence mechanisms and costs.
- Recurrence propagates information and errors; bidirectional processing also needs future frames.
- Compression artifacts and generated textures can accumulate or flicker across time.
- Report the degradation, color and border protocol with spatial and temporal metrics; blur alone can look stable.

## Exercises

1. **Integer versus sub-pixel motion.** In the multi-frame snippet, replace the random shifts by integer shifts (`rng.integers(0, s, size=(T, 2)).astype(float)`, keeping `shifts[0] =`

`0`) and rerun. Then try all shifts equal to zero. Explain the PSNR you get in each case using the sampling argument of Figure 1.

**Hint**

Integer shifts in HR pixels still land the LR samples on different HR phases (an HR shift of 1 is a quarter LR pixel), so they still help; what matters is the shift *modulo s* HR pixels. With all shifts zero every frame samples the same HR positions, so only the noise averages out and the result stays close to the single-frame solver. Try shifts that are multiples of 4 HR pixels to see the "no new information" case directly.

2. **Registration error budget.** Add Gaussian noise with standard deviation 0.25, 0.5 and 1.0 HR pixels to the true shifts before calling `reconstruct`, and plot PSNR against the shift error. Compare with the 0.13-pixel error of phase correlation. What does this suggest about how accurate alignment must be for super-resolution, compared with denoising?

**Hint**

PSNR should fall quickly once the shift error approaches the size of the detail being recovered (a fraction of an HR pixel), because samples land in the wrong places and the reconstruction blurs or rings. Denoising by averaging tolerates larger errors, since it only needs the same content, not its exact sub-pixel position. This is why VSR needs sub-pixel accurate flow, while many video denoisers can work with coarser alignment.

3. **Forward versus backward warping.** Write a forward-warping function that pushes each pixel of frame $t+1$ to $p - F(p)$ (rounded to the nearest pixel) and compare its output with `biline ar_warp` on the two-frame example. Count holes (pixels that receive no value) and collisions (pixels that receive several).

**Hint**

Holes appear where the scene expands or is disoccluded (behind the moving square), and collisions where it contracts or is occluded. Backward warping has neither, because every output pixel reads exactly one (interpolated) value, which is why networks use it; the price is that it needs the flow defined on the *reference* frame.

4. **Sliding window versus recurrent cost.** A sliding-window model with a 7-frame window runs a feature extractor on every input frame of the window and one flow network per neighbor. A bidirectional recurrent model runs one feature extractor per frame and two flow estimations per frame (one per direction). Count feature extractions and flow estimations per output frame for each design, with and without caching features between windows. Which design has lower latency for the first output of a 100-frame clip?

**Hint**

Without caching the window model does 7 feature extractions and 6 flow estimations per output; with caching, 1 new extraction per output but still 6 flows (each neighbor must be aligned to a new

center). The bidirectional model does 1 extraction and 2 flows per frame. But its first output needs the backward pass over all 100 frames, while the window model needs only frames 1 to 4. Cost per frame favors recurrence; latency favors the window.

5. **Flicker metric pitfalls.** In the flicker snippet, replace the zero-mean fake detail by a *fixed* texture pattern that moves with the scene (generate it once and roll it with the scene). Rerun and explain why the flicker measure changes while PSNR does not change much. What does this say about what tOF/tLP-style measures can and cannot detect?

**Hint**

A texture that moves consistently with the scene has the same temporal change as the ground truth, so the motion-compensated flicker drops to about zero, while PSNR stays roughly as low as before because the texture is still wrong. Temporal measures detect *inconsistency*, not *incorrectness*: a consistently hallucinated texture passes them. Fidelity (or perceptual) metrics and temporal metrics answer different questions.

## References


1. S. Farsiu, M. D. Robinson, M. Elad and P. Milanfar, "Fast and Robust Multiframe Super Resolution," *IEEE Transactions on Image Processing*, vol. 13, no. 10, pp. 1327–1344, 2004. doi
2. J. Caballero, C. Ledig, A. Aitken, A. Acosta, J. Totz, Z. Wang and W. Shi, "Real-Time Video Super-Resolution with Spatio-Temporal Networks and Motion Compensation," in *Proc. IEEE CVPR*, 2017. CVF ·arXiv
3. M. S. M. Sajjadi, R. Vemulapalli and M. Brown, "Frame-Recurrent Video Super-Resolution," in *Proc. IEEE/CVF CVPR*, 2018. CVF ·arXiv
4. M. Irani and S. Peleg, "Improving Resolution by Image Registration," *CVGIP: Graphical Models and Image Processing*, vol. 53, no. 3, pp. 231–239, 1991. doi
5. M. Haris, G. Shakhnarovich and N. Ukita, "Recurrent Back-Projection Network for Video Super-Resolution," in *Proc. IEEE/CVF CVPR*, 2019. CVF ·arXiv
6. M. Guizar-Sicairos, S. T. Thurman and J. R. Fienup, "Efficient Subpixel Image Registration Algorithms," *Optics Letters*, vol. 33, no. 2, p. 156, 2008. doi
7. K. C. K. Chan, X. Wang, K. Yu, C. Dong and C. C. Loy, "BasicVSR: The Search for Essential Components in Video Super-Resolution and Beyond," in *Proc. IEEE/CVF CVPR*, 2021. CVF · arXiv
8. B. K. P. Horn and B. G. Schunck, "Determining Optical Flow," *Artificial Intelligence*, vol. 17, no. 1–3, pp. 185–203, 1981. doi
9. G. Farnebäck, "Two-Frame Motion Estimation Based on Polynomial Expansion," in *Image Analysis* (Lecture Notes in Computer Science), pp. 363–370, Springer, 2003. doi
10. A. Ranjan and M. J. Black, "Optical Flow Estimation Using a Spatial Pyramid Network," in *Proc. IEEE CVPR*, 2017. CVF ·arXiv
11. Z. Teed and J. Deng, "RAFT: Recurrent All-Pairs Field Transforms for Optical Flow," in *Proc. ECCV* (Lecture Notes in Computer Science), pp. 402–419, 2020. doi ·arXiv

12. M. Jaderberg, K. Simonyan, A. Zisserman and K. Kavukcuoglu, "Spatial Transformer Networks," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2015. proceedings · arXiv
13. W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert and Z. Wang, "Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network," in *Proc. IEEE CVPR*, 2016. arXiv
14. T. Xue, B. Chen, J. Wu, D. Wei and W. T. Freeman, "Video Enhancement with Task-Oriented Flow," *International Journal of Computer Vision*, vol. 127, no. 8, pp. 1106–1125, 2019. doi ·arXiv
15. J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu and Y. Wei, "Deformable Convolutional Networks," in *Proc. IEEE ICCV*, 2017. CVF ·arXiv
16. X. Zhu, H. Hu, S. Lin and J. Dai, "Deformable ConvNets V2: More Deformable, Better Results," in *Proc. IEEE/CVF CVPR*, 2019. CVF ·arXiv
17. Y. Tian, Y. Zhang, Y. Fu and C. Xu, "TDAN: Temporally-Deformable Alignment Network for Video Super-Resolution," in *Proc. IEEE/CVF CVPR*, 2020. CVF ·arXiv
18. X. Wang, K. C. K. Chan, K. Yu, C. Dong and C. C. Loy, "EDVR: Video Restoration with Enhanced Deformable Convolutional Networks," in *Proc. IEEE/CVF CVPR Workshops*, 2019. CVF ·arXiv
19. K. C. K. Chan, X. Wang, K. Yu, C. Dong and C. C. Loy, "Understanding Deformable Alignment in Video Super-Resolution," in *Proc. AAAI Conference on Artificial Intelligence*, vol. 35, no. 2, pp. 973–981, 2021. doi ·arXiv
20. K. C. K. Chan, S. Zhou, X. Xu and C. C. Loy, "BasicVSR++: Improving Video Super-Resolution with Enhanced Propagation and Alignment," in *Proc. IEEE/CVF CVPR*, 2022. CVF ·arXiv
21. Y. Jo, S. W. Oh, J. Kang and S. J. Kim, "Deep Video Super-Resolution Network Using Dynamic Upsampling Filters Without Explicit Motion Compensation," in *Proc. IEEE/CVF CVPR*, 2018. CVF
22. P. Yi, Z. Wang, K. Jiang, J. Jiang and J. Ma, "Progressive Fusion Video Super-Resolution Network via Exploiting Non-Local Spatio-Temporal Correlations," in *Proc. IEEE/CVF ICCV*, 2019. CVF ·code and test data (Vid4, UDM10)
23. J. Liang, J. Cao, Y. Fan, K. Zhang, R. Ranjan, Y. Li, R. Timofte and L. Van Gool, "VRT: A Video Restoration Transformer," *IEEE Transactions on Image Processing*, vol. 33, pp. 2171–2182, 2024. doi ·arXiv
24. J. Liang, Y. Fan, X. Xiang, R. Ranjan, E. Ilg, S. Green, J. Cao, K. Zhang, R. Timofte and L. Van Gool, "Recurrent Video Restoration Transformer with Guided Deformable Attention," in *Advances in Neural Information Processing Systems* (*NeurIPS*), vol. 35, pp. 378–393, 2022. arXiv
25. T. Wiegand, G. J. Sullivan, G. Bjøntegaard and A. Luthra, "Overview of the H.264/AVC Video Coding Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, 2003. doi
26. Y. Li, P. Jin, F. Yang, C. Liu, M.-H. Yang and P. Milanfar, "COMISR: Compression-Informed Video Super-Resolution," in *Proc. IEEE/CVF ICCV*, 2021. CVF ·arXiv
27. K. C. K. Chan, S. Zhou, X. Xu and C. C. Loy, "Investigating Tradeoffs in Real-World Video Super-Resolution," in *Proc. IEEE/CVF CVPR*, 2022. CVF ·arXiv
28. S. Zhou, P. Yang, J. Wang, Y. Luo and C. C. Loy, "Upscale-A-Video: Temporal-Consistent Diffusion Model for Real-World Video Super-Resolution," in *Proc. IEEE/CVF CVPR*, pp. 2535–2545, 2024. CVF ·arXiv
29. X. Yang, C. He, J. Ma and L. Zhang, "Motion-Guided Latent Diffusion for Temporally Consis-

tent Real-World Video Super-Resolution," in *Proc. ECCV* (Lecture Notes in Computer Science), pp. 224–242, 2024. doi ·arXiv
30. R. Xie, Y. Liu, P. Zhou, C. Zhao, J. Zhou, K. Zhang, Z. Zhang, J. Yang, Z. Yang and Y. Tai, "STAR: Spatial-Temporal Augmentation with Text-to-Video Models for Real-World Video Super-Resolution," in *Proc. IEEE/CVF ICCV*, 2025. CVF ·arXiv
31. J. Wang, Z. Lin, M. Wei, Y. Zhao, C. Yang, C. C. Loy and L. Jiang, "SeedVR: Seeding Infinity in Diffusion Transformer Towards Generic Video Restoration," in *Proc. IEEE/CVF CVPR*, 2025. CVF ·arXiv
32. Y. Sun, L. Sun, S. Liu, R. Wu, Z. Zhang and L. Zhang, "One-Step Diffusion for Detail-Rich and Temporally Consistent Video Super-Resolution," in *Advances in Neural Information Processing Systems* (*NeurIPS*), vol. 38, 2025. arXiv
33. W.-S. Lai, J.-B. Huang, O. Wang, E. Shechtman, E. Yumer and M.-H. Yang, "Learning Blind Video Temporal Consistency," in *Proc. ECCV*, 2018. CVF ·arXiv
34. Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. doi
35. ITU-R, "Recommendation ITU-R BT.601-7: Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios," International Telecommunication Union, 2011. ITU
36. M. Chu, Y. Xie, J. Mayer, L. Leal-Taixé and N. Thuerey, "Learning Temporal Coherence via Self-Supervision for GAN-based Video Generation," *ACM Transactions on Graphics*, vol. 39, no. 4, article 75, 2020. doi ·arXiv
37. R. Zhang, P. Isola, A. A. Efros, E. Shechtman and O. Wang, "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric," in *Proc. IEEE/CVF CVPR*, 2018. CVF ·arXiv
38. S. Nah, S. Baik, S. Hong, G. Moon, S. Son, R. Timofte and K. M. Lee, "NTIRE 2019 Challenge on Video Deblurring and Super-Resolution: Dataset and Study," in *Proc. IEEE/CVF CVPR Workshops*, 2019. CVF
39. C. Liu and D. Sun, "On Bayesian Adaptive Video Super Resolution," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 2, pp. 346–360, 2014. doi
40. H. Liu, Z. Ruan, P. Zhao, C. Dong, F. Shang, Y. Liu, L. Yang and R. Timofte, "Video Super-Resolution Based on Deep Learning: A Comprehensive Survey," *Artificial Intelligence Review*, vol. 55, no. 8, pp. 5981–6035, 2022. doi ·arXiv
41. S. Nah, R. Timofte, S. Gu, S. Baik, S. Hong, G. Moon, S. Son and K. M. Lee, "NTIRE 2019 Challenge on Video Super-Resolution: Methods and Results," in *Proc. IEEE/CVF CVPR Workshops*, 2019. CVF
42. S. Son, S. Lee, S. Nah, R. Timofte and K. M. Lee, "NTIRE 2021 Challenge on Video Super-Resolution," in *Proc. IEEE/CVF CVPR Workshops*, 2021. CVF ·arXiv
43. A. Ignatov, R. Timofte et al., "Power Efficient Video Super-Resolution on Mobile NPUs with Deep Learning, Mobile AI & AIM 2022 Challenge: Report," arXiv:2211.05256, 2022. arXiv
44. M. V. Conde, Z. Lei, W. Li, C. Bampis, I. Katsavounidis, R. Timofte et al., "AIM 2024 Challenge on Efficient Video Super-Resolution for AV1 Compressed Content," in *Proc. ECCV Workshops* (Lecture Notes in Computer Science), pp. 304–325, 2025. doi ·arXiv
45. Z. Tu, H. Li, W. Xie, Y. Liu, S. Zhang, B. Li and J. Yuan, "Optical Flow for Video Super-Resolution: A Survey," *Artificial Intelligence Review*, vol. 55, no. 8, pp. 6505–6546, 2022. doi ·

arXiv
46. A. A. Baniya, T.-K. Lee, P. W. Eklund and S. Aryal, “A Survey of Deep Learning Video Super-Resolution,” *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 8, no. 4, pp. 2655–2676, 2024. doi ·arXiv
47. B. Wronski, I. Garcia-Dorado, M. Ernst, D. Kelly, M. Krainin, C.-K. Liang, M. Levoy and P. Milanfar, “Handheld Multi-Frame Super-Resolution,” *ACM Transactions on Graphics*, vol. 38, no. 4, 2019. doi ·arXiv

# DLCV 09B • Generative Image Restoration: Sampling a Plausible Answer

Why regression-trained restorers blur, what it means to sample from the posterior instead, and how GAN priors, conditional and zero-shot diffusion, latent-diffusion and flow-based restorers do it, with the evaluation, cost and hallucination risks that come with them.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/generative-restoration/

## What you'll learn

- Explain why squared-error restoration averages uncertain detail.
- Distinguish plausible posterior samples from faithful recovery.
- Compare GAN priors, conditional diffusion and pretrained-prior inverse solvers.
- Trace prior guidance and measurement consistency in a runnable toy example.
- Evaluate randomness, hallucination and complete inference cost.

## The big picture

DLCV 09 trained restoration networks to be close to the clean image on average. That works well for mild damage. When the damage destroys a lot of information (a 16-pixel-wide face, a heavily compressed photo, a large hole), "close on average" produces a smooth, waxy guess. Generative restoration takes a different stance: since many clean images are consistent with the input, *pick one of them*, and make sure it looks like a real photo.

> **Explain like I'm five**
>
> A friend shows you a tiny, blurry photo of a car's number plate and asks what it says. One way to answer is to imagine every plate that could look like this blur and mix them all together: you get a gray smudge, which is never exactly wrong but is useless to read. Another way is to draw one plate that really could look like that blur: crisp letters, nice and clear. That drawing *looks* great, but it is only one possibility, and the letters may not be the real ones. Regression restorers draw the smudge; generative restorers draw the crisp plate. This chapter is about drawing crisp plates well, and about never forgetting that they might be wrong.

Why it matters. The most convincing real-world super-resolution and face or old-photo restoration of recent years comes from generative models, GANs first and diffusion models since about 2021. The same machinery is used for medical and scientific inverse problems, where an invented detail can do real harm. So you need two skills at once: how these methods work, and how to measure what they invent.

**Where this chapter sits.** It builds on the degradation model, losses and metrics of DLCV 09 and the generative models of ML 07; diffusion and flow matching are derived in ML 08, and here we only use them. Super-resolution, which has its own chapters (DLCV 09A for images, DLCV 09A.3 for video), appears here only as an example. All code runs in seconds on a CPU with NumPy and SciPy, on toy problems whose exact answer is known, so every claim can be checked.

### Why regression gives blur: the posterior mean

**Plain version.** Squared-error training favors the conditional average. When several sharp structures explain the same input, their average can be blurry.

### One input, many answers

Write the measurement as $y = \mathcal{D}(x) + n$, as in DLCV 09. For super-resolution, deblurring, inpainting or heavy compression, $\mathcal{D}$ throws information away, so a whole set of clean images $x$ explains the same $y$ almost equally well. In the Bayesian language, the uncertainty that remains after seeing $y$ is the **posterior distribution**

$$p(x \mid y) = \frac{p(y \mid x)\, p(x)}{p(y)},$$

where $p(x)$ is the **prior** (how likely an image is before we look at $y$: natural images are likely, noise is not), $p(y \mid x)$ is the **likelihood** (how well $x$ explains the measurement; for Gaussian noise of standard deviation $\sigma$ it is proportional to $\exp(-\|y - \mathcal{D}(x)\|^2 / 2\sigma^2)$), and $p(y)$ is a normalizing constant that does not depend on $x$. Mild damage gives a narrow posterior, and every reasonable method agrees. Severe damage gives a wide posterior, often with several separate modes ("the edge is here" or "two pixels to the right").

### The math: squared error is minimized by the posterior mean

A restoration network $f_\theta$ trained on pairs $(y, x)$ with the squared error minimizes the expected loss $\mathbb{E}\,\|f_\theta(y) - x\|^2$ over the data distribution. For each fixed $y$, the best output $\hat{x}$ minimizes

$$\mathbb{E}[\|\hat{x} - x\|^2 \mid y] = \|\hat{x} - \mathbb{E}[x \mid y]\|^2 + \mathbb{E}[\|x - \mathbb{E}[x \mid y]\|^2 \mid y],$$

where the expectation is over the posterior $p(x \mid y)$ and $\mathbb{E}[x \mid y]$ is the posterior mean. The second term does not depend on $\hat{x}$ (it is the irreducible uncertainty), so the minimum is at

$$\hat{x}_{\mathrm{MMSE}}(y) = \mathbb{E}[x \mid y] = \int x\, p(x \mid y)\, dx,$$

the **minimum mean squared error (MMSE)** estimator. A large enough network trained long enough approaches it. (The L1 loss gives the per-pixel posterior *median* instead: sometimes sharper, but still chosen pixel by pixel rather than as one coherent image; DLCV 09 compares the losses.) The posterior mean is the right answer if you are scored by MSE or PSNR, and the wrong one if you want an image that looks real: averaging images with edges in different places gives an image with no sharp edge at all.

### Code: a toy where the mean is blurry and every sample is sharp

We need a problem whose posterior we can compute exactly. The clean "images" are 1-D signals of 64 pixels with one step edge (0 on the left, 1 on the right) at an unknown position between

pixels 8 and 55, all equally likely. The degradation blurs with a Gaussian of standard deviation 3 pixels, keeps one pixel in eight and adds Gaussian noise of standard deviation 0.2. With only 48 possible clean signals, the posterior is an exact list of 48 probabilities.

```
import numpy as np
from scipy.ndimage import gaussian_filter1d
from scipy.stats import wasserstein_distance

rng = np.random.default_rng(0)
n, factor, sigma = 64, 8, 0.2
positions = np.arange(8, 56)                                        # where the edge may be
X = (np.arange(n)[None, :] >= positions[:, None]).astype(float)  # every possible clean
 ↪  signal
A = lambda x: gaussian_filter1d(x, 3.0, axis=-1, mode="nearest")[..., ::factor]  # blur, keep
 ↪  1 of 8
AX = A(X)                                                           # every candidate, degraded

def posterior(y):
    """Exact p(edge position | y): uniform prior, Gaussian noise of std sigma."""
    logp = -np.sum((y - AX) ** 2, axis=1) / (2 * sigma**2)
    p = np.exp(logp - logp.max())
    return p / p.sum()

truth, mean, post, prior = [], [], [], []
for _ in range(2000):
    x = X[rng.integers(len(positions))]
    y = A(x) + sigma * rng.standard_normal(n // factor)
    p = posterior(y)
    truth.append(x)
    mean.append(p @ X)                                              # posterior mean (MMSE)
    post.append(X[rng.choice(len(positions), p=p)])                 # a sample from p(x | y)
    prior.append(X[rng.integers(len(positions))])                   # a sample that ignores y
truth = np.array(truth)

for name, est in [("posterior mean", mean), ("posterior sample", post), ("prior sample",
 ↪  prior)]:
    est = np.array(est)
    mse = np.mean((est - truth) ** 2)
    w1 = wasserstein_distance(est.ravel(), truth.ravel())           # pixel values like real
     ↪  ones?
    jump = np.abs(np.diff(est, axis=1)).max(axis=1).mean()          # sharpest step (real: 1.0)
    print(f"{name:16s} MSE {mse:.4f} | W1 {w1:.4f} | largest step {jump:.2f}")
# posterior mean   MSE 0.0169 | W1 0.0243 | largest step 0.22
# posterior sample MSE 0.0333 | W1 0.0010 | largest step 1.00
# prior sample     MSE 0.2485 | W1 0.0076 | largest step 1.00
```

Three estimators are compared over 2,000 random test signals. The columns are the MSE to the clean signal; a crude "realism" score, the 1-D Wasserstein distance W1 between the distribution of output pixel values and that of real pixel values (real signals contain only 0s and 1s, so in-between gray values are unrealistic); and the height of the sharpest step in the output (1.0 for every real signal). All numbers in this chapter's code blocks come from our own runs.

- The **posterior mean** has the lowest MSE, but its sharpest step averages 0.22: it is a ramp, not an edge, and its gray values make it statistically unlike any real signal.
- A **posterior sample** is a real edge (step 1.00, W1 near zero) placed where the data allow. Its MSE is 0.0333, almost exactly **twice** the posterior mean's 0.0169. That factor of two is not a coincidence (next section).
- A **prior sample** ignores $y$. It is perfectly realistic and useless: its MSE is about 15 times worse. Realism alone is not restoration. (Its W1 of 0.0076 instead of 0 is sampling noise of the 2,000-signal test.)

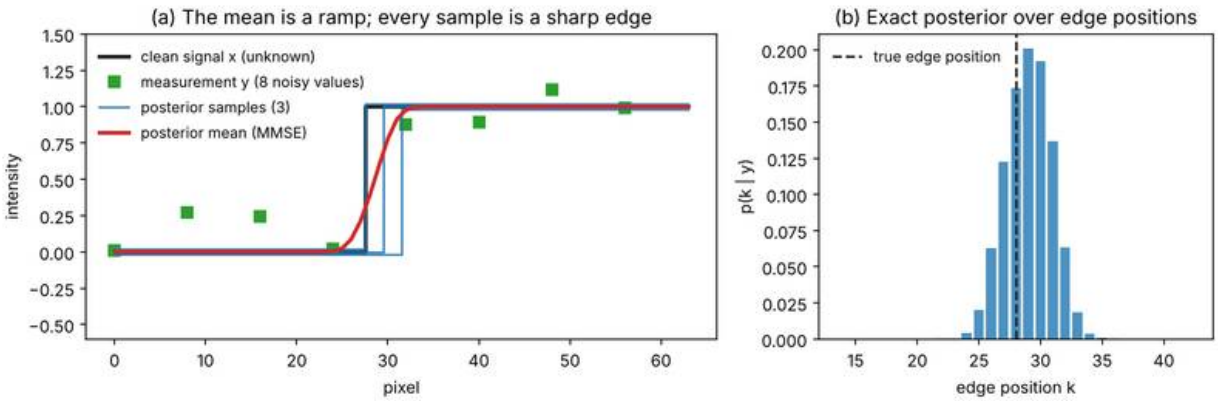



*Figure 1 — The step-edge toy for one input. (a) The posterior mean (red) is a smooth ramp; each posterior sample (blue) is a sharp edge, but they disagree about where. (b) The exact posterior over the edge position: the measurement narrows the position to a few pixels but cannot pin it down.*

### What "generative" restoration means

**Plain version.** A generative restorer selects a plausible answer. An ideal posterior sampler would reproduce the input's conditional uncertainty across repeated draws.

**Precise version.** A generative (or stochastic, or posterior-sampling) restorer is a random map: given $y$ and a random seed $z$, it returns $\hat{x} = G(y, z)$ whose distribution, for fixed $y$, is meant to approximate $p(x \mid y)$. Three ingredients appear in every method of this chapter:

1. **A prior** $p(x)$ of natural images, learned by a generative model (a GAN, a diffusion model, a flow, a codebook).
2. **A likelihood or data-consistency term** that ties the output to the input, either explicitly ($\|y - \mathcal{D}(\hat{x})\|^2$ with a known $\mathcal{D}$) or implicitly (a network that is shown $y$ and trained on pairs).
3. **A way to produce samples**: one forward pass of a generator, or many steps of a sampler.

A convenient identity connects the three. Taking the gradient of the logarithm of Bayes' rule with respect to $x$ removes the constant $p(y)$:

$$\nabla_x \log p(x \mid y) = \nabla_x \log p(x) + \nabla_x \log p(y \mid x),$$

where $\nabla_x \log p(x)$ is the **score** of the prior (a vector field pointing toward more likely images) and $\nabla_x \log p(y \mid x)$ is the gradient of the likelihood (pointing toward images that explain $y$ better). Every sampler that follows "uphill directions plus noise" can therefore be split into a *prior push* and a *data push*. This split is the backbone of zero-shot diffusion restoration below.

There are two broad ways to build such a restorer:

- **Learn the posterior directly** (supervised): train a conditional generator on pairs $(y, x)$, as conditional GANs and conditional diffusion models do. $\mathcal{D}$ never has to be written down, but a new degradation needs new training.
- **Combine a prior with a known likelihood** (zero-shot): train an *unconditional* generative model on clean images once, then steer its sampler with $\mathcal{D}$ and $y$ at test time, as GAN inversion and DPS-type methods do. One model serves many tasks, but $\mathcal{D}$ must be known and sampling is usually slower.

### The perception–distortion trade-off

**Plain version.** Realistic samples commit to uncertain detail. Under the ideal exact-posterior assumptions, sampling has twice the optimal mean estimator's expected MSE; learned generators have no such guarantee.

**Precise version.** Blau and Michaeli [1] define, for any restoration method that outputs $\hat{X}$ from $Y$, its **distortion** $\mathbb{E}[\Delta(X, \hat{X})]$, where $\Delta$ is any per-image distortion measure such as squared error, and its **perceptual quality** as a divergence $d(p_X, p_{\hat{X}})$ between the distribution of real images $p_X$ and the distribution of restored images $p_{\hat{X}}$ (zero means the outputs are statistically indistinguishable from real images, for example by a perfect discriminator). The **perception–distortion function** is

$$P(D) = \min_{p_{\hat{X}|Y}} d(p_X, p_{\hat{X}}) \quad \text{subject to} \quad \mathbb{E}[\Delta(X, \hat{X})] \le D,$$

where the minimum is over all, possibly random, restoration methods $p_{\hat{X}|Y}$ and $D$ is the distortion budget. They prove that $P(D)$ is non-increasing, and convex when $d$ is convex in its second argument [1]. So there is an **unattainable region** near the origin of the (distortion, perception) plane: lowering distortion past a point forces outputs away from the real-image distribution. They also show that this holds for *any* distortion measure, not only for MSE.

Two points on the curve matter most. Let $D_{\min}$ be the lowest possible distortion (achieved by the MMSE estimator for squared error) and $D_{\max}$ the lowest distortion achievable with perfect perceptual quality ($p_{\hat{X}} = p_X$). For squared error the paper proves (in its extended arXiv version [1])

$$D_{\max} \le 2\, D_{\min},$$

and the bound is reached by **posterior sampling**, $p_{\hat{X}|Y} = p_{X|Y}$. In PSNR terms, perfect realism never needs to cost more than about 3 dB, because $10 \log_{10} 2 \approx 3.01$. This is exactly the factor of two in our toy (0.0333 versus 0.0169). The paper also notes that the bound need not be tight: some problems allow perfect realism for less than twice the MMSE.

*Online only: Why posterior sampling exactly doubles the MSE — https://blog.shyandram.dev/en/learn/deep-learning/generative-restoration/notes/why-posterior-sampling-exactly-doubles-the-mse/*

Figure 2 places the toy's estimators on the plane. Between the posterior mean and the posterior sample we also plot simple blends $(1-\lambda)\,\hat{x}_{\mathrm{MMSE}} + \lambda\,\hat{x}_{\mathrm{sample}}$, a simple way to trade the two. Their expected MSE is $(1+\lambda^2)D_{\min}$, because a posterior sample's deviation from the mean is uncorrelated with the truth's deviation, so $\lambda = 1$ gives the factor of two. The blends are one achievable path; the true bound $P(D)$ may lie below it.

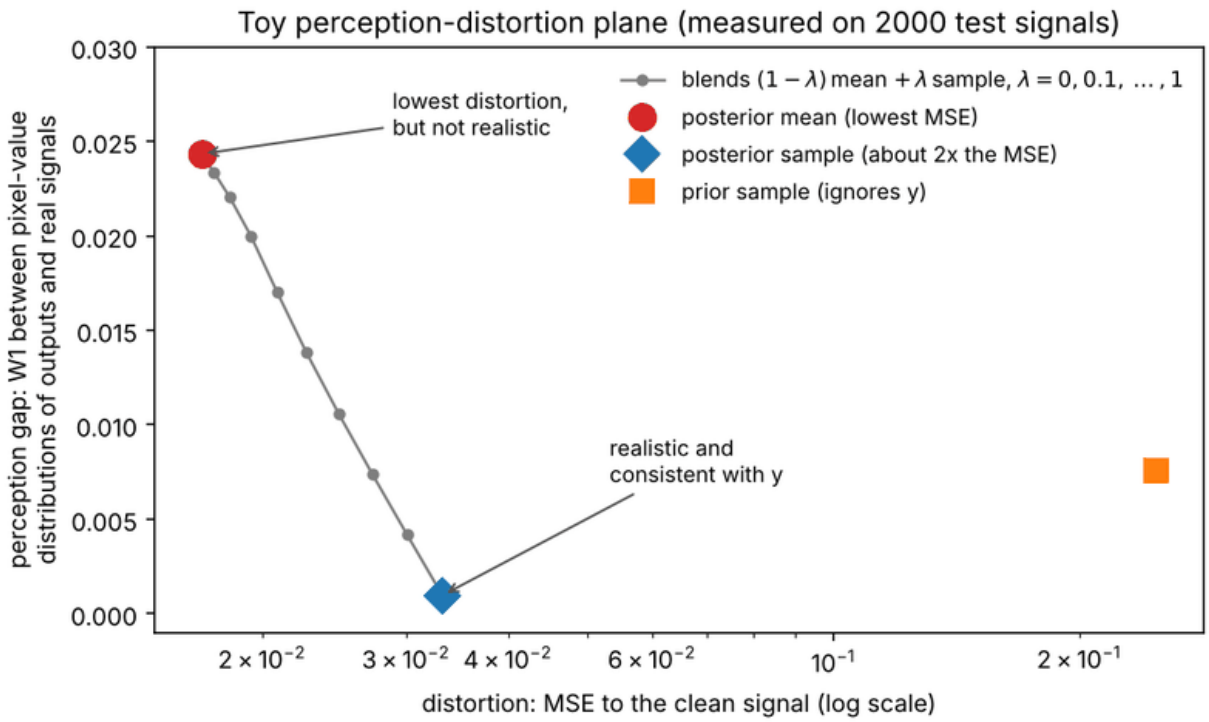



*Figure 2 — The perception–distortion plane for the step-edge toy, measured on 2,000 test signals. Moving from the posterior mean to a posterior sample lowers the perception gap to near zero and doubles the MSE. A prior sample is also realistic but ignores the input, so it is far to the right.*

What this means in practice:

- **A method is a point, not a score.** Report a distortion metric (PSNR, SSIM) *and* a perceptual one, and say which point your method aims for. Comparing a GAN restorer with an MSE restorer by PSNR alone is comparing two different targets.
- **"Better PSNR and better realism" is possible only for methods away from the bound.** Once a method is near the curve, it can only move along it.
- **Many generative restorers expose a knob** that slides along the curve: a GAN loss weight, a fidelity weight in CodeFormer [10], a feature-wrapping scalar in StableSR [24], a guidance scale in DiffBIR [25]. Choosing the knob is choosing a point on the trade-off.

*Online only: Blau and Michaeli, "The Perception-Distortion Tradeoff" (CVPR 2018) — https://blog.*

*shyandram.dev/en/learn/deep-learning/generative-restoration/papers/blau-and-michaeli-the-perception-distortion-tradeoff-cvpr-2018/*

### Family 1: GAN-based restoration and GAN priors

**Plain version.** A GAN adds a critic that has seen many real photos and complains whenever the restored image looks fake. The restorer learns to produce outputs the critic accepts, which pushes it off the blurry average and onto realistic texture. A second idea reuses a GAN that was already trained to *generate* faces or scenes: restoration becomes "find the image this generator can make that matches the damaged input".

#### Adversarial and perceptual losses

The first route keeps supervised training and changes the loss (DLCV 09 has the formulas). SRGAN [2] combined an adversarial loss with a loss on pretrained VGG features, following the perceptual-loss idea of Johnson et al. [3]; ESRGAN [4] improved the generator and discriminator; Real-ESRGAN [5] trained such a model on a rich synthetic degradation pipeline for real photos (DLCV 09A covers SRGAN and ESRGAN; blind and real-world SR covers the real-world degradation pipelines). In this chapter's terms, the total loss

$$\mathcal{L} = \mathcal{L}_{\text{pixel}} + \lambda_p \, \mathcal{L}_{\text{feat}} + \lambda_a \, \mathcal{L}_{\text{adv}}$$

has a pixel term $\mathcal{L}_{\text{pixel}}$ (separable L1 targets coordinate-wise posterior medians; squared L2 targets the posterior mean), a feature term $\mathcal{L}_{\text{feat}}$ (matching deep features, which tolerates small shifts), and an adversarial term $\mathcal{L}_{\text{adv}}$ (pulling the output distribution toward real images), with weights $\lambda_p, \lambda_a \geq 0$. Blau and Michaeli point out that the adversarial term approximates a divergence $d(p_X, p_{\hat{X}})$, so minimizing this loss is a Lagrangian form of the perception–distortion problem, and changing $\lambda_a$ moves along the curve [1].

Two caveats. Most such networks are **deterministic**, so they give *one* plausible answer, not samples with the right probabilities. And GAN training can **mode-collapse** (ML 07), while the adversarial term rewards any texture the discriminator likes, including textures that were never in the scene.

#### Pretrained GANs as image priors

The second route separates "what images look like" from "what this input says". A GAN generator $G$ trained on clean images, for example a StyleGAN [6] trained on faces, maps a latent code $w$ to an image $G(w)$. Its outputs form a learned **image manifold**: restoration can search on that manifold for an image consistent with $y$.

**GAN inversion by optimization.** Solve

$$\hat{w} = \arg\min_{w} \left\| \mathcal{D}(G(w)) - y \right\|^2 + \lambda \, R(w), \qquad \hat{x} = G(\hat{w}),$$

where $R$ keeps $w$ in the region the generator was trained on and $\lambda$ weights it. Nothing is trained per task, and $\mathcal{D}$ can be any differentiable degradation. PULSE [7] does this for face super-resolution,

searching a StyleGAN's latent space for faces that downscale to the input; different random starts give different faces that all fit, a direct picture of "many answers".

**Generators inside a feed-forward network.** To avoid one optimization per image, GLEAN [8] and GFP-GAN [9] put a pretrained, frozen GAN inside an encoder–decoder: the encoder reads the damaged image, the generator supplies realistic features, and one forward pass gives the result. CodeFormer [10] uses a learned discrete codebook of face parts instead of a GAN latent space and has a test-time knob that trades fidelity for quality.

*Online only: GAN-prior restorers in more detail — https://blog.shyandram.dev/en/learn/deep-learning/generative-restoration/notes/gan-prior-restorers-in-more-detail/*

**Strengths and limits.** GAN priors give sharp, realistic results in one pass on narrow domains with good generators, such as aligned faces. But they can only produce what the generator can produce: an input unlike the training faces is pulled toward a "typical" face, which is a fidelity and fairness problem, and the PULSE authors discuss such biases explicitly [7]. A diffusion-restoration survey [44] names unstable training and fake-looking textures as the two main weaknesses of GAN-based restoration, one motivation for the diffusion priors that follow. The survey of Xia et al. [47] reviews GAN inversion and its use for restoration and editing.

### Family 2: diffusion models as priors

**Plain version.** A diffusion model is a denoiser trained at every noise level. Starting from pure noise and denoising a little at a time turns noise into a realistic image. For restoration, we either *teach the denoiser to look at the damaged input* while it denoises (conditional diffusion), or we take a denoiser that only knows clean images and, after each small denoising step, *nudge* the current guess so that it agrees with the damaged input (zero-shot posterior sampling).

### Diffusion in one page

The full derivation is in ML 08; we need only four facts, in the variance-preserving DDPM convention [11].

**Forward noising.** For a clean image $x_0$ and steps $t = 1, \dots, T$,

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1 - \bar{\alpha}_t}\, \epsilon, \qquad \epsilon \sim \mathcal{N}(0, I),$$

where $\bar{\alpha}_t \in (0, 1)$ decreases from about 1 (almost clean) to about 0 (pure noise) along a chosen schedule and $\epsilon$ is standard Gaussian noise.

**A denoiser is a score estimator.** A network trained to predict $\epsilon$ from $x_t$ gives the score of the noisy distribution $p_t$: $\nabla_{x_t} \log p_t(x_t) \approx -\epsilon_\theta(x_t, t)/\sqrt{1 - \bar{\alpha}_t}$ (ML 08) [12].

**Tweedie's formula** [13] turns the score into a clean-image estimate,

$$\hat{x}_0(x_t) = \mathbb{E}[x_0 \mid x_t] = \frac{x_t + (1 - \bar{\alpha}_t)\, \nabla_{x_t} \log p_t(x_t)}{\sqrt{\bar{\alpha}_t}},$$

where $\hat{x}_0$ is the *posterior mean* of the clean image given the noisy one, so it is blurry when $t$ is large, exactly as in the first section.

**Sampling.** Start from $x_T \sim \mathcal{N}(0, I)$ and repeatedly draw $x_{t-1}$ from a Gaussian centered between $x_t$ and $\hat{x}_0(x_t)$ with a small, known variance (ML 08). Each step costs one network call; classic samplers use hundreds. Annealed Langevin dynamics with score networks [48] and the continuous-time SDE view [12] describe the same idea in other terms.

**Code: diffusion with an exact score.** Image-scale diffusion needs a trained network. On a 2-D toy we can skip the network: if the "image" distribution is a mixture of Gaussians, every noisy marginal $p_t$ is again a mixture of Gaussians, and its score has a closed form. The prior below has four blobs with weights 0.1, 0.2, 0.3, 0.4 and standard deviation 0.3.

```
import numpy as np

rng = np.random.default_rng(1)
mu = np.array([[-2.0, -2.0], [-2.0, 2.0], [2.0, -2.0], [2.0, 2.0]])  # a 2-D "image" prior:
w = np.array([0.1, 0.2, 0.3, 0.4])                                   # four Gaussian blobs
s2 = 0.3**2                                                          # each with variance s2
 ↪  * I

T = 200
beta = np.linspace(1e-4, 0.05, T)
alpha = 1 - beta
abar = np.cumprod(alpha)

def score(x, ab):
    """Exact grad log p_t(x) of the noised mixture, plus responsibilities r."""
    m, v = np.sqrt(ab) * mu, ab * s2 + (1 - ab)            # component k at time t: N(m_k, v
     ↪  I)
    d2 = ((x[:, None, :] - m[None]) ** 2).sum(-1)         # (N, K) squared distances
    logr = np.log(w) - d2 / (2 * v)
    r = np.exp(logr - logr.max(1, keepdims=True)); r /= r.sum(1, keepdims=True)
    return (r[:, :, None] * (m[None] - x[:, None, :])).sum(1) / v, r

def x0_hat(x, ab):                                          # Tweedie: E[x_0 | x_t]
    return (x + (1 - ab) * score(x, ab)[0]) / np.sqrt(ab)

def ddpm_step(x, t, x0):
    """Sample x_{t-1} from the Gaussian q(x_{t-1} | x_t, x_0 = x0)."""
    ab_prev = abar[t - 1] if t > 0 else 1.0
    mean = (np.sqrt(ab_prev) * beta[t] * x0 + np.sqrt(alpha[t]) * (1 - ab_prev) * x) / (1 -
     ↪  abar[t])
    var = beta[t] * (1 - ab_prev) / (1 - abar[t])
    return mean + np.sqrt(var) * rng.standard_normal(x.shape)

x = rng.standard_normal((4000, 2))                          # start from pure noise
for t in reversed(range(T)):
    x = ddpm_step(x, t, x0_hat(x, abar[t]))
_, r = score(x, 1.0)
print("signal left at t=T:", round(abar[-1], 5))
print("fraction per blob :", np.bincount(r.argmax(1), minlength=4) / len(x), "(target", w,
 ↪  ")")
```

```
print("spread inside blob:", round(float(np.sqrt(((x - mu[r.argmax(1)]) ** 2).mean())), 3),
 ↪  "(target 0.3)")
# signal left at t=T: 0.00612
# fraction per blob : [0.09675 0.2005  0.3075  0.39525] (target [0.1 0.2 0.3 0.4] )
# spread inside blob: 0.283 (target 0.3)
```

The sampler reproduces the blob weights to within sampling error. The spread inside each blob is 0.283 instead of 0.3: with only 200 coarse steps the reverse process is slightly too narrow (with 1,000 smaller steps we measured 0.296). This is the first lesson about **step count**: fewer steps are cheaper and less exact. Figure 3 shows the particle cloud contracting from noise to the four blobs.

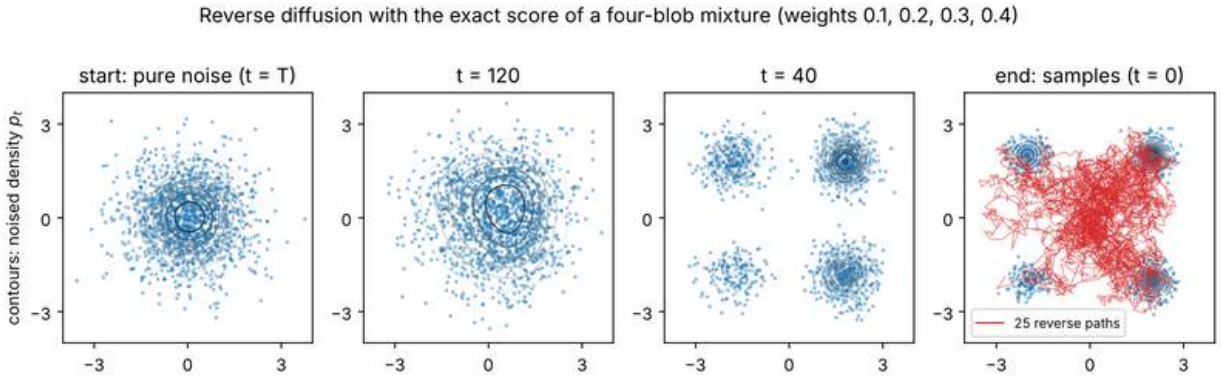



*Figure 3 — Reverse diffusion with the exact score of a four-blob prior. Gray contours show the noised density at each time; blue points are 1,500 samples; red lines (right) are 25 individual reverse paths. Paths are noisy because every step injects fresh noise.*

### Route A: train a conditional diffusion restorer

The direct approach lets the denoiser see the input: train $\epsilon_\theta(x_t, t, y)$ on pairs, so the sampler draws from an approximation of $p(x \mid y)$. **SR3** [14] did this for super-resolution: the upsampled low-resolution image is given to the U-Net with the noisy image, and sampling refines pure noise step by step. In the paper's human study on 8× face super-resolution, raters were fooled by SR3 close to 50% of the time, against at most 34% for GAN baselines [14]. **Palette** [15] applied the recipe to colorization, inpainting, uncropping and JPEG restoration, and showed that one multi-task model can match task-specific ones.

Conditional models need pairs and new training for each family of degradations, but they cope with noise and with operators that are hard to write down (camera processing, compression). Starting from pure noise wastes steps re-creating what $y$ already contains; **ResShift** [16] instead builds a chain that moves between the high- and low-resolution images by shifting their residual, and reports competitive results with only 15 sampling steps.

### Route B: zero-shot posterior sampling with a pretrained model

**Plain version.** Train one unconditional diffusion model on clean images. To restore, run its sampler as usual, but after every step check the current clean guess against the input and correct it a little. The model contributes "looks real"; the correction contributes "agrees with what we measured".

**Precise version.** Apply the score identity of the previous section to the noisy variable $x_t$:

$$\nabla_{x_t} \log p_t(x_t \mid y) = \nabla_{x_t} \log p_t(x_t) + \nabla_{x_t} \log p_t(y \mid x_t),$$

where the first term is the pretrained model's score and the second is the **noisy likelihood**: how well the noisy state $x_t$ explains $y$. The first term we have. The second is the hard part, because $y$ depends on the *clean* image, and

$$p_t(y \mid x_t) = \int p(y \mid x_0)\, p(x_0 \mid x_t)\, dx_0$$

averages the likelihood over all clean images that could have produced $x_t$, which is intractable for an image-scale model. The methods differ mainly in how they approximate this term, and they all end up with the same loop (Figure 4): a **prior step** (one reverse-diffusion step with the pretrained model) and a **data-consistency step** (a gradient step, a projection or a closed-form correction toward $\mathcal{D}(\hat{x}_0) \approx y$).

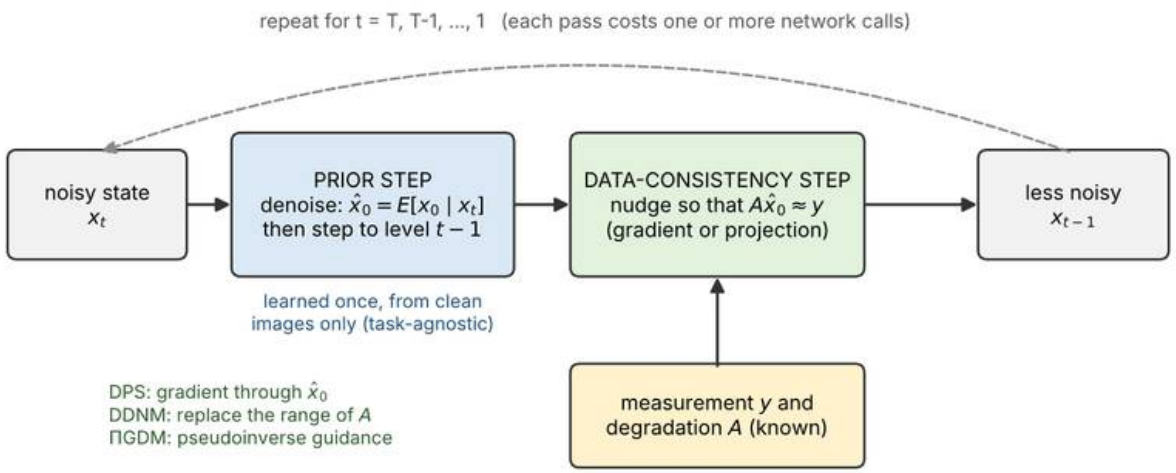



*Figure 4 — The loop behind zero-shot diffusion restoration. The prior step uses a model trained only on clean images; the data step uses the known degradation. Methods differ in exactly how, and where in the step, the data correction is applied.*

**DPS** (diffusion posterior sampling) [17] is the simplest to state. It replaces the average over $p(x_0 \mid x_t)$ by its mean, $p_t(y \mid x_t) \approx p(y \mid \hat{x}_0(x_t))$. With Gaussian noise this gives the data step

$$x_{t-1} \leftarrow x'_{t-1} - \zeta_t \nabla_{x_t} \| y - \mathcal{D}(\hat{x}_0(x_t)) \|^2,$$

where $x'_{t-1}$ is the result of the ordinary prior step and $\zeta_t > 0$ is a step size; the paper uses $\zeta_t = \zeta'/\|y - \mathcal{D}(\hat{x}_0)\|$ with a constant $\zeta'$. The gradient passes through $\hat{x}_0$, so it needs **backpropagation through the denoiser** at every step. Because it never projects exactly onto the measurements, DPS also works with noisy and nonlinear degradations (the paper includes Poisson noise, Fourier phase retrieval and non-uniform deblurring).

The other members differ mainly in the data step:

- **DDRM** (denoising diffusion restoration models) [18] handles linear degradations in the coordinates of the degradation's singular value decomposition, treating measured and unmeasured coordinates differently; no backpropagation, few steps.
- **DDNM** (denoising diffusion null-space model) [19] copies the part of $\hat{x}_0$ that the measurement determines from the data, $A^+y$ (with $A^+$ the pseudoinverse of a linear $A$), and keeps the model's guess only in the part the measurement cannot see.
- **ΠGDM** (pseudoinverse-guided diffusion models) [20] builds its guidance from a pseudoinverse of the degradation and an explicit uncertainty for $\hat{x}_0$; it backpropagates through the denoiser but not through the degradation, so it also covers non-differentiable measurements such as JPEG.
- **DiffPIR** [21] puts plug-and-play splitting (a data step and a denoiser step in turn, as described in DLCV 09) inside the diffusion sampler and reports good results with at most 100 network evaluations.

*Online only: How the zero-shot diffusion samplers differ — https://blog.shyandram.dev/en/learn/deep-learning/generative-restoration/notes/how-the-zero-shot-diffusion-samplers-differ/*

**Code: a DPS-style sampler where the true posterior is known.** We reuse the four-blob prior and observe one linear projection of the 2-D "image", $y = Ax + n$ with $A = [1, 1]/\sqrt{2}$ and noise standard deviation $\sigma_y = 0.2$. The blobs at $(-2, 2)$ and $(2, -2)$ both have $Ax \approx 0$, so a measurement near 0 is genuinely ambiguous. Prior and likelihood are Gaussian within each blob, so the exact posterior is again a four-blob mixture and we can check the sampler against it. The Jacobian of $\hat{x}_0$ is taken by finite differences, cheap in 2-D; at image scale this is the backpropagation step.

```
# Continues the previous snippet (same session): mu, w, s2, T, abar, x0_hat, ddpm_step.
rng = np.random.default_rng(3)
A = np.array([[1.0, 1.0]]) / np.sqrt(2)          # we only observe one projection of x
sig_y = 0.2
x_true = np.array([1.8, -2.1])
y = A @ x_true + sig_y * rng.standard_normal(1)

# Reference: the exact posterior of a Gaussian mixture under a linear-Gaussian measurement.
S = (A @ A.T)[0, 0] * s2 + sig_y**2               # variance of y within one blob
logpost = np.log(w) - (y - mu @ A.T)[:, 0] ** 2 / (2 * S)
post_w = np.exp(logpost - logpost.max()); post_w /= post_w.sum()
gain = s2 * A.T / S                               # Kalman gain, shape (2, 1)
post_mu = mu + (y - mu @ A.T) @ gain.T            # posterior mean inside each blob
post_cov = s2 * (np.eye(2) - gain @ A)
k = rng.choice(4, size=2000, p=post_w)
exact = post_mu[k] + rng.multivariate_normal(np.zeros(2), post_cov, size=2000)
mmse = post_w @ post_mu                           # posterior mean
```

```
least_sq = (np.linalg.pinv(A) @ y).ravel()        # minimum-norm solution of A x = y

def data_grad(x, ab, h=1e-4):
    """grad_x ||y - A x0_hat(x)||^2 through Tweedie's estimate (finite-difference
     ↪  Jacobian)."""
    J = np.stack([(x0_hat(x + h * e, ab) - x0_hat(x - h * e, ab)) / (2 * h) for e in
     ↪  np.eye(2)], -1)
    resid = y - x0_hat(x, ab) @ A.T                # (N, 1)
    return -2 * np.einsum("nij,ik,nk->nj", J, A.T, resid), np.abs(resid)

x = rng.standard_normal((2000, 2))
for t in reversed(range(T)):
    x_prior = ddpm_step(x, t, x0_hat(x, abar[t]))  # 1) prior step: one reverse-diffusion
     ↪  step
    g, res = data_grad(x, abar[t])
    x = x_prior - 0.1 / (res + 1e-8) * g           # 2) data-consistency step, DPS step size

blob = lambda pts: np.bincount(np.argmin(((pts[:, None] - mu) ** 2).sum(-1), 1), minlength=4)
 ↪  / len(pts)
print("least squares", least_sq.round(2), "| posterior mean", mmse.round(2), "| truth",
 ↪  x_true)
for name, pts in [("exact posterior", exact), ("DPS", x)]:
    print(f"{name:15s} blob shares {blob(pts)} | median |y - Ax| {np.median(np.abs(y - pts @
     ↪  A.T)):.3f}"
          f" | mean sq. error {np.mean(((pts - x_true) ** 2).sum(1)):.2f}")
for name, est in [("least squares", least_sq), ("posterior mean", mmse)]:
    print(f"{name:15s} sq. error {np.sum((est - x_true) ** 2):.2f}")
# least squares [0.14 0.14] | posterior mean [ 0.5 -0.3] | truth [ 1.8 -2.1]
# exact posterior blob shares [0.    0.409 0.591 0.   ] | median |y - Ax| 0.122 | mean sq.
 ↪  error 12.97
# DPS             blob shares [0.     0.4055 0.5945 0.    ] | median |y - Ax| 0.100 | mean
 ↪  sq. error 13.09
# least squares   sq. error 7.77
# posterior mean  sq. error 4.93
```

Read the output with Figure 5:

- The **least-squares** solution $A^{+}y$ and the **posterior mean** both lie *between* the two plausible blobs, where the prior is almost empty: the "blurry" answers of this toy. The posterior mean has the lowest squared error $(4.93)$ because it hedges.
- The **DPS-style samples** fall in the two right blobs with shares $0.41$ and $0.59$, matching the exact posterior. Their mean squared error $(13.09)$ is much higher, because about $40\%$ of them land in the other blob: a confident, wrong answer. That is posterior sampling working as intended, and also the 2-D version of a hallucinated detail.
- DPS fits the measurement slightly *more* tightly than the true posterior (median residual $0.100$ versus $0.122$): it is an approximation, and its residual depends on the step size $\zeta'$ (Exercise 3). In high dimensions such deviations are much harder to see.

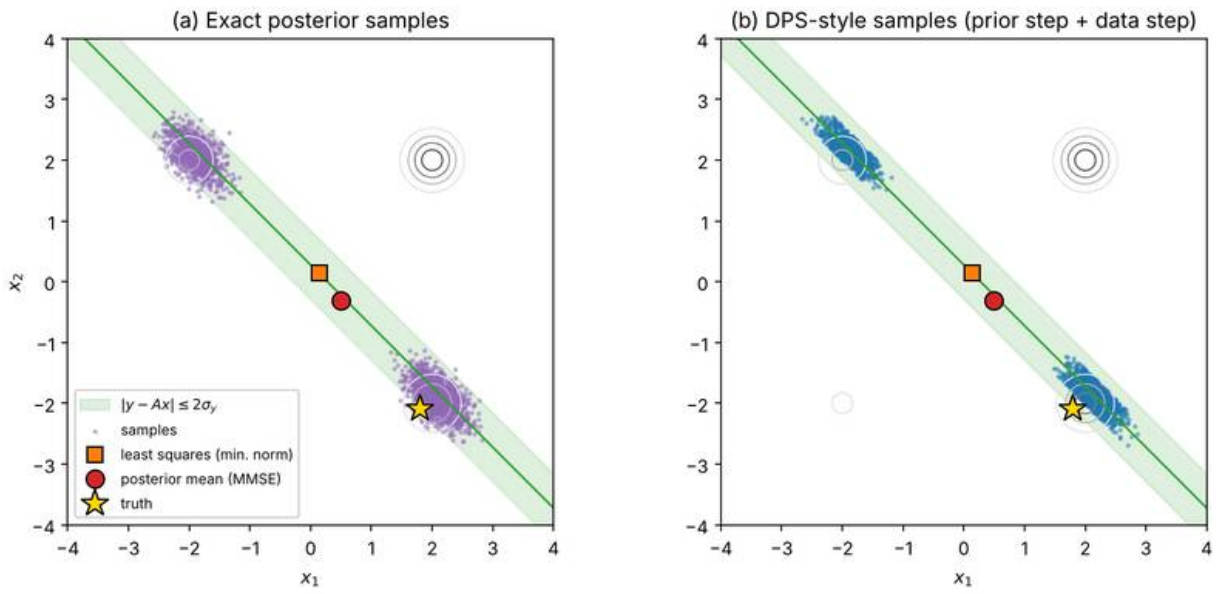



*Figure 5 — A linear inverse problem with a known posterior. (a) Exact posterior samples; (b) DPS-style samples from the loop of Figure 4. Both land in the two blobs the measurement allows, while least squares and the posterior mean land in the empty space between them.*

*Online only: Chung et al., "Diffusion Posterior Sampling for General Noisy Inverse Problems" (ICLR 2023) — https://blog.shyandram.dev/en/learn/deep-learning/generative-restoration/papers/chung-et-al-diffusion-posterior-sampling-for-general-noisy-inverse-problems-iclr-2023/*

### Choosing between conditional and zero-shot

| Question | Conditional (SR3, Palette, ResShift) | Zero-shot (DPS, DDRM, DDNM, ΠGDM, DiffPIR) |
|---|---|---|
| Training | One model per family of degradations, on pairs | One unconditional model on clean images |
| Degradation at test time | Implicit; can be unknown if it was in training | Must be known (or estimated) and usually differentiable |
| Noise and real camera pipelines | Handled if they were in training | Handled only as far as the likelihood model allows |
| Cost per image | Tens to hundreds of steps, no backpropagation | Often hundreds of steps; DPS-type methods also backpropagate |
| Good fit | One task with plenty of training pairs | Known physics, new operators, no retraining |

A rough rule: if you can collect or synthesize pairs for exactly your degradation, train a conditional model (or a non-generative one). If the operator is known physics (a blur kernel, a sampling mask, a tomography projector) and retraining for every setting is impractical, zero-shot sampling is attractive.

### Family 3: latent-diffusion and foundation-model restorers

**Plain version.** The strongest image generators are large text-to-image diffusion models trained on huge image collections. They know what grass, fabric and brick look like far better than any restoration dataset could teach. Recent real-world restorers keep such a model frozen (or almost frozen), add a small network that shows it the damaged image, and let it paint in the missing detail. The result can look stunning, and it can invent things.

#### Building on a pretrained text-to-image model

Latent diffusion [22] runs diffusion in the compressed latent space of an autoencoder instead of on pixels, which is what makes large text-to-image models affordable (ML 08). ControlNet [23] showed how to add a new image condition to such a model, which its authors describe as pretrained with billions of images, without damaging it: the pretrained model is locked, a trainable copy of its encoding layers receives the condition, and the two are joined by zero-initialized convolutions, so training starts from the unchanged model. Restoration methods adapt both ideas. **StableSR** [24] keeps the generator fixed and injects the low-quality image through a time-aware encoder, with a user-set scalar that trades quality for fidelity. **DiffBIR** [25] first removes the degradation with an ordinary restoration network and then regenerates lost detail with a ControlNet-style adapter, again with a tunable fidelity guidance. **SUPIR** [26] scales up the prior and the data (the paper describes 20 million high-resolution images with text annotations) and lets text prompts steer the restoration.

#### Fewer steps: from hundreds to one

Multi-step latent restorers are slow, and their random starting noise makes outputs vary from run to run. Distilled and re-designed models reduce both: **SinSR** [27], **OSEDiff** [28] and **TSD-SR** [30] restore in one step, and **InvSR** [29] in one to five, mostly by starting from the low-quality image instead of pure noise and by distilling a many-step teacher. The general distillation families (progressive, consistency, adversarial) are described in ML 08.

*Online only: Latent and one-step restorers, method by method — https://blog.shyandram.dev/en/learn/deep-learning/generative-restoration/notes/latent-and-one-step-restorers-method-by-method/*

Two consequences. First, a one-step restorer that starts from the input is nearly deterministic: it gives *one* plausible answer, like a GAN, rather than samples with the right spread. Second, step count is not the whole cost: a one-step model may have a larger network, and latent methods also pay for the autoencoder's encoder and decoder at full resolution.

#### The hallucination problem

**Plain version.** A strong prior fills gaps with what is *typical*, not with what was *there*. Where the input says little, the output is mostly prior: it looks real, and nothing in the image tells you it was invented. Where it shows up:

- **Text and symbols.** Letters on signs, plates and documents are very specific: a small error changes the meaning. A generative restorer can produce clean, legible, wrong characters.
- **Faces and identity.** Face priors produce *a* face from the training distribution; for a specific person, a slightly different nose or eye is an identity change.
- **Textures.** Brick, fabric and foliage may be replaced by a different but equally plausible pattern.
- **Latent decoders.** The autoencoder of a latent restorer has its own reconstruction error; encode and decode a clean image to see what it alone changes, as ML 08 recommends.
- **Medical and scientific images.** A plausible structure that is not there, or a real lesion smoothed away, is a diagnostic error. Bhadra et al. [35] analyze this in tomographic reconstruction by splitting an estimate into the part determined by the measurements and a **null-space** part supplied by the prior, and define a hallucination map from it.

The null-space view explains the mechanism in one line, using the decomposition behind DDNM: for a linear $A$, any estimate $\hat{x}$ satisfies $\hat{x} = A^{+}A\hat{x} + (I - A^{+}A)\hat{x}$. Data consistency constrains only the first part. Everything in the second part, which is large when $A$ discards a lot, comes from the prior, so **a perfectly data-consistent output can still be largely invented**. Consistency with $y$ is necessary, never sufficient.

Measuring hallucination is itself a research topic. A recent study of generative super-resolution argues that hallucinated details are poorly captured both by fidelity metrics and by no-reference quality models, and proposes a "Hallucination Score" obtained by prompting a multimodal large language model, which its authors found to align closely with human judgments [36].

### Family 4: flow matching and rectified flow restorers

**Plain version.** Flow matching learns a smooth "velocity field" that carries noise to images along nearly straight paths, so it needs few steps. For restoration, one can carry a *blurry best guess* to a realistic image, or plug a pretrained flow into the same prior-plus-data loop as diffusion.

Flow matching [31] and rectified flow [32] train a network to predict the velocity of a path between a simple distribution and the data distribution; samples come from integrating an ordinary differential equation, and straight paths need few integration steps (ML 08). Two verified restoration uses:

- **PMRF** (posterior-mean rectified flow) [33] targets the point $D_{\max}$ of the perception–distortion section directly: the estimator with the lowest MSE among those whose outputs are distributed like real images. Following a theoretical result the paper cites, it first predicts the **posterior mean** (an ordinary MMSE network) and then **transports** that blurry estimate to a realistic image with a rectified flow model that approximates the optimal transport map. This is a clean way to think about the whole chapter: first get the best average, then move it as little as possible to make it look real.
- **PnP-Flow** [34] defines a time-dependent denoiser from a pretrained flow-matching model and alternates gradient steps on the data-fidelity term, reprojections onto the learned flow path, and denoising: the loop of Figure 4 with a flow instead of a diffusion model, avoiding backpropagation through the ODE.

### Evaluating generative restoration

**Plain version.** Ask four separate questions: is the output close to the truth (when you have it)? Does it look real? Does it agree with the input? And how much does it change when you run it again? One number cannot answer all four, and a pretty picture answers none of them.

The metric definitions are in DLCV 09. For a generative restorer, use them as follows.

1. **Fidelity, when a reference exists.** PSNR and SSIM [39] (low distortion), LPIPS [38] (perceptual distance to the reference). An exact, conditionally independent posterior sample has twice the MMSE estimator's expected squared error under the same data model [1]. This gives about a 3 dB gap only when PSNR is computed from those expected MSEs, not necessarily when per-image PSNR is averaged. A learned generator and a trained regression baseline need not satisfy these ideal assumptions. Measure their actual trade-off; a lower PSNR alone is not evidence of greater realism.
2. **Realism.** No-reference metrics such as NIQE [41], MUSIQ [42] and CLIP-IQA [43] score single images; FID [40] compares Gaussian approximations to feature distributions of output and real-image sets. It is a practical distributional proxy, not an exact distance between the full image distributions $p_X$ and $p_{\hat{X}}$, and it needs many images. No-reference metrics can be gamed by adding texture, so never use them alone.
3. **Consistency with the input.** Re-apply the degradation and compare: $\|y - \mathcal{D}(\hat{x})\|$ should be about the noise level. This needs no reference and catches outputs that ignore the input. It cannot catch null-space hallucinations (previous section).
4. **Stochasticity.** A sampler gives a distribution of outputs. Report it.

**Reporting several samples.** The snippet continues the step-edge toy: one test input, eight restorations per run, and three runs with different seeds, as three users of the same method would see.

```
# Recreate the step-edge toy: the intervening DPS example uses a different A.
import numpy as np
from scipy.ndimage import gaussian_filter1d
n, factor, sigma = 64, 8, 0.2
positions = np.arange(8, 56)
X = (np.arange(n)[None, :] >= positions[:, None]).astype(float)
A = lambda x: gaussian_filter1d(x, 3.0, axis=-1, mode="nearest")[..., ::factor]
AX = A(X)

def posterior(y):
    logp = -np.sum((y - AX)**2, axis=1) / (2 * sigma**2)
    p = np.exp(logp - logp.max())
    return p / p.sum()

rng = np.random.default_rng(5)
x = X[20]                                                    # a test signal: edge at pixel 28
y = A(x) + sigma * rng.standard_normal(n // factor)
p = posterior(y)
for seed in range(3):                                        # three "users" with different
 ↪  seeds
    r = np.random.default_rng(seed)
```

```
    S = X[r.choice(len(positions), size=8, p=p)]           # 8 restorations of the same
     ↪ input
    err = ((S - x) ** 2).mean(axis=1)
    print(f"seed {seed}: MSE mean {err.mean():.4f} ± {err.std():.4f} | best of 8
     ↪ {err.min():.4f}"
          f" | edges at {sorted(set(np.argmax(S, axis=1).tolist()))}")
S = X[rng.choice(len(positions), size=200, p=p)]
std = S.std(axis=0)                                        # per-pixel spread across samples
print("pixels with std > 0.1:", np.flatnonzero(std > 0.1), "| true edge at", np.argmax(x))
# seed 0: MSE mean 0.0273 ± 0.0268 | best of 8 0.0000 | edges at [23, 24, 26, 27, 28, 29]
# seed 1: MSE mean 0.0273 ± 0.0103 | best of 8 0.0156 | edges at [25, 26, 27, 29, 30]
# seed 2: MSE mean 0.0293 ± 0.0213 | best of 8 0.0000 | edges at [24, 25, 26, 27, 28]
# pixels with std > 0.1: [22 23 24 25 26 27 28 29 30] | true edge at 28
```

Three lessons, each with a reporting rule:

- **The spread is as large as the mean.** On one input, the per-sample MSE varies by about as much as its average. Report the mean and the spread over samples (and over seeds), not one sample.
- **"Best of K" is an oracle.** Picking the sample closest to the ground truth gives an MSE of zero for two of the three seeds, a number no user can obtain because users do not have the ground truth. If a paper reports best-of-K, it must say so, and it should not be compared with single-sample results. (DPS, for example, reports the best of four samples for phase retrieval, following common practice for that problem, and says so [17].)
- **Sample spread is a diagnostic, not a calibrated error map.** The per-pixel standard deviation over 200 samples is large around the edge (pixels 22 to 30) in this exact-posterior toy. A learned sampler can instead repeat the same wrong detail, giving low spread despite high error; additional samples also cost computation. Angelopoulos et al. [37] calibrate image-valued prediction sets on held-out data: with a chosen confidence over calibration, they bound the expected fraction of uncovered pixels on fresh images from the same distribution. That is not a guarantee for every pixel of every individual image, nor a guarantee under arbitrary distribution shift.

**Compute.** Report the number of network evaluations (NFE), including guidance and any extra passes, the resolution, the hardware and wall-clock time, and for latent methods the autoencoder's cost. As a concrete anchor, the DPS paper reports about 95 seconds per 256 × 256 FFHQ image and about 600 seconds per ImageNet image, each with 1,000 network evaluations on one RTX 2080 Ti GPU [17]. Distilled one-step restorers cut the step count to one, but compare full pipelines, not step counts.

**A minimal honest report**: PSNR/SSIM and LPIPS on references; two or three no-reference metrics from different families, plus FID on a large enough set; the re-degradation residual; mean and spread over samples and seeds; NFE and time; a user study for headline claims; and a gallery that includes failures, not only the best-looking results.

### Failure modes and when NOT to use generative restoration

**Plain version.** Generative restoration is the right tool when the goal is a pleasant, plausible picture. It is the wrong tool when the picture will be used as evidence.

**Typical failure modes.**

- **Hallucinated specifics**: text, digits, faces, small objects, medical findings (see the hallucination section).
- **Ignoring the input**: weak data consistency, especially with strong guidance toward the prior or a prompt, gives realistic images of a slightly different scene; color shifts are a known issue of diffusion super-resolution [46].
- **Mismatch between assumed and real degradation**: zero-shot methods need the right $\mathcal{D}$. A wrong blur kernel or noise level is propagated by the data step, and the prior then makes the wrong answer look convincing.
- **Out-of-distribution inputs**: a prior trained on faces or natural photos pulls microscopy, satellite or medical images toward the wrong "typical" appearance, and faces unlike the training data toward the typical face (the bias discussed for PULSE [7]).
- **Instability across runs and sizes**: different seeds give different details, and models trained at a fixed size need special handling for larger images (StableSR's progressive aggregation sampling is one answer [24]).
- **Cost**: hundreds of network calls per image, or a very large one-step network.

**When not to use it**, or to use it only with explicit labeling and uncertainty:

- **Evidence and measurement**: forensic, legal, journalistic and surveillance use, license plates, documents to be read, anything where a person will act on a detail. Prefer an MMSE restorer (smooth, but it does not commit to specifics) or show a set of samples with an uncertainty map.
- **Medicine and science**: diagnosis, quantification and any downstream measurement. Use methods validated for the task, with hallucination analysis such as [35], and keep the measured data visible.
- **Mild degradations**: when the posterior is narrow (light noise, 2× super-resolution of a clean image), a regression network already gives a sharp answer, faster and with less risk.
- **Pipelines that feed another algorithm** (detection, OCR, measurement): restoration should be judged by the downstream task; invented detail can create false detections.

A short decision rule: if a wrong but realistic detail is worse than a blurry region, do not use a generative restorer without uncertainty reporting.

### Modern view

**Survey papers.**

- **Li et al., "Diffusion Models for Image Restoration and Enhancement: A Comprehensive Survey" (IJCV 2025)** [44]. Splits diffusion restoration into **supervised** methods (grouped by how they condition the sampler) and **zero-shot** methods (projection, decomposition and posterior estimation), adds blind and real-world restoration, and compares methods on several tasks. Takeaway: the best perceptual quality, with sampling cost, model size, real-world degradations and evaluation as the main obstacles.

- **Daras et al., "A Survey on Diffusion Models for Inverse Problems" (arXiv 2024)** [45]. Covers methods that use a *pretrained* diffusion model without further training, in four families by how they treat the measurement term: **explicit approximations** of $\nabla \log p_t(y \mid x_t)$ (DPS, ΠGDM and the DDRM family among them), **variational inference**, **CSGM-type** latent optimization and **asymptotically exact** Monte Carlo methods; a second taxonomy sorts them by how consistency is enforced (gradients, projections, sampling, optimization). Takeaway: many seemingly different methods compute a measurement error and "lift" it back to image space in different ways, and the field lacks a standard benchmark of quality versus compute.
- **Moser et al., "Diffusion Models, Image Super-Resolution, and Everything: A Survey" (IEEE TNNLS 2025)** [46]. Concentrates on super-resolution: foundations, input domains (pixel or latent), conditioning, guidance, corruption spaces and zero-shot methods. Takeaway: diffusion closed much of the gap to human preference, while computational cost, comparability between papers, explainability and color shifts are open.
- **Xia et al., "GAN Inversion: A Survey" (IEEE TPAMI 2023)** [47]. Reviews how images are mapped back into a pretrained GAN's latent space and how this supports restoration and editing. Takeaway: the GAN-prior methods of Family 1 are special cases of inversion, and their limits are the generator's coverage and the inversion's cost.

*Online only: Li et al., "Diffusion Models for Image Restoration and Enhancement: A Comprehensive Survey" (IJCV 2025) — https://blog.shyandram.dev/en/learn/deep-learning/generative-restoration/papers/li-et-al-diffusion-models-for-image-restoration-and-enhancement-a-comprehensive-survey-ijcv-2025/*

**State of the art, 2023–2026.**

1. **Pretrained text-to-image models became the default prior for real-world restoration**, connected through adapters (StableSR, DiffBIR, SUPIR), with explicit fidelity knobs because uncontrolled generation drifts from the input.
2. **Step counts fell from hundreds to one** through residual-shifting chains, distillation and inversion (ResShift, SinSR, OSEDiff, InvSR, TSD-SR). One-step models trade the posterior's spread for speed and repeatability.
3. **Zero-shot posterior sampling matured into a family** with shared structure (DPS, DDRM, DDNM, ΠGDM, DiffPIR) and was extended to flows (PnP-Flow), with surveys that unify the approximations [45].
4. **Theory re-entered practice**: the perception–distortion bound now guides method design directly, as in PMRF's "posterior mean, then transport" [33].
5. **Hallucination started to be measured**, not only mentioned, for example with dedicated scores [36].

**Open problems.**

- **Measuring faithfulness**: no accepted metric says "sharp *and* correct". Fidelity metrics penalize all detail; no-reference metrics reward invented detail.
- **True posterior sampling at image scale**: zero-shot methods are approximate (our toy shows a small bias even in 2-D), and few-step models give little diversity. Calibrated uncertainty for high-resolution images is largely open.

- **Unknown degradations**: zero-shot methods need $\mathcal{D}$; supervised ones need the right training distribution.
- **Cost**: high-resolution and video restoration with large priors remain expensive.
- **Responsible use**: domain shift, demographic bias of face priors and the labeling of generated content are unsolved in practice.

*Online only: DDNM: preserve measured components and generate the null space — https://blog.shyandram.dev/en/learn/deep-learning/generative-restoration/papers/ddnm-preserve-measured-components-and-generate-the-null-space/; Chung et al., "Diffusion Posterior Sampling for General Noisy Inverse Problems" (ICLR 2023) — https://blog.shyandram.dev/en/learn/deep-learning/generative-restoration/papers/chung-et-al-diffusion-posterior-sampling-for-general-noisy-inverse-problems-iclr-2023/; BlindDPS: infer an image and its degradation together — https://blog.shyandram.dev/en/learn/deep-learning/generative-restoration/papers/blinddps-infer-an-image-and-its-degradation-together/*

## Key takeaways

- The optimal squared-error estimate is the conditional mean; averaging ambiguous structures can blur them.
- A plausible generated detail may differ from the original scene.
- Perception–distortion bounds concern ideal distributions and estimators [1]; learned generators do not automatically satisfy them.
- Distinguish task-trained conditional models from pretrained priors coupled to measurement constraints.
- Input consistency cannot detect inventions in unobserved components; fewer sampling steps do not remove this ambiguity.
- Report fidelity, realism, consistency, seeds and full cost; never hide best-of-many selection.
- Choose conservative restoration when a realistic wrong detail would undermine the application.

## Exercises

1. **The factor of two, by hand.** In the step-edge toy, suppose that for one input the posterior puts probability 0.5 on an edge at pixel 30 and 0.5 on an edge at pixel 32, and the truth is at 30. Compute the MSE (per pixel, over 64 pixels) of the posterior mean and the *expected* MSE of a posterior sample. Is the ratio two? Explain why the factor of two in the theorem is an average over inputs, not a guarantee for each input.

   **Hint**

   The two candidates differ only at pixels 30 and 31. The mean is 0.5 there, so its MSE is $2 \times 0.25/64 \approx 0.0078$; a sample is right with probability 0.5 and wrong by 1 at two pixels otherwise, so its expected MSE is $0.5 \times 2/64 \approx 0.0156$: a ratio of two. If instead the posterior put 0.9 on the wrong position, the ratio for *this* truth would be $1.8/1.62 \approx 1.1$. The theorem averages over $x$ and $y$ jointly, where the truth itself is distributed like the posterior.

2. **Walking the trade-off.** Using the first snippet, compute MSE and W1 for (a) the posterior mean plus Gaussian noise of standard deviation 0.1, (b) the posterior *median* edge position (choose the position where the posterior's cumulative probability first exceeds 0.5), and (c) the posterior *mode*. Place them on Figure 2's plane. Which are near the posterior-sample point?

   **Hint**

   Adding noise does not create edges: in our run the largest step was only 0.40, W1 grew to about 0.095 (noise creates many unrealistic in-between values) and the MSE grew to 0.0274, so (a) moves *away* from both goals. The median and the mode are single real edges, so their W1 is near zero like a sample's (0.0009 and 0.0011 in our run), and their MSE (0.0245 and 0.0259) is *below* a random sample's (0.0338) because they never pick an unlikely position. They are deterministic, though: they do not reproduce the posterior's spread, which a set-level metric could detect (all outputs for similar inputs agree).

3. **DPS step size.** In the DPS-style snippet, replace the constant 0.1 in the data step by 0.02, 0.5 and 2.0. For each, report the blob shares, the median residual $|y - Ax|$ and the mean squared error, and compare with the exact posterior. Then replace the normalized step $0.1/(\text{res} + 10^{-8})$ $g$ by an unnormalized step $0.2\,g$. What do you conclude about how DPS's step size relates to the noise level $\sigma_y$?

   **Hint**

   With the normalized step, each data step has roughly the same length whatever the residual, so the final median residual settles near the constant itself (in our runs 0.021, 0.100, 0.496 and 1.994 for 0.02, 0.1, 0.5 and 2.0). Large values leave residuals far above the noise level and distort the blob shares (0.24 and 0.76 at 2.0, against 0.41 and 0.59 exactly); at 0.02 about 1% of the samples ended in a blob the measurement rules out, because the early corrections were too weak. The unnormalized step $0.2\,g$ gave a median residual of 0.007, far tighter than the noise level allows. Neither choice is tied to $\sigma_y$ the way an exact posterior is: the step size is a tuned hyperparameter, one reason DPS is an approximation.

4. **Steps and accuracy.** In the diffusion snippet, run the sampler with $T = 50$ (and `beta = np.linspace(1e-4, 0.2, T)`) and with $T = 1000$ (and `np.linspace(1e-4, 0.01, T)`). Record the spread inside each blob. Relate the result to the cost discussion and to why distilled one-step restorers are trained rather than obtained by simply using fewer steps.

   **Hint**

   In our runs the spread was about 0.263 for 50 steps, 0.283 for 200 and 0.296 for 1,000 (target 0.3): coarse steps make the reverse process too narrow even with the *exact* score. A one-step model cannot get its accuracy by taking one huge step of the same sampler; it must be trained (distilled or built differently) to jump correctly.

5. **Null-space hallucination.** Take a 1-D signal of length 16 and $A$ = average of consecutive pairs (an $8 \times 16$ matrix). Compute $P = A^{+}A$ and verify that $P$ is a projection ($P^2 = P$). Construct two different signals with exactly the same measurement $Ax$, one smooth and one with a sharp checkerboard pattern. Which one would a generative prior of natural signals pick, and which part of each signal is "invented"?

**Hint**

The null space of pair-averaging contains every pattern that alternates sign within a pair, $(+a, -a)$ with any amplitude $a$ per pair; adding it to a smooth signal leaves $Ax$ unchanged. The within-pair differences, $(I - P)x$, are invisible to the data and supplied entirely by the prior: a smooth-signal prior picks small ones, a texture prior may add some. This is the hallucination-map idea of [35].

6. **Design an evaluation.** You must decide whether a one-step latent-diffusion restorer can replace an MSE-trained network for restoring scanned archival documents. Write the evaluation protocol: data, metrics, number of samples and seeds, a check for hallucinated characters, compute measurements, and a decision rule.

**Hint**

Use real scans with references where possible (rescan at high resolution) plus synthetic degradations of clean documents. Report PSNR/SSIM/LPIPS, a no-reference metric, and, crucially, an OCR character error rate against the true text for both methods and for several seeds; also report how often two seeds disagree on a character (an uncertainty signal). Include NFE, time and memory including the autoencoder. Decide by the task: if the generative model raises character errors or changes characters between seeds, keep the MSE network for reading and use the generative one only for display, clearly labeled.

## References


1. Y. Blau and T. Michaeli, "The Perception-Distortion Tradeoff," in *Proc. IEEE/CVF CVPR*, 2018, pp. 6228–6237 (extended version on arXiv). doi ·arXiv
2. C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang and W. Shi, "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network," in *Proc. IEEE CVPR*, 2017, pp. 105–114. doi ·arXiv
3. J. Johnson, A. Alahi and L. Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution," in *Proc. ECCV*, 2016, pp. 694–711. doi ·arXiv
4. X. Wang, K. Yu, S. Wu, J. Gu, Y. Liu, C. Dong, Y. Qiao and C. C. Loy, "ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks," in *Proc. ECCV Workshops*, 2018 (LNCS, 2019), pp. 63–79. doi ·arXiv
5. X. Wang, L. Xie, C. Dong and Y. Shan, "Real-ESRGAN: Training Real-World Blind Super-Resolution with Pure Synthetic Data," in *Proc. IEEE/CVF ICCV Workshops*, 2021, pp. 1905–1914. doi ·arXiv
6. T. Karras, S. Laine and T. Aila, "A Style-Based Generator Architecture for Generative Adversarial Networks," in *Proc. IEEE/CVF CVPR*, 2019, pp. 4396–4405. doi ·arXiv
7. S. Menon, A. Damian, S. Hu, N. Ravi and C. Rudin, "PULSE: Self-Supervised Photo Upsampling via Latent Space Exploration of Generative Models," in *Proc. IEEE/CVF CVPR*, 2020, pp. 2434–2442. doi ·arXiv
8. K. C. K. Chan, X. Wang, X. Xu, J. Gu and C. C. Loy, "GLEAN: Generative Latent Bank for Large-Factor Image Super-Resolution," in *Proc. IEEE/CVF CVPR*, 2021, pp. 14240–14249. doi · arXiv

9. X. Wang, Y. Li, H. Zhang and Y. Shan, “Towards Real-World Blind Face Restoration with Generative Facial Prior,” in *Proc. IEEE/CVF CVPR*, 2021, pp. 9164–9174. doi ·arXiv
10. S. Zhou, K. C. K. Chan, C. Li and C. C. Loy, “Towards Robust Blind Face Restoration with Codebook Lookup Transformer,” in *Proc. NeurIPS*, vol. 35, 2022, pp. 30599–30611. doi ·arXiv
11. J. Ho, A. Jain and P. Abbeel, “Denoising Diffusion Probabilistic Models,” in *Proc. NeurIPS*, 2020. proceedings ·arXiv
12. Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon and B. Poole, “Score-Based Generative Modeling through Stochastic Differential Equations,” in *Proc. ICLR*, 2021. arXiv
13. B. Efron, “Tweedie’s Formula and Selection Bias,” *Journal of the American Statistical Association*, vol. 106, no. 496, pp. 1602–1614, 2011. doi
14. C. Saharia, J. Ho, W. Chan, T. Salimans, D. J. Fleet and M. Norouzi, “Image Super-Resolution via Iterative Refinement,” *IEEE TPAMI*, 2022 (early access). doi ·arXiv
15. C. Saharia, W. Chan, H. Chang, C. Lee, J. Ho, T. Salimans, D. Fleet and M. Norouzi, “Palette: Image-to-Image Diffusion Models,” in *Proc. ACM SIGGRAPH Conference*, 2022, pp. 1–10. doi · arXiv
16. Z. Yue, J. Wang and C. C. Loy, “ResShift: Efficient Diffusion Model for Image Super-resolution by Residual Shifting,” in *Proc. NeurIPS*, vol. 36, 2023, pp. 13294–13307. doi ·arXiv
17. H. Chung, J. Kim, M. T. McCann, M. L. Klasky and J. C. Ye, “Diffusion Posterior Sampling for General Noisy Inverse Problems,” in *Proc. ICLR*, 2023. arXiv
18. B. Kawar, M. Elad, S. Ermon and J. Song, “Denoising Diffusion Restoration Models,” in *Proc. NeurIPS*, vol. 35, 2022, pp. 23593–23606. doi ·arXiv
19. Y. Wang, J. Yu and J. Zhang, “Zero-Shot Image Restoration Using Denoising Diffusion Null-Space Model,” in *Proc. ICLR*, 2023. arXiv
20. J. Song, A. Vahdat, M. Mardani and J. Kautz, “Pseudoinverse-Guided Diffusion Models for Inverse Problems,” in *Proc. ICLR*, 2023. ICLR page
21. Y. Zhu, K. Zhang, J. Liang, J. Cao, B. Wen, R. Timofte and L. Van Gool, “Denoising Diffusion Models for Plug-and-Play Image Restoration,” in *Proc. IEEE/CVF CVPR Workshops*, 2023, pp. 1219–1229. doi ·arXiv
22. R. Rombach, A. Blattmann, D. Lorenz, P. Esser and B. Ommer, “High-Resolution Image Synthesis with Latent Diffusion Models,” in *Proc. IEEE/CVF CVPR*, 2022, pp. 10674–10685. doi · arXiv
23. L. Zhang, A. Rao and M. Agrawala, “Adding Conditional Control to Text-to-Image Diffusion Models,” in *Proc. IEEE/CVF ICCV*, 2023, pp. 3813–3824. doi ·arXiv
24. J. Wang, Z. Yue, S. Zhou, K. C. K. Chan and C. C. Loy, “Exploiting Diffusion Prior for Real-World Image Super-Resolution,” *International Journal of Computer Vision*, vol. 132, no. 12, pp. 5929–5949, 2024. doi ·arXiv
25. X. Lin, J. He, Z. Chen, Z. Lyu, B. Dai, F. Yu, Y. Qiao, W. Ouyang and C. Dong, “DiffBIR: Toward Blind Image Restoration with Generative Diffusion Prior,” in *Proc. ECCV*, 2024 (LNCS), pp. 430–448. doi ·arXiv
26. F. Yu, J. Gu, Z. Li, J. Hu, X. Kong, X. Wang, J. He, Y. Qiao and C. Dong, “Scaling Up to Excellence: Practicing Model Scaling for Photo-Realistic Image Restoration In the Wild,” in *Proc. IEEE/CVF CVPR*, 2024, pp. 25669–25680. doi ·arXiv
27. Y. Wang, W. Yang, X. Chen, Y. Wang, L. Guo, L.-P. Chau, Z. Liu, Y. Qiao, A. C. Kot and B. Wen, “SinSR: Diffusion-Based Image Super-Resolution in a Single Step,” in *Proc. IEEE/CVF CVPR*,

2024, pp. 25796–25805. doi ·arXiv
28. R. Wu, L. Sun, Z. Ma and L. Zhang, "One-Step Effective Diffusion Network for Real-World Image Super-Resolution," in *Proc. NeurIPS*, vol. 37, 2024, pp. 92529–92553. doi ·arXiv
29. Z. Yue, K. Liao and C. C. Loy, "Arbitrary-steps Image Super-resolution via Diffusion Inversion," in *Proc. IEEE/CVF CVPR*, 2025, pp. 23153–23163. doi ·arXiv
30. L. Dong, Q. Fan, Y. Guo, Z. Wang, Q. Zhang, J. Chen, Y. Luo and C. Zou, "TSD-SR: One-Step Diffusion with Target Score Distillation for Real-World Image Super-Resolution," in *Proc. IEEE/CVF CVPR*, 2025, pp. 23174–23184. doi ·arXiv
31. Y. Lipman, R. T. Q. Chen, H. Ben-Hamu, M. Nickel and M. Le, "Flow Matching for Generative Modeling," in *Proc. ICLR*, 2023. arXiv
32. X. Liu, C. Gong and Q. Liu, "Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow," in *Proc. ICLR*, 2023. arXiv
33. G. Ohayon, T. Michaeli and M. Elad, "Posterior-Mean Rectified Flow: Towards Minimum MSE Photo-Realistic Image Restoration," in *Proc. ICLR*, 2025. arXiv
34. S. Martin, A. Gagneux, P. Hagemann and G. Steidl, "PnP-Flow: Plug-and-Play Image Restoration with Flow Matching," in *Proc. ICLR*, 2025. arXiv
35. S. Bhadra, V. A. Kelkar, F. J. Brooks and M. A. Anastasio, "On Hallucinations in Tomographic Image Reconstruction," *IEEE Transactions on Medical Imaging*, vol. 40, no. 11, pp. 3249–3260, 2021. doi ·arXiv
36. W. Ren, R. Goyal, Z. Hu, T. T. Aumentado-Armstrong, I. Mohomed and A. Levinshtein, "Hallucination Score: Towards Mitigating Hallucinations in Generative Image Super-Resolution," arXiv:2507.14367, 2025. arXiv
37. A. N. Angelopoulos, A. P. Kohli, S. Bates, M. I. Jordan, J. Malik, T. Alshaabi, S. Upadhyayula and Y. Romano, "Image-to-Image Regression with Distribution-Free Uncertainty Quantification and Applications in Imaging," in *Proc. ICML*, PMLR 162, 2022. proceedings ·arXiv
38. R. Zhang, P. Isola, A. A. Efros, E. Shechtman and O. Wang, "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric," in *Proc. IEEE/CVF CVPR*, 2018, pp. 586–595. doi ·arXiv
39. Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. doi
40. M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler and S. Hochreiter, "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium," in *Proc. NeurIPS (NIPS)*, 2017. arXiv
41. A. Mittal, R. Soundararajan and A. C. Bovik, "Making a 'Completely Blind' Image Quality Analyzer," *IEEE Signal Processing Letters*, vol. 20, no. 3, pp. 209–212, 2013. doi
42. J. Ke, Q. Wang, Y. Wang, P. Milanfar and F. Yang, "MUSIQ: Multi-scale Image Quality Transformer," in *Proc. IEEE/CVF ICCV*, 2021, pp. 5128–5137. doi ·arXiv
43. J. Wang, K. C. K. Chan and C. C. Loy, "Exploring CLIP for Assessing the Look and Feel of Images," in *Proc. AAAI*, vol. 37, no. 2, 2023, pp. 2555–2563. doi ·arXiv
44. X. Li, Y. Ren, X. Jin, C. Lan, X. Wang, W. Zeng, X. Wang and Z. Chen, "Diffusion Models for Image Restoration and Enhancement: A Comprehensive Survey," *International Journal of Computer Vision*, vol. 133, no. 11, pp. 8078–8108, 2025. doi ·arXiv
45. G. Daras, H. Chung, C.-H. Lai, Y. Mitsufuji, J. C. Ye, P. Milanfar, A. G. Dimakis and M. Delbracio, "A Survey on Diffusion Models for Inverse Problems," arXiv:2410.00083, 2024. arXiv

46. B. B. Moser, A. S. Shanbhag, F. Raue, S. Frolov, S. Palacio and A. Dengel, “Diffusion Models, Image Super-Resolution, and Everything: A Survey,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 36, no. 7, pp. 11793–11813, 2025. doi ·arXiv
47. W. Xia, Y. Zhang, Y. Yang, J.-H. Xue, B. Zhou and M.-H. Yang, “GAN Inversion: A Survey,” *IEEE TPAMI*, vol. 45, no. 3, pp. 3121–3138, 2023. doi ·arXiv
48. Y. Song and S. Ermon, “Generative Modeling by Estimating Gradients of the Data Distribution,” in *Proc. NeurIPS*, 2019. arXiv
49. H. Chung, J. Kim, S. Kim and J. C. Ye, “Parallel Diffusion Models of Operator and Image for Blind Inverse Problems,” in *Proc. IEEE/CVF CVPR*, 2023, pp. 6059–6069. paper

# DLCV 09C • Low-Light Image Enhancement

From histogram equalization and Retinex to curve estimation, Retinex transformers and diffusion models: how to brighten dark photos without amplifying noise, shifting colors or fooling the metrics.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/

## What you'll learn

- Separate tone correction from noise, detail and color restoration.
- Compare histogram, gamma, Retinex and photographic-transform approaches.
- Classify learning methods by supervision and architecture independently.
- Choose evaluation protocols that expose exposure, color and noise failures.
- Measure full-resolution cost and assess suitability for downstream tasks.

## The big picture

A photo taken in the dark is not just a normal photo with the brightness turned down. The sensor collected very few photons, so the picture is dim, noisy, and often has a color cast. Low-light image enhancement aims to produce the image you *would* have gotten with good light: well exposed, with natural colors, clean details, and no new artifacts.

> **Explain like I'm five**
>
> Imagine you are drawing in a room with the lights almost off. Your drawing comes out faint and smudgy, and you are not sure which crayon was which color. Turning on the lights later and photocopying the drawing "brighter" won't help: the smudges get brighter too, and the wrong colors stay wrong. A good low-light enhancer is like a friend who looks at your faint drawing and *redraws* it as it would look in daylight: same picture, right colors, without the smudges.

Why it matters: phones shoot in the dark every day, cars and robots must see at night, and surveillance, astronomy-like scientific imaging and medical endoscopy all fight the same physics. Low light also hurts downstream models: a detector trained on daytime photos loses accuracy at night. That makes LLIE both a picture-quality task and a robustness task for computer vision.

> **About this tutorial**
>
> This tutorial is expanded from an earlier Medium post on LLIE [1], which is written in Traditional Chinese (read it here). It follows the post's outline and adds math, code, figures and a survey of the literature.

**Related chapters.** DLCV 09 supplies the restoration background. DLCV 10 combines complementary measurements from several images. Efficiency is discussed alongside every enhancement lineage here.

### Goal and difficulty

**Plain version.** Low-light enhancement combines tone adjustment with restoration of noisy detail and color. Brightening alone does not solve both.

**Precise version.** Unlike image-to-image translation, which changes an image's *attributes* (summer to winter, photo to painting), LLIE should keep the scene and its content exactly as they are and only undo the damage caused by insufficient light. The post [1] splits that damage into two difficulties:

1. **Tone enhancement.** Lift the exposure, globally and locally, without crushing highlights or flattening contrast. Bright regions such as lamps must stay bright, dark regions must open up, and the ordering of brightness between regions should be preserved.
2. **Detail restoration.** Recover texture and color that are buried in noise and quantization, while suppressing that noise rather than amplifying it.

In deep learning, detail restoration is what a network does anyway once it is trained to output a clean image, so most method design goes into the *tone* part (which image properties or physical model to build in) and the *learning mechanism* (which supervision and losses to use), rather than into a dedicated detail module.

### Why it is not simple brightening

A linear gain $y = c\,x$ with $c > 1$ fails in four ways, all visible in Figure 1:

- **Noise.** Photon arrival is random. With few photons, the signal-to-noise ratio is low, and a gain multiplies signal and noise equally. The image gets brighter, not cleaner.
- **Color cast.** Low light is often colored (tungsten, sodium street lamps), and the camera's white balance and the three color channels have different noise levels. Scaling the channels equally keeps the cast; scaling them separately may produce strange colors.
- **Saturation and over-exposure.** Pixels already near the top (lamps, windows, screens) clip at 1. A global gain big enough for the shadows destroys the highlights.
- **Quantization.** A dark 8-bit image may use only 20–40 distinct gray levels. Stretching them produces visible banding.

### A model of low-light capture

It helps to have an explicit degradation model, both for understanding and for **synthesizing training data**. Work in *linear* light (values proportional to photon counts), not on gamma-encoded pixels (see IP 03 on gamma). A widely used model is the Poisson–Gaussian model of Foi et al. [2]:

$$z = \frac{1}{\eta}\,\mathcal{P}(\eta\,k\,x) + n, \qquad n \sim \mathcal{N}(0, \sigma_r^2),$$

where $x \in [0, 1]$ is the clean linear image, $k \in (0, 1]$ the fraction of light that reached the sensor, $\eta$ a scale that converts intensity to photon counts (a larger $\eta$ means more photons and less relative

noise), $\mathcal{P}(\cdot)$ a Poisson random variable modeling **shot noise**, and $n$ Gaussian **read noise** with standard deviation $\sigma_r$. The camera then applies a tone curve and quantizes. The variance of $z$ is approximately $kx/\eta + \sigma_r^2$: it grows with the signal, so noise is *signal-dependent*, which is why "add Gaussian noise" is a poor stand-in for real low light.

```
import numpy as np
from skimage import data, img_as_float

rng = np.random.default_rng(0)
clean = img_as_float(data.coffee())            # H x W x 3, sRGB-encoded, in [0, 1]

def synth_low_light(x, k=0.05, peak=2000.0, read_std=8e-4, gamma=2.2):
    lin = k * x ** gamma                              # 1) less light, in linear units
    lin = rng.poisson(lin * peak) / peak              # 2) shot noise (Poisson)
    lin = lin + rng.normal(0, read_std, lin.shape) # 3) read noise (Gaussian)
    out = np.clip(lin, 0, 1) ** (1 / gamma)           # 4) re-encode for display
    return np.round(out * 255) / 255                  # 5) 8-bit quantization

dark = synth_low_light(clean)
print(clean.mean(), dark.mean())                      # about 0.39 -> 0.10
```

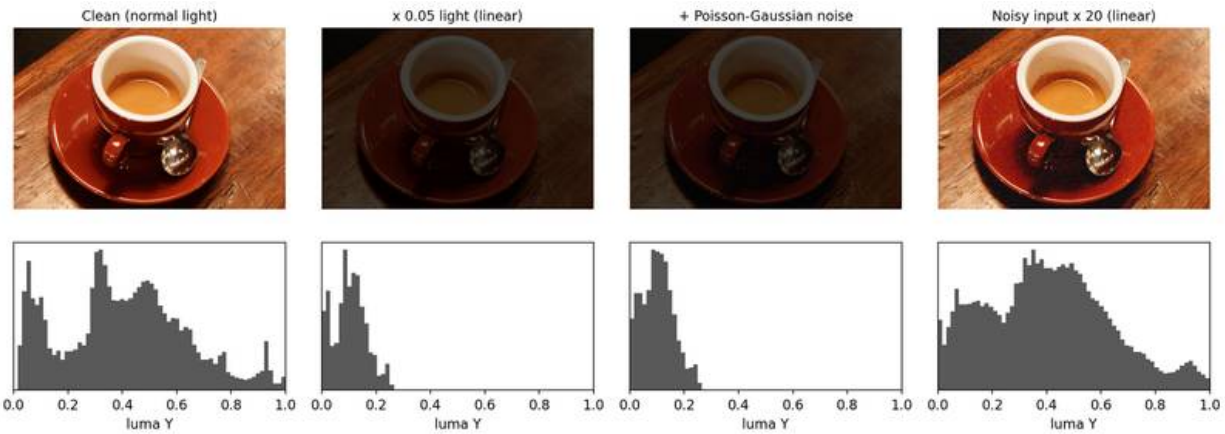



*Figure 1 — Synthetic low light. Removing 95% of the light squeezes the histogram into the bottom quarter. Noise is hard to see in the dark image, but undoing the exposure with a perfect gain (right) reveals it: brightening and denoising are separate problems.*

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

The right panel is the most important picture in this tutorial: even the *ideal* brightness correction leaves the noise, which is why LLIE is a restoration problem (see DLCV 09 ·Image Restoration) and not only a tone-mapping problem.

### Non-deep-learning LLIE and the characteristics of low-light images

**Plain version.** Histogram methods redistribute intensity; Retinex methods estimate and adjust illumination. Statistical assumptions determine how each behaves.

### Low-light images in color space

Many color spaces separate a "how bright" channel from "what color" channels. Different spaces call it value, intensity, luma or lightness, and each defines it differently. With $R, G, B \in [0, 1]$ (see IP 06 for the spaces themselves):

$$\begin{aligned} \text{HSV:} \quad & V = \max(R, G, B), \\ \text{HSI:} \quad & I = \tfrac{1}{3}(R + G + B), \\ \text{YCbCr (BT.601):} \quad & Y' = 0.299\,R' + 0.587\,G' + 0.114\,B', \\ \text{CIELAB:} \quad & L^* = 116\,f(Y_{\text{lin}}/Y_n) - 16. \end{aligned}$$

Here $R', G', B'$ are gamma-encoded values; $Y_{\text{lin}}$ is linear-light luminance and $Y_n$ that of the reference white; $f(t) = t^{1/3}$ above a small threshold and linear below it. Some remarks:

- $V$ **is the "bright channel".** It is the per-pixel maximum, the counterpart of the *dark channel* (a per-patch minimum) used in dehazing [3]. In a dark image, $V$ is the channel least damaged by the darkness, which is why LIME [4] uses it to initialize the illumination.
- **The luma weights.** ITU-R BT.601 [5] defines luma as $Y' = 0.299R' + 0.587G' + 0.114B'$. That full-range form is what JPEG/JFIF uses. For 8-bit *limited-range* ("studio") video, BT.601 places black at 16 and white at 235: $Y = 16 + 219\,Y'$. With $R', G', B' \in [0, 255]$ this becomes $Y = 16 + 0.257R' + 0.504G' + 0.098B'$, because $219/255 \times (0.299, 0.587, 0.114) \approx (0.257, 0.504, 0.098)$. Both formulas describe the same luma; they differ only in range and offset.
- $L^*$ is designed to be perceptually uniform, so equal steps look roughly equal to a person. That makes it a good channel for contrast enhancement.

The practical lesson: since brightness can be separated from color, you can **enhance only the brightness channel** and keep the chroma. Figure 2 shows that the choice of channel matters a lot.

```
R, G, B = dark[..., 0], dark[..., 1], dark[..., 2]
V = dark.max(axis=2)                          # HSV value
I = dark.mean(axis=2)                         # HSI intensity
Y = 0.299 * R + 0.587 * G + 0.114 * B         # BT.601 luma, full range (JPEG/JFIF)
Y_limited = 16 + 219 * Y                      # 8-bit limited ("studio") range, 16..235
print(np.allclose(Y_limited, 16 + (65.481 * R + 128.553 * G + 24.966 * B)))  # True
```

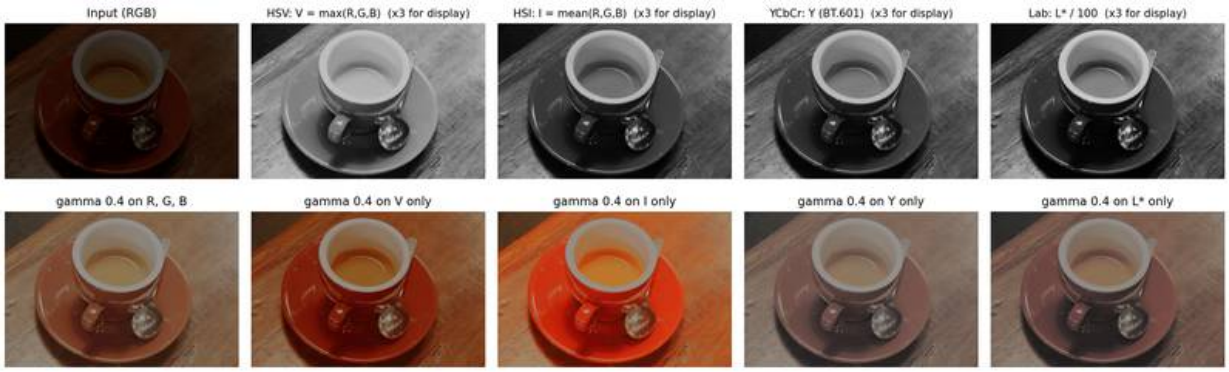



*Figure 2 — The same gamma curve applied in different color spaces. Raising V or I scales all three channels by one factor, which preserves hue and HSV saturation before clipping; perceived colorfulness can still change. Raising Y or L* with fixed chroma desaturates the result. Neither is 'correct': tone and color are coupled.*

Two effects are visible. Scaling RGB by a common factor (HSV, HSI) preserves hue and keeps the ratio between channels, so HSV saturation is unchanged before clipping; their perceived colorfulness can still increase with brightness. Changing only $Y'$ or $L^*$ while keeping Cb/Cr or a*/b* fixed adds brightness without adding color, so the image looks washed out. This coupling of tone and color is exactly what later methods such as HVI-CIDNet [6] try to handle with a learned color space.

### Histogram equalization and CLAHE

**Plain version.** Histogram equalization redistributes intensity values across the available range, expanding dark images' concentrated values but potentially amplifying noise.

**Precise version.** For an image with $L$ levels, normalized histogram $p_r(r_k) = n_k/(MN)$, HE uses the scaled cumulative distribution function as the mapping:

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p_r(r_j), \qquad k = 0, \dots, L-1,$$

where $n_k$ is the number of pixels at level $r_k$ and $MN$ the image size. The slope of $T$ is proportional to the histogram, so heavily populated gray levels get pulled apart. The derivation is in IP 03. For color images, the usual recipe is to convert to a space with a brightness channel, equalize that channel only, and convert back, for example RGB → Lab → equalize $L^*$ → RGB.

Global HE has two problems in low light: it over-stretches large flat regions (including their noise), and one mapping cannot serve both a dark corner and a bright lamp. **Adaptive HE** computes a mapping per tile, and **CLAHE** (contrast-limited AHE) clips each tile's histogram at a limit before computing the CDF, which caps the slope of $T$ and therefore the noise gain, then interpolates between neighboring tiles to avoid seams [7], [8].

*Online only: CLAHE: equalizing locally without amplifying noise — https://blog.shyandram.dev/en/*

*learn/deep-learning/low-light-enhancement/notes/clahe-equalizing-locally-without-amplifying-noise/*

### Gamma correction

A power law $s = r^{\gamma}$ with $\gamma < 1$ lifts shadows more than highlights; its slope $\gamma r^{\gamma-1}$ is large near $r = 0$. It is cheap and monotonic but global: one $\gamma$ for the whole picture, no denoising, and the same desaturation-or-oversaturation choice as above depending on where it is applied.

```
from skimage import color, exposure

gamma_out = dark ** 0.4                                    # power-law curve on every
 ↪  channel
lab = color.rgb2lab(dark)
L = lab[..., 0] / 100                                      # L* scaled to [0, 1]
he = lab.copy();    he[..., 0] = 100 * exposure.equalize_hist(L)
clahe = lab.copy(); clahe[..., 0] = 100 * exposure.equalize_adapthist(L, clip_limit=0.01)
he_out, clahe_out = color.lab2rgb(he), color.lab2rgb(clahe)
for name, im in [("gamma", gamma_out), ("HE", he_out), ("CLAHE", clahe_out)]:
    print(f"{name:6s} mean {im.mean():.3f}   std {im.std():.3f}")
```




*Figure 3 — Classical enhancers on the noisy input of Figure 1. All of them brighten; none of them removes noise, and the Retinex outputs show strong color noise where the input was darkest.*

### Retinex theory

**Plain version.** Retinex says that what the eye (or camera) receives is "the object's own colors" multiplied by "the light falling on it". If we can estimate the light, we can divide it out and see the object as it really is, then put back a better light.

**Precise version.** Land's Retinex theory [9] models an observed image $S$ as

$$S(\mathbf{x}) = R(\mathbf{x}) \circ L(\mathbf{x}),$$

where $\mathbf{x}$ is the pixel position, $R \in [0, 1]$ is the **reflectance** (a simplified surface-related component, assumed consistent across the paired lighting conditions), $L$ is the **illumination** (the light falling on the scene, which varies smoothly in space except at shadow boundaries), and $\circ$ is element-wise multiplication. Note the word: $L$ is *illumination*, not luminance. "Luminance" usually refers to a brightness channel of a color space, such as $Y$ above.

Recovering two unknowns from one product is ill-posed, so every Retinex method adds a prior. The classic one is "illumination is smooth".

**Single-scale Retinex (SSR)** [10] estimates $L$ with a Gaussian blur $G_\sigma$ and works in the log domain, where the product becomes a sum:

$$\log R_c(\mathbf{x}) = \log S_c(\mathbf{x}) - \log\left(G_\sigma * S_c\right)(\mathbf{x}),$$

for each color channel $c$, with $*$ convolution and $\sigma$ the surround scale. Small $\sigma$ gives strong local contrast and halos at edges; large $\sigma$ gives better tone but weaker detail. **Multi-scale Retinex (MSR)** [11] averages SSR outputs at several scales (typically three) with weights $w_n$:

$$\log R_c^{\mathrm{MSR}} = \sum_{n=1}^{N} w_n \left[\log S_c - \log(G_{\sigma_n} * S_c)\right].$$

Because SSR and MSR work per channel, they tend to gray out the colors. **MSRCR** (MSR with color restoration) [11] multiplies the result by a color-restoration factor based on each channel's share of $R + G + B$; other variants apply MSR to an intensity channel only and keep the input's chromaticity.

```
from scipy import ndimage as ndi

def ssr(img, sigma=40.0, eps=1e-3):
    log_s = np.log(img + eps)
    log_l = np.log(ndi.gaussian_filter(img, sigma=(sigma, sigma, 0)) + eps)  # L ~ Gaussian
     ↪  blur
    r = log_s - log_l                                          # log R = log S - log L
    lo, hi = np.percentile(r, [1, 99])                         # robust stretch for display
    return np.clip((r - lo) / (hi - lo), 0, 1)

msr = np.mean([ssr(dark, s) for s in (15, 80, 250)], axis=0)   # MSR, equal weights
 ↪  (simplified)
```

For *enhancement* we usually do not want $R$ alone, which looks flat and unnatural. We want to brighten $L$ and recombine: $\hat{S} = R \circ \hat{L}$ with, for example, $\hat{L} = L^\gamma$ and $\gamma < 1$. This "decompose, adjust $L$, recombine" pattern is the template for most Retinex-based deep networks.




*Figure 4 — A LIME-style Retinex decomposition. The illumination is initialized with $\max(R, G, B)$ and smoothed; the reflectance is S / L. Brightening L with a gamma of 0.35 and multiplying back gives a natural result, while R alone looks flat and over-saturated.*

### LIME: Retinex as optimization

LIME [4] estimates only the illumination. It initializes it with the bright channel, $\hat{T}(\mathbf{x}) = \max_c S_c(\mathbf{x})$, then refines it by solving a structure-aware smoothing problem: keep $T$ close to $\hat{T}$, but make it smooth except where the image has strong structure. The enhanced image is $S/T^\gamma$ (equivalently $RT^{1-\gamma}$ when $R = S/T$), and because dividing amplifies noise, LIME optionally denoises afterward. LIME's images (often called the LIME set) became a standard test set, and the method is a good example of the classical recipe: a physical model, a hand-designed prior, and an optimization.

### Another model: dehazing an inverted image

The Retinex model is not the only image-formation model. Dehazing uses the **atmospheric scattering model**, written out in DLCV 09: a hazy observation $I$ is the haze-free scene radiance $J$ attenuated by a transmission $t$, plus a global atmospheric light $A$. This model cannot be applied to a dark image directly, because a dark image does not look like a hazy one.

Dong et al. observed that an **inverted** low-light image, $1 - S$, *does* look hazy: dark regions become bright and low-contrast, like fog. They therefore invert the image, run a dehazing algorithm (in the spirit of the dark channel prior [3]), and invert the result back [12]. With $I_{\text{haze}}$ the hazy image and $J_{\text{haze}}$ the haze-free result, the inversions are

$$I_{\text{haze}} = 1 - S_{\text{Retinex}}, \qquad J_{\text{haze}} = 1 - \hat{S}_{\text{enhanced}}.$$

From the "what is reliable" angle, dehazing assumes the *dark* channel marks haze-free content; in a low-light image the dark channel is mostly damaged pixels, so attention shifts to the *bright* channel, the region least affected by the darkness. The LIME paper [4] analyzes this inversion approach and shows that it can be rewritten in Retinex form, so the two models make essentially the same assumption, with Retinex being the simpler statement.

### Statistical approaches

Many classical methods add statistics or priors on top of these models: naturalness constraints that keep the lightness order of a scene (NPE [13]), layered or two-dimensional histogram models

(the method behind the DICM set [14]), and fusion of several synthetic exposures. They are hand-crafted, need no training data, and generalize in a predictable way; their weak point is noise, which they usually ignore or hand to a separate denoiser.

## Deep-learning LLIE

### Two axes: learning signal and design lineage

A model's architecture and its supervision are separate choices. Following Li et al.'s survey [15], **supervised learning** (**SL**) uses paired dark/reference images; **reinforcement learning** (**RL**) uses rewards for a sequence of adjustments; **unsupervised learning** (**UL**) uses unpaired dark and normal-light sets; **zero-shot learning** (**ZSL**) in that survey uses low-light data alone, sometimes optimizing on the test image; and **semi-supervised learning** (**SSL**) combines paired and unpaired stages. Here SSL means semi-supervised, not self-supervised. Other papers use "unsupervised," "zero-reference" and "self-supervised" differently, so inspect their actual data requirements.

The design axis asks what is predicted: curve parameters, illumination, image residuals, a generative denoising trajectory, or a compact retouching transform. Retinex can appear under several learning strategies; diffusion is a model family, not a supervision category.

| Method | Design and learning signal | Quality mechanism | Published cost and scope |
|---|---|---|---|
| Retinex-Net [16] | Decomposition and illumination adjustment; paired images, no intrinsic-image labels | Reflectance consistency and structure-aware smoothness | Several learned stages plus optional reflectance denoising; measure the complete pipeline |
| EnlightenGAN [17] | Image generator; unrelated dark/normal-light sets | Global/local adversarial and self-feature constraints | Discriminators are training components; deploy the generator |
| Zero-DCE [18] | Spatially varying curves; no reference images | Exposure, color, local-contrast and parameter-smoothness priors | **79,416** parameters; **84.99 G** FLOPs at 1200 × 900 in [19], Table 4 |
| Zero-DCE++ [19] | Shared curves estimated on a thumbnail; no references | Same family of priors with less flexible curve prediction | **10,561** parameters; **0.115 G** FLOPs at 1200 × 900 in [19], Table 4 |
| RUAS [20] | Searched unrolled illumination/denoising modules; reference-free losses | Retinex structure plus a separate noise-removal stage | Illumination-only **0.001 M** parameters versus illumination-plus-denoising **0.003 M**, as rounded in its Table 2 |
| SCI [21] | Illumination estimation; unsupervised training cascade | Self-calibration during training; one basic block at inference | **0.0003 M** parameters, rounded in its Table 4; calibration module is not deployed |
| Retinexformer [22] | Illumination-guided Transformer; paired images | Noise/artifact restoration guided by illumination features | **1.61 M** parameters, **15.57 G** FLOPs at **256 × 256**, Table 4 |

| Method | Design and learning signal | Quality mechanism | Published cost and scope |
|---|---|---|---|
| LLFlow and diffusion methods [23], [24], [25], [26], [27] | Conditional generative restoration; supervision varies | Learned conditional distribution or image prior | Count all network evaluations, image encoding and decoding; not comparable from parameter count alone |
| HDRNet [28] | Local affine bilateral grid; paired photographic edits | Local/global context chooses a spatially varying transform | Thumbnail prediction plus full-resolution guide, slicing and application |
| Image-adaptive 3D LUT [29] | Basis color tables; paired or unpaired retouching data | Image-dependent global color transform | **1.66 ms** at 3840 × 2160 on one Titan RTX, its Table 7 |

These rows use different resolutions, counting conventions and timing hardware. They explain design choices, **not a common speed ranking**. The LUT timing is a photographic-retouching result; it does not establish denoising quality on a dark phone capture. For a first comparison, run a fixed gamma baseline, inspect the photon experiment, then try the curve toy. Read detailed architectures after the image and evaluation problems are clear.

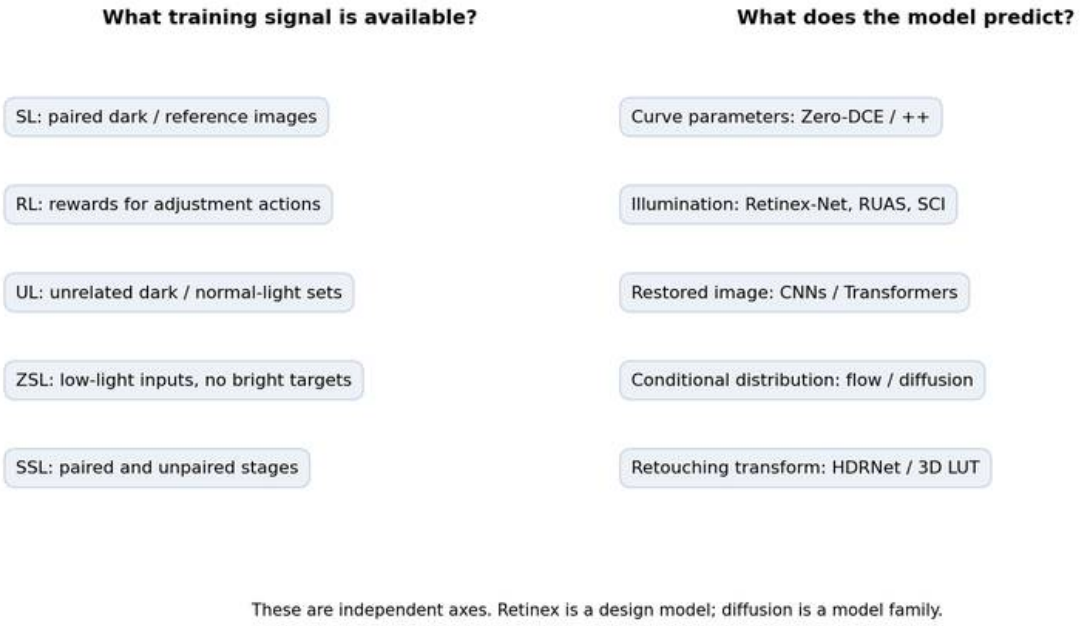



*Figure 5 — Separate supervision from architecture. The same design can use different training signals; generative models are a design family, while Retinex is a modeling choice.*

### Supervised methods (paired data)

Training on pairs $(x_i, y_i)$ of low-light and normal-light images follows the general recipe of DLCV 09: fit $f_\theta$ with a reconstruction loss, typically $\mathcal{L} = \|f_\theta(x) - y\|_1$ plus perceptual or structural terms (see DLCV 09 for these losses). Paired data is expensive (two exposures of a static scene, or

synthesis), and the network learns the *specific* brightness and color style of the reference images, which matters for evaluation later.

Representative examples: **LLNet** [30] (2017), one of the first deep LLIE models, learned brightening and denoising jointly with a denoising autoencoder; Tao et al. [31] (2017) denoised with a CNN and then enhanced contrast with a bright-channel model; **MIRNet** [32] (2020) is a general multi-resolution restoration backbone; **SNR-aware** [33] (2022) estimates a per-pixel signal-to-noise map and uses long-range attention where the local neighborhood is too noisy to trust; and **HVI-CIDNet** [6] (2025) learns a color space that decouples color from intensity, attacking the problem of Figure 2.

*Online only: More deep LLIE methods — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/notes/more-deep-llie-methods/*

### Efficient end-to-end designs

A direct network can learn restoration as well as tone adjustment, at a cost that grows with its spatial feature maps. IAT [34] combines local multiply/add maps with global color-matrix and gamma prediction. Thin convolutions, depthwise filtering and a small global branch reduce computation, while a compact network can still fail on noise patterns absent from training.

Structural re-parameterization trains several linear branches and merges them into one convolution for inference [35]. It changes deployment structure, whereas weight sharing changes stored parameters. Neither guarantees faster execution on every device. Measure the exported model, including its full-resolution operations.

*Online only: Efficient network building blocks — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/notes/efficient-network-building-blocks/; Structural re-parameterization step by step — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/notes/structural-re-parameterization-step-by-step/*

### Retinex-based networks

Retinex-Net [16] (2018) changed the goal from "estimate $R$" to "**both the input and the output have their own** $R$ **and** $L$". The pipeline is: decompose the low and normal images, enhance $L$ to $\hat{L}$, recombine with $R$. It also introduced the LOL dataset (below). Its paired-image training enforces reconstruction, reflectance consistency across exposures and structure-aware illumination smoothness. There are no measured intrinsic reflectance/illumination targets: the components are useful learned representations, not uniquely identified physical quantities. A separate illumination-adjustment network changes brightness, and reflectance denoising addresses amplified noise.

*Online only: Retinex-Net: decompose, adjust and denoise — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/papers/retinex-net-decompose-adjust-and-denoise/*

The pattern spread quickly. **KinD** [36] (2019) and **KinD++** [37] (2021) add a reflectance-restoration step, because the dark image's reflectance is noisy and color-shifted; **Retinexformer**

[22] (2023) adds perturbation terms for noise and artifacts to the Retinex model, trains in one stage, uses the illumination estimate to guide self-attention, and became a common baseline; **RetinexMamba** [38] (2024 preprint) swaps attention for Mamba state-space blocks. "Retinex decomposition plus a new backbone" has been a stable design pattern for years. A less common route builds lightweight networks on the atmospheric scattering model instead (CPGA-Net [39], CPGA-Net+ [40]).

*Online only: More deep LLIE methods — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/notes/more-deep-llie-methods/*

### Quality and cost within the same lineage

Predicting a smooth transform on a thumbnail can preserve full-resolution detail because the original pixels are still available when the transform is applied. It cannot recover information that clipping or noise removed. If the desired operation is denoising or deblurring, a pointwise transform alone is insufficient; a network that predicts that transform from a neighborhood can still use context, so "curve-based" does not imply that the complete system has no spatial reasoning.

For illumination models, division by a small illumination estimate amplifies both detail and noise. A separate reflectance-restoration module costs more but addresses a different problem from tone adjustment. The experiment below contrasts resizing a smooth illumination map with resizing the already enhanced image. Its reference is the same toy operator run at full resolution, not a real normal-light ground truth.

It estimates the illumination in a simple classical way: the bright channel (the per-pixel maximum of R, G and B), which LIME [4] uses as its initial estimate, smoothed here with a plain Gaussian instead of LIME's structure-aware refinement. It does so once at full resolution and once on an 8× smaller copy (1/64 of the pixels), and compares two ways of producing the full-resolution result.

```
import numpy as np
from scipy.ndimage import gaussian_filter
from skimage import data, img_as_float
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.transform import resize

dark = (0.15 * img_as_float(data.coffee()) ** 2.2) ** (1 / 2.2)   # 400 x 600, 15% of the
 ↪  light

def illumination(x, sigma):
    """A simple illumination estimate: the bright channel (max of R, G, B), smoothed."""
    return np.clip(gaussian_filter(x.max(axis=-1), sigma), 0.02, 1)

def relight(x, L, gamma=0.4):
    """Retinex relighting: O = R * L**gamma = I * L**(gamma - 1)."""
    return np.clip(x * L[..., None] ** (gamma - 1), 0, 1)

reference = relight(dark, illumination(dark, sigma=15))           # estimate L at full
 ↪  resolution
small = resize(dark, (50, 75), anti_aliasing=True)                # 8x smaller in each
 ↪  direction
```

```
L_small = illumination(small, sigma=15 / 8)                    # estimate L on 1/64 of the
 ↪  pixels
results = {
    "L at 1/8 size, resized up": relight(dark, resize(L_small, dark.shape[:2], order=1)),
    "relight at 1/8 size, resize output": resize(relight(small, L_small), dark.shape,
     ↪  order=1),
}
print(f"mean brightness {dark.mean():.2f} -> {reference.mean():.2f}")
for name, out in results.items():
    print(f"{name:36s} PSNR vs full-resolution estimate {psnr(reference, out,
     ↪  data_range=1):5.2f} dB")
```

```
mean brightness 0.16 -> 0.35
L at 1/8 size, resized up            PSNR vs full-resolution estimate 54.57 dB
relight at 1/8 size, resize output   PSNR vs full-resolution estimate 22.87 dB
```

Resizing the *map* is almost indistinguishable from the full-resolution estimate (above 54 dB on our toy), while resizing the *output* loses about 31.7 dB, because the image's detail is gone. It works because this light map is very smooth; one with sharp shadow boundaries suffers more, which is why some methods upsample it with an edge-aware step.

### Retinex-inspired unrolling

**Unrolling** (or unfolding) takes an iterative optimization algorithm for the Retinex problem and turns each iteration into a network stage with learnable parameters. The architecture inherits the algorithm's structure and interpretability, and training replaces hand-tuned priors.

- **RUAS** [20] (2021) unrolls optimization models for illumination estimation and noise removal, searches the architecture of the prior modules, and is trained with a cooperative *reference-free* strategy, so it also belongs in the unsupervised column. It is very small and fast.
- **URetinex-Net** [41] (2022) unfolds a Retinex decomposition optimization into a network with learned modules for initialization, the unfolded updates of $R$ and $L$, and illumination adjustment.

### Unpaired training: EnlightenGAN

**EnlightenGAN** [17] (2021) uses a generative adversarial network (GAN) so that it needs only a set of dark images and an *unrelated* set of well-lit images. A U-Net generator, guided by a self-regularized attention map derived from the input's illumination, produces the enhanced image. A **global** discriminator judges the whole image and a **local** discriminator judges random patches, so the output looks like a well-lit photo both overall and in detail. A **self-feature-preserving** loss compares VGG features of input and output, keeping content in place since there is no ground truth to do so.

### Zero-reference methods and curve estimation

**Plain version.** Zero-DCE does not output a new image. It outputs a *brightening curve per pixel*, applies it, and is trained with losses that judge the result without any reference photo: "is it well exposed, is the color balanced, is the curve smooth, is local contrast kept?"

**The LE curve.** Zero-DCE [18] (2020) defines a quadratic **light-enhancement curve**

$$\mathrm{LE}(x;\alpha) = x + \alpha\, x\,(1-x), \qquad \alpha \in [-1,1],$$

where $x \in [0,1]$ is a pixel value. The curve has three useful properties: $\mathrm{LE}(0) = 0$ and $\mathrm{LE}(1) = 1$, so the range is preserved and no clipping is needed; it is monotonic for $|\alpha| \le 1$, because $\mathrm{d\,LE}/\mathrm{d}x = 1 + \alpha(1-2x) \ge 0$, so a fixed curve preserves order; different spatial parameter maps need not preserve ordering between pixels; and it is differentiable, so it can be trained by backpropagation. One quadratic is not flexible enough, so it is applied **iteratively**:

$$\mathrm{LE}_n(\mathbf{x}) = \mathrm{LE}_{n-1}(\mathbf{x}) + \mathcal{A}_n(\mathbf{x})\,\mathrm{LE}_{n-1}(\mathbf{x})\,(1 - \mathrm{LE}_{n-1}(\mathbf{x})),$$

with $\mathrm{LE}_0$ the input, $n = 1, \ldots, 8$, and $\mathcal{A}_n$ a **per-pixel, per-channel** parameter map instead of a single $\alpha$. A seven-layer CNN (DCE-Net, 32 channels per layer, about 79k parameters, no downsampling) predicts $8 \times 3 = 24$ maps with a tanh output. Each pixel thus gets its own high-order curve, which is how local adjustment arises from a global-looking formula.

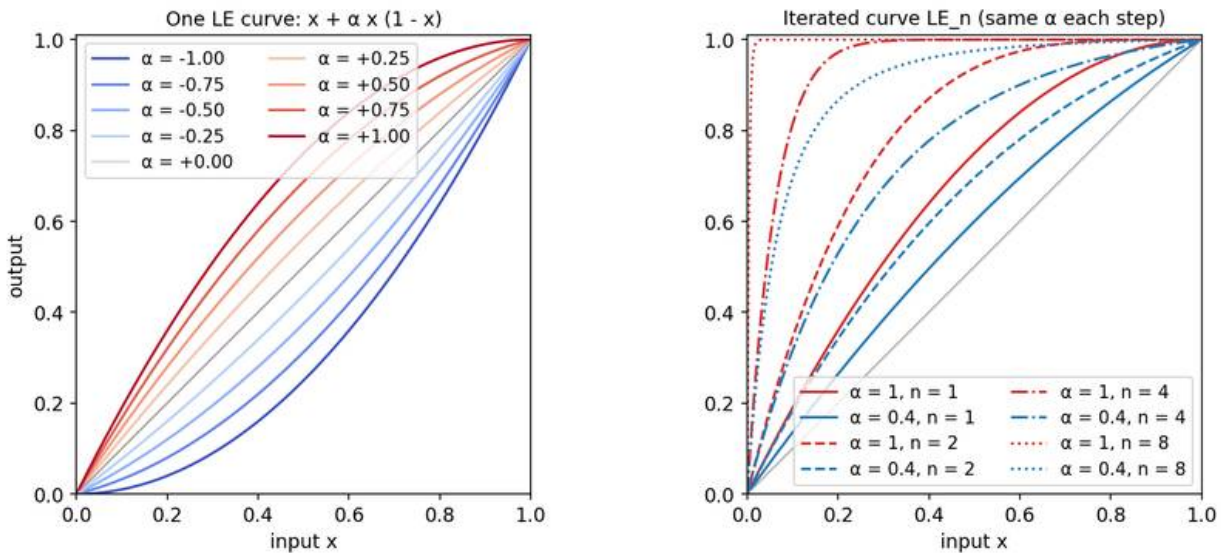



*Figure 6 — The LE curve. Left: one iteration for α from −1 to 1. Right: iterating sharpens the curve; with $\alpha = 1$ and $n = 8$, a value of 0.02 is already mapped to almost 1, so tiny noisy values in the shadows are amplified enormously.*

```
def le_iterate(x, alphas):
    """Zero-DCE curve: LE_n = LE_{n-1} + a_n * LE_{n-1} * (1 - LE_{n-1})."""
    y = x.copy()
    for a in alphas:              # a can be a scalar or a per-pixel map
```

```
        y = y + a * y * (1 - y)
    return y

x = np.array([0.0, 0.02, 0.1, 0.3, 0.6, 1.0])
print(le_iterate(x, [1.0] * 8).round(3))   # [0. 0.994 1. 1. 1. 1.]
print(le_iterate(x, [0.4] * 8).round(3))   # [0. 0.245 0.695 0.93 0.985 1.]
```

The first line shows the danger: for small $x$, each step multiplies by about $1 + \alpha$, so eight steps with $\alpha = 1$ give a gain near $2^8 = 256$. Whatever noise sits in the shadows is amplified by the same factor.

**Non-reference losses.** Training uses four complementary priors: retain local contrast, move patch-mean brightness toward a selected exposure target, discourage global RGB color imbalance, and smooth the parameter maps. None is a measured normal-light target. Gray-world balance can damage a colorful scene; the exposure target is a preference; smooth curves can still amplify noise. The original paper uses exposure target 0.6 and eight iterations [18].

*Online only: Zero-DCE and Zero-DCE++: learn curves without reference images — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/papers/zero-dce-and-zero-dce-learn-curves-without-reference-images/*

Follow-ups refined the idea:

- **Zero-DCE++** [19] (2022) uses depthwise separable convolutions, predicts only **3** curve maps and reuses them across all iterations, and estimates them on a downsampled input before resizing them back. This cuts the model to about 10k parameters and makes it fast enough to run at interactive rates even on a CPU.
- **SCI** [21] (2022) learns illumination with a cascade of weight-sharing stages and a **self-calibrated** module that makes the stages' outputs converge during training, so that at test time a **single** basic block suffices. It is trained with unsupervised losses.
- **CLIP-LIT** [42] (2023) targets *backlit* images. It uses the vision-language model CLIP and learns a pair of text prompts, one for backlit and one for well-lit images, then refines the prompts and the enhancement network iteratively, so the CLIP similarity to the "well-lit" prompt acts as the training signal.

### A zero-reference toy in PyTorch

To see these losses at work, the code below trains a tiny Zero-DCE-style network for about half a minute on CPU. It trains on random crops of four `skimage` images, each darkened by a random factor in linear light, and is tested on the coffee image, which it never sees during training. No ground truth is used. (The loss weights differ from the paper's because the terms are normalized differently: squared rather than absolute exposure error and mean rather than summed gradients.)

```
import numpy as np
import torch
```

```
torch.set_num_threads(1)
import torch.nn as nn
import torch.nn.functional as F
from skimage import data, img_as_float
from skimage.transform import resize

torch.manual_seed(0); rng = np.random.default_rng(0)
darken = lambda x, k: (k * x ** 2.2) ** (1 / 2.2)          # less light, in linear units
train = [resize(img_as_float(f()), (256, 256), anti_aliasing=True)
         for f in (data.astronaut, data.chelsea, data.rocket, data.immunohistochemistry)]

def batch(n=4, size=64):                                    # random crops, random darkness
    xs = []
    for _ in range(n):
        img = train[rng.integers(len(train))]
        i, j = rng.integers(0, 256 - size, 2)
        xs.append(darken(img[i:i + size, j:j + size], rng.uniform(0.02, 0.3)))
    return torch.tensor(np.stack(xs), dtype=torch.float32).permute(0, 3, 1, 2)

class TinyDCE(nn.Module):
    def __init__(self, n_iter=8, w=16):
        super().__init__()
        self.n_iter = n_iter
        self.body = nn.Sequential(
            nn.Conv2d(3, w, 3, padding=1), nn.ReLU(), nn.Conv2d(w, w, 3, padding=1),
            ↪  nn.ReLU(),
            nn.Conv2d(w, w, 3, padding=1), nn.ReLU(), nn.Conv2d(w, 3 * n_iter, 3, padding=1),
            ↪  nn.Tanh())
    def forward(self, x):
        alphas = self.body(x).chunk(self.n_iter, dim=1)    # 8 maps of shape B x 3 x H x W
        y = x
        for a in alphas:
            y = y + a * y * (1 - y)                         # the LE curve
        return y, torch.cat(alphas, 1)

def L_exp(y, E=0.6, p=16):   # mean brightness of 16x16 patches should be near E
    return (F.avg_pool2d(y.mean(1, keepdim=True), p) - E).pow(2).mean()
def L_col(y):                # gray-world: channel means should agree
    r, g, b = y.mean((2, 3)).unbind(1)
    return ((r - g) ** 2 + (r - b) ** 2 + (g - b) ** 2).mean()
def L_tv(a):                 # curve parameters should vary smoothly in space
    return (a[..., 1:, :] - a[..., :-1, :]).pow(2).mean() + (a[..., :, 1:] - a[..., :,
    ↪  :-1]).pow(2).mean()
def L_spa(y, x, p=4):        # keep the contrast between neighbouring 4x4 regions
    Y, X = (F.avg_pool2d(t.mean(1, keepdim=True), p) for t in (y, x))
    return sum(((torch.diff(Y, dim=d) - torch.diff(X, dim=d)) ** 2).mean() for d in (2, 3))

net = TinyDCE()
opt = torch.optim.Adam(net.parameters(), lr=1e-3)
for step in range(600):                                     # small CPU training experiment
    x = batch()
    y, a = net(x)
    loss = L_spa(y, x) + 10 * L_exp(y) + 5 * L_col(y) + 200 * L_tv(a)
    opt.zero_grad(); loss.backward(); opt.step()

test = darken(img_as_float(data.coffee()), 0.05)           # an image never used in training
```

```
with torch.no_grad():
    out, _ = net(torch.tensor(test, dtype=torch.float32).permute(2, 0, 1)[None])
print(f"mean brightness: {test.mean():.3f} -> {out.mean():.3f}")   # about 0.10 -> 0.50
```

Drag the slider to compare the dark input and the toy's output:



*Dark input*



*Zero-reference toy*

The dark coffee-cup image and the toy network's output, which is much brighter, with visible wood grain and a lighter, slightly greenish and less saturated color than the original photo.

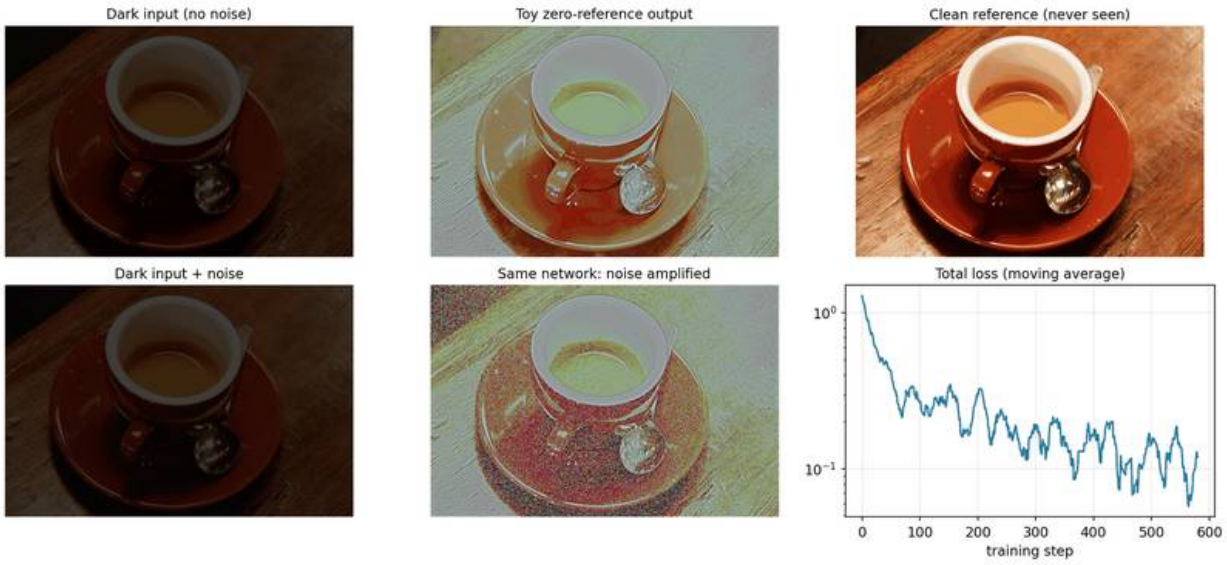



*Figure 7 — What the zero-reference losses do and don't do. The exposure loss gives good brightness, but the gray-world color loss pulls a red-brown scene toward gray (top middle versus top right), and with real*

*noise the per-pixel curves amplify it* (*bottom middle*).

The toy shows the strengths and the limits of the zero-reference idea in one picture. Without a single reference image, the network learns a sensible brightening that transfers to an unseen photo. But each loss is a *prior*, and priors can be wrong: gray-world fails for scenes that really are red or brown, a fixed exposure target $E$ flattens deliberately dark regions, and nothing in the losses asks for denoising. Real zero-reference methods are trained on many more images, and some add explicit noise handling, but these failure modes are visible in published results too.

### Normalizing flows and diffusion models

**LLFlow** [23] (2022) observes that one dark image has *many* acceptable bright versions, and a pixel-wise loss averages them into a blurry compromise. It learns a conditional normalizing flow: an invertible network that maps normal-light images to a Gaussian distribution, conditioned on features of the dark input. Those features are guided by priors that echo the classical methods: a histogram-equalized version of the input, a color map (close to the Retinex reflectance) and a noise map.

From 2023, **diffusion models** entered LLIE. They generate the output by iteratively denoising random noise, conditioned on the dark image, and so can synthesize plausible detail that regression networks blur out. Diff-Retinex [24] generates the Retinex reflectance and illumination with diffusion, LightenDiffusion [26] decomposes in a latent space and trains on unpaired data, Reti-Diff [27] uses latent diffusion only to produce compact priors for a restoration transformer, and GSAD [25] regularizes the diffusion trajectory with the image structure. The trade-off is cost and fidelity: many denoising steps make inference slow, and generated details may look right without being *true*.

*Online only: More deep LLIE methods — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/notes/more-deep-llie-methods/*

### Real-time photographic transforms

HDRNet [28] and image-adaptive 3D LUTs [29] originated in photographic retouching. They predict a compact transform from a thumbnail and apply it to the original pixels. This can retain input detail at high resolution, but a fast tone/color edit is not a complete sensor-noise restoration pipeline.

A bilateral grid stores spatially varying affine transforms. A full-resolution guide supplies the third coordinate, allowing pixels near an edge to select different transforms. After interpolating eight nearby nodes, apply the resulting $3 \times 4$ matrix to RGB plus a constant bias. HDRNet learns the guide and coefficients from paired edits; the optional experiment instead fits coefficients by weighted least squares, so it is a conceptual toy rather than the original learned model.

A 3D LUT is indexed by RGB rather than spatial position. It stores a transform on an $M \times M \times M$ color lattice, with three output values per node. Trilinear interpolation combines eight enclosing colors. An image-adaptive model predicts coefficients $a_n$ for basis tables $\Phi_n$ and forms

$$\Phi = \sum_{n=1}^{N} a_n \Phi_n.$$

Here $N$ is the number of bases and $\Phi$ is the image's combined table. Fusing the tables once and interpolating once per pixel equals interpolating every basis and combining the outputs, because interpolation is linear in table values. A global LUT maps identical RGB values identically everywhere in one image; it cannot independently brighten one of two same-colored regions.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

The default adaptive LUT uses three 33-node bases [29]. Table size grows as $3M^3$ values, so finer lattices trade memory for accuracy near steep dark-region curves. Learned nonuniform nodes and spatial/context extensions change that trade-off [43], [44], [45], [46], [47]. The full method and measured results are in the paper introductions.

*Online only: Bilateral grids and color-table mathematics — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/notes/bilateral-grids-and-color-table-mathematics/; A bilateral-grid experiment — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/notes/a-bilateral-grid-experiment/; Implementing and checking trilinear LUTs — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/notes/implementing-and-checking-trilinear-luts/*

### Raw-domain low light

Everything above works on processed sRGB images, where the camera's pipeline (demosaicking, white balance, denoising, tone curve, compression) has already discarded information. **Learning to See in the Dark** [48] (2018) instead feeds **raw** sensor data to a U-Net that replaces the whole pipeline. Its SID dataset has 5,094 short-exposure raw images paired with 424 distinct long-exposure references, from a Sony α7S II (Bayer sensor) and a Fujifilm X-T2 (X-Trans sensor), with exposure ratios of ×100, ×250 and ×300. Raw data keeps the linear relation between photons and values and has more bits, so extreme low light that is hopeless in sRGB becomes recoverable. The cost is that the model is tied to one sensor.

### All-in-one restoration

Low-light enhancement is also one task inside all-in-one restoration models, for example **InstructIR** [49] (2024), which restores an image according to a natural-language instruction such as "make this photo brighter". The all-in-one family as a whole is described in DLCV 09.

### Low light for downstream tasks

If the goal is *detection* rather than pretty pictures, the usual baseline is "enhance, then detect" (see DLCV 03 ·Object Detection). But enhancement tuned for people is not necessarily what a detector

needs. **MAET** [50] (2021) models the low-light degradation physically and trains the detector to also predict the degradation parameters, so that features become aware of it. **FeatEnHancer** [51] (2023) enhances hierarchical *features* inside the network, trained directly with the downstream loss, instead of enhancing pixels.




*Figure 8 — A timeline of the methods in this tutorial (by publication year). Colors mark Retinex-based or unrolled designs, unpaired or zero-reference training, flow and diffusion models, and challenges.*

### Measuring efficiency honestly

**Plain version.** Efficiency claims are easy to make and hard to compare. A fair comparison of lightweight designs says exactly what was counted, at which resolution, on which hardware, and includes every step from input frame to output frame.

Dehghani et al. show that parameters, FLOPs, throughput and latency can disagree, and that reporting only one of them gives a misleading picture [52]. For enhancement:

- **Count at the deployment resolution**, and also report MACs per megapixel so that readers can rescale. A FLOP count at 256 × 256 says nothing about 4K for a full-resolution network, and the opposite is true for thumbnail-based designs.
- **Include the full-resolution step.** Slicing, LUT interpolation, curve iteration, guided-filter upsampling and resizing are often custom operators that automatic FLOP counters skip. They are the part of a predict-then-apply design that grows with resolution.
- **Say what a "parameter" includes.** A 33-node 3D LUT is about 108K values, more than the whole of Zero-DCE++ at about 10K parameters [19] or IAT at about 90K [34].
- **Time the whole path on the target device**: fixed device, runtime, numeric precision, batch size 1, resolution and thread count, warm-up, percentiles, GPU synchronization, and color conversion and memory transfers included (the protocol of ML 11). Challenges that time on real phones make this concrete: the Mobile AI 2022 learned-ISP challenge, for example, measured runtime on a Snapdragon 8 Gen 1 GPU with TensorFlow Lite [53]. A GPU number from a paper (such as the Titan RTX or V100 timings above) tells you about relative speed on that GPU, not about your phone.

- **Judge quality at full resolution**, with the metrics of DLCV 09 and the low-light specifics of DLCV 09C, because halos and banding from coarse grids or tables may vanish when the output is downsampled for scoring.

*Online only: Counting resolution, arithmetic and memory — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/notes/counting-resolution-arithmetic-and-memory/*

### Failure modes

| Symptom | Why it happens | What to check |
|---|---|---|
| Bright grain and false texture | Shadow gain amplifies sensor noise | Flat-region crops and a noise-aware restoration stage |
| Posterized gradients | A few quantized dark levels are spread far apart | Input bit depth and near-black ramps |
| Halos | A coarse illumination map or bilateral cell spans a sharp boundary | Full-resolution edges, grid size and guide behavior |
| Unnatural color | Gray-world priors, clipping or sparsely trained LUT regions | Skin, neon lights and strongly colored scenes |
| Flickering video | Per-frame parameters change abruptly | Temporal parameter traces and moving scenes |
| Good scores, poor recognition | The metric objective differs from the downstream task | A fixed, disclosed task protocol as well as visual evaluation |

A global LUT cannot map identical colors differently by position, and a fixed pointwise transform cannot average independent spatial noise. Spatially predicted curve parameters can use neighboring evidence, so diagnose the complete architecture rather than assuming every curve model has the same limits.

*Online only: Banding, halos and temporal stability — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/notes/banding-halos-and-temporal-stability/*

### Evaluation

**Plain version.** The general evaluation toolkit (full-reference and no-reference metrics, learned and vision-language assessors, user studies, and the pitfalls and reporting practice that go with them) is explained once, in DLCV 09. This section does not repeat it. It covers what is different for dark images: why no-reference scores matter so much here, how the common ones mislead on exposure and color, measures made for enhancement, the GT-mean pitfall, and task-driven evaluation.

### Full-reference metrics

On paired sets, LLIE papers report PSNR, SSIM [54] and LPIPS [55] exactly as defined in DLCV 09. The one low-light-specific caution is brightness: a small global exposure difference from the reference costs a lot of PSNR, which is the subject of the GT-mean pitfall below.

### No-reference metrics: why low light depends on them

**Plain version.** A no-reference (NR) metric looks at one image and guesses how good it is, with no "correct answer" to compare against. Low-light papers need these metrics because their most realistic test photos have no correct answer, and because even when one exists, it is only one of many acceptable brightness levels.

**Precise version.** Two facts push LLIE toward NR evaluation:

1. **Real test sets are unpaired.** LIME [4], MEF [56], NPE [13], DICM [14] and VV [57] are small sets of real, often non-uniformly lit photos with no ground truth. Paired sets such as LOL [16] exist, but they were captured under controlled conditions (a static scene, two exposures), and a model can do well on them without generalizing (the generalization gap reported in [15]). Results on the unpaired sets are therefore shown in almost every LLIE paper, and only NR metrics and human judgments can score them.
2. **The "correct" brightness is ambiguous.** Even with a reference, the normal-light image is one photographer's choice of exposure and color. An output that is a little brighter or warmer can look as good, yet lose several dB of PSNR (this is the reason for the GT-mean protocol below). An NR metric at least does not punish a different but valid exposure.

The price is that an NR metric is a *model of human preference*, and every model has blind spots. The general theory of no-reference image quality assessment (IQA), including how NIQE, BRISQUE, MUSIQ and CLIP-IQA are built and how NR metrics are validated against human opinion, is covered in DLCV 09. The subsections below do not repeat it; they focus on how each family behaves on *enhanced low-light* images.

### Classical natural-scene-statistics metrics: NIQE, BRISQUE and PI

NIQE [58], BRISQUE [59] and the Perceptual Index (PI) [60], [61] are natural-scene-statistics (NSS) metrics: they measure how far the statistics of an image (its MSCN coefficients) are from those of clean photos, and lower is better for all three. The MSCN coefficients and the BRISQUE and NIQE constructions are given in DLCV 09, together with a runnable snippet showing how blur, noise and over-sharpening move the features; the definition of PI is in the DLCV 09 notes. Low-light papers use these metrics heavily; Zero-DCE [18], for example, reports PI on the unpaired sets alongside a user study.

**What they reward, and how they mislead on exposure.** All three were designed for *structural* distortions (blur, noise, compression, ringing), not for exposure or color:

- MSCN normalization divides out local mean and contrast by construction, so a global brightness change barely moves the features. An under-enhanced image and a well-exposed one can get similar scores.
- Grayscale features ignore color casts and saturation, the very errors of Figure 2.
- Noise and over-sharpening add high-frequency energy, which can make a smooth, noise-amplified output look "more natural" to these statistics than a clean but slightly soft one. HE and CLAHE outputs, with strong local contrast, are a typical case.

- The pristine corpus and training distortions contain few dark, night or backlit scenes, so "natural" means "daylight-like".

### No-reference measures need complementary checks

A low-light metric can reward the wrong change. Lightness order error (LOE) measures how often relative brightness ordering changes [13]; it does not check whether the result is bright enough. Colorfulness [62] measures opponent-color variation and can rise when chroma noise is amplified. Noise estimates depend on separating texture from noise [63]. Learned quality models discussed in the optional note inherit their training domains, which may contain few severely dark images.

Use several metric families, show full-resolution crops, and specify the preprocessing and implementation. For human comparisons, randomize order and state the viewing conditions and question; photographic preference and task usefulness are distinct outcomes. A source image is available at test time, but its brightness is not the desired brightness target.

*Online only: Understanding no-reference low-light metrics — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/notes/understanding-no-reference-low-light-metrics/*

### The GT-mean pitfall

**Plain version.** On paired benchmarks, a small difference in *overall brightness* changes PSNR a lot. Some papers rescale each output so its average brightness matches the ground truth before scoring. That is information you never have in practice.

**Precise version.** "GT-mean" evaluation scales the output by $\lambda = \mathbb{E}[y]/\mathbb{E}[f(x)]$ and computes the metric on $\lambda f(x)$, where $\mathbb{E}[\cdot]$ is the image mean. Because LLIE is ambiguous about the "right" brightness, this can raise PSNR substantially. The Retinexformer authors provide this mode in their code for comparability with methods such as LLFlow and KinD and recent diffusion models, while explicitly saying they do *not* recommend it, because it uses the ground truth to correct the output [22]. Liao et al. [64] (2025) study the brightness mismatch between outputs and references and propose a GT-mean *loss* for training, reporting both standard and GT-mean metrics. When reading tables: check which setting is used, never mix the two, and prefer methods that are good under the standard protocol.

### Task-driven evaluation

Since LLIE is often a preprocessing step, another test is to run a detector or segmenter on the enhanced images (on ExDark [65], for example) and report its accuracy. Liu and Fan's 2025 review [66] finds a disconnect: supervised methods often produce images with high perceptual quality but only modest gains for vision tasks, while zero-shot methods, despite lower image-quality scores, give more consistent gains across tasks. The benchmark of Liu et al. [67] also evaluates face detection in low light. A complete evaluation reports fidelity, no-reference quality, task performance, and runtime.

### Benchmark datasets

**Plain version.** Datasets differ in whether they have pairs, whether the darkness is real or synthetic, whether they are raw or sRGB, and whether they carry labels for detection.

| Dataset | Type | Content and what it is for |
|---|---|---|
| LOL (v1) [16] | Paired, real, sRGB | 500 low/normal pairs captured by changing exposure (485 train / 15 test), 400×600. The most common LLIE benchmark; small test set. |
| LOL-v2 [68] | Paired, real + synthetic subsets | A larger *Real* subset of captured pairs and a *Synthetic* subset of synthesized pairs. Report the two subsets separately. |
| LSRW [69] | Paired, real | 5,650 pairs from a Nikon D7500 (3,170) and a Huawei P40 Pro (2,480); 5,600 train, 50 test. Tests generalization across devices. |
| SDSD [70] | Paired video | Dynamic indoor and outdoor scenes captured twice with a mechatronic system for alignment; often split into frames for image tests. |
| SID [48] | Paired, raw | 5,094 short-exposure raw images with 424 long-exposure references (Sony and Fuji). Extreme low light, raw-to-sRGB. |
| MIT-Adobe FiveK [71] | Paired, retouching | 5,000 raw photos, each retouched by five experts. Used for tone and color enhancement (photo retouching). |
| SICE [72] | Multi-exposure sequences | Under- to over-exposed captures of each scene with a reference; used to train exposure correction (e.g. Zero-DCE). |
| VE-LOL [67] | Paired + detection | VE-LOL-L for enhancement and VE-LOL-H with face annotations for high-level evaluation. |
| ExDark [65] | Unpaired, labeled | 7,363 low-light images, 12 object classes, 10 lighting conditions. Task-driven evaluation (detection). |
| LIME [4], MEF [56], NPE [13], DICM [14], VV [57] | Unpaired test sets | Small collections of real low-light or non-uniformly lit photos for visual comparison and no-reference metrics. |

### Challenges: NTIRE 2024 and 2025

The NTIRE workshop at CVPR (see DLCV 09 for the series in general) ran LLIE challenges in consecutive years. The **NTIRE 2024** challenge [73] covered ultra-high-resolution (4K and beyond), non-uniform illumination, backlighting, extreme darkness and night scenes; it had 428 registered participants and 22 teams with valid submissions. The **NTIRE 2025** challenge [74] had 762 registered participants and 28 teams with valid entries. The reports summarize the winning designs

and are a good snapshot of what works under a fixed, fair protocol.

### Modern view

Li et al. [15] provide the learning-strategy taxonomy used above and compare generalization, cost and image quality under a common protocol. Their review includes real-phone image/video data and warns that a result on a paired benchmark need not transfer to a noisy phone capture. Zheng et al. [75] add mixed-exposure and video perspectives; VE-LOL [67] and the later task-oriented study [66] distinguish appearance from downstream utility. The diffusion review [76] organizes generative enhancement into six families, which differ in conditioning, representation and inference procedure. Read each survey as a dated map, then verify a recent method through its primary paper.

**Efficient enhancement in 2026.** The NTIRE efficient low-light challenge [77] uses a model-size constraint and rank aggregation over several quality metrics. A small file is not a measured phone latency. Its report illustrates compact encoder–decoders, color/illumination priors and re-parameterization under that particular protocol. HDRNet and LUT models instead originated in photographic retouching; they need separate low-light validation.

**State of the art (2023–2026).** Three directions dominate. First, **Retinex-guided backbones**: transformers (Retinexformer [22]) and state-space models (RetinexMamba [38]) that keep a light-versus-reflectance structure. Second, **generative models**: diffusion (Diff-Retinex [24], GSAD [25], LightenDiffusion [26], Reti-Diff [27]) and flows (LLFlow [23]). Third, **better representations and protocols**: learned color spaces (HVI-CIDNet [6]), brightness-mismatch-aware training and evaluation (GT-Mean loss [64]), language-guided and all-in-one restoration (CLIP-LIT [42], InstructIR [49]), and task-aware enhancement (MAET [50], FeatEnHancer [51]).

**Open problems.**

- **Real noise.** Most sRGB methods train on data whose noise does not match real sensors after a camera pipeline. Better noise models [2], raw-domain training [48], or joint enhancement and denoising remain central.
- **Video consistency.** Frame-by-frame enhancement flickers. Paired video data (SDSD [70]) is scarce, and temporal consistency is still hard, especially with generative models.
- **Evaluation.** The ill-posed brightness target, GT-mean protocols, unreliable no-reference metrics for exposure, and the gap between perceptual and task metrics make comparisons fragile.
- **Efficiency on edge devices.** Phones and cars need real-time, low-power models. Curve methods (Zero-DCE++ [19], SCI [21]) and lightweight prior-based networks [39] show that tiny models can work, while iterative diffusion models require careful accounting of all inference steps.
- **Generalization.** Models trained on one dataset or camera often fail on another (the generalization gap reported in [15]); mixed over- and under-exposure, backlight and colored artificial light are still difficult.

*Online only: KinD: repair reflectance while controlling illumination — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/papers/kind-repair-reflectance-while-controlling-*

*illumination/*

*Online only: IAT: local affine maps and a global camera-style transform — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/papers/iat-local-affine-maps-and-a-global-camera-style-transform/; Multinex: fuse analytic priors into a Retinex residual — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/papers/multinex-fuse-analytic-priors-into-a-retinex-residual/*

*Online only: CPGA-Net: channel priors, gamma estimation and learned fusion — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/papers/cpga-net-channel-priors-gamma-estimation-and-learned-fusion/; CPGA-Net+: attention, BDSF and stronger variants — https://blog.shyandram.dev/en/learn/deep-learning/low-light-enhancement/papers/cpga-net-attention-bdsf-and-stronger-variants/*

## Key takeaways

- Brightening changes tone; restoration must also address noise, damaged detail and color.
- Brightness-channel choices affect color and saturation; state the encoding and value range.
- Retinex decomposition depends on illumination/reflectance assumptions, not a uniquely observed physical separation.
- Keep supervision separate from design: paired, unpaired or zero-reference training can support different architectures.
- Spatially varying enhancement can change pixel ordering; exposure and smoothness priors can fail on noise or colorful scenes.
- Keep GT-mean and standard protocols separate, and evaluate downstream tasks rather than assuming brighter is better.
- Use complementary quality metrics and user studies; no single score captures exposure, color, noise and fidelity.

## Exercises

1. **Monotonic curves.** Show that $\mathrm{LE}(x;\alpha) = x + \alpha x(1-x)$ is monotonically non-decreasing on $[0, 1]$ if and only if $|\alpha| \leq 1$. What goes wrong for $\alpha = 1.5$?

   **Hint**

   The derivative is $1 + \alpha(1-2x)$, which is linear in $x$, so its minimum on $[0,1]$ is at an endpoint: $1+\alpha$ at $x=0$ or $1-\alpha$ at $x=1$. Both are $\geq 0$ exactly when $|\alpha| \leq 1$. For $\alpha = 1.5$ the slope at $x=1$ is $-0.5$, so near-white pixels would get darker as the input gets brighter, reversing the brightness order and creating artifacts in highlights.

2. **Gain of the iterated curve.** For a small input $x$, show that eight iterations with a constant $\alpha$ multiply $x$ by approximately $(1+\alpha)^8$. Use this to explain why zero-reference curve methods amplify shadow noise, and suggest one change to the losses or architecture that would reduce the problem.

**Hint**

For small $x$, $x(1-x) \approx x$, so each step gives $x \mapsto (1+\alpha)x$. With $\alpha = 1$ that is $2^8 = 256$. Noise with standard deviation $\sigma$ becomes $256\sigma$. Possible fixes: add a denoising branch or a noise-aware loss, limit $\alpha$ where the estimated SNR is low (in the spirit of SNR-aware), or penalize high-frequency energy in the output that is not present in a smoothed input.

3. **Two luma formulas.** A pixel has $(R', G', B') = (200, 100, 50)$ in 8 bits. Compute its full-range BT.601 luma and its limited-range luma. Show that they encode the same quantity.

**Hint**

Full range: $0.299 \cdot 200 + 0.587 \cdot 100 + 0.114 \cdot 50 = 124.2$. Limited range: $16 + 0.257 \cdot 200 + 0.504 \cdot 100 + 0.098 \cdot 50 \approx 122.7$. Converting back: $(122.7 - 16) \cdot 255/219 \approx 124.2$. The only differences are the scale $219/255$ and the offset 16.

4. **Retinex losses without ground truth.** In Retinex-Net, why is $\mathcal{L}_{ir} = \|R_{\text{low}} - R_{\text{normal}}\|_1$ needed? Describe a trivial decomposition that zeroes the self-reconstruction terms of $\mathcal{L}_{\text{recon}}$ and $\mathcal{L}_{is}$ but is useless, and explain how $\mathcal{L}_{ir}$ rules it out.

**Hint**

Set $I_i \equiv 1$ (perfectly smooth) and $R_i = S_i$. Then each image is reconstructed exactly and the smoothness loss is zero, but nothing has been decomposed. Since $S_{\text{low}} \neq S_{\text{normal}}$, this gives $R_{\text{low}} \neq R_{\text{normal}}$, so $\mathcal{L}_{ir}$ is large. Requiring equal reflectances forces the brightness difference between the pair into the illumination.

5. **GT-mean by hand.** Your model outputs an image with mean 0.40; the ground truth has mean 0.50, and otherwise the output is perfect up to that global scale. What is the GT-mean scale factor, and what is the PSNR before and after rescaling if the image is a constant? Why is the rescaled number not a fair comparison with a method evaluated normally?

**Hint**

$\lambda = 0.50/0.40 = 1.25$. Before: MSE $= 0.1^2 = 0.01$, PSNR $= 20$ dB. After rescaling the output equals the reference, so the error is 0 and PSNR is unbounded. A real method never has the ground-truth mean at test time, so GT-mean numbers reward the right *structure* while ignoring the hardest part of the task, choosing the brightness.

6. **Change the color prior.** In the PyTorch toy, replace the gray-world loss with a loss that keeps the *chromaticity* of the input, for example $\|y/\sum_c y_c - x/\sum_c x_c\|_1$. Train again and compare the coffee result with Figure 7. When is each prior preferable?

**Hint**

Add a small epsilon to the sums. Preserving input chromaticity keeps the red-brown cup red, but it also keeps any color cast of the light source (for example a sodium lamp). Gray world removes casts but desaturates scenes that are genuinely colorful. Real methods often combine a weak color prior with learned color correction from data.

## References


1. Shyandram, “簡介 Deep Learning Low-Light Image Enhancement (LLIE)” (Introduction to deep-learning low-light image enhancement, in Chinese), Medium post, 2024; updated 2026. link
2. A. Foi, M. Trimeche, V. Katkovnik and K. Egiazarian, “Practical Poissonian-Gaussian noise modeling and fitting for single-image raw-data,” *IEEE Transactions on Image Processing*, vol. 17, no. 10, pp. 1737–1754, 2008. doi
3. K. He, J. Sun and X. Tang, “Single image haze removal using dark channel prior,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 12, pp. 2341–2353, 2011. doi
4. X. Guo, Y. Li and H. Ling, “LIME: Low-light image enhancement via illumination map estimation,” *IEEE Transactions on Image Processing*, vol. 26, no. 2, pp. 982–993, 2017. doi
5. ITU-R, “Recommendation BT.601-7: Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios,” 2011. ITU
6. Q. Yan, Y. Feng, C. Zhang, et al., “HVI: A new color space for low-light image enhancement,” in *Proc. IEEE/CVF CVPR*, 2025. arXiv
7. S. M. Pizer, E. P. Amburn, J. D. Austin, et al., “Adaptive histogram equalization and its variations,” *Computer Vision, Graphics, and Image Processing*, vol. 39, no. 3, pp. 355–368, 1987. doi
8. K. Zuiderveld, “Contrast limited adaptive histogram equalization,” in *Graphics Gems IV*, Academic Press, 1994, pp. 474–485. doi
9. E. H. Land, “The retinex theory of color vision,” *Scientific American*, vol. 237, no. 6, pp. 108–128, 1977. doi
10. D. J. Jobson, Z. Rahman and G. A. Woodell, “Properties and performance of a center/surround retinex,” *IEEE Transactions on Image Processing*, vol. 6, pp. 451–462, 1997. doi
11. D. J. Jobson, Z. Rahman and G. A. Woodell, “A multiscale retinex for bridging the gap between color images and the human observation of scenes,” *IEEE Transactions on Image Processing*, vol. 6, no. 7, pp. 965–976, 1997. doi
12. X. Dong, Y. Pang and J. Wen, “Fast efficient algorithm for enhancement of low lighting video,” *ACM SIGGRAPH 2010 Posters*, 2010. SIGGRAPH history doi
13. S. Wang, J. Zheng, H.-M. Hu and B. Li, “Naturalness preserved enhancement algorithm for non-uniform illumination images,” *IEEE Transactions on Image Processing*, vol. 22, no. 9, pp. 3538–3548, 2013. doi
14. C. Lee, C. Lee and C.-S. Kim, “Contrast enhancement based on layered difference representation of 2D histograms,” *IEEE Transactions on Image Processing*, vol. 22, no. 12, pp. 5372–5384, 2013. doi
15. C. Li, C. Guo, L. Han, J. Jiang, M.-M. Cheng, J. Gu and C. C. Loy, “Low-light image and video enhancement using deep learning: A survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 12, pp. 9396–9416, 2022. arXiv
16. C. Wei, W. Wang, W. Yang and J. Liu, “Deep retinex decomposition for low-light enhancement,” in *Proc. British Machine Vision Conference (BMVC)*, 2018. arXiv project
17. Y. Jiang, X. Gong, D. Liu, et al., “EnlightenGAN: Deep light enhancement without paired supervision,” *IEEE Transactions on Image Processing*, vol. 30, pp. 2340–2349, 2021. arXiv
18. C. Guo, C. Li, J. Guo, et al., “Zero-reference deep curve estimation for low-light image enhancement,” in *Proc. IEEE/CVF CVPR*, 2020, pp. 1777–1786. arXiv

19. C. Li, C. Guo and C. C. Loy, “Learning to enhance low-light image via zero-reference deep curve estimation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 8, 2022. arXiv
20. R. Liu, L. Ma, J. Zhang, X. Fan and Z. Luo, “Retinex-inspired unrolling with cooperative prior architecture search for low-light image enhancement,” in *Proc. IEEE/CVF CVPR*, 2021, pp. 10556–10565. arXiv
21. L. Ma, T. Ma, R. Liu, X. Fan and Z. Luo, “Toward fast, flexible, and robust low-light image enhancement,” in *Proc. IEEE/CVF CVPR*, 2022, pp. 5627–5636. arXiv
22. Y. Cai, H. Bian, J. Lin, H. Wang, R. Timofte and Y. Zhang, “Retinexformer: One-stage Retinex-based transformer for low-light image enhancement,” in *Proc. IEEE/CVF ICCV*, 2023. arXiv code
23. Y. Wang, R. Wan, W. Yang, H. Li, L.-P. Chau and A. C. Kot, “Low-light image enhancement with normalizing flow,” in *Proc. AAAI Conference on Artificial Intelligence*, vol. 36, no. 3, pp. 2604–2612, 2022. arXiv
24. X. Yi, H. Xu, H. Zhang, L. Tang and J. Ma, “Diff-Retinex: Rethinking low-light image enhancement with a generative diffusion model,” in *Proc. IEEE/CVF ICCV*, 2023. arXiv
25. J. Hou, Z. Zhu, J. Hou, H. Liu, H. Zeng and H. Yuan, “Global structure-aware diffusion process for low-light image enhancement,” in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2023. arXiv
26. H. Jiang, A. Luo, X. Liu, S. Han and S. Liu, “LightenDiffusion: Unsupervised low-light image enhancement with latent-Retinex diffusion models,” in *Proc. European Conference on Computer Vision* (*ECCV*), 2024. arXiv
27. C. He, C. Fang, Y. Zhang, et al., “Reti-Diff: Illumination degradation image restoration with Retinex-based latent diffusion model,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2025. proceedings
28. M. Gharbi, J. Chen, J. T. Barron, S. W. Hasinoff and F. Durand, “Deep bilateral learning for real-time image enhancement,” *ACM Transactions on Graphics* (Proc. SIGGRAPH), vol. 36, no. 4, art. 118, 2017. doi arXiv
29. H. Zeng, J. Cai, L. Li, Z. Cao and L. Zhang, “Learning image-adaptive 3D lookup tables for high performance photo enhancement in real-time,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 4, 2022. doi arXiv
30. K. G. Lore, A. Akintayo and S. Sarkar, “LLNet: A deep autoencoder approach to natural low-light image enhancement,” *Pattern Recognition*, vol. 61, pp. 650–662, 2017. doi
31. L. Tao, C. Zhu, J. Song, T. Lu, H. Jia and X. Xie, “Low-light image enhancement using CNN and bright channel prior,” in *Proc. IEEE International Conference on Image Processing* (*ICIP*), 2017, pp. 3215–3219. doi
32. S. W. Zamir, A. Arora, S. Khan, et al., “Learning enriched features for real image restoration and enhancement,” in *Proc. European Conference on Computer Vision* (*ECCV*), 2020, pp. 492–511. arXiv
33. X. Xu, R. Wang, C.-W. Fu and J. Jia, “SNR-aware low-light image enhancement,” in *Proc. IEEE/CVF CVPR*, 2022, pp. 17693–17703. doi
34. Z. Cui, K. Li, L. Gu, S. Su, P. Gao, Z. Jiang, Y. Qiao and T. Harada, “You only need 90K parameters to adapt light: A light weight transformer for image enhancement and exposure correction,” in *Proc. British Machine Vision Conference* (*BMVC*), 2022. paper arXiv

35. X. Ding, X. Zhang, N. Ma, J. Han, G. Ding and J. Sun, "RepVGG: Making VGG-style ConvNets great again," in *Proc. IEEE/CVF CVPR*, 2021, pp. 13728–13737. doi arXiv
36. Y. Zhang, J. Zhang and X. Guo, "Kindling the darkness: A practical low-light image enhancer," in *Proc. ACM International Conference on Multimedia (ACM MM)*, 2019, pp. 1632–1640. doi
37. Y. Zhang, X. Guo, J. Ma, W. Liu and J. Zhang, "Beyond brightening low-light images," *International Journal of Computer Vision*, vol. 129, no. 4, pp. 1013–1037, 2021. doi
38. J. Bai, Y. Yin, Q. He, Y. Li and X. Zhang, "RetinexMamba: Retinex-based Mamba for low-light image enhancement," arXiv:2405.03349, 2024. arXiv
39. S.-E. Weng, S.-G. Miaou and R. Christanto, "A lightweight low-light image enhancement network via channel prior and gamma correction," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 39, no. 12, art. 2554013, 2025. doi code manuscript
40. S.-E. Weng, C.-Y. Hsiao, L.-W. Lu, et al., "Rethinking theoretical illumination for efficient low-light image enhancement" (CPGA-Net+), arXiv:2409.05274, 2024. arXiv
41. W. Wu, J. Weng, P. Zhang, X. Wang, W. Yang and J. Jiang, "URetinex-Net: Retinex-based deep unfolding network for low-light image enhancement," in *Proc. IEEE/CVF CVPR*, 2022, pp. 5891–5900. doi
42. Z. Liang, C. Li, S. Zhou, R. Feng and C. C. Loy, "Iterative prompt learning for unsupervised backlit image enhancement," in *Proc. IEEE/CVF ICCV*, 2023. arXiv
43. C. Yang, M. Jin, X. Jia, Y. Xu and Y. Chen, "AdaInt: Learning adaptive intervals for 3D lookup tables on real-time image enhancement," in *Proc. IEEE/CVF CVPR*, 2022, pp. 17522–17531. CVF arXiv
44. C. Yang, M. Jin, Y. Xu, R. Zhang, Y. Chen and H. Liu, "SepLUT: Separable image-adaptive lookup tables for real-time image enhancement," in *Proc. European Conference on Computer Vision (ECCV)*, 2022, pp. 201–217. doi arXiv
45. T. Wang, Y. Li, J. Peng, Y. Ma, X. Wang, F. Song and Y. Yan, "Real-time image enhancer via learnable spatial-aware 3D lookup tables," in *Proc. IEEE/CVF ICCV*, 2021, pp. 2471–2480. CVF arXiv
46. C. Liu, H. Yang, J. Fu and X. Qian, "4D LUT: Learnable context-aware 4D lookup table for image enhancement," *IEEE Transactions on Image Processing*, vol. 32, pp. 4742–4756, 2023. doi arXiv
47. W. Kim and N. I. Cho, "Image-adaptive 3D lookup tables for real-time image enhancement with bilateral grids," in *Proc. European Conference on Computer Vision (ECCV)*, 2024, pp. 91–108. doi ECCV
48. C. Chen, Q. Chen, J. Xu and V. Koltun, "Learning to see in the dark," in *Proc. IEEE/CVF CVPR*, 2018, pp. 3291–3300. arXiv
49. M. V. Conde, G. Geigle and R. Timofte, "InstructIR: High-quality image restoration following human instructions," in *Proc. European Conference on Computer Vision (ECCV)*, 2024. arXiv
50. Z. Cui, G.-J. Qi, L. Gu, S. You, Z. Zhang and T. Harada, "Multitask AET with orthogonal tangent regularity for dark object detection," in *Proc. IEEE/CVF ICCV*, 2021. arXiv
51. K. A. Hashmi, G. Kallempudi, D. Stricker and M. Z. Afzal, "FeatEnHancer: Enhancing hierarchical features for object detection and beyond under low-light vision," in *Proc. IEEE/CVF ICCV*, 2023. arXiv
52. M. Dehghani, A. Arnab, L. Beyer, A. Vaswani and Y. Tay, "The efficiency misnomer," in *Proc. International Conference on Learning Representations (ICLR)*, 2022. ICLR arXiv
53. A. Ignatov, R. Timofte, S. Liu, et al., "Learned smartphone ISP on mobile GPUs with deep

learning, Mobile AI & AIM 2022 challenge: Report," in *Proc. European Conference on Computer Vision (ECCV) Workshops*, 2023, pp. 44–70. doi arXiv
54. Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. doi
55. R. Zhang, P. Isola, A. A. Efros, E. Shechtman and O. Wang, "The unreasonable effectiveness of deep features as a perceptual metric," in *Proc. IEEE/CVF CVPR*, 2018. arXiv
56. K. Ma, K. Zeng and Z. Wang, "Perceptual quality assessment for multi-exposure image fusion," *IEEE Transactions on Image Processing*, vol. 24, no. 11, pp. 3345–3356, 2015. doi
57. V. Vonikakis, "Datasets" (source of the VV test images), personal web page. link
58. A. Mittal, R. Soundararajan and A. C. Bovik, "Making a 'completely blind' image quality analyzer," *IEEE Signal Processing Letters*, vol. 20, no. 3, pp. 209–212, 2013. doi
59. A. Mittal, A. K. Moorthy and A. C. Bovik, "No-reference image quality assessment in the spatial domain," *IEEE Transactions on Image Processing*, vol. 21, no. 12, pp. 4695–4708, 2012. doi
60. Y. Blau, R. Mechrez, R. Timofte, T. Michaeli and L. Zelnik-Manor, "The 2018 PIRM challenge on perceptual image super-resolution," in *Proc. European Conference on Computer Vision (ECCV) Workshops*, 2018. arXiv
61. C. Ma, C.-Y. Yang, X. Yang and M.-H. Yang, "Learning a no-reference quality metric for single-image super-resolution," *Computer Vision and Image Understanding*, vol. 158, pp. 1–16, 2017. doi arXiv
62. D. Hasler and S. E. Süsstrunk, "Measuring colorfulness in natural images," in *Proc. SPIE 5007, Human Vision and Electronic Imaging VIII*, 2003. doi
63. J. Immerkær, "Fast noise variance estimation," *Computer Vision and Image Understanding*, vol. 64, no. 2, pp. 300–302, 1996. doi
64. J. Liao, S. Hao, R. Hong and M. Wang, "GT-Mean loss: A simple yet effective solution for brightness mismatch in low-light image enhancement," in *Proc. IEEE/CVF ICCV*, 2025. arXiv
65. Y. P. Loh and C. S. Chan, "Getting to know low-light images with the Exclusively Dark dataset," *Computer Vision and Image Understanding*, vol. 178, pp. 30–42, 2019. arXiv
66. F. Liu and L. Fan, "A review of advancements in low-light image enhancement using deep learning," *Neurocomputing*, vol. 652, art. 131052, 2025. arXiv
67. J. Liu, D. Xu, W. Yang, M. Fan and H. Huang, "Benchmarking low-light image enhancement and beyond," *International Journal of Computer Vision*, vol. 129, no. 4, pp. 1153–1184, 2021. doi project
68. W. Yang, W. Wang, H. Huang, S. Wang and J. Liu, "Sparse gradient regularized deep Retinex network for robust low-light image enhancement," *IEEE Transactions on Image Processing*, vol. 30, pp. 2072–2086, 2021. doi
69. J. Hai, Z. Xuan, R. Yang, et al., "R2RNet: Low-light image enhancement via Real-low to Real-normal Network," *Journal of Visual Communication and Image Representation*, vol. 90, art. 103712, 2023. arXiv
70. R. Wang, X. Xu, C.-W. Fu, et al., "Seeing dynamic scene in the dark: A high-quality video dataset with mechatronic alignment," in *Proc. IEEE/CVF ICCV*, 2021, pp. 9680–9689. doi code
71. V. Bychkovsky, S. Paris, E. Chan and F. Durand, "Learning photographic global tonal adjustment with a database of input/output image pairs," in *Proc. IEEE CVPR*, 2011, pp. 97–104. dataset

72. J. Cai, S. Gu and L. Zhang, "Learning a deep single image contrast enhancer from multi-exposure images," *IEEE Transactions on Image Processing*, vol. 27, no. 4, pp. 2049–2062, 2018. doi
73. X. Liu, Z. Wu, A. Li, et al., "NTIRE 2024 challenge on low light image enhancement: Methods and results," in *Proc. IEEE/CVF CVPR Workshops*, 2024. arXiv
74. X. Liu, Z. Wu, F.-A. Vasluianu, et al., "NTIRE 2025 challenge on low light image enhancement: Methods and results," in *Proc. IEEE/CVF CVPR Workshops*, 2025. arXiv
75. S. Zheng, Y. Ma, J. Pan, C. Lu and G. Gupta, "Low-light image and video enhancement: A comprehensive survey and beyond," arXiv:2212.10772, 2022. arXiv
76. E. Adhikarla, Y. Liu and B. D. Davison, "Diffusion models for low-light image enhancement: A multi-perspective taxonomy and performance analysis," arXiv:2510.05976, 2025. arXiv
77. J. Yan, C. Tu, W. Zhang, Z. Wang, P. Cao, Q. Lin, Y. Fang, X. Liu, Z. Wu, Z. Zhou and R. Timofte, "NTIRE 2026 challenge on efficient low light image enhancement: Methods and results," NTIRE 2026 workshop report, arXiv:2605.02212, 2026. arXiv
78. O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," in *Proc. Medical Image Computing and Computer-Assisted Intervention* (*MICCAI*), 2015, pp. 234–241. doi arXiv
79. J. Chen, A. Adams, N. Wadhwa and S. W. Hasinoff, "Bilateral guided upsampling," *ACM Transactions on Graphics* (Proc. SIGGRAPH Asia), vol. 35, no. 6, 2016. doi project
80. K. He, J. Sun and X. Tang, "Guided image filtering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 6, pp. 1397–1409, 2013. doi
81. C. Chen and J. Mo, "IQA-PyTorch: PyTorch toolbox for image quality assessment," software (pip package `pyiqa`). GitHub
82. J. Ke, Q. Wang, Y. Wang, P. Milanfar and F. Yang, "MUSIQ: Multi-scale image quality transformer," in *Proc. IEEE/CVF ICCV*, 2021. arXiv
83. J. Wang, K. C. K. Chan and C. C. Loy, "Exploring CLIP for assessing the look and feel of images," in *Proc. AAAI Conference on Artificial Intelligence*, 2023. arXiv
84. V. Kannan, S. Malik, N. C. Babu and R. Soundararajan, "Quality assessment of low-light restored images: A subjective study and an unsupervised model," *IEEE Access*, vol. 11, pp. 68216–68230, 2023. doi arXiv
85. ITU-R, "Recommendation BT.500-15: Methodologies for the subjective assessment of the quality of television images," 2023. ITU
86. R. A. Bradley and M. E. Terry, "Rank analysis of incomplete block designs: The method of paired comparisons," *Biometrika*, vol. 39, no. 3–4, pp. 324–345, 1952. doi
87. H. Talebi and P. Milanfar, "NIMA: Neural image assessment," *IEEE Transactions on Image Processing*, 2018. arXiv
88. S. Yang, T. Wu, S. Shi, et al., "MANIQA: Multi-dimension attention network for no-reference image quality assessment," in *Proc. IEEE/CVF CVPR Workshops*, 2022, pp. 1190–1199. doi arXiv
89. H. Wu, Z. Zhang, W. Zhang, C. Chen, et al., "Q-Align: Teaching LMMs for visual scoring via discrete text-defined levels," in *Proc. International Conference on Machine Learning* (*ICML*), PMLR 235, 2024. arXiv
90. A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," arXiv:1704.04861, 2017. arXiv

91. M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF CVPR*, 2018, pp. 4510–4520. doi arXiv
92. W.-S. Lai, J.-B. Huang, O. Wang, E. Shechtman, E. Yumer and M.-H. Yang, "Learning blind video temporal consistency," in *Proc. European Conference on Computer Vision (ECCV)*, 2018, pp. 179–195. doi arXiv
93. S. W. Hasinoff, D. Sharlet, R. Geiss, A. Adams, J. T. Barron, F. Kainz, J. Chen and M. Levoy, "Burst photography for high dynamic range and low-light imaging on mobile cameras," *ACM Transactions on Graphics* (Proc. SIGGRAPH Asia), vol. 35, no. 6, 2016. doi
94. J. Chen, S. Paris and F. Durand, "Real-time edge-aware image processing with the bilateral grid," *ACM Transactions on Graphics* (Proc. SIGGRAPH), vol. 26, no. 3, art. 103, 2007. doi
95. C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Proc. IEEE International Conference on Computer Vision (ICCV)*, 1998, pp. 839–846. doi
96. S. Paris and F. Durand, "A fast approximation of the bilateral filter using a signal processing approach," *International Journal of Computer Vision*, vol. 81, no. 1, pp. 24–52, 2009. doi
97. H. Wu, S. Zheng, J. Zhang and K. Huang, "Fast end-to-end trainable guided filter," in *Proc. IEEE/CVF CVPR*, 2018, pp. 1838–1847. doi arXiv
98. R. Wang, Q. Zhang, C.-W. Fu, X. Shen, W.-S. Zheng and J. Jia, "Underexposed photo enhancement using deep illumination estimation," in *Proc. IEEE/CVF CVPR*, 2019, pp. 6842–6850. doi CVF
99. Y. Lin, Z. Fu, K. Wen, T. Ye, S. Chen, G. Meng, Y. Wang, et al., "DPLUT: Unsupervised low-light image enhancement with lookup tables and diffusion priors," in *Proc. AAAI Conference on Artificial Intelligence*, vol. 39, no. 5, pp. 5316–5324, 2025. doi arXiv
100. K. Dabov, A. Foi, V. Katkovnik and K. Egiazarian, "Image denoising by sparse 3-D transform-domain collaborative filtering," *IEEE Transactions on Image Processing*, 2007. doi
101. K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *ICLR*, 2015. arXiv
102. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," *CVPR*, 2009. doi
103. A. Brateanu, T. Mu, C. O. Ancuti and C. Ancuti, "Multinex: Lightweight Low-light Image Enhancement via Multi-prior Retinex," in *Proc. IEEE/CVF CVPR*, 2026. paper supplement

# DLCV 10 • Image Fusion: Complementary Images, Useful Outputs

How to combine several images of one scene into a single better image: what infrared–visible, multi-focus and multi-exposure fusion ask for, the classical rules (decision maps, Laplacian pyramids, wavelets, exposure fusion), deep fusion without ground truth, and why fusion metrics disagree and can be gamed.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/

## What you'll learn

- Define the information to retain in infrared–visible, multi-focus or multi-exposure fusion.
- Compare averaging, decision maps, quality weighting and multi-scale blending.
- Understand how losses define learned fusion when no target image exists.
- Assess registration assumptions and downstream-task objectives.
- Interpret conflicting fusion metrics and inspect characteristic artifacts.

## The big picture

Many cameras see the same scene in different ways. A thermal camera sees a person in the dark but not the texture of the wall behind them. A lens focused on the foreground blurs the background. A short exposure keeps the sky but loses the shadows. **Image fusion** takes two or more such images and builds one image that keeps the useful part of each.

> **Explain like I'm five**
>
> Imagine three friends each took a photo of the same playground. One photo is sharp only on the swings, one is sharp only on the slide, and one was taken with special glasses that show which children are warm. You want one picture to put on the wall. You could cut out the best piece of each photo and glue the pieces together, but you have to be careful: if you cut badly, people will see the seams, and if the photos were not taken from the same spot, the pieces will not line up. Image fusion is that cutting and gluing, done carefully, pixel by pixel.

Why it matters:

- **Night driving, surveillance and search-and-rescue** use visible and thermal cameras together. A driver or an operator looks at one screen, so the two views must be combined into one image that shows both the warm pedestrian and the road markings.

- **Photography fuses images too.** A bracket of exposures can become one picture of a bright window and a dark room; focus stacking gives macro and microscope photos that are sharp from front to back.
- **Fusion is unusual among image tasks: there is no correct answer to copy.** Nobody owns an image that is simultaneously thermal and visible. That single fact shapes the whole field: how methods are designed, how networks are trained and why evaluation is so hard.

This chapter builds on two earlier ones. DLCV 09 placed image fusion in the restoration taxonomy in one paragraph; here it gets the full treatment. IP 07 built the wavelet transform that wavelet fusion uses; this chapter adds the Laplacian pyramid, which IP 07 does not cover. For no-reference image quality assessment in general, see DLCV 09; here we focus on the metrics specific to fusion.

## What image fusion is

**Plain version.** Fusion combines complementary observations of the same scene into one image. Define which information the output must preserve from each source.

**Precise version.** Given $K$ registered source images $I_1, \dots, I_K$ of size $H \times W$, a fusion method is a map

$$F = \Phi(I_1, \dots, I_K), \qquad F \in \mathbb{R}^{H \times W \times C},$$

where $F$ is the fused image, $C$ its number of channels (1 for gray, 3 for color) and $\Phi$ the fusion rule, which may be hand-designed or learned. "Registered" means that the same pixel position $(i, j)$ shows the same point of the scene in every source. Many classical rules use a **weighted combination**, either of pixels or of transform coefficients. A pixel-space example is:

$$F(i,j) = \sum_{k=1}^{K} w_k(i,j)\, I_k(i,j), \qquad \sum_{k=1}^{K} w_k(i,j) = 1, \quad w_k(i,j) \geq 0,$$

where $w_k(i, j)$ is the weight of source $k$ at pixel $(i, j)$. This convex-combination model describes many pixel-blending rules. A general learned decoder need not remain inside the sources' pixel-wise range, and remote-sensing fusion may produce a multiband cube rather than an image of the same shape.

This chapter is about **pixel-level** fusion [1], which outputs an image. The alternative is to feed both sources to a network and combine them only at the level of its decisions, which can serve one task well but gives up the fused image that a person can look at and that other tasks can reuse [2]. Deep fusion networks sit in between: they combine features inside the network but still output pixels.

> **Start with one small experiment**
>
> For a first pass, read the task comparison and averaging example, then run the **multi-focus experiment and interactive lab**. Next learn pyramid blending and try exposure fusion. Return to evaluation before choosing a deep model: the metric examples explain why a sharper-looking result can still lose useful evidence. The paper introductions are optional pages for a second pass.

### Three tasks

The three tasks of this chapter differ in what each source contributes, in how hard alignment is, and in whether any ground truth can exist (Figure 1).

| | Infrared–visible | Multi-focus | Multi-exposure |
|---|---|---|---|
| Sources | Thermal (long-wave infrared) image + visible image, two different sensors | Two or more shots of one camera, focused at different depths | Two or more shots of one camera, with different exposure times |
| What each source contributes | IR: thermal contrast under suitable conditions; visible: reflected-light texture, color and detail | The region that is sharp in each shot | Shadows from long exposures, highlights from short ones |
| Typical rule | Keep the salient (warm) targets of IR and the texture of visible | **Select**, per region, the sharpest source | **Blend** with weights that favor well-exposed, detailed pixels |
| Alignment | Hard: two lenses, two resolutions, different appearance | Easy if camera and scene are still | Easy for static scenes; moving objects cause ghosts |
| Ground truth? | No | Only for synthetic data (blur a sharp image) | No (a tone-mapped HDR image is one possible reference, not *the* answer) |

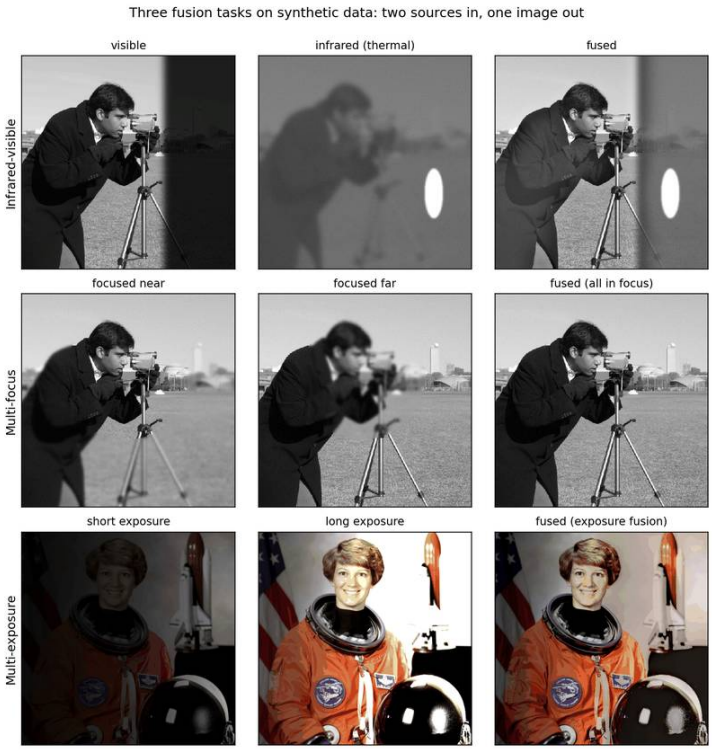



*Figure 1 — The three fusion tasks on synthetic data built from scikit-image test images: infrared–visible*

(*pyramid fusion*), *multi-focus* (*decision map*) *and multi-exposure* (*exposure fusion of the two shots*). *All three are produced by code in this chapter.*

**Infrared–visible fusion.** A thermal camera measures infrared radiance, influenced by temperature, emissivity and reflected radiation, so a sufficiently contrasting target can stand out in darkness; visibility still depends on temperature contrast, atmosphere and sensor characteristics, while a visible camera measures reflected light and shows texture and detail in a way that matches human vision, but little in the dark [3], [2]. The goal is an image in which warm targets stand out and the background keeps its visible detail. Because the two images look so different, "keep the best of each" is ambiguous: the fused image cannot be both as bright as the thermal target and as dark as the visible shadow at the same pixel.

**Multi-focus fusion.** A lens has a limited depth of field: objects at one distance are sharp, nearer and farther ones are blurred. Taking several shots with different focus settings and fusing them gives an **all-in-focus** image (also called focus stacking or extended depth of field). An all-in-focus target is a useful ideal, and source selection approximates it where one source contains the required sharp evidence. At depth boundaries, real defocus can corrupt both sources, so neither is necessarily correct. That is why multi-focus methods often *select* rather than blend.

**Multi-exposure fusion (MEF).** A camera sensor records a limited range of brightness in one exposure. A bracket of exposures covers a wider range. The classical route is **high-dynamic-range (HDR) imaging**: recover the camera response and the scene radiance from the bracket [4], then compress it to a displayable image with a tone-mapping operator. Multi-exposure fusion skips the radiance step and directly blends the 8-bit exposures into a well-exposed 8-bit image [5]. It needs no camera calibration and is fast; the price is that its output is a pleasant picture, not a physical measurement.

### Medical and remote-sensing fusion ask different questions

The three photographic tasks are not the whole field. The desired output can also change its number of channels and its physical interpretation.

| Scenario | Complementary evidence | Desired output | Main check |
|---|---|---|---|
| Medical multimodal fusion | CT emphasizes attenuation-related structure; MRI provides different tissue contrasts; PET/SPECT depict tracer distributions | A display combining anatomical and functional information, or features for a specified task | Anatomical alignment and preservation of modality-specific evidence |
| Pan-sharpening | A finer-resolution broad-band panchromatic image plus coarser multispectral bands | Multispectral data on a finer spatial grid | Spatial detail **and** spectral fidelity |
| Hyperspectral–multispectral fusion | Many narrow spectral bands at coarse resolution plus fewer broad bands at finer resolution | A finer-resolution hyperspectral cube | Agreement with both sensors' spatial and spectral measurements |

The medical setting appears in U2Fusion's PET–MRI evaluation [32] and DDFM's CT/MRI and functional-imaging experiments [41]. It cannot be reduced to "copy the brighter pixel." A bright value has a different meaning in each modality. Display fusion also does not preserve every quantitative value from both sources: retain the original data for measurements and assess any claimed diagnostic benefit through an appropriate task study.

Remote-sensing fusion often asks for an estimated multiband image rather than one pleasant grayscale display [54]. A simplified observation model makes the difference explicit:

$$Y_H = \mathcal{D}(\mathcal{B}(X)) + N_H, \qquad Y_M = \mathcal{R}(X) + N_M.$$

$X \in \mathbb{R}^{H\times W\times L}$ is the unknown fine-resolution cube with $L$ narrow spectral bands. $\mathcal{B}$ spatially blurs each band, $\mathcal{D}$ downsamples it, and $Y_H$ is the coarse hyperspectral observation. $\mathcal{R}$ mixes narrow bands through spectral responses into the smaller number of multispectral bands, producing $Y_M$. $N_H, N_M$ are measurement errors. Real models also need sensor calibration, registration and potentially band-dependent blur.

For pan-sharpening, the fine-resolution observation has only one broad panchromatic band. Detail injection must account for its spectral response; its intensity is not automatically the average of the multispectral channels. At full resolution, the desired cube is usually unavailable as ground truth. Reduced-resolution experiments can create reference comparisons by degrading the available data, but conclusions depend on whether that degradation represents the real sensors.

These tasks share feature extraction and combination ideas with photographic fusion, while requiring different losses and evaluation. A model tested on infrared–visible pairs is not thereby validated for medical fusion or spectral reconstruction. "General fusion" must state whether it means a reusable architecture, separately trained models, or one checkpoint across tasks.

### Registration: the step everything assumes

**Plain version.** Fusion compares the sources pixel by pixel, so the same pixel must show the same point in every source. If one image is shifted by a few pixels, every edge appears twice.

Registration (alignment) estimates the geometric transform between the sources and warps them onto a common grid; for moving scenes it becomes dense motion estimation, which DLCV 09A.3 explains. A static tripod reduces alignment errors, but focus changes, lens distortion and moving objects can still require registration. For infrared–visible pairs it is a real problem, and a prerequisite of fusion [3]: the two sensors differ in viewpoint, pixel distribution and resolution, so accurately registered pairs are hard to obtain, yet most existing methods need pixel-level registered pairs and existing benchmarks consist of pixel-aligned pairs [2]. We return to methods that relax this assumption at the end of the deep-learning section.

## The simplest rule: averaging, and why it fades contrast

**Plain version.** Averaging the sources keeps a bit of everything, but anything that appears in only one source comes out at half strength. Fusion's whole difficulty is to avoid that.

**Precise version.** Take two sources $I_A$ and $I_B$ that share a background $b$, and suppose a feature of amplitude $a$ (an edge, a texture, a warm target) appears only in $I_A$: $I_A = b + a\,s$, $I_B = b$, where $s$ is the feature's shape. The average is

$$F_{\text{avg}} = \tfrac{1}{2}(I_A + I_B) = b + \tfrac{a}{2}\,s,$$

so the feature's contrast is halved. With $K$ sources and the feature in one of them, it drops to $a/K$. In multi-focus fusion it is worse: the blurred source adds a blurred copy of the edge, so the average shows a sharp edge with a soft halo. In infrared–visible fusion, a target that is bright in IR and dark in the visible image ends up gray (Figure 6, last column). In multi-exposure fusion, averaging an under- and an over-exposed shot gives a flat, low-contrast image.

The fix is to make the weights $w_k(i,j)$ depend on *where* each source is informative: high weight for the sharp source, the warm target, the well-exposed pixel. The rest of the classical section is about how to measure "informative" and how to change the weights without creating seams.

## Multi-focus fusion: focus measures and decision maps

**Plain version.** Sharp regions have strong fine detail; blurred ones do not. Measure the detail around every pixel in every shot, pick the shot with the most, and copy its pixel.

**Precise version.** A **focus measure** $M_k(i,j) \geq 0$ scores how much high-frequency detail source $k$ has around pixel $(i,j)$. A simple choice, of the kind used in classical multi-focus fusion [5], is the local energy of the Laplacian,

$$M_k = G_\sigma * (\nabla^2 I_k)^2,$$

where $\nabla^2 I_k = \partial^2 I_k/\partial x^2 + \partial^2 I_k/\partial y^2$ is the Laplacian of the source (a second-derivative filter, see IP 03), the square makes it an energy, and $G_\sigma *$ denotes smoothing with a Gaussian of standard deviation $\sigma$, which averages the energy over a neighborhood. Blur removes high frequencies, so the in-focus source has the larger $M_k$. The **decision map** records the winner,

$$D(i,j) = \arg\max_k M_k(i,j), \qquad F(i,j) = I_{D(i,j)}(i,j),$$

where $D(i,j) \in \{1,\dots,K\}$ is the index of the source chosen at pixel $(i,j)$.

A raw decision map is noisy: in flat regions (sky, a plain wall) no source has detail, and the winner is decided by noise. Two post-processing steps are standard. **Cleaning** removes small islands, for example with a majority vote (median filter) or morphological operations, on the assumption that in-focus regions are large and connected. **Softening** blurs the binary map slightly near its boundaries so that the switch from one source to another does not leave a visible seam.

The snippet builds a synthetic pair from the camera test image: inside a disk shot A is sharp, outside it shot B is sharp, and defocus is modeled as a Gaussian blur. Because we made the blur ourselves, we have the all-in-focus truth and can measure PSNR, which is impossible with real multi-focus pairs.

```
import numpy as np
```

```
from scipy import ndimage as ndi
from skimage import data, img_as_float
from skimage.metrics import peak_signal_noise_ratio as psnr

rng = np.random.default_rng(0)
x = img_as_float(data.camera())                      # 512 x 512: the all-in-focus "truth"
yy, xx = np.mgrid[:512, :512]
near = (yy - 200) ** 2 + (xx - 250) ** 2 < 150 ** 2   # the part that is in focus in shot A
blurred = ndi.gaussian_filter(x, 3.0)                # defocus modeled as a Gaussian blur
A = np.where(near, x, blurred) + 0.01 * rng.standard_normal(x.shape)  # sharp inside the disk
B = np.where(near, blurred, x) + 0.01 * rng.standard_normal(x.shape)  # sharp outside it

def focus_measure(img, s=2.0):
    """Local energy of the Laplacian: large where the image has sharp detail."""
    return ndi.gaussian_filter(ndi.laplace(img) ** 2, s)

raw = focus_measure(A) > focus_measure(B)                # decision map: True = take A
clean = ndi.median_filter(raw.astype(float), 31) > 0.5   # majority vote removes small
 ↪  islands
w = ndi.gaussian_filter(clean.astype(float), 2.0)        # soften the seam a little

results = {"average": (A + B) / 2,
           "raw decision map": np.where(raw, A, B),
           "cleaned, soft map": w * A + (1 - w) * B}
for name, F in results.items():
    print(f"{name:18s} PSNR = {psnr(x, np.clip(F, 0, 1), data_range=1):.2f} dB")
textured = ndi.gaussian_filter(np.abs(ndi.laplace(x)), 2) > 0.01   # where the choice matters
for name, m in [("raw", raw), ("cleaned", clean)]:
    print(f"{name:7s} map agrees with the true focus map on {np.mean(m == near):.1%} of all
     ↪  pixels, "
          f"{np.mean((m == near)[textured]):.1%} of textured pixels")
```

```
average            PSNR = 29.97 dB
raw decision map   PSNR = 39.22 dB
cleaned, soft map  PSNR = 39.81 dB
raw     map agrees with the true focus map on 80.9% of all pixels, 91.3% of textured pixels
cleaned map agrees with the true focus map on 91.2% of all pixels, 98.4% of textured pixels
```

The decision map wins by about 10 dB over averaging, because it copies sharp pixels instead of mixing them with blurred ones (Figure 2). Cleaning makes the map much more faithful (98.4% instead of 91.3% agreement on textured pixels), but the PSNR gain is small, 0.6 dB, because most of the raw map's mistakes are in flat regions where shots A and B look the same anyway. These numbers are from our toy, not from any paper.

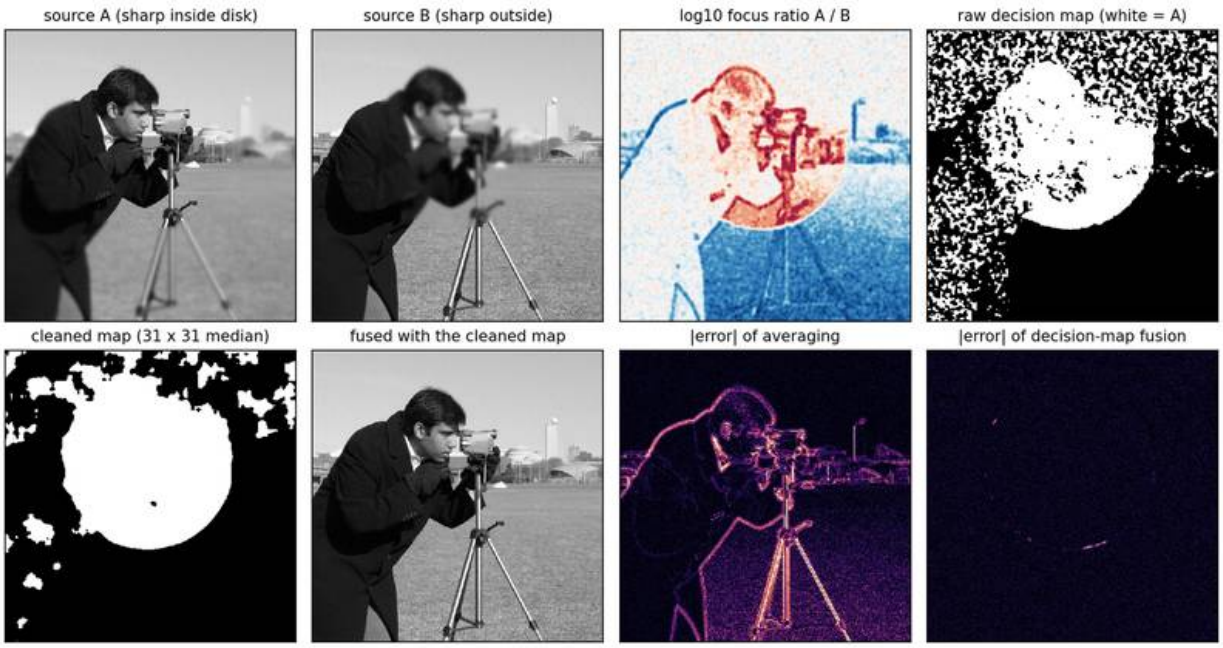



*Figure 2 — Multi-focus fusion with a focus measure. The focus ratio is decided by noise in the flat sky, which the median filter mostly cleans up; there both sources are equally good, so the leftover blobs do no harm. The error maps (same color scale) show why selection beats averaging: averaging leaves an error at every edge.*

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

**What the toy hides.** Real defocus is not a Gaussian blur pasted onto a mask. Near a depth edge, the blurred foreground spreads over the background in the far-focused shot, so *neither* shot is sharp there. This **defocus spread effect** is why methods that do well on simulated data or on the popular Lytro pairs often fail on real photographs with strong defocus spread [6], [7]; Figure 6 shows it on a layered version of our toy.

*Online only: The defocus spread effect — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/notes/the-defocus-spread-effect/*

## Multi-scale fusion: Laplacian pyramids and wavelets

**Plain version.** Fuse at multiple scales: select useful fine detail and blend coarse brightness smoothly to reduce visible transitions.

### The Laplacian pyramid

**Precise version.** The **Laplacian pyramid** of Burt and Adelson [8] represents an image as a stack of band-pass images plus a small low-pass residual. Let $G_0 = I$ and

$$G_{\ell+1} = \mathrm{REDUCE}(G_\ell), \qquad L_\ell = G_\ell - \mathrm{EXPAND}(G_{\ell+1}), \quad \ell = 0, \dots, N-1,$$

where REDUCE blurs with a small low-pass kernel and keeps every second row and column (OpenCV's `pyrDown`), EXPAND upsamples by a factor of 2 and interpolates with the same kernel (`pyrUp`), $G_\ell$ is the **Gaussian pyramid** level $\ell$ (a blurred, smaller copy of the image) and $L_\ell$ the **Laplacian pyramid** level $\ell$, the detail that $G_\ell$ has and the coarser level does not. The top of the pyramid is the residual $L_N = G_N$. Level $\ell$ has $2^{-\ell}$ times the width and height of the image. The image is recovered exactly by the **collapse**

$$G_N = L_N, \qquad G_\ell = L_\ell + \mathrm{EXPAND}(G_{\ell+1}), \quad \ell = N-1, \dots, 0,$$

and $I = G_0$. Exactness holds for any EXPAND, because each $L_\ell$ stores exactly what EXPAND fails to predict. The name comes from the authors' observation that the encoding is equivalent to sampling the image with Laplacian operators of many scales [8].

*Online only: Building and collapsing a Laplacian pyramid — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/notes/building-and-collapsing-a-laplacian-pyramid/*

### Fusing two pyramids

Pyramid fusion decomposes each source, combines the pyramids level by level, and collapses the result. Classic pyramid fusion selects the most salient Laplacian coefficients of the sources [9], [5]; a simple version of the rule has two parts:

$$L_\ell^F(i,j) = \begin{cases} L_\ell^A(i,j) & \text{if } |L_\ell^A(i,j)| \geq |L_\ell^B(i,j)|, \\ L_\ell^B(i,j) & \text{otherwise,} \end{cases} \qquad L_N^F = \tfrac{1}{2}(L_N^A + L_N^B),$$

where $L_\ell^A$, $L_\ell^B$ and $L_\ell^F$ are level $\ell$ of the pyramids of source A, source B and the fused image. The **max-abs** rule keeps, at every position and scale, the coefficient with the larger magnitude, which is the stronger local feature; the coarse residual is averaged because it carries overall brightness, where no source should simply win. Selecting per coefficient at every scale means the switch between sources is spread over the support of the pyramid's filters, which is wider at coarser levels; this is the multiresolution blending idea that Burt and Adelson introduced for joining image mosaics without visible seams [10]. Burt and Kolczynski later refined pyramid fusion to make it more robust to small shifts and noise and to handle features of similar salience but opposite contrast [9].

The snippet continues from the multi-focus example (it uses `A`, `B`, `x` and `psnr`).

```
import cv2
import pywt
```

```
def laplacian_pyramid(img, levels=5):
    G = [img.astype(np.float32)]
    for _ in range(levels):
        G.append(cv2.pyrDown(G[-1]))                      # blur, then keep every 2nd pixel
    L = [G[i] - cv2.pyrUp(G[i + 1], dstsize=G[i].shape[::-1]) for i in range(levels)]
    return L + [G[-1]]                                    # band-pass details + coarse base

def collapse(L):
    img = L[-1]
    for detail in reversed(L[:-1]):
        img = cv2.pyrUp(img, dstsize=detail.shape[::-1]) + detail
    return img

def pick_stronger(p, q):
    return np.where(np.abs(p) >= np.abs(q), p, q)         # "max-abs" selection rule

def pyramid_fuse(a, b, levels=5):
    La, Lb = laplacian_pyramid(a, levels), laplacian_pyramid(b, levels)
    fused = [pick_stronger(p, q) for p, q in zip(La[:-1], Lb[:-1])]
    fused.append((La[-1] + Lb[-1]) / 2)                   # base: average the coarse level
    return collapse(fused).astype(float)

def wavelet_fuse(a, b, wavelet="db2", levels=4):
    ca, cb = pywt.wavedec2(a, wavelet, level=levels), pywt.wavedec2(b, wavelet, level=levels)
    fused = [(ca[0] + cb[0]) / 2]                         # approximation band: average
    for da, db in zip(ca[1:], cb[1:]):                    # (H, V, D) detail bands per level
        fused.append(tuple(pick_stronger(p, q) for p, q in zip(da, db)))
    return pywt.waverec2(fused, wavelet)

print(f"pyramid reconstruction error: {np.abs(collapse(laplacian_pyramid(A)) -
 ↪  A).max():.1e}")
for name, F in [("Laplacian pyramid", pyramid_fuse(A, B)), ("wavelet (db2)", wavelet_fuse(A,
 ↪  B))]:
    print(f"{name:18s} PSNR = {psnr(x, np.clip(F, 0, 1), data_range=1):.2f} dB")
```

```
pyramid reconstruction error: 5.9e-08
Laplacian pyramid  PSNR = 37.60 dB
wavelet (db2)      PSNR = 36.43 dB
```

The reconstruction error of about $10^{-7}$ is single-precision rounding: the pyramid itself is lossless. Pyramid fusion beats averaging by more than 7 dB but stays about 2 dB below the decision map. The reason is visible in Figure 3: the per-coefficient choice is noisy, especially at fine levels, where the blurred source's noise sometimes wins in flat areas. Comparing smoothed coefficient energies instead of single coefficients, the "area-based" selection of Li et al. [11], closes most of the gap (Exercise 3). The general lesson: when the true answer is a selection (multi-focus), spatially coherent decisions beat per-coefficient ones.

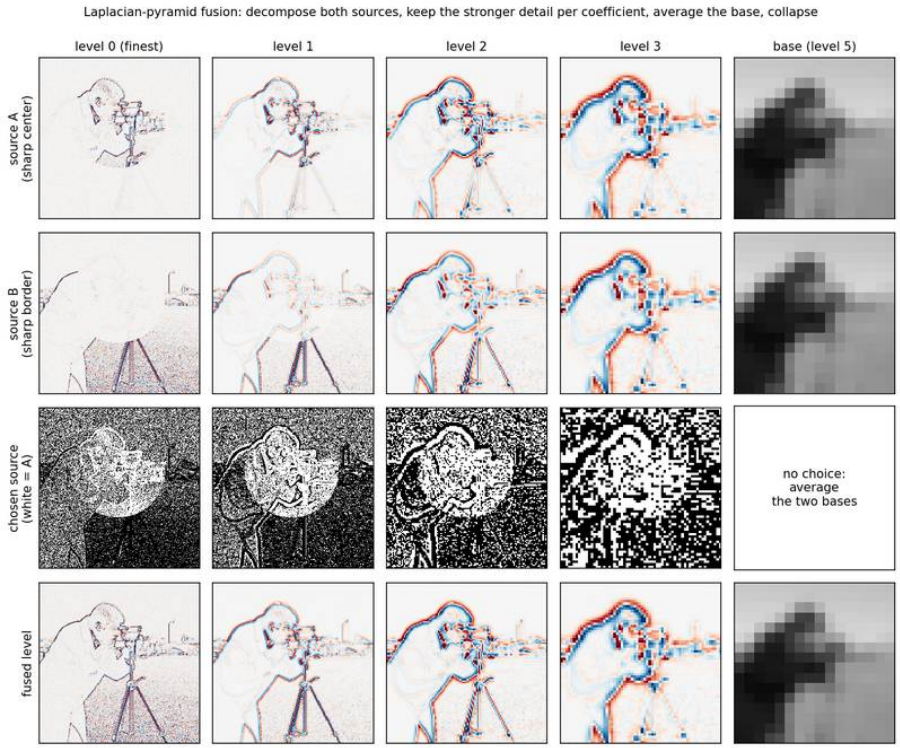



*Figure 3 — Laplacian-pyramid fusion of the multi-focus pair. Each source is split into band-pass levels (red and blue are positive and negative coefficients); the max-abs rule picks the stronger coefficient per position (row 3, white = source A), the base is averaged, and collapsing the fused pyramid gives the result.*

### Wavelet fusion

Wavelet fusion follows the same plan with the discrete wavelet transform of IP 07 instead of the pyramid: decompose both sources, keep the stronger detail coefficient in each horizontal, vertical and diagonal band, average the approximation band, and invert. Li, Manjunath and Mitra [11] proposed it with an **area-based** maximum-selection rule (compare activity over a small window, not single coefficients) and a **consistency verification** step that makes a coefficient follow the choice of its neighbors, and argued that the wavelet transform's compactness, orthogonality and directional selectivity make it better suited than the Laplacian pyramid. In our toy, the simple per-coefficient rule gives 36.43 dB, about 1 dB below the pyramid. One known weakness is that the decimated wavelet transform is not shift-invariant: shifting the image by one pixel changes its coefficients, so a per-coefficient decision can change too. Greater shift invariance was also one of the aims of Burt and Kolczynski's refinement of pyramid fusion [9].

## Exposure fusion: well-exposed images without HDR

**Plain version.** Exposure fusion weights contrast, saturation and well-exposedness, then blends through image pyramids to reduce seams.

**Precise version.** Mertens, Kautz and Van Reeth [5] compute, for exposure $k$ at pixel $(i, j)$, three **quality measures**:

- **contrast** $C_{ij,k}$: the absolute response of a Laplacian filter applied to the grayscale exposure (high at edges and texture);
- **saturation** $S_{ij,k}$: the standard deviation of the R, G and B values at the pixel (vivid colors have a large spread, washed-out ones a small spread);
- **well-exposedness** $E_{ij,k} = \prod_{c\in\{R,G,B\}} \exp\!\left(-\frac{(I^c_{ij,k}-0.5)^2}{2\sigma^2}\right)$: a Gaussian bump around mid-gray, applied to each color channel $I^c_{ij,k} \in [0,1]$ and multiplied, with $\sigma = 0.2$ in their implementation.

They are combined by a product with exponents and normalized over the $N$ exposures:

$$W_{ij,k} = (C_{ij,k})^{\omega_C}\,(S_{ij,k})^{\omega_S}\,(E_{ij,k})^{\omega_E}, \qquad \hat{W}_{ij,k} = \frac{W_{ij,k}}{\sum_{k'=1}^{N} W_{ij,k'}},$$

where $\omega_C, \omega_S, \omega_E \geq 0$ set the influence of each measure (an exponent of 0 switches a measure off) and $\hat{W}_{ij,k}$ is the normalized weight, which sums to 1 over the exposures at each pixel. The product acts like an "and": a pixel gets a high weight only if it is detailed *and* colorful *and* well exposed [5].

**Why a pyramid.** Blending pixels directly, $R_{ij} = \sum_k \hat{W}_{ij,k} I_{ij,k}$, produces visible seams wherever the weights change quickly, because the exposures have very different absolute brightness [5]. Smoothing the weights with a Gaussian removes the seams but creates halos around edges. Exposure fusion instead blends **per pyramid level**: each Laplacian level of each exposure is weighted by the matching level of a Gaussian pyramid of its weight map,

$$L\{R\}_\ell = \sum_{k=1}^{N} G\{\hat{W}_k\}_\ell \; L\{I_k\}_\ell,$$

where $L\{\cdot\}_\ell$ is level $\ell$ of a Laplacian pyramid and $G\{\cdot\}_\ell$ level $\ell$ of a Gaussian pyramid; the result $R$ is obtained by collapsing $L\{R\}$ [5]. Fine detail is blended with sharp weights and coarse brightness with very smooth weights, which reduces seams and halos when alignment and pyramid settings are appropriate.

The snippet builds a synthetic scene with an 8-stop light falloff (dark on the left, bright on the right), photographs it with three exposure times through a simple camera model (gamma, clipping, 8-bit quantization), and fuses the bracket. It checks the result against OpenCV's implementation, `cv2.createMergeMertens`.

```
import numpy as np
import cv2
from skimage import data, img_as_float
```

```
# A synthetic high-dynamic-range scene: linear radiance times a light falloff of 8 stops.
rgb = img_as_float(data.astronaut()) ** 2.2                      # undo display gamma ->
 ↪  linear
light = 2.0 ** np.linspace(-6, 2, rgb.shape[1])[None, :, None]    # dark left, bright right
radiance = rgb * light

def camera(E, t):
    """Expose for time t, apply a 1/2.2 display gamma, clip and quantize to 8 bits."""
    return np.round(np.clip((t * E) ** (1 / 2.2), 0, 1) * 255) / 255

stack = [camera(radiance, t) for t in (1 / 8, 1.0, 8.0)]          # under, middle, over

def mertens_weights(imgs, sigma=0.2, eps=1e-12):
    W = []
    for I in imgs:
        gray = I.mean(axis=2).astype(np.float32)
        C = np.abs(cv2.Laplacian(gray, cv2.CV_32F))               # contrast
        S = I.std(axis=2)                                         # saturation
        E = np.prod(np.exp(-(I - 0.5) ** 2 / (2 * sigma ** 2)), axis=2)  # well-exposedness
        W.append(C * S * E + eps)
    W = np.array(W)
    return W / W.sum(axis=0)                                      # weights sum to 1 per
     ↪  pixel

def gauss_pyr(img, n):
    G = [img.astype(np.float32)]
    for _ in range(n):
        G.append(cv2.pyrDown(G[-1]))
    return G

def lap_pyr(img, n):
    G = gauss_pyr(img, n)
    return [G[i] - cv2.pyrUp(G[i + 1], dstsize=G[i].shape[1::-1]) for i in range(n)] +
     ↪  [G[-1]]

def exposure_fusion(imgs):
    n = int(np.log2(min(imgs[0].shape[:2])))                      # full depth: 9 levels for
     ↪  512 px
    fused = None
    for I, Wk in zip(imgs, mertens_weights(imgs)):
        level = [g[..., None] * l for g, l in zip(gauss_pyr(Wk, n), lap_pyr(I, n))]
        fused = level if fused is None else [f + l for f, l in zip(fused, level)]
    out = fused[-1]
    for detail in reversed(fused[:-1]):
        out = cv2.pyrUp(out, dstsize=detail.shape[1::-1]) + detail
    return np.clip(out, 0, 1)

R = exposure_fusion(stack)
naive = np.clip(np.sum(mertens_weights(stack)[..., None] * np.array(stack), axis=0), 0, 1)
ref = cv2.createMergeMertens(1.0, 1.0, 1.0).process(
    [(255 * I[..., ::-1]).round().astype(np.uint8) for I in stack])[..., ::-1]   # OpenCV
     ↪  uses BGR
for name, I in [("under", stack[0]), ("middle", stack[1]), ("over", stack[2]),
                ("per-pixel blend", naive), ("pyramid blend", R)]:
```

```
    ok = np.mean(np.all((I > 0.05) & (I < 0.95), axis=2))        # no channel near black or
     ↪  clipped
    print(f"{name:15s} well-exposed pixels: {100 * ok:4.1f}%")
print(f"mean |ours - OpenCV MergeMertens| = {np.abs(R - np.clip(ref, 0, 1)).mean():.4f}")
```

```
under           well-exposed pixels: 45.8%
middle          well-exposed pixels: 58.7%
over            well-exposed pixels: 41.2%
per-pixel blend well-exposed pixels: 72.3%
pyramid blend   well-exposed pixels: 73.8%
mean |ours - OpenCV MergeMertens| = 0.0027
```

Our implementation agrees with OpenCV's to within 0.003 on a $[0, 1]$ scale; the small difference presumably comes from implementation details such as pyramid depth, border handling and input rounding. The crude "well-exposed pixels" count rises from at most 59% for any single exposure to 74% after fusion. Note what the count misses: the per-pixel blend scores almost as well (72%) yet looks badly broken, with seams and speckle where the weights jump (Figure 4, bottom left). A single number does not see artifacts, a theme the evaluation section returns to.

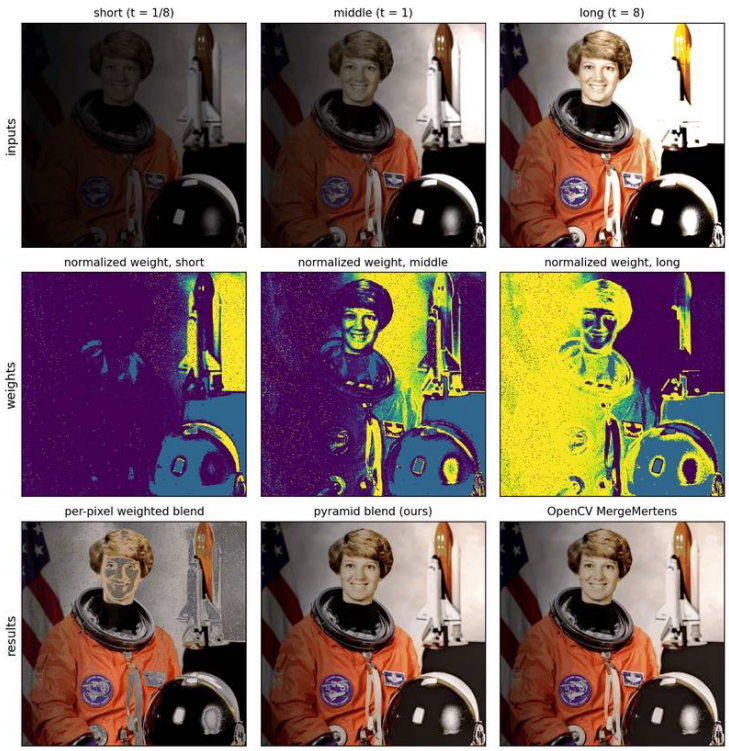



*Figure 4 — Exposure fusion of a synthetic bracket. The weights pick the well-exposed, detailed parts of each exposure (in flat black areas all weights are tiny and the normalization gives each exposure one third). Blending pixels directly with these weights fails; blending pyramid levels works and matches OpenCV's implementation.*

**Exposure fusion is not HDR.** The output keeps no radiance values, so it cannot be relit or re-exposed, and with strongly varying brightness the multiresolution blend can create a spurious low-frequency brightness change; Mertens et al. note that a taller pyramid partly solves it [5]. Exposure fusion also assumes perfectly aligned exposures, possibly after registration [5]; with camera or object motion, the result shows ghosts (Figure 6). Finally, Mertens et al. compare their blending with earlier pyramid fusion that *selects* the most salient coefficients and find the selection results too dark and with color shifts on exposure stacks [5]: for exposures, blending beats selecting, the opposite of multi-focus fusion.

*Online only: Exposure fusion (Mertens, Kautz and Van Reeth) — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/papers/exposure-fusion-mertens-kautz-and-van-reeth/*

## Two more classical tools: guided filtering and sparse codes

**Plain version.** Two later ideas improved the classical rules: make the weight maps follow object edges using an edge-aware filter, and describe image patches with a learned dictionary so that "how much detail" is measured on a richer representation.

**Guided-filtering fusion (GFF).** Li, Kang and Hu [12] split each source into a **base layer** (large-scale intensity variations) and a **detail layer** (small-scale detail), and fuse each layer with a weighted average whose weight maps are refined by the guided filter [13] so that they respect spatial consistency, that is, neighboring pixels with similar intensities get similar weights. The guided filter is explained in IP 03, and its learned and fast variants in DLCV 09C. The authors report state-of-the-art results at the time for multispectral, multi-focus, multimodal and multi-exposure images with one method [12]. The design shows the general recipe of the classical era: a two- or multi-scale decomposition, an activity measure, and a smoothing step that turns the activity into clean weights.

**Sparse-representation (SR) fusion.** Yang and Li [14] represent the sources with sparse coefficients over an overcomplete dictionary, combine the coefficients with a choose-max rule, and reconstruct the fused image from the combined coefficients and the dictionary; by changing the approximation criterion of the sparse coding, the same scheme also restores noisy sources while fusing them. Later work learned the dictionary from the source patches themselves [15]. SR methods were among the strongest classical multi-focus methods, but they are slow: in one benchmark they took more than seven minutes to fuse an image pair [7].

*Online only: Sparse-representation fusion — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/notes/sparse-representation-fusion/*

## Evaluating fusion without ground truth

**Plain version.** Without a correct fused image to compare with, fusion is scored by rules of thumb: does the result have a lot of detail, a lot of contrast, a lot of information from the sources? Each rule captures one aspect, so they disagree, and a method can be tuned to look good on the chosen ones.

This section comes before deep learning on purpose: deep fusion networks are trained with losses built from the same ingredients as these metrics, so understanding the metrics explains the losses.

### Why fusion has no single right answer

For multi-focus fusion on synthetic data, the all-in-focus image is known and PSNR or SSIM [16] against it is meaningful, as in our first snippet. For infrared–visible and multi-exposure fusion, no such image exists, and benchmark studies agree that no single metric is better than all others [17], [18], [7]. Benchmarks therefore report many metrics: VIFB implements 13 [17], and the multi-exposure benchmark MEFB and the multi-focus benchmark MFIFB 20 each [18], [7]. VIFB groups them into four families: information-theory-based, image-feature-based, structural-similarity-based and human-perception-based [17]. Its results show the consequence: one method wins the information-theory metrics, another the feature-based ones and a third the structural ones, and no method wins all [17].

*Online only: VIFB: a visible and infrared image fusion benchmark — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/papers/vifb-a-visible-and-infrared-image-fusion-benchmark/*

### Metrics that look only at the fused image

Let $F$ be the fused image, of size $M \times N$, with values in $[0, 1]$.

**Entropy (EN)** measures how spread out the histogram is:

$$\mathrm{EN} = -\sum_{l=0}^{L-1} p_l \log_2 p_l,$$

where $L$ is the number of gray levels (histogram bins) and $p_l$ the fraction of pixels in bin $l$. EN is in bits; a larger value means the gray levels are used more evenly. It is easily affected by noise, so benchmark studies use it only as an auxiliary metric [18].

**Standard deviation (SD)** measures global contrast, $\mathrm{SD} = \sqrt{\frac{1}{MN}\sum_{i,j}(F(i,j)-\mu)^2}$, where $\mu$ is the mean of $F$ (some implementations omit the $\frac{1}{MN}$, which changes the scale; rankings are unchanged only when the compared images have the same dimensions).

**Spatial frequency (SF)** measures the overall activity of neighboring differences [19], [18]:

$$\mathrm{SF} = \sqrt{\mathrm{RF}^2 + \mathrm{CF}^2}, \qquad \mathrm{RF}^2 = \frac{1}{MN}\sum_{i,j}\big(F(i,j) - F(i,j-1)\big)^2, \quad \mathrm{CF}^2 = \frac{1}{MN}\sum_{i,j}\big(F(i,j) - F(i-1,j)\big)^2,$$

where RF (row frequency) uses horizontal differences and CF (column frequency) vertical ones; normalizations differ between implementations. Large SF means many strong edges and much texture.

These three never look at the sources. They reward "more": more gray levels, more contrast, more edges. Noise supplies all three for free.

### Metrics that compare with the sources

**Mutual information (MI)** measures how much the fused image tells about each source [20]:

$$\mathrm{MI} = \mathrm{MI}_{A,F} + \mathrm{MI}_{B,F}, \qquad \mathrm{MI}_{X,F} = \sum_{x,f} p_{X,F}(x,f) \log_2 \frac{p_{X,F}(x,f)}{p_X(x)\, p_F(f)},$$

where $p_{X,F}$ is the joint histogram of source $X \in \{A, B\}$ and $F$ (normalized to sum to 1), and $p_X$, $p_F$ its marginals. MI is large when the gray level of $F$ predicts the gray level of each source well (see IT 01 for the definition). Hossny, Nahavandi and Creighton showed analytically that this classic MI sum is not a valid fusion performance measure on its own [21], and benchmark toolkits implement a normalized version attributed to them, which divides each term by the sum of the two entropies [18].

*Online only: Mutual information as a fusion metric — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/notes/mutual-information-as-a-fusion-metric/*

**Gradient-based fusion performance** $Q^{AB/F}$ (Xydeas and Petrović [22]) measures how much **edge information** of the sources survives in $F$. At every pixel it compares the edge strength $g$ and orientation $\alpha$ (from image gradients) of each source $X$ with those of $F$, turns the comparison into a preservation value $Q^{XF}(i,j) = Q_g^{XF}(i,j)\, Q_\alpha^{XF}(i,j)$, the product of an edge-strength and an orientation preservation value, and averages with weights $w^X(i,j)$ that express the importance of each source [18]:

$$Q^{AB/F} = \frac{\sum_{i,j} \left( Q^{AF}(i,j)\, w^A(i,j) + Q^{BF}(i,j)\, w^B(i,j) \right)}{\sum_{i,j} \left( w^A(i,j) + w^B(i,j) \right)}.$$

A value near 1 means the edges of both sources are kept with their strength and direction. The snippet below implements a simplified version, which we call $Q_G$: the strength preservation is the ratio of the smaller to the larger gradient magnitude, the orientation preservation falls linearly with the angle between the edges, and the weights are the source gradient magnitudes. Use a reference implementation when you report $Q^{AB/F}$.

**Structure-based and perception-based metrics.** Piella and Heijmans [23] proposed three variants of a fusion quality metric based on an image quality index of Wang and Bovik; they need no ground-truth image because they compare $F$ with the sources, and the authors found them consistent with subjective evaluations. VIFF [24] adapts the visual information fidelity criterion to fusion and computes it over several resolutions. For multi-exposure fusion, **MEF-SSIM** [25] was designed from a subjective study: its authors found that general fusion metrics predicted the perceived quality of multi-exposure fusion poorly, and built an SSIM-style index around a patch structure consistency measure that correlates much better with human judgments. Liu et al. [26] compared twelve fusion metrics over six multiresolution fusion algorithms for night-vision applications.

**VIF is an information-transfer model, not raw entropy.** It estimates how much modeled source information survives through the candidate image and a visual channel [55]. Fusion variants aggregate source comparisons differently, so values above one or different paper scales require checking the implementation. A source still is not a true fused target.

*Online only: VIF and source-referenced fusion scores — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/notes/vif-and-source-referenced-fusion-scores/*

### A toy that fools the metrics

The snippet builds an infrared–visible pair from the camera image: the visible image has a deep shadow on the right in which a person is invisible; the thermal image is smooth, with little texture, and the person glows. It scores four candidates: no fusion at all (the visible image), the average, pyramid fusion (max-abs details, and the brighter of the two coarse bases so that the warm target keeps its level), and the average with added noise. As an independent check of what a user cares about, it also measures the **person contrast**: the mean of $F$ on the person minus the mean on a ring around them.

```
import numpy as np
import cv2
from scipy import ndimage as ndi
from skimage import data, img_as_float

rng = np.random.default_rng(1)
scene = img_as_float(data.camera())
yy, xx = np.mgrid[:512, :512]
person = ((yy - 330) / 60) ** 2 + ((xx - 420) / 22) ** 2 < 1        # a warm "person", 120 px
 ↪  tall
shadow = ndi.gaussian_filter((xx > 340).astype(float), 8)          # a dark area on the right
vis = scene * (1 - 0.85 * shadow)                                  # visible: person hidden
 ↪  in shadow
ir = 0.25 + 0.3 * ndi.gaussian_filter(scene, 6)                    # thermal: smooth, little
 ↪  texture
ir = np.clip(ir + 0.6 * ndi.gaussian_filter(person.astype(float), 2)  # the warm body glows
             + 0.01 * rng.standard_normal(ir.shape), 0, 1)

def lap_pyr(img, n=5):
    G = [img.astype(np.float32)]
    for _ in range(n):
        G.append(cv2.pyrDown(G[-1]))
    return [G[i] - cv2.pyrUp(G[i + 1], dstsize=G[i].shape[::-1]) for i in range(n)] + [G[-1]]

def collapse(L):
    img = L[-1]
    for d in reversed(L[:-1]):
        img = cv2.pyrUp(img, dstsize=d.shape[::-1]) + d
    return img

def pyramid_fuse(a, b):                    # max-abs details, and the brighter coarse base
    La, Lb = lap_pyr(a), lap_pyr(b)
    out = [np.where(np.abs(p) >= np.abs(q), p, q) for p, q in zip(La[:-1], Lb[:-1])]
    out.append(np.maximum(La[-1], Lb[-1]))
    return np.clip(collapse(out), 0, 1).astype(float)

def entropy(F, bins=256):
    p = np.histogram(F, bins, (0, 1))[0] / F.size
```

```
    p = p[p > 0]
    return -np.sum(p * np.log2(p))

def spatial_frequency(F):
    rf = np.mean(np.diff(F, axis=1) ** 2)           # row frequency: horizontal differences
    cf = np.mean(np.diff(F, axis=0) ** 2)           # column frequency: vertical differences
    return np.sqrt(rf + cf)

def mutual_info(X, F, bins=64):
    pxy = np.histogram2d(X.ravel(), F.ravel(), bins, [[0, 1], [0, 1]])[0] / X.size
    px, pf = pxy.sum(1, keepdims=True), pxy.sum(0, keepdims=True)
    nz = pxy > 0
    return np.sum(pxy[nz] * np.log2(pxy[nz] / (px @ pf)[nz]))

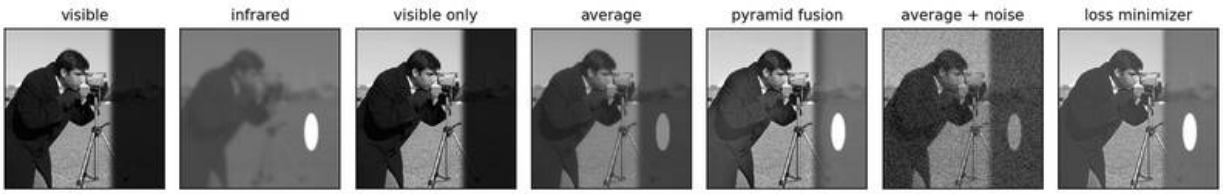
def q_gradient(A, B, F):
    """A simplified Q^{AB/F}: how well F keeps the edge strength and direction of A and B."""
    def grad(I):
        gx, gy = ndi.sobel(I, 1), ndi.sobel(I, 0)
        return np.hypot(gx, gy), np.arctan2(gy, gx)
    gF, aF = grad(F)
    num = den = 0.0
    for X in (A, B):
        gX, aX = grad(X)
        strength = np.minimum(gX, gF) / (np.maximum(gX, gF) + 1e-12)
        d = np.abs(aX - aF) % np.pi                 # edge direction is defined modulo pi
        direction = 1 - np.minimum(d, np.pi - d) / (np.pi / 2)
        num += np.sum(strength * direction * gX)    # weight each pixel by the source edge
         ↪  strength
        den += np.sum(gX)
    return num / den

ring = ndi.binary_dilation(person, iterations=15) & ~person
candidates = {"visible only": vis,
              "average": (vis + ir) / 2,
              "pyramid fusion": pyramid_fuse(vis, ir),
              "average + noise": np.clip((vis + ir) / 2 + 0.08 *
               ↪  rng.standard_normal(vis.shape), 0, 1)}
print(f"{'':16s}  EN    SF     SD     MI    Q_G   person contrast")
for name, F in candidates.items():
    mi = mutual_info(vis, F) + mutual_info(ir, F)
    tc = F[person].mean() - F[ring].mean()          # task check: does the person stand out?
    print(f"{name:16s} {entropy(F):4.2f}  {spatial_frequency(F):.3f}  {F.std():.3f}
     ↪  {mi:4.2f}  "
          f"{q_gradient(vis, ir, F):.3f}  {tc:6.3f}")
```

```
                  EN    SF     SD     MI    Q_G   person contrast
visible only     6.68  0.065  0.293  5.72  0.784  -0.000
average          6.69  0.034  0.172  4.33  0.455   0.257
pyramid fusion   7.48  0.067  0.221  3.75  0.779   0.513
average + noise  7.49  0.163  0.189  1.41  0.296   0.259
```

Read the table column by column (all numbers are from this toy):

- **Noise wins EN and SF.** Adding noise to the average gives the highest entropy and almost five times the average's spatial frequency, while making the image worse.
- **Not fusing at all wins SD, MI and $Q_G$.** The visible image alone has the most contrast, the highest mutual information (copying a source makes $\mathrm{MI}_{A,F}$ equal to that source's entropy) and even the highest edge score, because the strong visible edges carry most of the weight $w^X$. Yet the person is invisible: its contrast is zero (printed as -0.000).
- **Pyramid fusion wins only the task check.** It doubles the person's contrast relative to the average while keeping the visible texture ($Q_G$ almost as high as the visible image), but it is not the best on any of the five standard metrics.




Higher is 'better' for every column; the shaded cell is each column's winner

| candidate | EN (bits) | SF | SD | MI (bits) | Q_G | person contrast |
|---|---|---|---|---|---|---|
| visible only | 6.68 | 0.065 | 0.293 | 5.72 | 0.784 | 0.000 |
| average | 6.69 | 0.034 | 0.172 | 4.33 | 0.455 | 0.257 |
| pyramid fusion | 7.48 | 0.067 | 0.221 | 3.75 | 0.779 | 0.513 |
| average + noise | 7.49 | 0.163 | 0.189 | 1.41 | 0.296 | 0.259 |
| loss minimizer | 6.99 | 0.065 | 0.200 | 4.24 | 0.718 | 0.513 |

*Figure 5 — Five candidate fusions of the synthetic infrared–visible pair and their scores (our toy; the loss minimizer is computed in the deep-learning section). Every metric is 'higher is better', yet the winners are a noisy image and an image that ignores the infrared source. Only the task check identifies the useful results.*

## What to report

The toy is extreme on purpose, but each effect appears in practice: noise and over-sharpening raise EN, SF and SD; results that look like one source score well on MI; and metrics with source-dependent weights favor the source with stronger edges. Benchmark authors draw the same conclusion from real methods: researchers tend to pick the metrics that support their method [17], [7], and quantitative and qualitative comparisons may disagree [7]. Practical advice:

1. Report several metrics from **different families**, including at least one that compares with the sources ($Q^{AB/F}$, an SSIM-based score or VIFF), never only EN, SD or SF.
2. Use **reference-based** metrics where a reference can exist: synthetic multi-focus pairs, or multi-exposure datasets with selected references (with the caveat that the reference is one acceptable answer).

3. Add a **downstream task** when fusion feeds one, such as detection or segmentation accuracy on the fused images, as recent infrared–visible benchmarks do [27], [28].
4. Show **crops** of hard regions (depth edges, highlights, warm targets in shadow), and run a user study when the claim is about visual quality.

## Deep image fusion

**Plain version.** Deep networks learn the weights, the features and the combination rule from data. The catch is that there are no examples of the correct output, so the network is trained to satisfy rules of the same kind as the metrics above.

### The missing ground truth

A restoration network (DLCV 09) learns from pairs (degraded input, clean target). A fusion network has inputs but no target. The field uses four ways around this:

1. **Synthesize a task that has ground truth.** For multi-focus fusion, blur parts of sharp images: the sharp image is the target and the blur mask the true decision map. Most deep multi-focus methods are trained this way [7]; IFCNN trains a general fusion network on a synthetic multi-focus dataset built from an RGB-D dataset, so that ground-truth fused images exist [29].
2. **Train something else, then fuse features by a rule.** Train an autoencoder to reconstruct ordinary images, then at test time encode both sources, merge the features with a hand-made rule and decode [30].
3. **Unsupervised losses.** Define a loss that measures how well $F$ keeps the intensity, gradients or structure of each source, and train the network to minimize it on unlabeled source pairs [31], [32].
4. **Let another task supervise.** Feed $F$ to a detector or segmenter and add its loss, so that the fusion is judged by its usefulness [33], [27].

### Unsupervised losses

A recent survey groups the losses of unsupervised infrared–visible fusion into pixel-level terms (L1 or MSE), terms built from evaluation metrics (above all SSIM) and terms that target data characteristics such as image gradients [2]. A typical combination is

$$\mathcal{L}(F) = \lambda_{\text{int}} \|F - T\|_1 + \lambda_{\text{grad}} \big\| |\nabla F| - |\nabla T_g| \big\|_1 + \lambda_{\text{ssim}} \sum_X \omega_X (1 - \text{SSIM}(F, I_X)),$$

where $T$ is an intensity target built from the sources (for example a weighted average, or the pixel-wise maximum, which keeps the brighter thermal target), $|\nabla T_g|$ a gradient-magnitude target (for example the larger of the two sources' gradient magnitudes at each pixel), $\text{SSIM}(F, I_X)$ the structural similarity [16] between $F$ and source $X$, $\omega_X$ per-source weights, and the $\lambda$s weight the terms. Methods differ mainly in these choices: DeepFuse [31] trains with MEF-SSIM [25] alone,

SwinFusion [34] with SSIM, texture and intensity terms, and U2Fusion [32] weights a similarity loss by an estimate of how much information each source holds.

*Online only: Training signals of representative fusion networks — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/notes/training-signals-of-representative-fusion-networks/*

### The loss defines the answer

**Plain version.** If nobody gives the network a target, the loss *is* the target. The best a network can do is to output, for each pair, the image that minimizes the loss. For a simple quadratic loss we can compute that image directly, without any network, and look at it.

The snippet continues from the toy of the evaluation section (it uses `vis`, `ir`, `person`, `ring` and the metric functions). It minimizes a squared-error version of the loss above, an intensity term toward the pixel-wise maximum of the sources plus a gradient term toward the stronger source gradient,

$$F^\star = \arg\min_F \|F - \max(I_A, I_B)\|_2^2 + \lambda \, \|\nabla F - \mathbf{g}\|_2^2,$$

where $\nabla F$ stacks the horizontal and vertical forward differences of $F$, $\mathbf{g}$ takes, at each position and in each direction, the difference of whichever source has the larger one in magnitude, and $\lambda > 0$ trades intensity against gradients. Setting the gradient of the objective to zero gives a sparse linear system,

$(\mathbf{I} + \lambda \mathbf{D}^\top \mathbf{D}) F = \max(I_A, I_B) + \lambda \mathbf{D}^\top \mathbf{g}$, where $\mathbf{D}$ is the difference operator; conjugate gradients solves it in seconds.

```
import scipy.sparse as sp
from scipy.sparse.linalg import cg

def diff_ops(H, W):
    d = lambda n: sp.diags([-np.ones(n - 1), np.ones(n - 1)], [0, 1], shape=(n - 1, n))
    return sp.kron(sp.eye(H), d(W)).tocsr(), sp.kron(d(H), sp.eye(W)).tocsr()  # Dx, Dy

def fuse_by_loss(a, b, lam=10.0):
    """Minimize ||F - max(a, b)||^2 + lam * ||grad F - g||^2, where g is the stronger source
     ↪  gradient."""
    H, W = a.shape
    Dx, Dy = diff_ops(H, W)
    gx = [Dx @ s.ravel() for s in (a, b)]
    gy = [Dy @ s.ravel() for s in (a, b)]
    pick_x = np.abs(gx[0]) >= np.abs(gx[1])
    pick_y = np.abs(gy[0]) >= np.abs(gy[1])
    tx, ty = np.where(pick_x, gx[0], gx[1]), np.where(pick_y, gy[0], gy[1])
    target = np.maximum(a, b).ravel()                # intensity term: the brighter source
    M = sp.eye(H * W) + lam * (Dx.T @ Dx + Dy.T @ Dy)  # normal equations of the quadratic
     ↪  loss
    rhs = target + lam * (Dx.T @ tx + Dy.T @ ty)
    f, info = cg(M, rhs, x0=target, rtol=1e-6, maxiter=2000)
    if info != 0: raise RuntimeError("Conjugate gradient did not converge")
    return np.clip(f.reshape(H, W), 0, 1)
```

```
for lam in (1.0, 10.0):
    F = fuse_by_loss(vis, ir, lam)
    mi = mutual_info(vis, F) + mutual_info(ir, F)
    print(f"loss minimizer, lam={lam:4.1f}: EN {entropy(F):.2f}  SF
     ↪  {spatial_frequency(F):.3f}  "
          f"SD {F.std():.3f}  MI {mi:.2f}  Q_G {q_gradient(vis, ir, F):.3f}  "
          f"person contrast {F[person].mean() - F[ring].mean():.3f}")
```

```
loss minimizer, lam= 1.0: EN 6.91  SF 0.060  SD 0.195  MI 4.68  Q_G 0.659  person contrast
 ↪  0.513
loss minimizer, lam=10.0: EN 6.99  SF 0.065  SD 0.200  MI 4.24  Q_G 0.718  person contrast
 ↪  0.513
```

The loss minimizer has the person as bright as pyramid fusion does (contrast 0.513) and keeps the visible texture (Figure 5, right). Raising $\lambda$ from 1 to 10 raises $Q_G$ and SF and lowers MI: the weight between loss terms moves the result along the metrics, and choosing it is choosing what the "correct" fusion is. This is the honest summary of unsupervised deep fusion. The network adds speed, the ability to generalize from many training pairs and features richer than raw gradients, but the definition of a good result is still written by hand, in the loss.

### Families of deep fusion networks

The surveys organize the field by architecture and by training strategy [35], [2]. The table lists representative members of each family; the infrared–visible survey guide (https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/papers/the-infraredvisible-fusion-survey-of-liu-et-al-tpami-2025/) in the Paper guides covers the taxonomy in more depth.

| Family | Idea | Representative methods |
|---|---|---|
| Learned decision maps or weights | A network predicts which source is in focus, or a per-pixel weight; the output is still a selection or blend of source pixels | CNN focus map for multi-focus [36]; MEF-Net predicts weights at low resolution and upsamples them with a guided filter [37] |
| Autoencoder + fusion rule | Train an encoder–decoder to reconstruct images; fuse encoded features with a hand-made rule | DenseFuse [30] |
| End-to-end, unsupervised | A network maps the sources to $F$, trained with an intensity/gradient/SSIM-type loss | DeepFuse [31], U2Fusion [32], SwinFusion [34], CDDFuse [38] |
| Supervised on synthetic data | Train on synthesized pairs with a known fused image | IFCNN [29] |
| Adversarial (GAN) | A discriminator pushes $F$ to look like one source's texture while a content loss keeps the other's intensity | FusionGAN [39], TarDAL [27] |
| Task-driven | A detection or segmentation loss on $F$ guides the fusion network | SeAFusion [33], TarDAL [27], SegMiF [28], TDFusion [40] |

| Family | Idea | Representative methods |
|---|---|---|
| Generative priors and language | A pretrained diffusion model or a text encoder guides the fusion | DDFM [41], Text-IF [42] |

Two families rely on generative models explained elsewhere: GANs in ML 07 and diffusion models in ML 08. Across all families, the architecture matters less for the character of the result than the training signal, which is what the table's middle column really distinguishes.

### What each learned family actually learns

The architecture table is a map, not a training recipe. Follow the source of supervision to understand the result.

**Autoencoder plus a rule.** DenseFuse learns features and reconstruction from ordinary images [30]. At inference, a fixed rule merges source features. It needs no fused target for training, but reconstruction alone does not teach which modality should win. Addition and activity weighting encode different preferences even with identical network weights.

**Supervised convolutional fusion.** IFCNN learns from synthetic pairs with a known target and merges source features through an element-wise operation [29]. This supplies direct supervision but transfers the synthetic data's assumptions into the model. A blur process that ignores foreground spread or sensor differences may teach the wrong decision on real pairs.

**Direct unsupervised fusion.** U2Fusion trains the generated image to resemble sources under adaptive information weights [32]. Both its structural and intensity terms are needed to understand the output. Its VGG-based weights and continual-learning penalty operate during training; the final image generator does not need VGG or EWC at inference.

**Adversarial fusion.** FusionGAN adds a discriminator to a source-preservation objective, encouraging visible-like texture while retaining infrared intensity [39]. A realistic appearance is a distributional preference, not proof that a particular edge is supported by the sources. TarDAL splits the adversarial roles between thermal targets and visible details and adds downstream supervision [27].

**Attention-based fusion.** SwinFusion learns within-source and cross-source interactions with windowed and shifted-window attention [34]. Queries, keys and values let one feature use evidence from another location or modality. Windowing controls cost; successive shifted windows extend interaction across boundaries. Attention still needs a fusion objective and aligned evidence, and it can transfer noise as well as useful texture.

**Diffusion-based fusion.** DDFM repeatedly combines a pretrained denoising prior with source-informed corrections [41]. It avoids fusion-specific denoiser training but pays for iterative inference. One EM correction inside a diffusion step is not one-step fusion. The modeled likelihood and image prior both influence what the sampler produces.

**Unified and task-aware systems.** "Unified" may mean one loss design or one architecture rather than one set of weights; check the checkpoint and training setting. U2Fusion explicitly targets

shared parameters across tasks [32]. Task-aware models instead judge fusion partly through a detector or segmenter [27], [33]. They need the corresponding supervision and can specialize to that task. Neither label guarantees that a single model handles arbitrary modalities or unregistered inputs.

*Online only: DenseFuse: learn reconstruction, choose fusion in feature space — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/papers/densefuse-learn-reconstruction-choose-fusion-in-feature-space/; U2Fusion: one unsupervised model across fusion tasks — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/papers/u2fusion-one-unsupervised-model-across-fusion-tasks/; TarDAL: fuse for both appearance and object detection — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/papers/tardal-fuse-for-both-appearance-and-object-detection/; DDFM: a diffusion prior with likelihood rectification — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/papers/ddfm-a-diffusion-prior-with-likelihood-rectification/*

### Fusion for downstream tasks

**Plain version.** If the fused image feeds a detector, the best fusion is the one that makes the detector work, not the one with the best entropy.

**SeAFusion** [33] cascades a real-time fusion network with a segmentation network and adds a semantic loss, training the two in a loop [2]. **TarDAL** [27] writes fusion and detection as one bilevel optimization problem, unrolls it into a fusion network with one generator and two discriminators (one for the infrared targets, one for visible texture) followed by a detector, and introduces the M3FD benchmark captured with a synchronized infrared–visible system. **SegMiF** [28] exchanges intermediate features between a fusion and a segmentation sub-network through hierarchical attention, and collects a full-time benchmark with 15 annotated pixel-level classes. **TDFusion** [40] goes one step further and *learns the fusion loss itself*: a loss-generation module is trained by meta-learning so that the fusion network it supervises minimizes the downstream task loss. Task-driven training answers the "what is correct?" question with a measurable goal, but it optimizes for one task and one network, and task performance does not fully reflect visual quality [2].

### Fusing images that are not aligned

**Plain version.** Real infrared and visible cameras are rarely perfectly aligned. Registration-robust methods estimate the misalignment and correct it inside the fusion network.

Pixel-wise fusion of misaligned pairs produces ghosts and double edges around objects [43], [44]. Two strategies appear in recent work [2]. The **style-transfer route** first translates one modality into the other, so that cross-modal registration becomes a single-modality one: UMF-CMGR [43] generates a pseudo-infrared image from the visible image, registers the real infrared image to it with a multi-level registration network, and then fuses. The **shared-feature route** maps both modalities into a common feature space and predicts deformation fields there. SuperFusion [44] combines registration (bidirectional deformation fields), fusion and a semantic constraint from a pretrained segmentation model in one framework. Multi-exposure fusion has the analogous problem with camera shake and moving objects (ghosting).

## Datasets and benchmarks

**Plain version.** Infrared–visible fusion has several public collections of aligned pairs, some with object labels; multi-focus and multi-exposure fusion have smaller test sets and benchmark toolkits. None has a ground-truth fused image for real infrared–visible data.

| Dataset | Task | Content (as described by its authors) |
|---|---|---|
| TNO [45] | IR–visible | Intensified visual, near-infrared and long-wave infrared night-time images of military and surveillance scenes |
| RoadScene [32] | IR–visible | 221 aligned pairs of road scenes selected from FLIR videos, released with U2Fusion |
| MSRS [46] | IR–visible | 1,444 aligned pairs of road scenes, built from the MFNet dataset [47] by removing 125 misaligned pairs and enhancing the infrared images |
| LLVIP [48] | IR–visible | 15,488 strictly aligned pairs, mostly taken in very dark scenes, with pedestrian labels |
| M3FD [27] | IR–visible | 4,200 registered pairs with detection labels, plus 300 pairs from independent scenes for fusion; images mostly 1024 × 768 |
| FMB [28] | IR–visible | A full-time benchmark with 15 annotated pixel-level classes for fusion and segmentation |
| VIFB [17] | IR–visible | A test set of 21 pairs, a code library of 20 fusion algorithms and 13 metrics |
| Lytro [15] | Multi-focus | 20 pairs of multi-focus images, long the most widely used test set [7] |
| MFFW [6] | Multi-focus | 19 real pairs collected from the Internet with a strong defocus spread effect |
| MFIFB [7] | Multi-focus | A benchmark of 105 test pairs, 30 algorithms and 20 metrics |
| SICE [49] | Multi-exposure | 589 multi-exposure sequences (4,413 images), each with a reference chosen by subjective comparison of 13 fusion and HDR algorithms |
| MEFB [18] | Multi-exposure | A benchmark of 100 test image pairs, 16 algorithms and 20 metrics |

Read the table with three cautions. **Preprocessing is part of the data:** RoadScene's infrared images had their background thermal noise processed, and MSRS's infrared images were contrast-enhanced with a dark-channel-prior-based method, so "the infrared source" is already processed. **Scenes are narrow:** most infrared–visible data are urban roads and campuses, and the 2025 survey lists more diverse scenes and registration ground truth among the missing benchmarks [2]. **Old test sets are saturated:** on the 20 Lytro pairs, popular methods perform very similarly because

the defocus spread effect is weak there [6], [7]. SICE is also used in DLCV 09C for exposure correction.

## Failure modes

**Plain version.** Fusion fails where its assumptions fail: sources not aligned, sources that are not simply "sharp or blurred", weights that change too fast or too slowly, and rules that average what should be selected.

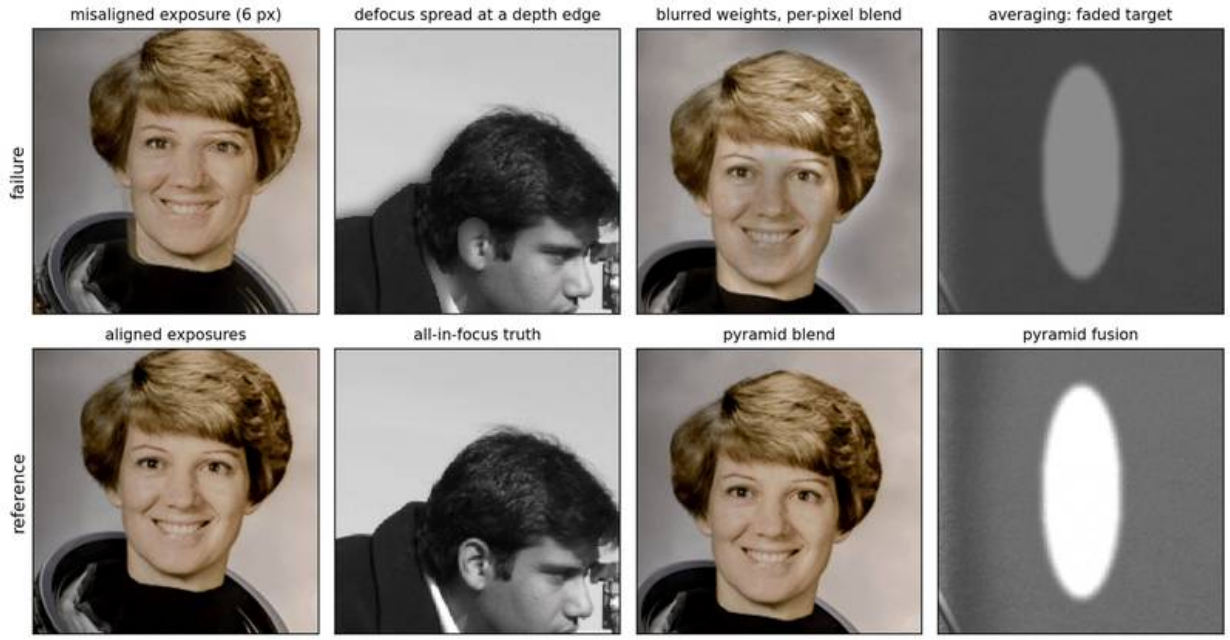



*Figure 6 — Four failure modes on synthetic data (top) and a reference for each (bottom): misregistration ghosts, the defocus spread effect at a depth edge (a layered defocus model in which the blurred foreground spills over the background), halos from smoothed weights, and a washed-out thermal target.*

- **Misregistration ghosts.** A shift of a few pixels between sources turns edges into double edges (Figure 6, first column). In multi-exposure fusion, moving objects cause the same effect. Check alignment before blaming the fusion rule.
- **Decision-map boundary artifacts.** Near depth edges, the real blur of the foreground spills over the background, so neither source is sharp and any binary decision leaves a soft ring or a blocky edge (second column). Hard switches also show small seams where the map is wrong.
- **Halos.** Weights that are smoothed too much, or a base layer fused at too fine a scale, create bright or dark bands along strong edges (third column) [5]. Multi-scale blending can reduce them, but a pyramid that is too shallow brings back seams and low-frequency errors (Exercise 4).
- **Washed-out thermal targets.** Averaging, or a loss that weights the visible source heavily, halves the contrast of warm targets (fourth column); the opposite error makes the background look

like an infrared image, flat and textureless.

- **Color shifts.** A common way to handle color is to fuse only the luminance (for example the Y channel of YCbCr) and to combine or copy the chroma separately, as DeepFuse does for exposures [31]. Where the fused luminance differs a lot from the source whose chroma is used, for example a bright thermal target on a dark visible background, the colors no longer fit and look washed out or tinted. For exposure stacks, selecting pyramid coefficients instead of blending them produced color shifts in the comparison of Mertens et al. [5].
- **Noise amplification.** Max-abs selection and gradient losses prefer strong high frequencies, and noise can dominate high-frequency responses in a dark visible image, so a fused night image can look sharper but noisier than either source. This is also why EN and SF reward such results.
- **Metric gaming.** A method tuned to EN, SF or SD can produce exactly the artifacts above while reporting better numbers (the toy of Figure 5).

## Modern view

**Surveys.** Six surveys together cover the field:

- **Ma, Ma and Li (Information Fusion 2019) [3]** review classical and early deep infrared–visible fusion, introduce registration as a prerequisite, summarize applications and evaluation metrics, and compare eighteen methods with nine metrics. Takeaway: a reference point for classical methods and their evaluation habits before deep learning took over.
- **Liu et al. (Information Fusion 2018) [1]** wrote an early review (2018) of deep learning for pixel-level fusion across tasks. They list the difficulties of conventional fusion (hand-designed activity measures and fusion rules) and argue that learned features can replace them, grouping the early methods (CNNs, convolutional sparse representation, stacked autoencoders) into generic frameworks.
- **Zhang and Demiris (TPAMI 2023) [35]** review deep visible–infrared fusion by architecture (CNN-, autoencoder-, GAN- and transformer-based), with datasets and evaluation methods. Takeaway: deep methods have become the dominant approach for the task.
- **Liu et al. (TPAMI 2025) [2]** organize more than 180 learning-based infrared–visible methods along three dimensions, data compatibility (registration), fusion (architectures and losses) and task adaptation (detection, segmentation), and compare them on registration, fusion and downstream tasks.
- **Zhang (TPAMI 2022) [50]** surveys deep multi-focus fusion (methods, datasets, metrics) and compares deep and conventional methods experimentally.
- **Xu et al. (Remote Sensing 2022) [51]** review multi-exposure fusion for static and dynamic scenes and compare 18 methods with nine metrics.

*Online only: The infrared–visible fusion survey of Liu et al. (TPAMI 2025) — https://blog.shyandram.dev/en/learn/deep-learning/image-fusion/papers/the-infraredvisible-fusion-survey-of-liu-et-al-tpami-2025/*

**State of the art (2023–2026).** Four trends stand out. First, **fusion is increasingly judged by downstream tasks**: methods are trained with detection or segmentation losses, benchmarks such

as M3FD and FMB carry labels, and the fusion loss itself can be learned from the task loss [27], [28], [40]. Second, **generative priors and language** have entered: diffusion-based fusion without task-specific training [41], self-supervised training from imaging priors [52], and text-guided fusion that also handles degraded sources [42]. Third, **registration is moving inside the model**, with joint registration–fusion networks for unaligned infrared–visible pairs [43], [44]. Fourth, **video fusion** is starting: a CVPR 2026 work introduces M3SVD, 220 synchronized and registered infrared–visible videos with 153,797 frames, and a model whose bi-temporal attention mitigates the temporal inconsistency of frame-by-frame image fusion [53]. For multi-focus fusion, a 2020 benchmark study found that deep methods trained on simulated data did not generalize well to challenging real pairs, while conventional methods remained competitive [7]. Treat claims of large improvements with care, and check which metrics they rest on.

**Open problems.**

- **Evaluation.** There is still no metric, or set of metrics, that agrees with human judgment and with task performance; the 2025 survey names better evaluation metrics as an essential direction [2].
- **Real misalignment and degradations.** Most methods assume registered, clean sources. Benchmarks with registration ground truth, and fusion that is robust to noise, blur, low light and adverse weather in either source, are open [2], [42].
- **Real defocus.** Training data that reproduces the defocus spread effect, and methods that handle depth edges, are needed for multi-focus fusion to generalize [6], [7].
- **Efficiency.** Many recent networks are large; fusion on drones, cars and handheld devices needs lightweight, hardware-friendly designs [2] (see DLCV 09C for the general techniques).
- **Video.** Research has fused images rather than videos, largely because large synchronized multi-sensor video datasets were missing [53]; temporal consistency at low cost is open.

## Key takeaways

- Choose what each source should contribute before selecting a fusion rule.
- Averaging can weaken source-specific features; weights should reflect useful information.
- Multi-scale blending separates fine-detail selection from coarse brightness blending, reducing seam risk.
- Without a unique fused target, training losses encode the desired answer and its compromises.
- Registration errors create ghosts; inspect boundaries, halos, colors, noise and source-specific targets.
- Use several metric families, crops and task results; a high score can reward noise or copying one source.

## Exercises

1. **Contrast under averaging.** A warm target has intensity $b + a$ in the infrared image and $b$ in the visible one, on a background of intensity $b$ in both. (a) Show that the average has target

contrast $a/2$ and that any per-pixel weighted average with weights $w_{\mathrm{IR}}, w_{\mathrm{VIS}}$ gives $w_{\mathrm{IR}}\, a$. (b) In our toy, the person contrast of the average is 0.257 and that of pyramid fusion 0.513. Which weight on the infrared source does pyramid fusion behave like on the target, and why does it achieve this without darkening the visible texture elsewhere?

**Hint**

(a) $F = w_{\mathrm{IR}}(b + a) + w_{\mathrm{VIS}}\, b = b + w_{\mathrm{IR}}\, a$, using $w_{\mathrm{IR}} + w_{\mathrm{VIS}} = 1$. (b) 0.513 is about twice 0.257, so on the target pyramid fusion behaves like $w_{\mathrm{IR}} \approx 1$. It can do so because the weights are not global: the coarse base takes the brighter source (the infrared target), while the fine levels take whichever detail is stronger, which is the visible texture away from the target. A global weight would have to trade the two against each other.

2. **Focus window and cleaning.** In the multi-focus snippet, vary the focus-measure smoothing $\sigma \in \{1, 2, 4\}$ and the median window $k \in \{1, 15, 31, 61\}$ ($k = 1$ means no cleaning), always followed by the soft blend with a Gaussian of standard deviation 2. Report PSNR. Which setting wins, and why might the answer differ on real photographs?

**Hint**

In our run all settings lie between 39.43 and 40.05 dB; the best is $\sigma = 2$ with no median cleaning (40.05 dB), and the largest window, $k = 61$, is the worst for every $\sigma$ (about 39.4–39.5 dB). Without cleaning, the soft blend mixes A and B in the speckled flat regions, which averages their independent noise and slightly raises PSNR there; a large median window misplaces the map near the boundary of the disk. On real photographs, uncleaned speckle shows up as visible blotches where the sources differ (noise, small misalignment, the defocus spread effect), so cleaning usually matters more for appearance than for PSNR on a toy.

3. **Area-based selection.** Replace `pick_stronger` in the pyramid snippet by a rule that compares *local energies*: smooth $(L_\ell^A)^2$ and $(L_\ell^B)^2$ with a Gaussian of standard deviation $s$ and choose the coefficient whose energy is larger. Try $s \in \{1, 1.5, 3\}$ and compare with the per-coefficient rule.

**Hint**

Use `ndi.gaussian_filter(p**2, s)`. In our run PSNR rose from 37.60 dB (per coefficient) to 39.14, 39.42 and 39.62 dB for $s = 1, 1.5, 3$, close to the decision map. Smoothing the activity makes neighboring coefficients take the same decision, which is what the area-based rule and consistency verification of Li et al. aim for: in multi-focus fusion, focus is a property of regions, not of single coefficients.

4. **Pyramid depth and quality measures in exposure fusion.** (a) Rerun exposure fusion with pyramids of $n = 0, 1, 3, 5, 7, 9$ levels ($n = 0$ is the per-pixel blend) and report the well-exposed fraction; look at the images. (b) Use one quality measure at a time (contrast, saturation or well-exposedness only) at full depth. What do you observe?

**Hint**

(a) In our run the fraction was 72.3% for $n = 0$, then 72.2%, 70.9%, 69.6% and 68.5% for $n = 1, 3, 5, 7$, and 73.8% for $n = 9$; the mean absolute difference from the 9-level result was still 0.068 at $n = 7$. Shallow pyramids keep seams and low-frequency brightness errors, because the coarsest levels, where the weights are blended most smoothly, are missing; the result changes a lot even between 7 and 9 levels on a 512-pixel image. (b) Well-exposedness alone gave the highest fraction (77.5%), contrast alone 69.1% and saturation alone 71.5%. Mertens et al. also found that well-exposedness by itself gives fairly good images; contrast and saturation matter for detail and color, which this count does not measure.

5. **How noise inflates spatial frequency.** Let $F' = F + n$ where $n$ is independent zero-mean noise with variance $\sigma^2$ at every pixel. (a) Show that the expected $\mathrm{SF}^2$ grows by about $4\sigma^2$. (b) Check it on the toy: add noise with $\sigma = 0.02, 0.04, 0.08$ to the average and compare the measured SF with $\sqrt{\mathrm{SF}^2 + 4\sigma^2}$.

**Hint**

(a) A horizontal difference of $F'$ is the difference of $F$ plus $n(i,j) - n(i,j-1)$, whose variance is $2\sigma^2$ and which is uncorrelated with $F$; so $\mathbb{E}[\mathrm{RF}'^2] \approx \mathrm{RF}^2 + 2\sigma^2$, the same for CF, and $\mathbb{E}[\mathrm{SF}'^2] \approx \mathrm{SF}^2 + 4\sigma^2$. (b) In our run (with a different random seed from the snippet), the measured and predicted SF were 0.0528 and 0.0528 for $\sigma = 0.02$, 0.0871 and 0.0871 for 0.04, and 0.1631 and 0.1637 for 0.08; the small gap at 0.08 comes from clipping to $[0, 1]$. EN rose from 6.69 to 6.99, 7.20 and 7.49 bits.

6. **Mutual information rewards copying.** (a) Show that if $F = I_A$, then $\mathrm{MI}_{A,F} + \mathrm{MI}_{B,F} = H(I_A) + \mathrm{MI}(I_A, I_B)$, where $H$ is the entropy of the binned image. (b) Verify it on the toy with 64 bins. (c) Compute the MI score of the average and of the visible image with 16, 64 and 256 bins. Does the ranking depend on the bin count? Does the value?

**Hint**

(a) $\mathrm{MI}(X, X) = H(X)$, because knowing $X$ removes all uncertainty about $X$. (b) In our run $H(\mathrm{vis}) = 4.728$ bits and $\mathrm{MI}(\mathrm{ir}, \mathrm{vis}) = 0.995$ bits, which add up to 5.723, the MI score of "visible only" in the snippet. (c) The average scored 2.64, 4.33 and 5.00 bits and the visible image 3.42, 5.72 and 7.82 bits: the ranking stayed, but the values more than doubled from 16 to 256 bins for the visible image, so MI values from papers that use different bin counts or implementations cannot be compared.

## References

1. Y. Liu, X. Chen, Z. Wang, Z. J. Wang, R. K. Ward and X. Wang, "Deep learning for pixel-level image fusion: Recent advances and future prospects," *Information Fusion*, vol. 42, pp. 158–173, 2018. doi
2. J. Liu, G. Wu, Z. Liu, D. Wang, Z. Jiang, L. Ma, W. Zhong, X. Fan and R. Liu, "Infrared and visible image fusion: From data compatibility to task adaption," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 47, no. 4, pp. 2349–2369, 2025. doi arXiv

3. J. Ma, Y. Ma and C. Li, “Infrared and visible image fusion methods and applications: A survey,” *Information Fusion*, vol. 45, pp. 153–178, 2019. doi
4. P. E. Debevec and J. Malik, “Recovering high dynamic range radiance maps from photographs,” in *Proc. SIGGRAPH*, 1997, pp. 369–378. doi
5. T. Mertens, J. Kautz and F. Van Reeth, “Exposure fusion: A simple and practical alternative to high dynamic range photography,” *Computer Graphics Forum*, vol. 28, no. 1, pp. 161–171, 2009. doi
6. S. Xu, X. Wei, C. Zhang, J. Liu and J. Zhang, “MFFW: A new dataset for multi-focus image fusion,” arXiv:2002.04780, 2020. arXiv
7. X. Zhang, “Multi-focus image fusion: A benchmark,” arXiv:2005.01116, 2020. arXiv
8. P. J. Burt and E. H. Adelson, “The Laplacian pyramid as a compact image code,” *IEEE Transactions on Communications*, vol. 31, no. 4, pp. 532–540, 1983. doi
9. P. J. Burt and R. J. Kolczynski, “Enhanced image capture through fusion,” in *Proc. 4th International Conference on Computer Vision (ICCV)*, 1993, pp. 173–182. doi
10. P. J. Burt and E. H. Adelson, “A multiresolution spline with application to image mosaics,” *ACM Transactions on Graphics*, vol. 2, no. 4, pp. 217–236, 1983. doi
11. H. Li, B. S. Manjunath and S. K. Mitra, “Multisensor image fusion using the wavelet transform,” *Graphical Models and Image Processing*, vol. 57, no. 3, pp. 235–245, 1995. doi
12. S. Li, X. Kang and J. Hu, “Image fusion with guided filtering,” *IEEE Transactions on Image Processing*, vol. 22, no. 7, pp. 2864–2875, 2013. doi
13. K. He, J. Sun and X. Tang, “Guided image filtering,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 6, pp. 1397–1409, 2013. doi
14. B. Yang and S. Li, “Multifocus image fusion and restoration with sparse representation,” *IEEE Transactions on Instrumentation and Measurement*, vol. 59, no. 4, pp. 884–892, 2010. doi
15. M. Nejati, S. Samavi and S. Shirani, “Multi-focus image fusion using dictionary-based sparse representation,” *Information Fusion*, vol. 25, pp. 72–84, 2015. doi
16. Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, “Image quality assessment: From error visibility to structural similarity,” *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. doi
17. X. Zhang, P. Ye and G. Xiao, “VIFB: A visible and infrared image fusion benchmark,” in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2020. CVF arXiv
18. X. Zhang, “Benchmarking and comparing multi-exposure image fusion algorithms,” *Information Fusion*, vol. 74, pp. 111–131, 2021. doi arXiv
19. A. M. Eskicioglu and P. S. Fisher, “Image quality measures and their performance,” *IEEE Transactions on Communications*, vol. 43, no. 12, pp. 2959–2965, 1995. doi
20. G. Qu, D. Zhang and P. Yan, “Information measure for performance of image fusion,” *Electronics Letters*, vol. 38, no. 7, pp. 313–315, 2002. doi
21. M. Hossny, S. Nahavandi and D. Creighton, “Comments on ‘Information measure for performance of image fusion’,” *Electronics Letters*, vol. 44, no. 18, pp. 1066–1067, 2008. doi
22. C. S. Xydeas and V. Petrović, “Objective image fusion performance measure,” *Electronics Letters*, vol. 36, no. 4, pp. 308–309, 2000. doi
23. G. Piella and H. Heijmans, “A new quality metric for image fusion,” in *Proc. IEEE International Conference on Image Processing (ICIP)*, 2003, vol. 3, pp. III-173–176. doi

24. Y. Han, Y. Cai, Y. Cao and X. Xu, “A new image fusion performance metric based on visual information fidelity,” *Information Fusion*, vol. 14, no. 2, pp. 127–135, 2013. doi
25. K. Ma, K. Zeng and Z. Wang, “Perceptual quality assessment for multi-exposure image fusion,” *IEEE Transactions on Image Processing*, vol. 24, no. 11, pp. 3345–3356, 2015. doi
26. Z. Liu, E. Blasch, Z. Xue, J. Zhao, R. Laganiere and W. Wu, “Objective assessment of multiresolution image fusion algorithms for context enhancement in night vision: A comparative study,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 1, pp. 94–109, 2012. doi
27. J. Liu, X. Fan, Z. Huang, G. Wu, R. Liu, W. Zhong and Z. Luo, “Target-aware dual adversarial learning and a multi-scenario multi-modality benchmark to fuse infrared and visible for object detection,” in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022, pp. 5802–5811. CVF arXiv M3FD
28. J. Liu, Z. Liu, G. Wu, L. Ma, R. Liu, W. Zhong, Z. Luo and X. Fan, “Multi-interactive feature learning and a full-time multi-modality benchmark for image fusion and segmentation,” in *Proc. IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023, pp. 8115–8124. CVF arXiv
29. Y. Zhang, Y. Liu, P. Sun, H. Yan, X. Zhao and L. Zhang, “IFCNN: A general image fusion framework based on convolutional neural network,” *Information Fusion*, vol. 54, pp. 99–118, 2020. doi
30. H. Li and X.-J. Wu, “DenseFuse: A fusion approach to infrared and visible images,” *IEEE Transactions on Image Processing*, vol. 28, no. 5, pp. 2614–2623, 2019. doi arXiv
31. K. R. Prabhakar, V. S. Srikar and R. V. Babu, “DeepFuse: A deep unsupervised approach for exposure fusion with extreme exposure image pairs,” in *Proc. IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 4724–4732. doi CVF
32. H. Xu, J. Ma, J. Jiang, X. Guo and H. Ling, “U2Fusion: A unified unsupervised image fusion network,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 1, pp. 502–518, 2022. doi RoadScene
33. L. Tang, J. Yuan and J. Ma, “Image fusion in the loop of high-level vision tasks: A semantic-aware real-time infrared and visible image fusion network,” *Information Fusion*, vol. 82, pp. 28–42, 2022. doi
34. J. Ma, L. Tang, F. Fan, J. Huang, X. Mei and Y. Ma, “SwinFusion: Cross-domain long-range learning for general image fusion via Swin Transformer,” *IEEE/CAA Journal of Automatica Sinica*, vol. 9, no. 7, pp. 1200–1217, 2022. doi
35. X. Zhang and Y. Demiris, “Visible and infrared image fusion using deep learning,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 8, pp. 10535–10554, 2023. doi
36. Y. Liu, X. Chen, H. Peng and Z. Wang, “Multi-focus image fusion with a deep convolutional neural network,” *Information Fusion*, vol. 36, pp. 191–207, 2017. doi
37. K. Ma, Z. Duanmu, H. Zhu, Y. Fang and Z. Wang, “Deep guided learning for fast multi-exposure image fusion,” *IEEE Transactions on Image Processing*, vol. 29, pp. 2808–2819, 2020. doi
38. Z. Zhao, H. Bai, J. Zhang, Y. Zhang, S. Xu, Z. Lin, R. Timofte and L. Van Gool, “CDDFuse: Correlation-driven dual-branch feature decomposition for multi-modality image fusion,” in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023, pp. 5906–5916. CVF arXiv
39. J. Ma, W. Yu, P. Liang, C. Li and J. Jiang, “FusionGAN: A generative adversarial network for infrared and visible image fusion,” *Information Fusion*, vol. 48, pp. 11–26, 2019. doi ·public

article transcription
40. H. Bai, J. Zhang, Z. Zhao, Y. Wu, L. Deng, Y. Cui, T. Feng and S. Xu, "Task-driven image fusion with learnable fusion loss," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025, pp. 7457–7468. CVF arXiv
41. Z. Zhao, H. Bai, Y. Zhu, J. Zhang, S. Xu, Y. Zhang, K. Zhang, D. Meng, R. Timofte and L. Van Gool, "DDFM: Denoising diffusion model for multi-modality image fusion," in *Proc. IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023, pp. 8082–8093. CVF arXiv
42. X. Yi, H. Xu, H. Zhang, L. Tang and J. Ma, "Text-IF: Leveraging semantic text guidance for degradation-aware and interactive image fusion," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024, pp. 27026–27035. CVF arXiv
43. D. Wang, J. Liu, X. Fan and R. Liu, "Unsupervised misaligned infrared and visible image fusion via cross-modality image generation and registration," in *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, 2022, pp. 3508–3515. doi arXiv
44. L. Tang, Y. Deng, Y. Ma, J. Huang and J. Ma, "SuperFusion: A versatile image registration and fusion network with semantic awareness," *IEEE/CAA Journal of Automatica Sinica*, vol. 9, no. 12, pp. 2121–2137, 2022. doi
45. A. Toet, "The TNO multiband image data collection," *Data in Brief*, vol. 15, pp. 249–251, 2017. doi
46. L. Tang, J. Yuan, H. Zhang, X. Jiang and J. Ma, "PIAFusion: A progressive infrared and visible image fusion network based on illumination aware," *Information Fusion*, vol. 83–84, pp. 79–92, 2022. doi MSRS
47. Q. Ha, K. Watanabe, T. Karasawa, Y. Ushiku and T. Harada, "MFNet: Towards real-time semantic segmentation for autonomous vehicles with multi-spectral scenes," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 5108–5115. doi
48. X. Jia, C. Zhu, M. Li, W. Tang and W. Zhou, "LLVIP: A visible-infrared paired dataset for low-light vision," in *Proc. IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, 2021, pp. 3496–3504. CVF arXiv
49. J. Cai, S. Gu and L. Zhang, "Learning a deep single image contrast enhancer from multi-exposure images," *IEEE Transactions on Image Processing*, vol. 27, no. 4, pp. 2049–2062, 2018. doi
50. X. Zhang, "Deep learning-based multi-focus image fusion: A survey and a comparative study," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 9, 2022. doi
51. F. Xu, J. Liu, Y. Song, H. Sun and X. Wang, "Multi-exposure image fusion techniques: A comprehensive review," *Remote Sensing*, vol. 14, no. 3, art. 771, 2022. doi
52. Z. Zhao, H. Bai, J. Zhang, Y. Zhang, K. Zhang, S. Xu, D. Chen, R. Timofte and L. Van Gool, "Equivariant multi-modality image fusion," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024, pp. 25912–25921. CVF arXiv
53. L. Tang, Y. Wang, M. Gong, Z. Li, Y. Deng, X. Yi, C. Li, H. Xu, H. Zhang and J. Ma, "VideoFusion: A spatio-temporal collaborative network for multi-modal video fusion," in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2026, pp. 19559–19569. CVF
54. N. Yokoya, C. Grohnfeldt and J. Chanussot, "Hyperspectral and multispectral data fusion: A comparative review of the recent literature," *IEEE Geoscience and Remote Sensing Magazine*, 2017. doi
55. H. R. Sheikh and A. C. Bovik, "Image information and visual quality," *IEEE Transactions on*

*Image Processing*, 2006. doi
56. H. Xu et al., "U2Fusion," official implementation accompanying the TPAMI paper, 2022. code
57. K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *ICLR*, 2015. arXiv
58. J. Kirkpatrick et al., "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, 2017. doi
59. Y. Zhang et al., "IFCNN," official implementation and errata accompanying the Information Fusion paper, 2020. code
60. K. He, X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition," *CVPR*, 2016. CVF
61. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," *CVPR*, 2009. doi
62. N. Silberman, D. Hoiem, P. Kohli and R. Fergus, "Indoor segmentation and support inference from RGBD images," *ECCV*, 2012. project
63. P. Dhariwal and A. Nichol, "Diffusion models beat GANs on image synthesis," *NeurIPS*, 2021. arXiv
64. J. Liu et al., "TarDAL," official implementation accompanying the CVPR paper, 2022. code
65. G. Jocher et al., "YOLOv5," Ultralytics object-detection software, 2020 onward. code
66. J. Ma et al., "FusionGAN," official implementation accompanying the Information Fusion paper, 2019. code
67. L. Tang, J. Yuan and J. Ma, "SeAFusion," official implementation accompanying the Information Fusion paper, 2022. code
68. T. Mertens, J. Kautz and F. Van Reeth, "Exposure Fusion," in *Proc. Pacific Graphics*, 2007. paper ·author project

# DLCV 11 • 3D Reconstruction: From Multi-View Geometry to NeRF and Gaussian Splatting

How a set of photos becomes a 3D model: camera geometry, structure from motion and multi-view stereo, neural radiance fields and signed distance fields, 3D Gaussian Splatting, mesh extraction, feed-forward reconstruction models, and how reconstructions are evaluated, with tested NumPy and OpenCV code.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/3d-reconstruction/

## What you'll learn

- Choose a scene representation for rendering, geometry or editing.
- Use camera projection, epipolar geometry and triangulation.
- Trace structure from motion and multi-view stereo.
- Compare neural fields, Gaussian splatting and feed-forward reconstruction.
- Evaluate held-out views and geometric accuracy separately.

## The big picture

You walk around a statue and take thirty photos with your phone. 3D reconstruction turns those photos into a 3D model you can spin, measure, 3D-print or place in a game. Two questions must be answered together: *where was the camera for each photo*, and *where is each piece of the statue's surface*. Neither is given, and each helps answer the other.

> **Explain like I'm five**
>
> Close one eye and hold up a finger: it jumps when you switch eyes. Things that are close jump a lot and things far away hardly move. Your brain uses this jump to feel depth. A computer does the same with photos taken from different places: it finds the same spot in two photos, measures how much it "jumped", and works out how far away it is. Do that for millions of spots and you have the shape. The newer methods add one more trick: they keep guessing the 3D scene, *draw* what each photo should look like, and fix the guess wherever the drawing does not match the real photo.

**Why it matters.** Reconstruction is how maps, film sets, cultural-heritage archives, robot environments, augmented-reality anchors and digital twins of factories are built. It is also a clean example of how deep learning changed computer vision: the classical pipeline is pure geometry and optimization; neural radiance fields (NeRF) [5] made the scene itself a trained network; 3D Gaussian Splatting (3DGS) [6] replaced the network with millions of small, explicit blobs that render in real time; and the newest models predict cameras and geometry in a single forward pass.

Figure 1 shows the roadmap of this tutorial. The classical stages on the left still feed almost every modern method: NeRF and 3DGS are trained on camera poses computed by structure-from-motion software such as COLMAP [2], [4].

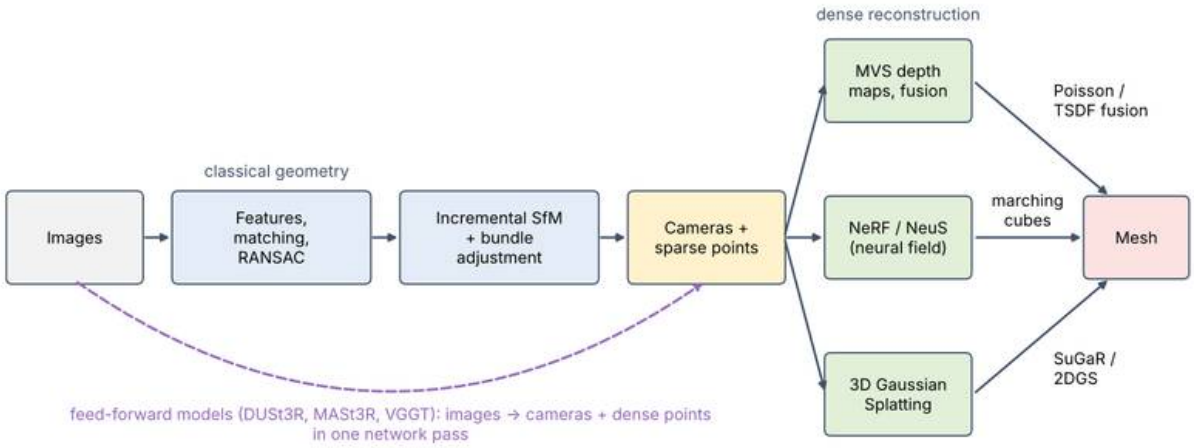



*Figure 1 — The reconstruction pipeline covered in this tutorial. Classical geometry (left) gives cameras and sparse points; dense methods (right) give a full model; feed-forward networks (dashed) shortcut the classical stages.*

This tutorial assumes you know keypoints and descriptor matching from IP 12 and the basics of deep networks from DLCV 01. The standard textbook for the geometry is Hartley and Zisserman [1].

**Related chapters.** DLCV 11A treats the dense correspondence problems behind reconstruction — stereo matching, monocular depth and optical flow — in depth, and DLCV 03B uses 3D geometry to detect objects.

## What 3D reconstruction is

**Plain version.** From photos of a static scene, estimate camera poses and a 3D representation of its shape and appearance.

**Precise version.** We are given $N$ images $I_1, \dots, I_N$ of a static scene. We want camera parameters $(K_i, R_i, \mathbf{t}_i)$ for each image (defined in the next section) and a scene model $\mathcal{S}$ such that rendering $\mathcal{S}$ with camera $i$ reproduces $I_i$. Two variants matter. **Geometric reconstruction** asks for an accurate surface; it is judged against laser or structured-light scans. **Novel-view synthesis** asks for realistic images from camera positions that were *not* photographed; it is judged against held-out photos. NeRF and 3DGS were designed for the second goal, which is why extracting a good surface from them took extra work.

The problem has a built-in limitation: from photos alone, the whole scene can be scaled up or

down without changing any image. A toy house photographed from close by and a real house photographed from far away can produce the same pictures. Reconstruction from images is therefore defined only **up to scale** (and up to the choice of the world coordinate frame) unless something else, such as a known distance or a calibrated stereo rig, fixes it [1].

### Seven ways to store a 3D scene

**Plain version.** Choose the representation for the operation: rendering, measurement, editing or learning. Figure 2 compares common forms on the same scene.

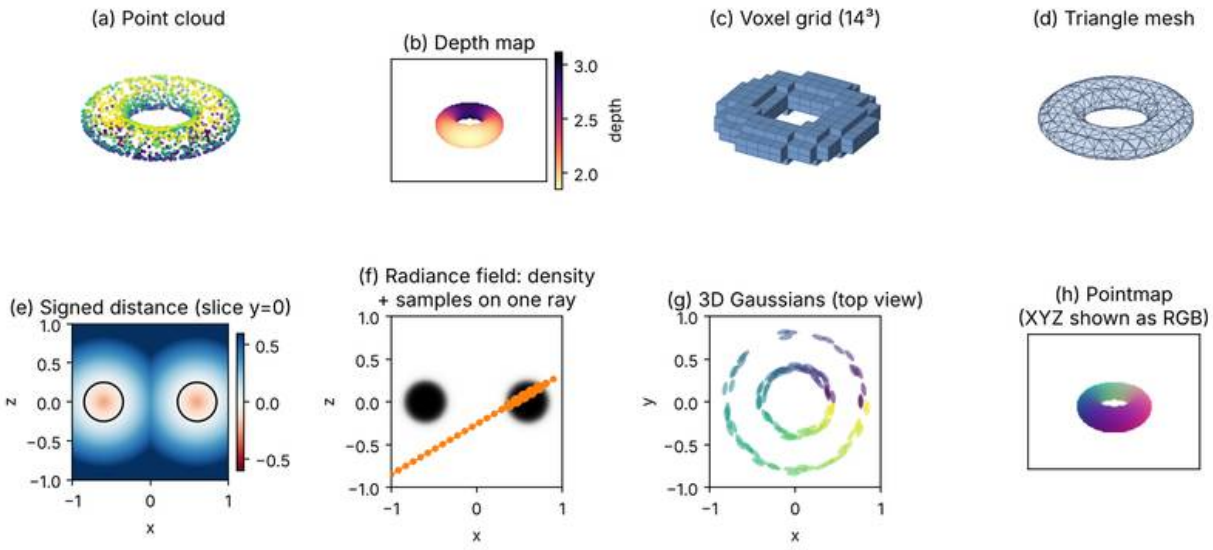



*Figure 2 — One shape, eight encodings: (a) point cloud, (b) depth map, (c) voxel grid, (d) triangle mesh, (e) signed distance function, (f) radiance field (density slice and samples along a ray), (g) 3D Gaussians, (h) a pointmap, the per-pixel 3D-point image predicted by feed-forward models.*

| Representation | What is stored | Strengths | Weaknesses |
|---|---|---|---|
| Point cloud | A list of 3D points, often with colour and normal | Simple; direct output of triangulation and depth sensors | No surface or connectivity; holes; hard to render nicely |
| Depth map | One distance per pixel of one camera | Fits image networks; easy to estimate per view | Covers one viewpoint only; must be fused |
| Voxel grid | A value (occupancy, colour) per cell of a 3D grid | Regular, easy to process with 3D convolutions | Memory grows with the cube of resolution |
| Mesh | Vertices and triangles | Compact, editable, what graphics engines and printers use | Fixed topology; hard to optimize directly from images |
| Signed distance function (SDF) | Distance to the nearest surface, negative inside | Watertight surface as the zero level set; normals from the gradient | Needs a closed surface; must be converted for rendering |

| Representation | What is stored | Strengths | Weaknesses |
|---|---|---|---|
| Radiance field | Density and view-dependent colour at every 3D point | Photorealistic novel views, including thin and fuzzy structures | Slow to render with ray marching; surface is implicit |
| 3D Gaussians | Many small ellipsoids with colour and opacity | Real-time rendering; explicit and editable | Many primitives (memory); surfaces are not well defined |

An SDF can be a stored grid or a function computed by a network: DeepSDF [20] learns a network that outputs signed distance, and Occupancy Networks [21] learn the probability that a point is inside the object. Both are called **neural implicit** representations, because the surface is given implicitly as a level set of a learned function.

## Camera geometry essentials

**Plain version.** The pinhole model maps a 3D point to image coordinates through camera transformation, projection and perspective division.

### The pinhole camera: intrinsics and extrinsics

A camera does two things to a 3D point. It first describes the point relative to itself (how far to the left, up and ahead of the camera it is), then flattens it onto the sensor by dividing by its distance. In symbols, a world point $\mathbf{X} \in \mathbb{R}^3$ is first moved into the camera's coordinate frame, then projected:

$$\mathbf{X}_c = R\,\mathbf{X} + \mathbf{t}, \qquad \lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K\mathbf{X}_c, \qquad K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix},$$

where $R$ (a $3 \times 3$ rotation) and $\mathbf{t}$ (a translation) are the **extrinsics**: they say where the camera is and which way it points; $\mathbf{X}_c$ is the point in camera coordinates, with the $z$ axis along the viewing direction; $K$ holds the **intrinsics**: focal lengths $f_x, f_y$ in pixels, the principal point $(c_x, c_y)$ where the optical axis meets the image, and a skew $s$ that is zero for real sensors; $(u, v)$ is the pixel; and $\lambda$ equals the depth $z_c$ of the point. Dividing by depth is what makes far things look small. The camera centre in world coordinates is $C = -R^\top \mathbf{t}$. Combining both steps gives the $3 \times 4$ **projection matrix** $P = K[R \mid \mathbf{t}]$.

Real lenses also bend straight lines (radial distortion); calibration tools estimate extra distortion coefficients and undistort the images first, so the rest of this tutorial assumes an ideal pinhole. The homogeneous coordinates used here, with their trailing 1, are the same as in the homography note of IP 02 (Homography: the projective transform).

### Two views: epipolar geometry

**Plain version.** If you see a point in one photo, you do not know how far away it is, so in 3D it could lie anywhere on a ray. In the second photo, that ray appears as a *line*. The true match must sit on that line. This turns a 2D search into a 1D search and, in reverse, lets us recover how the two cameras are placed from matched points alone.

Figure 3 (a) shows the geometry. The two camera centres and the 3D point **X** span a plane, the **epipolar plane**. It cuts each image in an **epipolar line**. Any point **X**′ on the same ray of camera 1 has the same image **x** in camera 1, and its image in camera 2 moves along that line.

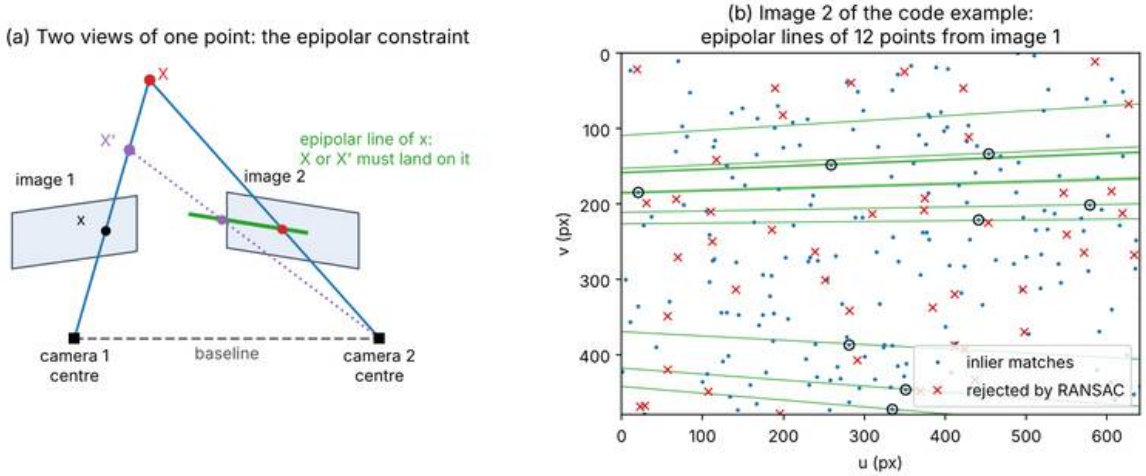



*Figure 3 — (a) The epipolar constraint: all points on the ray through x project onto one line in image 2. (b) Image 2 of the code example below: epipolar lines computed from the estimated essential matrix pass through the matching points (circled).*

In algebra, let $\hat{\mathbf{x}}_1 = K_1^{-1}(u_1, v_1, 1)^\top$ and $\hat{\mathbf{x}}_2 = K_2^{-1}(u_2, v_2, 1)^\top$ be the *normalized* coordinates of a matched pair (pixels with the intrinsics removed). If camera 2 sits at rotation $R$ and translation $\mathbf{t}$ relative to camera 1, every correct match satisfies

$$\hat{\mathbf{x}}_2^\top E\,\hat{\mathbf{x}}_1 = 0, \qquad E = [\mathbf{t}]_\times R, \qquad [\mathbf{t}]_\times = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix},$$

where $E$ is the $3 \times 3$ **essential matrix** [7] and $[\mathbf{t}]_\times$ is the matrix that computes the cross product with $\mathbf{t}$. The equation says that the two viewing rays and the baseline lie in one plane. In pixel coordinates the same constraint reads $\mathbf{x}_2^\top F \mathbf{x}_1 = 0$ with the **fundamental matrix** $F = K_2^{-\top} E K_1^{-1}$, where $\mathbf{x}_1, \mathbf{x}_2$ are homogeneous pixel coordinates. $F\mathbf{x}_1$ is exactly the epipolar line in image 2 drawn in Figure 3 (b).

In plain words: $E$ **encodes the relative pose of two calibrated cameras, and** $F$ **does the same when the intrinsics are unknown.** $E$ has five degrees of freedom: three for the rotation and two for the *direction* of the translation, whose length is lost to the scale ambiguity. Five matches therefore suffice, the minimal case solved by Nistér's five-point algorithm [9]. With eight or more matches

the constraint is linear in the nine entries of $E$ or $F$: the eight-point algorithm [7], [8] stacks one equation per match and takes the null vector with the singular value decomposition.

*Online only: The normalized eight-point algorithm — https://blog.shyandram.dev/en/learn/deep-learning/3d-reconstruction/notes/the-normalized-eight-point-algorithm/*

Once $E$ is known, it can be factored into $R$ and the direction of $\mathbf{t}$. The factorization gives four candidate poses; only one places the points in front of both cameras, which is called the **cheirality** (or chirality) check [1].

### Triangulation

**Plain version.** With both cameras known, a matched pair gives two rays. Where they (almost) cross is the 3D point. Noise means they never meet exactly, so we pick the point that best explains both observations.

The simplest method is linear. Each image gives two linear equations in the unknown point (the pinhole equation with the unknown depth eliminated), and the four equations are solved, like the eight-point system, as the null vector of a small matrix with the SVD [1]. The quality of a triangulated point is judged by its **reprojection error**: project it back into each image and measure the pixel distance to the observed point.

*Online only: Linear triangulation (DLT) — https://blog.shyandram.dev/en/learn/deep-learning/3d-reconstruction/notes/linear-triangulation-dlt/*

Accuracy depends strongly on the angle between the two rays. For two parallel cameras with focal length $f$ and baseline $B$, a point at depth $Z$ shifts by the **disparity** $d = fB/Z$ pixels between the images, so a small disparity error $\delta d$ gives a depth error $\delta Z \approx Z^2 \delta d/(fB)$. Depth error grows with the square of distance and shrinks with a wider baseline; Exercise 1 puts numbers on this.

### Code: relative pose and triangulation

The snippet builds a synthetic scene, projects it into two cameras with 0.5-pixel noise, replaces 15% of the matches by random wrong ones, and then recovers the relative pose with OpenCV. `cv2.findEssentialMat` runs RANSAC around the five-point solver (its documentation names Nistér's algorithm [9]); `cv2.recoverPose` performs the cheirality check. RANSAC's answer comes from a minimal sample, so we refit $E$ linearly on all inliers, the "refit on the consensus set" step described in the RANSAC note of IP 12 (RANSAC: fitting a model despite outliers).

```
import numpy as np
import cv2

rng = np.random.default_rng(0)
K = np.array([[800.0, 0, 320], [0, 800.0, 240], [0, 0, 1]])   # focal length 800 px,
 ↪  principal point (320, 240)

def look_at(C, target):
    """Rotation of a camera at centre C that looks at target (rows: camera x, y, z axes)."""
```

```
    z = (target - C) / np.linalg.norm(target - C)
    x = np.cross([0.0, 1.0, 0.0], z); x /= np.linalg.norm(x)
    return np.stack([x, np.cross(z, x), z])

def project(K, R, t, X):
    """Pinhole projection of world points X (N, 3) to pixels (N, 2)."""
    x = (X @ R.T + t) @ K.T          # world -> camera -> homogeneous pixels
    return x[:, :2] / x[:, 2:]       # divide by depth

def linear_E(x1, x2, K):
    """Least-squares E from many matches: each match gives one linear equation in E's 9
     ↪  entries."""
    Ki = np.linalg.inv(K)
    a = np.c_[x1, np.ones(len(x1))] @ Ki.T              # normalized camera coordinates
    b = np.c_[x2, np.ones(len(x2))] @ Ki.T
    A = np.einsum('ni,nj->nij', b, a).reshape(-1, 9)     # row n encodes b_n^T E a_n = 0
    E = np.linalg.svd(A)[2][-1].reshape(3, 3)            # best null vector of A
    U, _, Vt = np.linalg.svd(E)
    return U @ np.diag([1.0, 1.0, 0.0]) @ Vt             # nearest valid essential matrix

X = rng.uniform([-3, -2, 4], [3, 2, 8], size=(300, 3))          # 300 scene points
C2 = np.array([2.0, 0.0, 0.0])                                   # camera 2 centre: 2 units
 ↪  to the right
R2 = look_at(C2, np.array([0.0, 0.0, 6.0])); t2 = -R2 @ C2       # camera 1 is the world
 ↪  frame
x1 = project(K, np.eye(3), np.zeros(3), X) + rng.normal(0, 0.5, (300, 2))   # 0.5 px noise
x2 = project(K, R2, t2, X) + rng.normal(0, 0.5, (300, 2))
x2[:45] = rng.uniform([0, 0], [640, 480], (45, 2))               # 45 wrong matches (15 %)

def pose_errors(R, t):
    rot = np.degrees(np.arccos(np.clip((np.trace(R.T @ R2) - 1) / 2, -1, 1)))
    return rot, np.degrees(np.arccos(t.ravel() @ t2 / np.linalg.norm(t2)))

E, mask = cv2.findEssentialMat(x1, x2, K, method=cv2.RANSAC, prob=0.999, threshold=1.5)
_, R, t, mask = cv2.recoverPose(E, x1, x2, K, mask=mask)
inl = mask.ravel() > 0
print(f"inliers kept: {inl.sum()} of 300 | wrong matches rejected: {(~inl[:45]).sum()} of
 ↪  45")
print("RANSAC estimate      : rotation error %.2f deg, translation direction error %.2f deg"
 ↪  % pose_errors(R, t))
_, R, t, _ = cv2.recoverPose(linear_E(x1[inl], x2[inl], K), x1[inl], x2[inl], K)
print("refit on all inliers : rotation error %.2f deg, translation direction error %.2f deg"
 ↪  % pose_errors(R, t))
print(f"length of estimated t: {np.linalg.norm(t):.2f} (true baseline
 ↪  {np.linalg.norm(C2):.2f})")

P1 = K @ np.hstack([np.eye(3), np.zeros((3, 1))])     # P1 = K [I | 0]
P2 = K @ np.hstack([R, t])                            # P2 = K [R | t]
Xh = cv2.triangulatePoints(P1, P2, x1[inl].T, x2[inl].T)
Xe = (Xh[:3] / Xh[3]).T                               # homogeneous -> Euclidean
res = np.r_[np.linalg.norm(project(K, np.eye(3), np.zeros(3), Xe) - x1[inl], axis=1),
            np.linalg.norm(project(K, R, t.ravel(), Xe) - x2[inl], axis=1)]
print(f"reprojection error: RMS {np.sqrt(np.mean(res ** 2)):.2f} px")
rel = np.linalg.norm(2.0 * Xe - X[inl], axis=1) / np.linalg.norm(X[inl], axis=1)   # scale
 ↪  fixed by the true baseline
```

```
print(f"3D error after fixing the scale: median {100 * np.median(rel):.2f} % of the
 ↪  distance")
```

```
inliers kept: 250 of 300 | wrong matches rejected: 44 of 45
RANSAC estimate      : rotation error 0.30 deg, translation direction error 0.42 deg
refit on all inliers : rotation error 0.07 deg, translation direction error 0.46 deg
length of estimated t: 1.00 (true baseline 2.00)
reprojection error: RMS 0.40 px
3D error after fixing the scale: median 0.31 % of the distance
```

These numbers are from this synthetic run only. RANSAC rejects 44 of the 45 wrong matches. Refitting on all inliers reduces the rotation error from 0.30° to 0.07°; the translation direction stays within half a degree either way. The estimated translation has length exactly 1: OpenCV returns only its direction, because the images cannot tell the true baseline of 2. Once the scale is supplied from outside, the triangulated points are within about 0.3% of their true distance. The reprojection error of 0.40 px is below the 0.5 px noise level, as expected when the points are fitted to their own observations.

## Structure from motion

**Plain version.** Structure from motion (SfM) does the two-view trick for hundreds of photos at once: it finds matching points across all of them, places the cameras one by one, triangulates the points, and keeps polishing everything so that all 3D points project onto their observed pixels.

### Features, matching and geometric verification

First, SfM must know which pixels in different photos show the same physical point. It starts with local features. Each image yields keypoints with descriptors, such as SIFT [11] (IP 12), and descriptors are matched between image pairs (IP 12, Matching). Many matches are wrong, so each pair is **geometrically verified**: RANSAC [10] fits a fundamental matrix (or an essential matrix or a homography) and keeps the pair only if enough matches agree [2]. The verified pairs form a **scene graph**: images are nodes, and an edge means "these two images see the same points".

### Incremental reconstruction

COLMAP's paper describes the standard incremental pipeline [2], which grew out of earlier systems for Internet photo collections such as Photo Tourism [12]:

1. **Initialize** with a carefully chosen pair of images that share many matches and have a wide baseline; estimate their relative pose and triangulate the shared points.
2. **Register** a new image by solving the Perspective-n-Point (PnP) problem: find the camera pose from matches between its keypoints and already triangulated 3D points, inside RANSAC.

3. **Triangulate** new points seen by the new image and at least one registered image.
4. **Bundle adjust** (below) to refine all cameras and points together, then **filter** points with large reprojection error.
5. Repeat from step 2 until no more images can be registered.

The COLMAP paper improves each of these steps, including how the next image is chosen and how points are triangulated. Its central lesson is in step 4: registration and triangulation must be interleaved with repeated bundle adjustment, re-triangulation and outlier filtering, because without that refinement small errors accumulate as images are added and the reconstruction drifts [2].

### Bundle adjustment

**Plain version.** Bundle adjustment (BA) is the final polish: move every camera and every 3D point a little until all the "bundles" of rays line up with the pixels where the points were actually seen.

BA solves the nonlinear least-squares problem

$$\min_{\{P_c\},\{\mathbf{X}_k\}} \sum_{(c,k)\in O} \rho\Big(\|\pi(P_c,\mathbf{X}_k)-\mathbf{x}_{ck}\|^2\Big),$$

where $P_c$ are the parameters of camera $c$ (pose, and intrinsics if unknown), $\mathbf{X}_k$ are the 3D points, $O$ is the set of observations (point $k$ seen in image $c$), $\mathbf{x}_{ck}$ is the observed pixel, $\pi$ is the pinhole projection, and $\rho$ is a robust loss that down-weights outliers [2], [13]. The standard solver is Levenberg–Marquardt, a damped Gauss–Newton method. It is fast because each residual depends on only one camera and one point, so the Jacobian is almost all zeros [2], [13]. BA cannot fix the global scale and coordinate frame, and it finds only a local minimum, so it needs the good starting point that the incremental steps provide.

*Online only: Bundle adjustment in code — https://blog.shyandram.dev/en/learn/deep-learning/3d-reconstruction/notes/bundle-adjustment-in-code/*

### COLMAP in practice

COLMAP is the open-source SfM and MVS pipeline released with [2] and [3], with a graphical and a command-line interface [4]. It outputs the camera intrinsics and poses plus a sparse point cloud, the standard input of the dense methods below: the NeRF paper takes the poses of its real scenes from COLMAP [5], and 3DGS initializes its Gaussians from the SfM points [6]. Every later stage inherits the errors made here (see *Pose errors* under failure modes).

## Multi-view stereo

**Plain version.** SfM gives cameras and a sparse set of points. Multi-view stereo (MVS) fills in the rest: for every pixel of a reference image it searches, along the epipolar line in the other images, for the depth at which all views agree on what the surface looks like.

### Photo-consistency and plane sweep

The core test is **photo-consistency**: if a pixel's surface point is at depth $Z$, then the small patch around it should look the same in every image that sees that point. A depth hypothesis is scored by a matching cost such as the sum of squared differences (SSD) or normalized cross-correlation between patches.

The **plane sweep** algorithm [14] tests many depths efficiently. Sweep a plane, parallel to the reference image, through a range of depths $Z_1, \dots, Z_D$. For each depth, warp every source image onto the reference view as if the whole scene lay on that plane; the warp for one plane is a homography. Compute the cost per pixel, stack the costs into a **cost volume** of size $D \times H \times W$, and choose for each pixel the depth with the lowest cost:

$$\hat{Z}(u,v) = \arg\min_{k} \sum_{(u',v') \in \mathcal{W}(u,v)} \left( I_{\text{ref}}(u',v') - \tilde{I}^{(k)}_{\text{src}}(u',v') \right)^2,$$

where $\tilde{I}^{(k)}_{\text{src}}$ is the source image warped by the plane at depth $Z_k$ and $\mathcal{W}(u,v)$ is a small window around the pixel. For two parallel cameras (a *rectified* pair), the plane homography reduces to a horizontal shift by the disparity $fB/Z_k$, which keeps the code short.

*Online only: The plane-induced homography — https://blog.shyandram.dev/en/learn/deep-learning/3d-reconstruction/notes/the-plane-induced-homography/*

```
import numpy as np
from scipy import ndimage as ndi

rng = np.random.default_rng(0)
H, W, f, B = 120, 160, 150.0, 0.5                    # image size, focal length (px),
 ↪  baseline
tex = ndi.gaussian_filter(rng.random((400, 400)), 1.5)   # random texture painted on the
 ↪  world
tex = (tex - tex.min()) / np.ptp(tex)

def render(cx):
    """Ray-cast a camera at (cx, 0, 0) looking down +Z: a square at Z=5 in front of a wall at
     ↪  Z=8."""
    v, u = np.mgrid[0:H, 0:W].astype(float)
    dx, dy = (u - W / 2) / f, (v - H / 2) / f          # ray direction (dx, dy, 1)
    Xs, Ys = cx + 5 * dx, 5 * dy                        # hit point on the plane Z = 5
    front = (np.abs(Xs) < 1) & (np.abs(Ys) < 1)         # inside the square?
    Z = np.where(front, 5.0, 8.0)
    X, Y = cx + Z * dx, Z * dy
    img = ndi.map_coordinates(tex, [200 + 40 * Y + 100 * front, 200 + 40 * X], order=1)
    img[~front & (X > 1.5) & (Y < -0.5)] = 0.5          # a textureless patch on the wall
    return img, Z

ref, Z_true = render(0.0)                               # reference camera
src, _ = render(B)                                      # source camera, B to the right

depths = 1 / np.linspace(1 / 10, 1 / 3.5, 80)           # depth hypotheses, uniform in 1/Z
cost = np.empty((len(depths), H, W))
for k, Z in enumerate(depths):
```

```
    d = f * B / Z                                        # the plane Z induces a horizontal
     ↪  shift d
    warped = ndi.shift(src, (0, d), order=1, mode='nearest')   # src pixel u - d lands on u
    cost[k] = ndi.uniform_filter((ref - warped) ** 2, 7)       # windowed SSD
     ↪  photo-consistency
Z_est = depths[cost.argmin(0)]                           # winner-take-all depth per pixel

ok = np.abs(Z_est - Z_true) / Z_true < 0.05
flat = (Z_true == 8) & (ref == 0.5)
inner = np.zeros_like(ok); inner[8:-8, 20:-8] = True    # ignore borders the source camera
 ↪  cannot see
print(f"pixels within 5% of the true depth: {100 * ok[inner & ~flat].mean():.1f}% (textured),
 ↪  "
      f"{100 * ok[inner & flat].mean():.1f}% (textureless patch)")
```

```
pixels within 5% of the true depth: 95.3% (textured), 14.8% (textureless patch)
```

On textured surfaces 95.3% of the evaluated pixels get a depth within 5% of the truth in this synthetic run. In the textureless patch only 14.8% do, because every depth hypothesis matches a flat grey patch equally well. Figure 4 (d) shows this directly: the cost curve of the textured pixel has one sharp minimum at the true depth; the curve of the textureless pixel is flat. The remaining errors on textured surfaces sit at the left border (seen by one camera only) and next to the square, where part of the background is hidden from the second camera (occlusion).

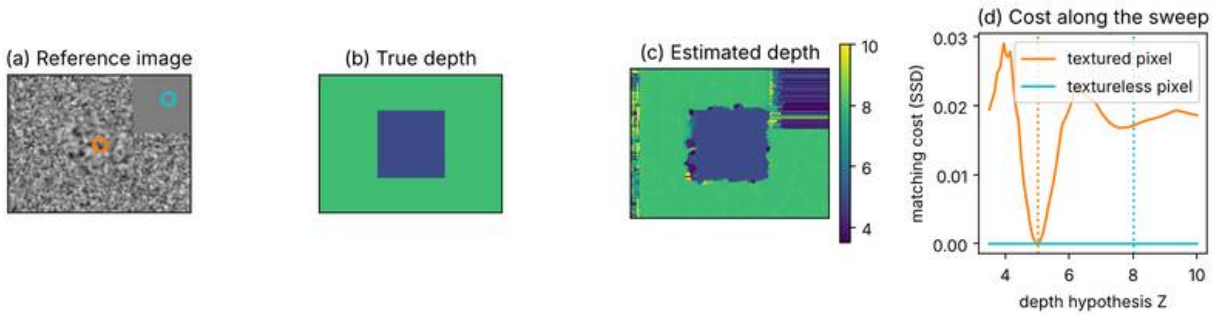



*Figure 4 — Plane-sweep stereo on a synthetic rectified pair. (a) Reference image with a textureless patch (top right). (b) True depth. (c) Winner-take-all depth from a 7 × 7 SSD window. (d) Cost curves for the two circled pixels; dotted lines mark the true depths.*

### From depth maps to one model

A depth map per image is not yet one model. **Depth-map fusion** merges them. The volumetric method of Curless and Levoy [15] stores, in a voxel grid, a running weighted average of the signed distance to the surface measured by each depth map:

$$D_{i+1}(\mathbf{x}) = \frac{W_i(\mathbf{x}) D_i(\mathbf{x}) + w_{i+1}(\mathbf{x})\, d_{i+1}(\mathbf{x})}{W_i(\mathbf{x}) + w_{i+1}(\mathbf{x})}, \qquad W_{i+1}(\mathbf{x}) = W_i(\mathbf{x}) + w_{i+1}(\mathbf{x}),$$

where $d_{i+1}(\mathbf{x})$ is the signed distance from voxel $\mathbf{x}$ to the surface seen in depth map $i+1$ along the viewing ray, $w_{i+1}$ is its confidence weight, and $D_i$, $W_i$ are the accumulated distance and weight after $i$ maps. The surface is the zero level set of $D$, extracted by marching cubes (below). In practice the distances are truncated near the surface, hence the name **TSDF** (truncated signed distance function) fusion. COLMAP's MVS [3] instead estimates a depth and a normal per pixel with per-pixel view selection and a multi-view geometric consistency check, then fuses the consistent depths into a dense point cloud.

### Learning-based MVS

**Plain version.** The hand-made parts of MVS, the patch cost and the rules that clean up the cost volume, can be learned from data.

MVSNet [16] keeps the plane-sweep structure but makes every step differentiable. A shared CNN extracts features from all images, and the features of each source view are warped onto fronto-parallel planes of the reference camera (differentiable homography warping). A **variance-based cost** merges any number of views into one cost volume: low variance means the views agree. A 3D CNN regularizes the volume, a softmax along the depth axis gives a probability per depth, and the expected depth under that distribution (the "soft argmin") is a continuous output that can be trained with a regression loss [16]. MVSNet was trained on DTU [37] and evaluated on Tanks and Temples [38] without fine-tuning [16]. Its authors motivate learning with the low-textured, specular and reflective regions where hand-made costs fail [16].

## Neural radiance fields (NeRF)

**Plain version.** NeRF stores a scene as a small neural network that answers one question for any point in space and any viewing direction: "how much stuff is here, and what colour does it send toward me?" To make an image, it shoots a ray through each pixel, asks the network at many points along the ray, and blends the answers. Training adjusts the network until the rendered images match the photos.

### The scene as a function

NeRF [5] is a multilayer perceptron $F_\Theta : (\mathbf{x}, \mathbf{d}) \mapsto (\sigma, \mathbf{c})$, where $\mathbf{x} \in \mathbb{R}^3$ is a position, $\mathbf{d}$ a viewing direction, $\sigma \geq 0$ the **volume density** at $\mathbf{x}$ and $\mathbf{c}$ the RGB colour emitted toward $\mathbf{d}$. The network is built so that density depends on position only, while colour also depends on direction, which lets NeRF reproduce shiny highlights that change with the viewpoint [5]. One network represents one scene; there is no training set of many scenes.

*Online only: NeRF (Mildenhall et al., 2020) — https://blog.shyandram.dev/en/learn/deep-learning/3d-reconstruction/papers/nerf-mildenhall-et-al-2020/*

### Volume rendering along a ray

**Plain version.** Picture the ray as passing through fog of varying thickness. Each bit of fog both glows with its own colour and blocks the light from behind it. The pixel colour is the sum of all the glows, each dimmed by the fog in front of it.

A camera ray is $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ with origin $\mathbf{o}$ (the camera centre), direction $\mathbf{d}$ and distance $t$ between near and far bounds $t_n$ and $t_f$. Following classical volume rendering [17], the expected colour is

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\,\sigma(\mathbf{r}(t))\,\mathbf{c}(\mathbf{r}(t),\mathbf{d})\,dt, \qquad T(t) = \exp\!\left(-\int_{t_n}^{t}\sigma(\mathbf{r}(s))\,ds\right),$$

where $T(t)$ is the **transmittance**, the probability that the ray travels from $t_n$ to $t$ without being stopped [5]. NeRF evaluates it with $N$ samples $t_1 < \ldots < t_N$ along the ray:

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} w_i\,\mathbf{c}_i, \qquad w_i = T_i\,\alpha_i, \qquad \alpha_i = 1 - e^{-\sigma_i\delta_i}, \qquad T_i = \prod_{j=1}^{i-1}(1-\alpha_j),$$

where $\sigma_i, \mathbf{c}_i$ are the network outputs at sample $i$, $\delta_i = t_{i+1} - t_i$ is the spacing, $\alpha_i$ is the **opacity** of segment $i$, $T_i$ is the light that survives the segments before it, and $w_i$ is the weight of sample $i$ in the pixel. This is exactly front-to-back **alpha compositing** of semi-transparent layers [5]. The sum of the weights, $\sum_i w_i$, is the pixel's total opacity, and $\sum_i w_i t_i$ is an estimate of depth. Every step is differentiable, so the photo error can be pushed back into the network weights.

The snippet checks the quadrature against an exact answer. The ray crosses blue fog, then a red surface layer, then reaches a white background; for piecewise-constant density the integrals have closed forms.

```
import numpy as np

# One ray from t = 0 to t = 6: blue fog on [1, 2] (density 0.5), a red surface on [3, 3.5]
 ↪  (density 4),
# white background behind everything.
slabs = [(1.0, 2.0, 0.5, np.array([0.2, 0.4, 1.0])),
         (3.0, 3.5, 4.0, np.array([1.0, 0.1, 0.1]))]
bg = np.ones(3)

def field(t):
    """Density sigma(t) and colour c(t) at sample positions t."""
    sigma, c = np.zeros_like(t), np.zeros(t.shape + (3,))
    for a, b, s, col in slabs:
        inside = (t >= a) & (t < b)
        sigma[inside], c[inside] = s, col
    return sigma, c

def render(n):
    """NeRF quadrature: alpha_i = 1 - exp(-sigma_i delta_i), T_i = prod_{j<i} (1 - alpha_j),
     ↪  w_i = T_i alpha_i."""
    edges = np.linspace(0, 6, n + 1)
    t, delta = (edges[:-1] + edges[1:]) / 2, np.diff(edges)
    sigma, c = field(t)
```

```
    alpha = 1 - np.exp(-sigma * delta)
    T = np.concatenate([[1.0], np.cumprod(1 - alpha)[:-1]])
    w = T * alpha
    T_end = T[-1] * (1 - alpha[-1])                    # light that reaches the background
    return w @ c + T_end * bg, w.sum(), w @ t

# Analytic answer for piecewise-constant density (exact integrals of the volume rendering
 ↪  equation)
C, acc, depth, T0 = np.zeros(3), 0.0, 0.0, 1.0
for a, b, s, col in slabs:
    L = b - a
    absorbed = T0 * (1 - np.exp(-s * L))
    C, acc = C + absorbed * col, acc + absorbed
    depth += T0 * (a * (1 - np.exp(-s * L)) + (1 - np.exp(-s * L)) / s - L * np.exp(-s * L))
    T0 *= np.exp(-s * L)
C += T0 * bg
print(f"analytic : colour {np.round(C, 4)}, opacity {acc:.4f}, depth {depth:.4f}")
for n in [16, 64, 256, 1024]:
    Cq, accq, dq = render(n)
    print(f"N = {n:4d} : colour {np.round(Cq, 4)}, opacity {accq:.4f}, depth {dq:.4f}")
```

```
analytic : colour [0.6852 0.2919 0.528 ], opacity 0.9179, depth 2.2373
N =   16 : colour [0.7498 0.3318 0.5195], opacity 0.8466, depth 2.1655
N =   64 : colour [0.7006 0.2986 0.5232], opacity 0.9040, depth 2.2263
N =  256 : colour [0.689  0.2934 0.5267], opacity 0.9146, depth 2.2348
N = 1024 : colour [0.6862 0.2923 0.5277], opacity 0.9171, depth 2.2367
```

The quadrature converges to the analytic colour, opacity and depth as the number of samples grows. With 16 samples it is visibly off, because the thin surface layer falls into one or two coarse bins. This is why NeRF does not sample uniformly: a coarse network with 64 samples per ray first finds where the weights are large, and a fine network adds 128 samples concentrated there (hierarchical sampling) [5]. Note also the depth: 2.24 (2.23 with the 256 samples of Figure 5), between the fog and the surface, where no surface exists. Semi-transparent "fog" in the wrong place is the origin of the floater artifacts discussed under failure modes. Figure 5 plots the density, transmittance and weights for this ray.

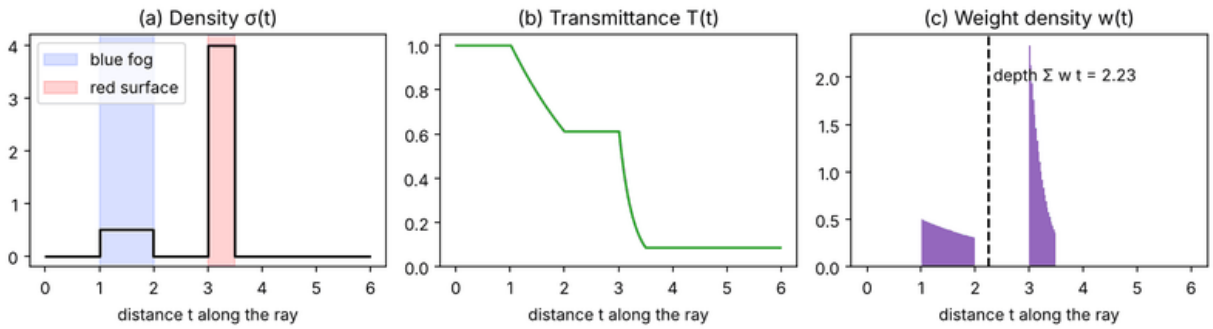



*Figure 5 — Volume rendering of the ray in the code: (a) density, (b) transmittance, (c) weights per unit length. The expected depth (dashed) lies between the fog and the surface, where nothing is.*

### Positional encoding

**Plain version.** A plain MLP fed with raw $(x, y, z)$ learns smooth, blurry functions. Feeding it sines and cosines of the coordinates at many frequencies lets it represent sharp edges and fine texture.

NeRF maps each coordinate $p$ (normalized to $[-1, 1]$) to

$$\gamma(p) = (\sin(2^0\pi p), \cos(2^0\pi p), \ldots, \sin(2^{L-1}\pi p), \cos(2^{L-1}\pi p)),$$

where $L$ is the number of frequency bands: $L = 10$ for position and $L = 4$ for direction [5]. Without the encoding, the paper's results lose their fine detail [5]. Tancik et al. [18] explained why: coordinate MLPs are biased toward low frequencies (a spectral bias), and a Fourier feature mapping of the input removes this bias. The sinusoids resemble the positional information of Transformers (ML 05), but here they encode continuous coordinates rather than token positions.

### Training, and why it is slow

Training needs posed images and nothing else. Each step samples a batch of pixels (rays) from the training images, renders them with the coarse and fine networks, and minimizes the squared colour error [5]. The cost is the problem. Every pixel needs hundreds of network evaluations, and a scene needs a few hundred thousand steps. The paper reports 100–300k iterations, about 1–2 days on one NVIDIA V100 GPU, per scene [5]. Rendering is slow for the same reason: the paper uses 256 network queries per ray, between 150 and 200 million per image, and reports about 30 seconds per frame on a V100 [5].

### Making it fast: hash grids

**Plain version.** Instead of asking one big network to remember everything, store learned features in lookup tables at several resolutions, and use a tiny network to decode them. Looking up a table is far cheaper than running many network layers.

Instant-NGP [19] replaces the frequency encoding with a **multiresolution hash encoding**. Space is covered by $L$ grids, from coarse to fine. Each grid stores trainable feature vectors at its corners, but in a table of fixed size $T$; on fine grids, where there are far more corners than table entries, a hash function decides which entry a corner uses, so several corners share one entry. To encode a point, the features of the corners around it are interpolated on every level, concatenated and passed to a small MLP. The paper does not resolve the shared entries explicitly; it relies on the coarse levels and the MLP to sort them out [19]. The authors report training of high-quality results "in a matter of seconds" and rendering in tens of milliseconds at 1920 × 1080 on one GPU [19]. Grid-based encodings of this kind are now common in fast NeRF variants.

*Online only: The multiresolution hash encoding — https://blog.shyandram.dev/en/learn/deep-learning/3d-reconstruction/notes/the-multiresolution-hash-encoding/*

## Neural implicit surfaces: NeuS

**Plain version.** NeRF's density does not have to form a clean surface: it can be a cloud. If you want a surface, let the network output a *signed distance* instead, and convert distance to opacity in a way that makes the rendered weight peak exactly on the surface.

NeuS [22] represents the scene as an SDF network $f(\mathbf{x})$ whose zero level set is the surface. It converts the SDF into opacity through the sigmoid $\Phi_s(x) = (1 + e^{-sx})^{-1}$, where $s$ is a learned sharpness that grows during training. The authors show that the obvious choice, using the logistic density $\phi_s(f) = \Phi_s'(f)$ directly as the volume density, gives weights that peak *before* the surface (the bias), and they derive an opacity that is unbiased to first order and still handles occlusion [22]. In discrete form,

$$\alpha_i = \max\!\left( \frac{\Phi_s(f(\mathbf{p}_i)) - \Phi_s(f(\mathbf{p}_{i+1}))}{\Phi_s(f(\mathbf{p}_i))},\ 0 \right),$$

where $\mathbf{p}_i$ is the $i$-th sample on the ray; the weights then follow the same compositing formula as NeRF. An **Eikonal** term $(\|\nabla f\| - 1)^2$ keeps $f$ a valid distance function [22]. Because the surface is a level set, a mesh follows directly from marching cubes. NeuS reconstructs objects without foreground masks, and it was evaluated on DTU [37] among other data [22].

*Online only: Why NeuS weights peak on the surface — https://blog.shyandram.dev/en/learn/deep-learning/3d-reconstruction/notes/why-neus-weights-peak-on-the-surface/*

## 3D Gaussian Splatting

**Plain version.** Instead of a network you must query along every ray, 3DGS represents the scene with millions of small, coloured, semi-transparent ellipsoids, like a cloud of soft paint drops. To make an image it projects each drop onto the screen as a soft ellipse and stacks them from front to back. No ray marching, no network: just sorting and blending, which a GPU does very fast.

### The Gaussian primitive

Each blob is a 3D bell curve: dense at its centre, fading smoothly outward, and stretched more in some directions than in others. Precisely, each primitive [6] has a centre $\boldsymbol{\mu} \in \mathbb{R}^3$, a covariance $\Sigma$ that sets its size and orientation, an opacity $o \in [0, 1]$ and a colour stored as spherical-harmonic coefficients (a small set of smooth functions on the sphere, so the colour can change with the viewing direction). Its footprint in space is

$$G(\mathbf{x}) = \exp\!\left(-\tfrac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right), \qquad \Sigma = R\, S\, S^\top R^\top,$$

where $S = \mathrm{diag}(s_1, s_2, s_3)$ holds the three axis lengths and $R$ is a rotation stored as a unit quaternion [6]. The factorization guarantees that $\Sigma$ stays a valid (positive semi-definite) covariance during gradient descent, which optimizing the six entries of $\Sigma$ directly would not [6].

### Projection to the image

**Plain version.** A 3D ellipsoid seen through a camera looks like a 2D ellipse. 3DGS computes that ellipse with a local linear approximation of the camera projection, which is accurate for small Gaussians.

Following EWA splatting [23], the 2D covariance of the projected Gaussian is

$$\Sigma' = J\,W\,\Sigma\,W^{\top}J^{\top},$$

where $W$ is the rotation part of the world-to-camera transform, $J$ is the Jacobian of the perspective projection evaluated at the Gaussian's centre, and keeping only the top-left $2\times 2$ block of $\Sigma'$ gives the image-space ellipse [6], [23]. The projected centre is simply the pinhole projection of $\boldsymbol{\mu}$.

*Online only: Projecting a 3D Gaussian: the local affine approximation — https://blog.shyandram.dev/en/learn/deep-learning/3d-reconstruction/notes/projecting-a-3d-gaussian-the-local-affine-approximation/*

### Alpha compositing and real-time rendering

The pixel colour uses the same compositing rule as NeRF, but over the Gaussians that cover the pixel, sorted by depth:

$$C = \sum_i \mathbf{c}_i\,\alpha_i \prod_{j=1}^{i-1}(1-\alpha_j), \qquad \alpha_i = o_i \exp\!\Big(-\tfrac{1}{2}(\mathbf{u}-\boldsymbol{\mu}'_i)^{\top}\Sigma_i'^{-1}(\mathbf{u}-\boldsymbol{\mu}'_i)\Big),$$

where $\mathbf{u}$ is the pixel position, $\boldsymbol{\mu}'_i$ and $\Sigma'_i$ are the projected centre and covariance of Gaussian $i$, and $\mathbf{c}_i$ its view-dependent colour. Why is it fast? The renderer [6] splits the screen into $16\times 16$-pixel tiles, assigns each visible Gaussian to the tiles it touches, sorts all (tile, depth) keys once with a GPU radix sort, and then blends each tile's list front to back in parallel, stopping a pixel once it is nearly opaque. There is no network to evaluate and no empty space to sample. The paper reports real-time (at least 30 frames per second) novel-view synthesis at 1080p with quality comparable to the best NeRF variant of the time [6].

*Online only: 3D Gaussian Splatting (Kerbl et al., 2023) — https://blog.shyandram.dev/en/learn/deep-learning/3d-reconstruction/papers/3d-gaussian-splatting-kerbl-et-al-2023/*

### Optimization and adaptive density control

**Plain version.** Training starts from the sparse SfM points, one Gaussian per point, and then grows and prunes the set: where detail is missing it adds Gaussians, where one big Gaussian covers too much it splits it, and invisible ones are removed.

All parameters are optimized by gradient descent on the loss $(1-\lambda)\mathcal{L}_1 + \lambda\mathcal{L}_{\text{D-SSIM}}$ between rendered and real images, with $\lambda = 0.2$ [6]. Every 100 iterations the method densifies Gaussians whose view-space position gradient is large, a sign that the region is not yet well explained [6]:

- a *small* Gaussian in an under-reconstructed region is **cloned** (copied and moved along the gradient);
- a *large* Gaussian in a region of high variance is **split** into two smaller ones, with scales divided by 1.6;
- Gaussians with opacity below a threshold are **pruned**; in addition, all opacities are reset close to zero every 3000 iterations, which limits the growth of Gaussians caused by floaters close to the input cameras [6].

### Code: a toy 2D splatting fit

The snippet fits 40 two-dimensional Gaussians to a 48 × 48 image with the Adam optimizer [49], using exactly the compositing rule above (with a fixed depth order and no densification). The backward pass is written out by hand with the chain rule (the 3DGS paper likewise derives its gradients explicitly instead of relying on automatic differentiation [6]); the gradient was checked against finite differences while writing this tutorial.

```
import numpy as np
from skimage.draw import disk, polygon, ellipse

# Target: a small synthetic RGB image (48 x 48) on a black background
H = W = 48
target = np.zeros((H, W, 3))
target[disk((16, 16), 9)] = [0.9, 0.2, 0.2]
target[ellipse(30, 30, 6, 14, rotation=np.deg2rad(30))] = [0.2, 0.4, 0.9]
target[polygon([40, 44, 34], [6, 22, 14])] = [0.2, 0.8, 0.3]
ys, xs = np.mgrid[0:H, 0:W].astype(float)

def sig(z):
    return 1 / (1 + np.exp(-z))

def render(p, target=None):
    """Front-to-back alpha compositing of K 2D Gaussians (index order = depth order).
    p: mu (K,2), log_s (K,2), theta (K,), op (K,) opacity logits, col (K,3) colour logits."""
    c, s, th = sig(p['col']), np.exp(p['log_s']), p['theta']
    dx = xs[None] - p['mu'][:, 0, None, None]; dy = ys[None] - p['mu'][:, 1, None, None]
    ct, st = np.cos(th)[:, None, None], np.sin(th)[:, None, None]
    u1, u2 = ct * dx + st * dy, -st * dx + ct * dy                  # offsets in the
     ↪  Gaussian's own axes
    sx2, sy2 = (s[:, 0] ** 2)[:, None, None], (s[:, 1] ** 2)[:, None, None]
    g = np.exp(-0.5 * (u1 ** 2 / sx2 + u2 ** 2 / sy2))              # Gaussian footprint
    o = sig(p['op'])[:, None, None]
    a = np.minimum(o * g, 0.99)                                      # alpha of each
     ↪  Gaussian at each pixel
    T = np.cumprod(np.concatenate([np.ones((1, H, W)), 1 - a[:-1]]), axis=0)   #
     ↪  transmittance in front
    w = a * T                                                         # compositing weights
    img = np.einsum('khw,kc->hwc', w, c)                             # black background adds
     ↪  nothing
    if target is None:
        return img
```

```
    # backward pass for the loss L = mean squared error (the chain rule written out by hand)
    G = 2 * (img - target) / target.size                                # dL/dimg, shape (H,
     ↪  W, 3)
    contrib = w[..., None] * c[:, None, None, :]                       # (K, H, W, 3)
    behind = np.cumsum(contrib[::-1], axis=0)[::-1] - contrib           # colour from
     ↪  Gaussians behind k
    dC_da = T[..., None] * c[:, None, None, :] - behind / (1 - a[..., None])
    da = np.einsum('khwc,hwc->khw', dC_da, G) * (o * g < 0.99)
    dq = da * o * g * -0.5                                              # through g = exp(-q /
     ↪  2)
    grads = {
        'col': np.einsum('khw,hwc->kc', w, G) * c * (1 - c),
        'op': (da * g).sum((1, 2)) * sig(p['op']) * (1 - sig(p['op'])),
        'log_s': np.stack([(dq * -2 * u1 ** 2 / sx2).sum((1, 2)), (dq * -2 * u2 ** 2 /
         ↪  sy2).sum((1, 2))], 1),
        'theta': (dq * 2 * u1 * u2 * (1 / sx2 - 1 / sy2)).sum((1, 2)),
        'mu': -np.stack([(dq * (2 * u1 * ct / sx2 - 2 * u2 * st / sy2)).sum((1, 2)),
                         (dq * (2 * u1 * st / sx2 + 2 * u2 * ct / sy2)).sum((1, 2))], 1),
    }
    return img, grads

rng = np.random.default_rng(0)
K = 40
p = {'mu': rng.uniform(4, 44, (K, 2)), 'log_s': np.full((K, 2), np.log(3.0)), 'theta':
 ↪  rng.uniform(0, np.pi, K),
     'op': np.zeros(K), 'col': np.zeros((K, 3))}
lr = {'mu': 0.5, 'log_s': 0.05, 'theta': 0.05, 'op': 0.1, 'col': 0.1}
m = {k: np.zeros_like(v) for k, v in p.items()}; v2 = {k: np.zeros_like(v) for k, v in
 ↪  p.items()}

def psnr(img):
    return 10 * np.log10(1 / np.mean((img - target) ** 2))

for it in range(1, 501):
    img, gr = render(p, target)
    for k in p:                                                         # Adam update
        m[k] = 0.9 * m[k] + 0.1 * gr[k]; v2[k] = 0.999 * v2[k] + 0.001 * gr[k] ** 2
        p[k] -= lr[k] * (m[k] / (1 - 0.9 ** it)) / (np.sqrt(v2[k] / (1 - 0.999 ** it)) +
         ↪  1e-8)
    if it in (1, 50, 100, 200, 500):
        print(f"iteration {it:3d}: PSNR {psnr(img):.2f} dB")
```

```
iteration   1: PSNR 11.32 dB
iteration  50: PSNR 20.88 dB
iteration 100: PSNR 21.85 dB
iteration 200: PSNR 23.12 dB
iteration 500: PSNR 23.57 dB
```

PSNR (defined in DLCV 09) rises quickly and then levels off near 23.6 dB in this run. The plateau is the point of the exercise: with a fixed number of soft blobs, sharp corners and the thin tip of the triangle cannot be fitted, and blobs that started in the wrong place stay there. That is exactly what adaptive density control fixes, and Exercise 4 asks you to add it. Figure 6 shows the fit and

the final ellipses.

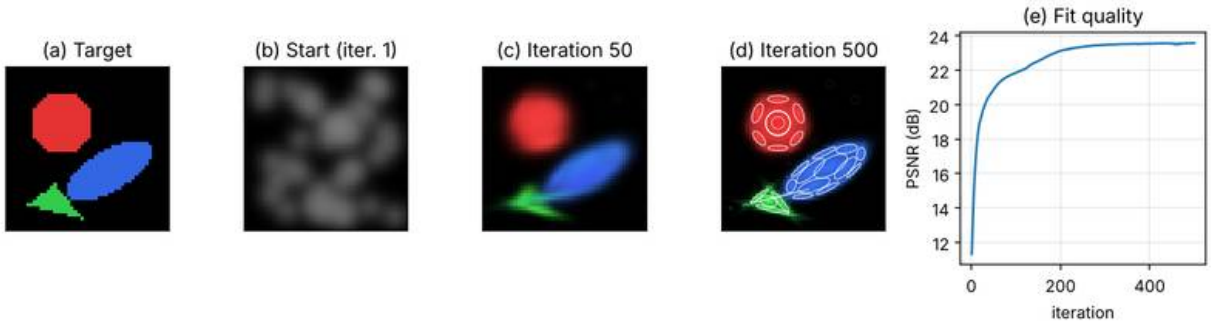



*Figure 6 — Toy splatting with 40 Gaussians and alpha compositing: (a) target, (b) start, (c) after 50 and (d) 500 iterations, with each Gaussian's one-standard-deviation ellipse; (e) PSNR against iteration. Numbers are from this toy run only.*

## Getting meshes

**Plain version.** Graphics engines, physics simulators and 3D printers want triangles. Every representation above needs a conversion step to produce them, and the conversion can add its own errors.

### Marching cubes

Marching cubes [24] turns a sampled scalar field (an SDF, an occupancy or a fused TSDF) into a triangle mesh of one level set. It visits each cube of eight neighbouring grid samples, looks at which corners are inside and which are outside, places vertices on the cube edges where the sign changes (by linear interpolation of the field values), and connects them with triangles from a lookup table. scikit-image provides an implementation [46]. The snippet extracts the zero level set of an exact torus SDF at several grid resolutions and measures the result with the geometric metrics defined in the evaluation section below.

```
import numpy as np
from scipy.spatial import cKDTree
from skimage.measure import marching_cubes

R, r = 0.6, 0.25                                    # torus: ring radius R, tube radius r

def sdf_torus(x, y, z):
    """Exact signed distance to the torus (negative inside)."""
    return np.hypot(np.hypot(x, y) - R, z) - r

def sample_mesh(verts, faces, n, rng):
    """n points spread uniformly over the area of a triangle mesh."""
    tri = verts[faces]
```

```
    area = 0.5 * np.linalg.norm(np.cross(tri[:, 1] - tri[:, 0], tri[:, 2] - tri[:, 0]),
     ↪ axis=1)
    k = rng.choice(len(faces), n, p=area / area.sum())
    a, b = rng.random((2, n, 1))
    flip = a + b > 1; a[flip], b[flip] = 1 - a[flip], 1 - b[flip]
    return tri[k, 0] + a * (tri[k, 1] - tri[k, 0]) + b * (tri[k, 2] - tri[k, 0])

def sample_torus(n, rng):
    """n points uniform over the true torus surface (rejection sampling on the area
     ↪ element)."""
    u, v = rng.uniform(0, 2 * np.pi, (2, 3 * n))
    ok = rng.random(3 * n) * (R + r) < R + r * np.cos(v)   # keep with probability
     ↪ proportional to R + r cos v
    u, v = u[ok][:n], v[ok][:n]
    return np.c_[(R + r * np.cos(v)) * np.cos(u), (R + r * np.cos(v)) * np.sin(u), r *
     ↪ np.sin(v)]

def geometry_metrics(P, G, tau):
    """Accuracy, completeness, Chamfer (their mean) and F-score at threshold tau."""
    d_pg = cKDTree(G).query(P)[0]                      # each reconstructed point -> nearest
     ↪ true point
    d_gp = cKDTree(P).query(G)[0]                      # each true point -> nearest
     ↪ reconstructed point
    prec, rec = (d_pg < tau).mean(), (d_gp < tau).mean()
    return d_pg.mean(), d_gp.mean(), (d_pg.mean() + d_gp.mean()) / 2, 2 * prec * rec / (prec
     ↪ + rec)

rng = np.random.default_rng(0)
G = sample_torus(100_000, rng)
acc, comp, cd, fs = geometry_metrics(sample_torus(100_000, rng), G, tau=0.01)
print(f"true surface vs itself: Chamfer {cd:.4f} | F-score@0.01 {fs:.3f}   (the floor set by
 ↪ sampling)")
for n in [12, 24, 48, 96]:
    ax = np.linspace(-1, 1, n)
    vol = sdf_torus(*np.meshgrid(ax, ax, ax, indexing='ij'))
    verts, faces, _, _ = marching_cubes(vol, level=0.0, spacing=(ax[1] - ax[0],) * 3)
    verts -= 1.0                                       # grid index space -> world
     ↪ coordinates [-1, 1]
    P = sample_mesh(verts, faces, 100_000, rng)
    acc, comp, cd, fs = geometry_metrics(P, G, tau=0.01)
    print(f"grid {n:2d}^3: {len(faces):6d} triangles | acc {acc:.4f} comp {comp:.4f} "
          f"Chamfer {cd:.4f} | F-score@0.01 {fs:.3f} | max |SDF| on vertices
           ↪ {np.abs(sdf_torus(*verts.T)).max():.4f}")
```

```
true surface vs itself: Chamfer 0.0039 | F-score@0.01 0.995   (the floor set by sampling)
grid 12^3:    448 triangles | acc 0.0154 comp 0.0154 Chamfer 0.0154 | F-score@0.01 0.299 |
 ↪ max |SDF| on vertices 0.0084
grid 24^3:   2272 triangles | acc 0.0052 comp 0.0052 Chamfer 0.0052 | F-score@0.01 0.988 |
 ↪ max |SDF| on vertices 0.0032
grid 48^3:   9568 triangles | acc 0.0040 comp 0.0039 Chamfer 0.0040 | F-score@0.01 0.995 |
 ↪ max |SDF| on vertices 0.0008
grid 96^3:  38832 triangles | acc 0.0039 comp 0.0039 Chamfer 0.0039 | F-score@0.01 0.995 |
 ↪ max |SDF| on vertices 0.0002
```

Two lessons come from this run. First, the vertices are almost exactly on the surface even on the $12^3$ grid (the largest |SDF| is 0.0084), yet the Chamfer distance is 0.0154: the flat triangles *between* the vertices cut across the curved tube. Second, the metric stops improving at about 0.0039, because comparing two independent random samplings of the *same* true surface already gives that value. A measured error is meaningful only relative to this floor, which depends on how many points are sampled. Figure 7 shows the meshes and the curves.

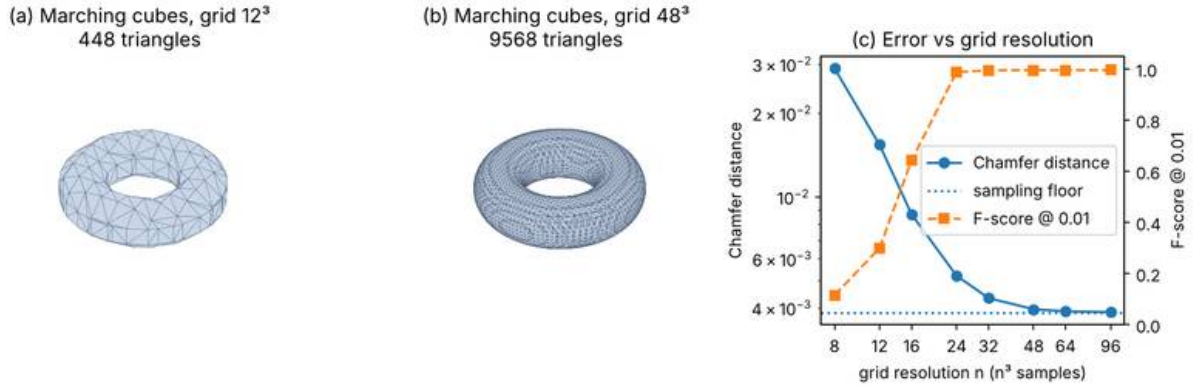



*Figure 7 — Marching cubes on an exact torus SDF: (a) $12^3$ and (b) $48^3$ grids; (c) Chamfer distance and F-score at $\tau = 0.01$ against grid resolution, with the floor obtained by comparing two samplings of the true surface.*

### Poisson surface reconstruction

When the input is a point cloud with normals, as from MVS, Poisson surface reconstruction [25] is the standard tool. It seeks an indicator function $\chi$ (1 inside, 0 outside) whose gradient matches the normals, which leads to the Poisson equation $\Delta\chi = \nabla \cdot \mathbf{V}$, where $\Delta$ is the Laplacian and $\mathbf{V}$ the vector field built from the normals; the surface is a level set of $\chi$ [25]. Because it uses all points at once, it is robust to noise [25]; the screened variant [26] also pulls the surface toward the points. The result is watertight, which is convenient but means that holes in the data are filled with invented surface.

### Meshes from Gaussians

3D Gaussians do not define a surface: after optimization they are unorganized, overlapping and often not aligned with the true geometry. Two lines of work address this. **SuGaR** [27] adds a regularization term that encourages the Gaussians to lie flat on the surface, then extracts a mesh with Poisson reconstruction from points sampled on the level set of the Gaussians' density, and optionally binds new Gaussians to the mesh and refines both together [27]. **2D Gaussian Splatting** [28] changes the primitive: each Gaussian is a flat, oriented 2D disk (a surface element), rendered with an exact ray–disk intersection, plus depth-distortion and normal-consistency losses. Meshes are then obtained by fusing the rendered depth maps with TSDF fusion [28], the same idea as Curless and Levoy's method above.

## Feed-forward reconstruction

**Plain version.** Everything so far optimizes one scene at a time. The newest models are trained once on large collections of 3D data and then, given new photos, output the cameras and a dense 3D point for every pixel in a single pass, without matching, RANSAC or bundle adjustment.

**DUSt3R** [29] takes two images with unknown intrinsics and poses and regresses two **pointmaps**: for each pixel of each image, the 3D point it sees, both expressed in the coordinate frame of the first camera [29]. A pointmap is an image with three channels $(X, Y, Z)$ (Figure 2 (h)). From it, depth, focal length, relative pose and pixel correspondences all follow. The network is a Transformer [48] encoder–decoder: both images go through the same Vision Transformer encoder [47], and two decoders exchange information through cross-attention before regression heads output the pointmaps and a confidence per pixel [29]. The output is defined only up to an unknown scale [29]. For more than two images, DUSt3R predicts pairwise pointmaps and then runs a *global alignment* that solves for one consistent set of pointmaps, poses and per-pair scales, a learned-geometry counterpart of bundle adjustment [29].

**MASt3R** [30] adds a head that outputs dense local features trained with a matching loss, and a fast reciprocal matching scheme, so that the pointmap network also yields accurate pixel correspondences [30]. **VGGT** [31] goes further: a large Transformer with alternating frame-wise and global attention takes one, a few or hundreds of images and predicts camera parameters, depth maps, pointmaps and point tracks for all of them at once; the authors report that this takes less than a second and often beats optimization-based alternatives without further processing [31].

These models do not make geometry obsolete. Their predictions can still be refined: the VGGT authors report that its cameras and tracks are a good initialization for bundle adjustment, which improves accuracy further [31]. Their accuracy also depends on how close a scene is to their training data. They are, however, the best tools available when images are few, overlap little or have no calibration.

## Dynamic scenes (brief)

**Plain version.** If the scene moves, a single 3D model is not enough: the model must also depend on time.

The common recipe is a canonical (static) scene plus a deformation over time. D-NeRF [32] adds time as an input and learns two networks together: one encodes the scene in a canonical space, the other maps that canonical scene to its deformed state at a given time; it works from a single camera moving around the scene [32]. 4D Gaussian Splatting [33] keeps one set of 3D Gaussians and predicts their deformation at each timestamp with a lightweight MLP fed by a factorized space–time feature grid; the paper reports real-time rendering, for example 82 frames per second at $800 \times 800$ on an RTX 3090 GPU [33]. Dynamic reconstruction is much less constrained than static reconstruction, because each moment is seen from fewer viewpoints.

## Evaluating reconstructions

**Plain version.** Two different things can be measured: whether the *pictures* rendered from new viewpoints look like real photos, and whether the *shape* is geometrically correct. A method can do well on one and badly on the other.

### Novel-view quality

Hold out some photos, render the reconstruction from their camera poses, and compare. The standard metrics are PSNR, SSIM [34] and LPIPS [35], all defined in DLCV 09. PSNR and SSIM measure pixel and structural agreement; LPIPS measures perceptual similarity in deep-feature space. A common protocol, which 3DGS adopts from Mip-NeRF 360, holds out every 8th photo of a capture as the test set [6]. Novel-view metrics depend on the held-out poses being correct, so they also reward good SfM.

### Geometric accuracy

Let $\mathcal{P}$ be points sampled from the reconstruction and $\mathcal{G}$ points from the ground-truth scan. Define the two directed distances

$$\text{acc} = \frac{1}{|\mathcal{P}|} \sum_{\mathbf{p} \in \mathcal{P}} \min_{\mathbf{g} \in \mathcal{G}} \|\mathbf{p} - \mathbf{g}\|, \qquad \text{comp} = \frac{1}{|\mathcal{G}|} \sum_{\mathbf{g} \in \mathcal{G}} \min_{\mathbf{p} \in \mathcal{P}} \|\mathbf{g} - \mathbf{p}\|,$$

where $|\cdot|$ counts points. **Accuracy** asks "are my points near the truth?"; **completeness** asks "is all of the truth covered?". Their sum or mean is the **Chamfer distance**, widely used as a loss and metric for point sets since Fan et al. [36], who define it with squared distances; papers differ in squaring and in sum versus mean, so check the definition before comparing numbers. On DTU, MVSNet reports the mean of accuracy and completeness as an "overall" score [16].

Tanks and Temples [38] uses a threshold $\tau$ instead of averages. With $e_{\mathbf{p}}$ the distance from a reconstructed point to the ground truth and $e_{\mathbf{g}}$ the distance from a ground-truth point to the reconstruction,

$$P(\tau) = \frac{100}{|\mathcal{P}|} \sum_{\mathbf{p} \in \mathcal{P}} [\, e_{\mathbf{p}} < \tau \,], \qquad R(\tau) = \frac{100}{|\mathcal{G}|} \sum_{\mathbf{g} \in \mathcal{G}} [\, e_{\mathbf{g}} < \tau \,], \qquad F(\tau) = \frac{2P(\tau)R(\tau)}{P(\tau) + R(\tau)},$$

where $[\cdot]$ is 1 if the condition holds and 0 otherwise, $P$ is precision (in percent), $R$ is recall and $F$ is their harmonic mean, the **F-score** [38]. Because it is a harmonic mean, $F$ is high only if the reconstruction is both accurate and complete; a sparse set of perfect points or a cloud that fills all space both score poorly [38]. The F-score is also robust to a few far outliers, which can dominate a Chamfer average (Exercise 5).

### Datasets and benchmarks

- **DTU** [37]: a multi-view stereo dataset of 80 tabletop scenes, each with 49 or 64 precisely known camera positions from a six-axis industrial robot and reference geometry from a structured-light scanner [37]. Standard for MVS and neural surface methods (NeuS uses 15 of its scenes [22]).
- **Tanks and Temples** [38]: high-resolution video of large indoor and outdoor scenes captured outside the lab, with ground truth from an industrial laser scanner, split into an *intermediate* and an *advanced* group, and evaluated by the F-score above on an online server [38].
- **Mip-NeRF 360** [39]: 9 unbounded scenes (5 outdoor, 4 indoor), each with a central object and a detailed background, captured for novel-view synthesis [39]. 3DGS evaluates on it together with Tanks and Temples scenes, among others [6].

## Failure modes

**Plain version.** The methods above rely on different evidence and assumptions. Classical matching benefits from stable appearance, texture and overlapping views; SfM estimates camera poses rather than requiring them all in advance [2]. NeRF explicitly models view-dependent color but its original formulation uses known poses [5]; DUSt3R can infer geometry without prior camera calibration or poses [29]. Static-scene methods need an extension for motion. The failures below show which information is missing or which assumption is being strained; they are not shared restrictions of every method.

- **Textureless surfaces** (white walls, sky). Photo-consistency is equal at every depth, as in the plane-sweep run; reconstructions show holes or noise. Learned MVS and feed-forward models fill them with learned priors, which may be wrong.
- **Reflective and transparent surfaces.** A mirror or a glossy car shows a reflected scene that moves differently from the surface, so matching and density both go wrong. NeRF tends to fake a highlight with light sources hidden *inside* the object under a semi-transparent surface; Ref-NeRF [40] instead models the radiance reflected about the surface normal, which improves specular surfaces [40].
- **Few views.** With two or three images, many scenes explain the photos equally well, and per-scene optimization overfits (good training views, broken new views). Feed-forward models such as DUSt3R [29] are the usual remedy.
- **Floaters.** Semi-transparent blobs of density in free space, often close to the cameras, that look right from training views but appear as fog from new views. Mip-NeRF 360 names them and adds a distortion-based regularizer to suppress them [39]; 3DGS resets opacities periodically for the same reason [6].
- **Scale ambiguity.** Monocular image collections fix geometry only up to scale; metric size needs a known distance, a calibrated rig or other sensors [1].
- **Pose errors.** NeRF and 3DGS assume accurate poses; small errors blur fine detail. BARF [41] jointly optimizes the poses and the radiance field, using a coarse-to-fine schedule on the positional encoding, because the high frequencies otherwise hurt registration [41].

- **Moving objects and lighting changes**, such as people walking through a capture or the sun moving, break the static-scene assumption itself.

## Modern view

### What the surveys say

Four reviews frame this tutorial. Seitz et al. [42] gave multi-view stereo its classical taxonomy, which still describes every method here: the scene representation, the photo-consistency measure, the visibility model, the shape prior, the reconstruction algorithm and the initialization requirements. Furukawa and Hernández [43] present MVS as an image–geometry consistency optimization built from two ingredients, robust photometric consistency measures and efficient optimization, and survey the successful algorithms and remaining challenges. Tewari et al. [44] review neural rendering, from scene representations and differentiable image formation to applications such as novel-view synthesis, generalization across scenes, dynamic content, editing and relighting, and list open problems such as integration with classical graphics tools and interpretability. Chen and Wang [45] survey 3D Gaussian Splatting: its principles and optimization, seven directions that extend it, its applications, and performance comparisons; they identify a shift from per-scene optimization toward feed-forward Gaussian prediction.

*Online only: Guide to the Modern view surveys — https://blog.shyandram.dev/en/learn/deep-learning/3d-reconstruction/papers/guide-to-the-modern-view-surveys/*

### The state of the art, 2023–2026

- **Explicit, rasterizable representations** won the speed race. Since 3DGS [6], a large share of view-synthesis work builds on Gaussian primitives. The 3DGS survey [45] groups the follow-up work into sparse-input, memory-efficient and photorealistic variants, better optimization, Gaussians with extra properties, hybrid representations and new rendering algorithms; geometry (SuGaR [27], 2DGS [28]) and dynamics (4DGS [33]) are active directions of their own.
- **Neural implicit surfaces** in the NeuS line [22] remain the choice when a watertight, accurate surface matters more than rendering speed.
- **Feed-forward geometry** is the largest change. DUSt3R [29], MASt3R [30] and VGGT [31] estimate cameras and dense geometry directly from images; they are increasingly used to initialize or replace parts of the classical SfM pipeline, and the 3DGS survey [45] describes the same trend for Gaussians.
- **The classical pipeline remains the reference.** COLMAP [2], [3] is still the usual source of camera poses for training and evaluation, and bundle adjustment is still used to refine learned predictions.

### Open problems

1. **Geometry and appearance together.** Methods tuned for novel views produce poor surfaces, and accurate-surface methods render less realistically. One representation that does both, fast, is still open.
2. **Reflections, transparency and lighting.** Separating surface, material and illumination (inverse rendering) is unsolved for real captures.
3. **Generalization of feed-forward models**: accuracy far from the training distribution, metric scale, and very large scenes.
4. **Dynamic and in-the-wild captures** with moving people, changing light and few viewpoints.
5. **Evaluation.** Pose errors in the "ground truth", sampling-dependent Chamfer numbers and inconsistent metric definitions make cross-paper comparisons fragile.

## Key takeaways

- Recovering a scene requires camera geometry and a representation suited to the output; image-only reconstruction can leave scale ambiguous.
- Projection, correspondence and triangulation connect image measurements to 3D structure.
- Structure from motion estimates cameras and sparse structure; dense reconstruction adds further assumptions about correspondence.
- Neural fields optimize rendered observations; good view synthesis does not alone establish accurate surfaces.
- Gaussian and feed-forward representations change rendering or inference cost, but still require geometry checks.
- Report view metrics and geometric accuracy/completeness with explicit alignment, sampling and thresholds.

## Exercises

1. **Depth error by hand.** A rectified stereo pair has $f = 800$ px and baseline $B = 0.1$ m. Using $\delta Z \approx Z^2 \delta d/(fB)$, how large is the depth error for a point at $Z = 5$ m if the disparity is off by 0.5 px? How large must $B$ be to bring it under 2 cm?

   **Hint**

   $\delta Z \approx 25 \cdot 0.5/(800 \cdot 0.1) = 0.156$ m. Solving $25 \cdot 0.5/(800B) < 0.02$ gives $B > 0.78$ m. A wider baseline helps, but it also makes the two views look more different, which makes matching harder.

2. **See the scale ambiguity.** In the two-view snippet, multiply all scene points `X` and the camera centre `C2` by 3 before projecting. Show that the pixel coordinates do not change and that the estimated pose is the same. Name three pieces of outside information that would fix the scale.

   **Hint**

Projection divides by depth, so scaling $\mathbf{X}$ and $\mathbf{t}$ together cancels: $K(R(3\mathbf{X}) + 3\mathbf{t}) = 3K(R\mathbf{X} + \mathbf{t})$. Outside information: a measured distance in the scene (a ruler or a marker of known size), the known baseline of a stereo rig, or another sensor such as an inertial measurement unit or GPS.

3. **Read the weights.** In the volume-rendering snippet, prove that $\sum_i w_i = 1 - \prod_i (1 - \alpha_i)$. Then remove the fog slab and recompute the depth. Why is the "depth" of a ray that passes through fog not the depth of any surface?

   **Hint**

   By induction: $T_{i+1} = T_i(1 - \alpha_i)$ so $w_i = T_i - T_{i+1}$, and the sum telescopes to $T_1 - T_{N+1} = 1 - T_{N+1}$. Without fog, all weight sits on the red layer and the depth lands near 3. With fog, $\sum_i w_i t_i$ averages two separate places, so it can fall in empty space; this is how a floater corrupts a depth map.

4. **Add adaptive density control.** Extend the toy splatting snippet: every 100 iterations, find the Gaussians with the largest accumulated position gradient; clone the small ones and split the large ones (divide the scales by 1.6), and remove those whose opacity falls below 0.02. Compare the final PSNR with the fixed 40 Gaussians, and with simply starting from 80 Gaussians.

   **Hint**

   When the arrays grow, the Adam moment arrays `m` and `v2` must grow with them (initialize the new rows at zero). For reference, simply starting from 80 Gaussians instead of 40, with nothing else changed, reached 27.00 dB at iteration 500 in our run (23.57 dB with 40). Densification should place the new Gaussians where the error is, for example at the triangle's corners.

5. **Chamfer versus F-score.** In the marching-cubes snippet (grid 48), first add 1,000 "floater" points near $(0, 0, 0.6)$, then instead delete the half of the mesh samples with $x < 0$. Recompute accuracy, completeness, Chamfer distance and F-score. Which metric reacts to which defect?

   **Hint**

   In our run, the floaters (1% of the points) raised accuracy from 0.0040 to 0.0097 and the Chamfer distance from 0.0040 to 0.0068, while the F-score at 0.01 barely moved (0.995 to 0.990). Deleting half the surface left accuracy at 0.0040 but raised completeness to 0.2226, the Chamfer distance to 0.1133, and lowered the F-score to 0.665. Averages are sensitive to a few far outliers; thresholded scores are not, but they report missing parts clearly through recall.

6. **Window size in plane sweep.** Run the plane-sweep snippet with SSD windows of 3, 7 and 15 pixels. How do the textured and textureless scores change, and what happens to the edges of the square?

   **Hint**

   In our run the textured score was 95.2%, 95.3% and 96.6%, and the textureless score 4.9%, 14.8% and 27.7%. Large windows reach texture outside the flat patch, so its pixels borrow depth from their surroundings; the price is that depth edges are blurred and the square appears "fattened" into the background. Learned MVS uses a 3D CNN to get this context without the blur.

The scene models built here are also a building block of the world models in DLCV 13, where an agent must predict how a 3D environment changes.

## References


1. R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed., Cambridge University Press, 2004. doi
2. J. L. Schönberger and J.-M. Frahm, "Structure-from-motion revisited," in *Proc. IEEE CVPR*, 2016. CVF
3. J. L. Schönberger, E. Zheng, J.-M. Frahm and M. Pollefeys, "Pixelwise view selection for unstructured multi-view stereo," in *Proc. ECCV*, 2016. doi
4. J. L. Schönberger et al., "COLMAP," official documentation (general-purpose structure-from-motion and multi-view stereo pipeline). link
5. B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi and R. Ng, "NeRF: Representing scenes as neural radiance fields for view synthesis," in *Proc. ECCV*, 2020; also *Communications of the ACM*, vol. 65, no. 1, pp. 99–106, 2022. doi ·CACM doi ·arXiv
6. B. Kerbl, G. Kopanas, T. Leimkühler and G. Drettakis, "3D Gaussian Splatting for real-time radiance field rendering," *ACM Transactions on Graphics*, vol. 42, no. 4, 2023. doi ·arXiv
7. H. C. Longuet-Higgins, "A computer algorithm for reconstructing a scene from two projections," *Nature*, vol. 293, pp. 133–135, 1981. doi
8. R. I. Hartley, "In defense of the eight-point algorithm," *IEEE TPAMI*, vol. 19, no. 6, pp. 580–593, 1997. doi
9. D. Nistér, "An efficient solution to the five-point relative pose problem," *IEEE TPAMI*, vol. 26, no. 6, pp. 756–770, 2004. doi
10. M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981. doi
11. D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004. doi
12. N. Snavely, S. M. Seitz and R. Szeliski, "Photo tourism: Exploring photo collections in 3D," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 835–846, 2006. doi
13. B. Triggs, P. F. McLauchlan, R. I. Hartley and A. W. Fitzgibbon, "Bundle adjustment — A modern synthesis," in *Vision Algorithms: Theory and Practice*, LNCS 1883, pp. 298–372, Springer, 2000. doi
14. R. T. Collins, "A space-sweep approach to true multi-image matching," in *Proc. IEEE CVPR*, pp. 358–363, 1996. doi
15. B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *Proc. SIGGRAPH*, pp. 303–312, 1996. doi
16. Y. Yao, Z. Luo, S. Li, T. Fang and L. Quan, "MVSNet: Depth inference for unstructured multi-view stereo," in *Proc. ECCV*, 2018. CVF ·doi
17. N. Max, "Optical models for direct volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, 1995. doi
18. M. Tancik, P. P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ra-

mamoorthi, J. T. Barron and R. Ng, “Fourier features let networks learn high frequency functions in low dimensional domains,” in *Proc. NeurIPS*, 2020. proceedings
19. T. Müller, A. Evans, C. Schied and A. Keller, “Instant neural graphics primitives with a multiresolution hash encoding,” *ACM Transactions on Graphics*, vol. 41, no. 4, 2022. doi ·arXiv
20. J. J. Park, P. Florence, J. Straub, R. Newcombe and S. Lovegrove, “DeepSDF: Learning continuous signed distance functions for shape representation,” in *Proc. IEEE/CVF CVPR*, 2019. CVF
21. L. Mescheder, M. Oechsle, M. Niemeyer, S. Nowozin and A. Geiger, “Occupancy networks: Learning 3D reconstruction in function space,” in *Proc. IEEE/CVF CVPR*, 2019. CVF
22. P. Wang, L. Liu, Y. Liu, C. Theobalt, T. Komura and W. Wang, “NeuS: Learning neural implicit surfaces by volume rendering for multi-view reconstruction,” in *Proc. NeurIPS*, 2021. proceedings ·arXiv
23. M. Zwicker, H. Pfister, J. van Baar and M. Gross, “EWA volume splatting,” in *Proc. IEEE Visualization*, 2001. doi
24. W. E. Lorensen and H. E. Cline, “Marching cubes: A high resolution 3D surface construction algorithm,” in *Proc. SIGGRAPH*, pp. 163–169, 1987. doi
25. M. Kazhdan, M. Bolitho and H. Hoppe, “Poisson surface reconstruction,” in *Proc. Eurographics Symposium on Geometry Processing*, 2006. doi
26. M. Kazhdan and H. Hoppe, “Screened Poisson surface reconstruction,” *ACM Transactions on Graphics*, vol. 32, no. 3, 2013. doi
27. A. Guédon and V. Lepetit, “SuGaR: Surface-aligned Gaussian splatting for efficient 3D mesh reconstruction and high-quality mesh rendering,” in *Proc. IEEE/CVF CVPR*, 2024. CVF ·arXiv
28. B. Huang, Z. Yu, A. Chen, A. Geiger and S. Gao, “2D Gaussian splatting for geometrically accurate radiance fields,” in *Proc. SIGGRAPH (Conference Papers)*, 2024. doi ·arXiv
29. S. Wang, V. Leroy, Y. Cabon, B. Chidlovskii and J. Revaud, “DUSt3R: Geometric 3D vision made easy,” in *Proc. IEEE/CVF CVPR*, 2024. CVF ·arXiv
30. V. Leroy, Y. Cabon and J. Revaud, “Grounding image matching in 3D with MASt3R,” in *Proc. ECCV*, 2024. doi ·arXiv
31. J. Wang, M. Chen, N. Karaev, A. Vedaldi, C. Rupprecht and D. Novotny, “VGGT: Visual geometry grounded transformer,” in *Proc. IEEE/CVF CVPR*, 2025. CVF ·arXiv
32. A. Pumarola, E. Corona, G. Pons-Moll and F. Moreno-Noguer, “D-NeRF: Neural radiance fields for dynamic scenes,” in *Proc. IEEE/CVF CVPR*, 2021. CVF
33. G. Wu, T. Yi, J. Fang, L. Xie, X. Zhang, W. Wei, W. Liu, Q. Tian and X. Wang, “4D Gaussian splatting for real-time dynamic scene rendering,” in *Proc. IEEE/CVF CVPR*, 2024. CVF ·arXiv
34. Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, “Image quality assessment: From error visibility to structural similarity,” *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. doi
35. R. Zhang, P. Isola, A. A. Efros, E. Shechtman and O. Wang, “The unreasonable effectiveness of deep features as a perceptual metric,” in *Proc. IEEE/CVF CVPR*, 2018. CVF
36. H. Fan, H. Su and L. J. Guibas, “A point set generation network for 3D object reconstruction from a single image,” in *Proc. IEEE CVPR*, 2017. CVF
37. R. Jensen, A. Dahl, G. Vogiatzis, E. Tola and H. Aanæs, “Large scale multi-view stereopsis evaluation,” in *Proc. IEEE CVPR*, pp. 406–413, 2014. doi
38. A. Knapitsch, J. Park, Q.-Y. Zhou and V. Koltun, “Tanks and Temples: Benchmarking large-scale scene reconstruction,” *ACM Transactions on Graphics*, vol. 36, no. 4, 2017. doi ·benchmark

39. J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan and P. Hedman, "Mip-NeRF 360: Unbounded anti-aliased neural radiance fields," in *Proc. IEEE/CVF CVPR*, 2022. CVF ·arXiv
40. D. Verbin, P. Hedman, B. Mildenhall, T. Zickler, J. T. Barron and P. P. Srinivasan, "Ref-NeRF: Structured view-dependent appearance for neural radiance fields," in *Proc. IEEE/CVF CVPR*, 2022. CVF
41. C.-H. Lin, W.-C. Ma, A. Torralba and S. Lucey, "BARF: Bundle-adjusting neural radiance fields," in *Proc. IEEE/CVF ICCV*, 2021. CVF
42. S. M. Seitz, B. Curless, J. Diebel, D. Scharstein and R. Szeliski, "A comparison and evaluation of multi-view stereo reconstruction algorithms," in *Proc. IEEE CVPR*, vol. 1, pp. 519–528, 2006. doi
43. Y. Furukawa and C. Hernández, "Multi-view stereo: A tutorial," *Foundations and Trends in Computer Graphics and Vision*, vol. 9, no. 1–2, pp. 1–148, 2015. doi
44. A. Tewari, J. Thies, B. Mildenhall, P. Srinivasan, E. Tretschk, W. Yifan, C. Lassner, V. Sitzmann et al., "Advances in neural rendering," *Computer Graphics Forum*, vol. 41, no. 2, pp. 703–735, 2022. doi ·arXiv
45. G. Chen and W. Wang, "A survey on 3D Gaussian splatting," *ACM Computing Surveys*, vol. 58, no. 12, 2026. doi ·arXiv
46. scikit-image contributors, "skimage.measure.marching_cubes," *scikit-image documentation (API reference)*. docs
47. A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai et al., "An image is worth 16x16 words: Transformers for image recognition at scale," in *Proc. ICLR*, 2021. arXiv
48. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin, "Attention is all you need," in *Proc. NeurIPS*, 2017. proceedings
49. D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICLR*, 2015. arXiv

# DLCV 11A • Depth Estimation and Optical Flow

Dense correspondence for depth and motion: rectified stereo and disparity, matching costs, semi-global matching and learned cost volumes, monocular relative and metric depth with scale-and-shift-invariant losses, self-supervised depth from video, optical flow from FlowNet to RAFT, scene flow, and the metrics, benchmarks and failure modes of all three, with tested NumPy and OpenCV code.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/depth-optical-flow/

### What you'll learn

- Distinguish stereo correspondence, temporal flow and monocular depth inference.
- Relate disparity to calibrated camera geometry and depth error.
- Compare matching costs, cost volumes and iterative refinement.
- Separate relative depth, metric depth and view-synthesis supervision.
- Evaluate with explicit alignment, visibility masks, units and dataset splits.

### The big picture

A depth map gives, for every pixel, how far away the surface is. An optical flow field gives, for every pixel, where that point moves in the next frame. Robots, cars, phones and video editors need both: depth to know where things are, flow to know how they move. Both are answers to the same question: *which pixel in the other image shows the same point?*

> **Explain like I'm five**
>
> Hold a pencil in front of your face and close one eye, then the other. The pencil jumps sideways, while the window far behind it hardly moves. How far something jumps tells you how close it is. A stereo camera does the same with two lenses, and a computer measures the jump for every pixel. If you have only one eye, you can still guess distances, because you know how big a car or a door usually is. But you can be fooled by a toy car close up. Optical flow is the same jumping game played over time: compare this video frame with the next one and measure how far each bit moved.

**Why it matters.** Stereo depth lets robots and cars measure distance without a laser scanner; single-image depth blurs portrait backgrounds on phones and places objects into video; optical flow drives video super-resolution, frame interpolation and stabilization.

**How this tutorial relates to others.** DLCV 11 covers camera geometry, triangulation and multi-view stereo; DLCV 09A.3 introduces brightness constancy and warping; long-range point tracking is in DLCV 07 and sparse keypoint matching in IP 12. We link to them instead of repeating them. Our focus is the *dense* case: one answer per pixel.

Figure 1 shows the four model families of this tutorial: three match two images, the fourth relies on what it has learned.

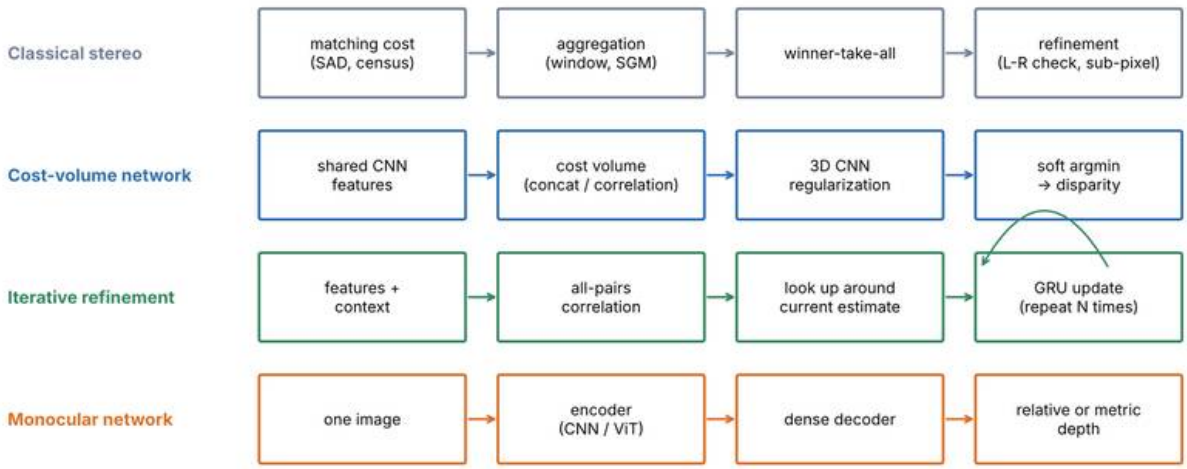



*Figure 1 — Four families of dense correspondence and depth models. Classical stereo (grey) and cost-volume networks (blue) follow the same four stages; iterative models (green) repeat a lookup-and-update step; monocular networks (orange) predict depth from a single image.*

## Three dense correspondence problems

**Plain version.** Rectified stereo matches pixels along the same row. Optical flow matches across frames; monocular depth estimates geometry without a second image.

**Precise version.** Let $I_1, I_2$ be two images of $H \times W$ pixels and $\mathbf{p} = (u, v)$ a pixel, with column $u$ and row $v$.

- **Optical flow** is a vector field $\mathbf{F}(\mathbf{p}) = (\Delta u, \Delta v)$ such that the point seen at $\mathbf{p}$ in $I_1$ is seen at $\mathbf{p}+\mathbf{F}(\mathbf{p})$ in $I_2$. Two unknowns per pixel; the cameras and the scene may both move.
- **Stereo disparity** is the special case of two cameras that see the scene at the same instant and have been *rectified* (next section), so that every match lies on the same row: $\mathbf{F}(\mathbf{p}) = (-d(\mathbf{p}), 0)$. One unknown per pixel, the disparity $d \geq 0$, which converts to depth.
- **Monocular depth** asks for a depth map $Z(\mathbf{p})$ from $I_1$ alone. No correspondence is available, so the answer comes from learned regularities, and it is only defined up to an unknown scale unless the network has learned metric cues.
- **Scene flow** asks for the 3D motion of the point seen at each pixel; it combines depth at two times with optical flow (see Scene flow below).

The two-image problems share one structure: a **cost** that scores a candidate match, a search over candidates, and a way to resolve pixels where the cost is ambiguous.

### Stereo geometry: rectification and disparity

**Plain version.** In a calibrated rectified pair, nearby points have larger disparity. Focal length and camera baseline convert that disparity into depth.

#### Rectification

For a general pair of cameras, the match of a pixel lies on its **epipolar line** in the other image (DLCV 11, epipolar geometry). Rectification warps both images with a homography each so that all epipolar lines become horizontal and corresponding lines have the same row index [1]. After rectification the two virtual cameras have the same intrinsics, parallel optical axes and a purely horizontal baseline. Stereo rigs are calibrated once, and every frame is rectified before matching; all the stereo benchmarks below distribute rectified pairs. In the rest of this tutorial the pair is always rectified.

#### Disparity and depth

Place the left camera centre at the origin and the right camera at distance $B$ along the $x$ axis (Figure 2a). A point at depth $Z$ and horizontal position $X$ projects to column $x_L = fX/Z$ in the left image and $x_R = f(X - B)/Z$ in the right image, both measured from the principal point. Subtracting gives

$$d = x_L - x_R = \frac{fB}{Z}, \qquad Z = \frac{fB}{d},$$

where $f$ is the focal length in pixels, $B$ is the **baseline** (distance between the camera centres, in metres), $Z$ is the depth along the optical axis and $d$ is the **disparity** in pixels. Depth and disparity are inversely proportional: disparity is an *inverse depth*, scaled by $fB$. Large disparities belong to near objects, and a disparity of zero means a point at infinity.

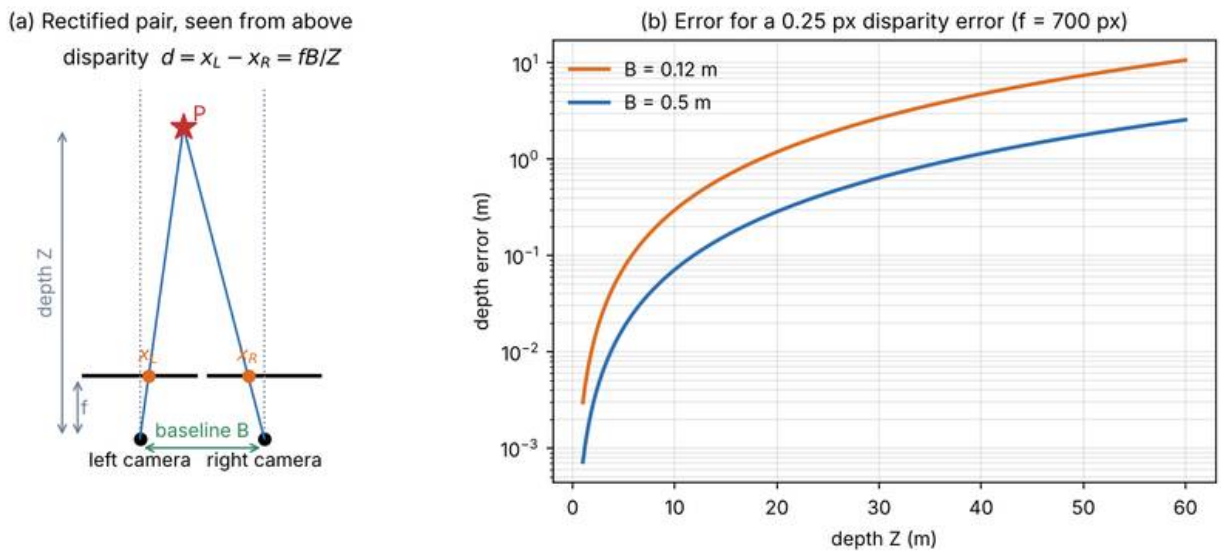



*Figure 2 — (a) A rectified stereo pair seen from above: a point shifts by the disparity d between the two images. (b) Depth error for a disparity error of 0.25 px with a focal length of 700 px; the error grows with the square of the distance and shrinks with a wider baseline.*

Because $Z = fB/d$, a small disparity error $\delta d$ becomes a depth error of about $\delta Z \approx Z^2 \delta d/(fB)$, the same relation that DLCV 11, Triangulation derives. The snippet puts numbers on it for a hypothetical camera with $f = 700$ px.

```
import numpy as np

f = 700.0                                   # focal length in pixels (a hypothetical rig)
for B in [0.12, 0.5]:                       # baselines in metres
    print(f"baseline {B} m: depth at a disparity of 1 px = {f * B:.0f} m")
    for Z in [2, 10, 30, 60]:
        d = f * B / Z                       # disparity in pixels
        dZ = Z ** 2 * 0.25 / (f * B)        # depth error for a 0.25 px disparity error
        print(f"   Z = {Z:2d} m  d = {d:6.2f} px  depth error for 0.25 px: {dZ:6.3f} m ({100
        ↪  * dZ / Z:4.1f}%)")
```

```
baseline 0.12 m: depth at a disparity of 1 px = 84 m
   Z =  2 m  d =  42.00 px  depth error for 0.25 px:  0.012 m ( 0.6%)
   Z = 10 m  d =   8.40 px  depth error for 0.25 px:  0.298 m ( 3.0%)
   Z = 30 m  d =   2.80 px  depth error for 0.25 px:  2.679 m ( 8.9%)
   Z = 60 m  d =   1.40 px  depth error for 0.25 px: 10.714 m (17.9%)
baseline 0.5 m: depth at a disparity of 1 px = 350 m
   Z =  2 m  d = 175.00 px  depth error for 0.25 px:  0.003 m ( 0.1%)
   Z = 10 m  d =  35.00 px  depth error for 0.25 px:  0.071 m ( 0.7%)
   Z = 30 m  d =  11.67 px  depth error for 0.25 px:  0.643 m ( 2.1%)
   Z = 60 m  d =   5.83 px  depth error for 0.25 px:  2.571 m ( 4.3%)
```

At 10 m, a quarter-pixel disparity error costs 30 cm with a 12 cm baseline and 7 cm with a 50 cm

baseline. At 60 m, the short baseline sees a disparity of only 1.4 px and its depth is uncertain by 18%. This is why stereo cameras for driving use wide baselines and high resolution, and why sub-pixel accuracy matters: the last fraction of a pixel decides the far range.

### Classical stereo matching

**Plain version.** Classical block matching searches candidate disparities, compares local patches and refines unreliable matches.

Scharstein and Szeliski organized dense two-frame stereo algorithms into four steps that most methods perform in some form [2]:

1. **Matching cost computation**: a per-pixel, per-disparity dissimilarity.
2. **Cost (support) aggregation**: combine costs over a neighbourhood to reduce noise.
3. **Disparity computation or optimization**: choose a disparity per pixel, either locally or by optimizing an energy over the whole image.
4. **Disparity refinement**: sub-pixel estimates, consistency checks, filling and smoothing.

The Middlebury stereo website, which accompanies that paper, still hosts an online evaluation [2], [3]. The plane sweep in DLCV 11 is exactly steps 1–3 with windowed SSD; here we look at each step more closely and add the refinement.

### Matching costs and the cost volume

The simplest cost compares intensities in a small window. The **sum of absolute differences** (SAD) at pixel $(u, v)$ and disparity $d$ is

$$C(u, v, d) = \sum_{(i,j)\in\mathcal{W}} |I_L(u+i, v+j) - I_R(u+i-d, v+j)|,$$

where $I_L, I_R$ are the left and right images, $\mathcal{W}$ is the set of offsets of a $w \times w$ window centred on the pixel, and $d$ ranges over integer hypotheses $0, 1, \ldots, D-1$. Squared differences give SSD. Computing $C$ for every pixel and every $d$ fills a **cost volume** of size $D \times H \times W$. The **winner-take-all** (WTA) rule picks $\hat{d}(u, v) = \arg\min_d C(u, v, d)$.

SAD assumes both cameras record the same brightness for the same point. Real cameras differ in gain and exposure, so practical systems use **zero-mean normalized cross-correlation** (ZNCC) or the **census transform**, which encodes which neighbours are darker than the pixel as a bit string and compares bit strings [4].

*Online only: Robust matching costs: ZNCC and census — https://blog.shyandram.dev/en/learn/deep-learning/depth-optical-flow/notes/robust-matching-costs-zncc-and-census/*

Figure 3a shows one row of a cost volume: dark valleys trace the surfaces. Figure 3b shows four typical cost curves. A textured pixel has one sharp minimum; a textureless pixel a flat curve; a repetitive pattern several equal minima, one per period; and an occluded pixel, hidden in the right image, no correct minimum at all.

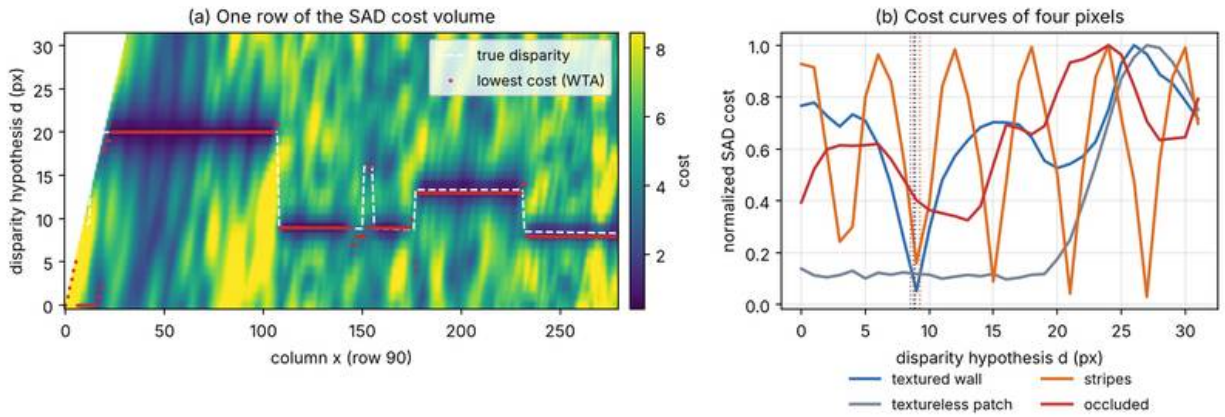



*Figure 3 — (a) One row of the SAD cost volume of the synthetic scene below; low cost (dark) follows the true disparity (dashed). (b) Cost curves of four pixels; dotted lines mark their true disparities.*

### Code: block matching with a left–right check

The snippet ray-casts a rectified pair of a synthetic scene with known depth: a slanted textured wall, a box, a disc and a thin pole in front of it, plus a textureless patch and a patch of fine stripes on the wall. It builds the SAD cost volume, takes the WTA disparity, and runs the **left–right consistency check**: match again with the right image as reference, and reject pixels where the two answers disagree by more than one pixel. Occluded pixels usually fail this check, because their "match" in the right image belongs to a different surface whose own match points elsewhere. For comparison it also runs OpenCV's semi-global matcher (next subsection). Errors are reported as **bad-1**, the percentage of pixels whose disparity is off by more than 1 px, separately for each kind of region.

```
import numpy as np
import cv2
from scipy import ndimage as ndi

rng = np.random.default_rng(0)
H, W, f, B = 180, 280, 320.0, 0.25                    # image size, focal length (px),
 ↪  baseline
tex = ndi.gaussian_filter(rng.random((700, 700)), 1.2)
tex = (tex - tex.min()) / np.ptp(tex)                 # random texture painted on every
 ↪  surface

def render(cx):
    """Ray-cast a rectified camera at (cx, 0, 0) looking down +Z: image, depth Z, region
     ↪  labels."""
    v, u = np.mgrid[0:H, 0:W].astype(float)
    dx, dy = (u - W / 2) / f, (v - H / 2) / f            # ray direction (dx, dy, 1)
    Z = (9 + 0.15 * cx) / (1 - 0.15 * dx)                # slanted wall Z = 9 + 0.15 X
    lab = np.zeros((H, W), int)                          # 0 wall, 1 box, 2 disc, 3 thin pole
    for k, z, inside in [(2, 6.0, lambda X, Y: (X - 1.2) ** 2 + (Y - 0.3) ** 2 < 0.36),
```

```
                        (3, 5.0, lambda X, Y: np.abs(X - 0.2) < 0.04),
                        (1, 4.0, lambda X, Y: (np.abs(X + 1.0) < 0.6) & (np.abs(Y) < 0.6))]:
        hit = inside(cx + z * dx, z * dy) & (z < Z)
        Z, lab = np.where(hit, z, Z), np.where(hit, k, lab)
    X, Y = cx + Z * dx, Z * dy
    img = ndi.map_coordinates(tex, [350 + 40 * Y + 60 * lab, 350 + 40 * X], order=1)
    flat = (lab == 0) & (X > 2.2) & (X < 3.6) & (Y > -2.0) & (Y < -0.9)
    rep = (lab == 0) & (X > -3.6) & (X < -2.2) & (Y > 0.9) & (Y < 2.0)
    img[flat] = 0.5                                                  # textureless patch:
     ↪  label 4
    img[rep] = 0.5 + 0.4 * np.sin(2 * np.pi * X[rep] / 0.15)        # repetitive stripes:
     ↪  label 5
    lab[flat], lab[rep] = 4, 5
    return img + rng.normal(0, 0.01, img.shape), Z, lab

left, Z_left, lab = render(0.0)
right, Z_right, _ = render(B)
d_true = f * B / Z_left                                  # ground-truth disparity of the left
 ↪  view
rows, cols = np.mgrid[0:H, 0:W]
xr = np.round(cols - d_true).astype(int)                 # where each left pixel lands in the
 ↪  right view
occluded = (xr < 0) | (np.abs(Z_right[rows, np.clip(xr, 0, W - 1)] - Z_left) > 0.05)

def sad_volume(ref, other, D, win, sign):
    """cost[d](y, x) = sum over a win x win window of |ref(y, x) - other(y, x - sign*d)|."""
    cost = np.full((D, H, W), np.inf)
    for d in range(D):
        c = ndi.uniform_filter(np.abs(ref - np.roll(other, sign * d, axis=1)), win) * win **
         ↪  2
        if sign > 0: c[:, :d] = np.inf                   # no partner pixel exists there
        else:        c[:, W - d:] = np.inf
        cost[d] = c
    return cost

D, win = 32, 7
d_left = sad_volume(left, right, D, win, +1).argmin(0)     # winner-take-all, left image as
 ↪  reference
d_right = sad_volume(right, left, D, win, -1).argmin(0)    # winner-take-all, right image as
 ↪  reference
lr_ok = np.abs(d_left - d_right[rows, np.clip(cols - d_left, 0, W - 1)]) <= 1   # left-right
 ↪  check

sgbm = cv2.StereoSGBM_create(minDisparity=0, numDisparities=D, blockSize=5,
                             P1=8 * 25, P2=32 * 25, uniquenessRatio=0, disp12MaxDiff=-1)
u8 = lambda a: np.clip(a * 255, 0, 255).astype(np.uint8)
d_sgbm = sgbm.compute(u8(left), u8(right)).astype(float) / 16    # OpenCV stores 16 x
 ↪  disparity

valid = ~occluded & (cols >= D)                          # visible in both views, away from the
 ↪  left strip
regions = {"textured": valid & np.isin(lab, [0, 1, 2]), "textureless": valid & (lab == 4),
           "stripes": valid & (lab == 5), "thin pole": valid & (lab == 3)}
bad = lambda d, m: 100 * np.mean(np.abs(d - d_true)[m] > 1)      # % of pixels off by > 1 px
print(f"disparity {d_true.min():.1f}-{d_true.max():.1f} px; occluded: {100 *
 ↪  occluded.mean():.1f}% of pixels")
```

```
print("bad-1 (%)          " + "".join(f"{k:>12s}" for k in regions))
for name, d in [("SAD + WTA", d_left), ("OpenCV SGBM", d_sgbm)]:
    print(f"{name:19s}" + "".join(f"{bad(d, m):12.1f}" for m in regions.values()))
print(f"left-right check: keeps {100 * lr_ok[valid].mean():.1f}% of visible pixels, "
      f"rejects {100 * (~lr_ok[occluded]).mean():.1f}% of occluded ones")
kept = valid & lr_ok
print(f"SAD + WTA bad-1 on all visible pixels: {bad(d_left, valid):.1f}% -> after the check
 ↪ {bad(d_left, kept):.1f}%")
```

```
disparity 8.3-20.0 px; occluded: 7.7% of pixels
bad-1 (%)              textured textureless     stripes   thin pole
SAD + WTA                   1.8        67.7        89.6        17.4
OpenCV SGBM                 1.7        18.2        53.3        67.2
left-right check: keeps 96.6% of visible pixels, rejects 93.0% of occluded ones
SAD + WTA bad-1 on all visible pixels: 6.1% -> after the check 3.6%
```

These numbers come from this synthetic scene only. On textured surfaces, SAD block matching is right on 98% of the pixels; in the textureless patch it is wrong on two thirds of them and on the stripes on nine out of ten, as the cost curves predicted. The left–right check rejects 93% of the occluded pixels while keeping 96.6% of the visible ones, and bad-1 on the kept pixels drops from 6.1% to 3.6%. Rejected pixels are *holes*, not errors: later stages know not to trust them and can fill them from neighbours.

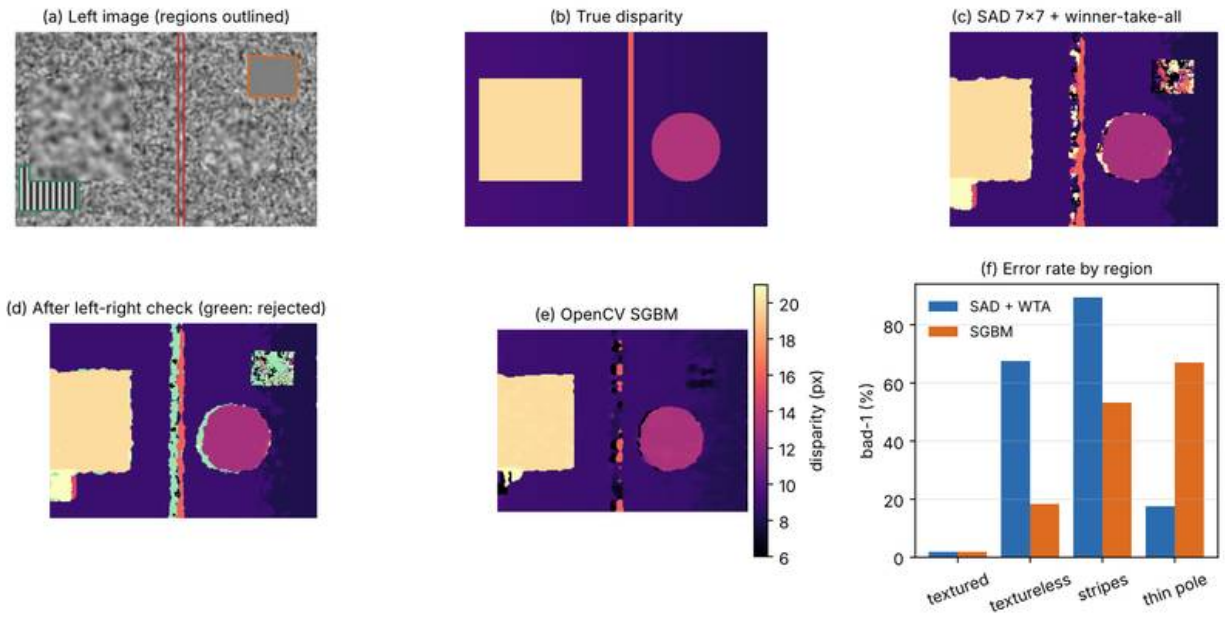



*Figure 4 — Block matching on the synthetic rectified pair. (c) SAD 7 × 7 with winner-take-all; (d) after the left–right check (rejected pixels in green); (e) OpenCV's semi-global matcher; (f) bad-1 per region from the snippet. The left strip, which has no partner pixels, is not evaluated.*

**Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

### Cost aggregation and semi-global matching

**Plain version.** Window matching decides each pixel alone. A better rule lets neighbouring pixels "vote": a pixel in a blank region should take the disparity of its textured neighbours, unless there is good evidence for a depth jump.

The classical way to say this is an energy over the whole disparity map $D$,

$$E(D) = \sum_{\mathbf{p}} C(\mathbf{p}, D(\mathbf{p})) + \sum_{(\mathbf{p},\mathbf{q})} P_1 \, [\, |D(\mathbf{p}) - D(\mathbf{q})| = 1 \,] + P_2 \, [\, |D(\mathbf{p}) - D(\mathbf{q})| > 1 \,],$$

where the second sum runs over pairs of neighbouring pixels $\mathbf{p}, \mathbf{q}$, $[\cdot]$ is 1 when the condition holds and 0 otherwise, $P_1$ is a small penalty for a change of one disparity level (slanted surfaces) and $P_2 > P_1$ is a larger penalty for a jump (depth edges) [5], [6]. Minimizing this energy exactly over a 2D image is intractable in general.

**Semi-global matching** (SGM) by Hirschmüller [5] approximates it by minimizing the energy along many straight 1D paths through each pixel, each of which can be solved exactly by dynamic programming, and summing the results. Along a path with direction $\mathbf{r}$ the aggregated cost is

$$L_{\mathbf{r}}(\mathbf{p}, d) = C(\mathbf{p}, d) + \min\Big(L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d),\; L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d \pm 1) + P_1,\; \min_k L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, k) + P_2\Big) - \min_k L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, k),$$

where $\mathbf{p} - \mathbf{r}$ is the previous pixel on the path and the last term only keeps the numbers bounded [6]. The costs of all directions are summed and WTA picks the disparity, so each pixel receives evidence from far away in many directions at a cost linear in the size of the cost volume. OpenCV's `StereoSGBM` is a modified version: by default 5 directions instead of the full algorithm's 8, small blocks instead of single pixels, and the Birchfield–Tomasi cost instead of mutual information [7].

The snippet shows SGM's trade-off. In the textureless patch, bad pixels drop from 67.7% to 18.2%, because the smoothness term carries the wall's disparity into it. But the thin pole, about five pixels wide, is mostly erased (17.4% → 67.2%): it costs two $P_2$ jumps for little evidence (Figure 4e). Neither method resolves the stripes, a genuine ambiguity of the data. Smoothness priors bet that the world is made of large surfaces, and that bet loses on thin structures.

*Online only: Semi-global matching step by step — https://blog.shyandram.dev/en/learn/deep-learning/depth-optical-flow/notes/semi-global-matching-step-by-step/*

### Refinement: sub-pixel disparity and post-processing

WTA returns integer disparities, but Figure 2b showed that fractions of a pixel matter. The usual fix fits a parabola through the cost at the best integer disparity and its two neighbours and takes the parabola's minimum. Further post-processing steps include the left–right check, a median

filter and filling of rejected pixels from their neighbours; MC-CNN's pipeline, for example, combines cross-based aggregation, SGM, a left–right check, sub-pixel enhancement, a median filter and a bilateral filter [6].

*Online only: Sub-pixel disparity and the soft argmin — https://blog.shyandram.dev/en/learn/deep-learning/depth-optical-flow/notes/sub-pixel-disparity-and-the-soft-argmin/*

### Learned stereo matching

**Plain version.** Every step of the classical pipeline has a hand-made rule. Deep stereo replaces them, one at a time, by learned ones: first the cost, then the aggregation, and finally the whole pipeline trained end to end on large synthetic datasets.

The surveys of Laga et al. [8] and Tosi et al. [9] follow the same order as this section: learned parts of the pipeline, then end-to-end networks, then the architectures of the 2020s.

### Learning the matching cost

MC-CNN by Žbontar and LeCun [6] trains a convolutional network to decide whether two small patches show the same point, using pairs built from ground-truth disparities as a binary classification set. The network's output replaces SAD as the matching cost, and the rest of the classical pipeline (cross-based aggregation, SGM, left–right check, sub-pixel enhancement, filtering) follows. The lesson: a learned similarity is far more discriminative than raw intensities, but the hand-made aggregation remained.

### End-to-end networks with cost volumes

End-to-end training needs dense ground truth for many images, which real scans rarely give. Mayer et al. [10] rendered three datasets (FlyingThings3D, Monkaa, Driving); FlyingThings3D alone has about 25,000 stereo frames of random objects flying along random 3D paths, and with it they trained DispNet, an encoder–decoder that regresses disparity [10]. Together these "Scene Flow" datasets became the usual pre-training data for stereo networks.

**GC-Net** by Kendall et al. [11] made the cost volume explicit and learnable. A shared 2D CNN extracts features from both images. For every disparity level the left feature is **concatenated** with the right feature shifted by that level, which gives a 4D volume of size height × width × (max disparity + 1) × feature size [11]. 3D convolutions over this volume learn the aggregation, using context in space and in disparity. Finally, a **soft argmin** turns the regularized costs $c_d$ into a sub-pixel disparity:

$$\hat{d} = \sum_{d=0}^{D_{\max}} d \cdot \sigma(-c_d), \qquad \sigma(-c_d) = \frac{e^{-c_d}}{\sum_{d'} e^{-c_{d'}}},$$

where $\sigma$ is the softmax over the disparity dimension and $D_{\max}$ the largest disparity [11]. Unlike the argmin it is differentiable, so the network trains end to end with a regression loss on $\hat{d}$. **PSM-**

**Net** [12] added a spatial pyramid pooling module that gathers context at several scales before the cost volume and a stacked-hourglass 3D CNN with intermediate supervision to regularize it, trained with the smooth L1 loss [12]. Both target ill-posed regions such as occlusions, repeated patterns, textureless and reflective surfaces, where local patches carry no answer [12].

The price of 3D convolutions is memory and computation proportional to $D \times H \times W$, so these networks work at reduced resolution and a fixed maximum disparity.

### Iterative refinement and zero-shot stereo

**RAFT-Stereo** [13] adapts the RAFT optical flow network (described in the optical flow section below) to rectified stereo: it replaces RAFT's 4D all-pairs volume by a 3D volume that only compares pixels on the same row, and refines a disparity field step by step with recurrent units that look up the volume around the current estimate, adding multi-level GRUs that pass information across the image more efficiently [13]. There is no 3D convolution and no fixed disparity range to regularize. Tosi et al. call this design "a game changer", noting that it brought much higher robustness to domain shift and that most recent frameworks follow it [9]. IGEV-Stereo [14] combines the two ideas: a geometry encoding volume that encodes geometry and context gives the recurrent updates a good starting point and is indexed by them at every iteration [14].

The open problem these surveys emphasized was **generalization**: networks fine-tuned on one benchmark often fail on another [9], [15]. FoundationStereo [16] targets zero-shot use: it is trained on a large synthetic dataset of about one million stereo pairs with automatic removal of ambiguous samples, and it adapts features from a monocular vision foundation model through a side-tuning backbone to narrow the gap between synthetic and real images [16].

## Monocular depth estimation

**Plain version.** With one image there is nothing to match. A network has to *recognize* depth the way people do with one eye closed: from perspective, the known sizes of things, occlusion, shading, haze and texture that gets finer with distance. These cues tell you which things are nearer, but they cannot tell a toy car from a real one.

### Why one image is not enough: scale ambiguity

Scale the whole scene by a factor $s$ and the image does not change, because projection divides by depth (Figure 5a; see also DLCV 11 and, for 3D detection, DLCV 03B). Eigen et al. called the global scale "a fundamental ambiguity in depth prediction" and showed that much of their error came from the mean depth alone [17].

A network can still learn *metric* cues, such as typical object sizes or the camera height of a driving dataset, but they fail when the camera or scene type changes. So two kinds of output are used:

- **Relative (affine-invariant) depth.** The prediction is correct up to an unknown scale and shift. Many such models, MiDaS and Depth Anything among them, predict *inverse* depth (disparity-

like values) [18], [19], so the output $\hat{y}$ satisfies $1/Z \approx s\hat{y} + t$ for unknown $s > 0$ and $t$. This is enough for ordering, segmentation, editing and many robot tasks when combined with a few metric measurements.
- **Metric depth.** The prediction is in metres. It requires either a known camera or a model that estimates the camera from the image.

### Losses that ignore scale and shift

To train on datasets whose depth is known only up to scale, or in different units, the loss must ignore the unknown factors. Eigen et al. introduced a **scale-invariant log error** [17]:

$$D(y, y^*) = \frac{1}{n}\sum_{i=1}^{n}(\log y_i - \log y_i^* + \alpha(y, y^*))^2, \qquad \alpha(y, y^*) = \frac{1}{n}\sum_{i=1}^{n}(\log y_i^* - \log y_i),$$

where $y_i$ and $y_i^*$ are the predicted and true depths at pixel $i$, $n$ is the number of pixels with ground truth, and $\alpha$ is the log-scale offset that best aligns the two maps; multiplying $y$ by any constant leaves $D$ unchanged [17]. For training they used a blend of this error and the ordinary log-space L2 error, weighted by $\lambda = 0.5$, which they found gave good absolute-scale predictions [17].

**MiDaS** by Ranftl et al. [18] went one step further: it predicts inverse depth (disparity) up to scale *and* shift, and aligns each prediction to the ground truth before computing the loss. With $\mathbf{d} \in \mathbb{R}^M$ the predicted disparities and $\mathbf{d}^*$ the ground truth at the $M$ valid pixels, the alignment solves

$$(s, t) = \arg\min_{s,t}\sum_{i=1}^{M}(sd_i + t - d_i^*)^2, \qquad \hat{d}_i = sd_i + t,$$

which is an ordinary least-squares fit of a line with a closed-form solution [18]. The **scale-and-shift-invariant loss** is then $\mathcal{L}_{\text{ssi}} = \frac{1}{2M}\sum_i \rho(\hat{d}_i - d_i^*)$ with a per-pixel loss $\rho$, here the squared error [18]. MiDaS also has a robust variant (median-based alignment, trimmed absolute loss), a gradient-matching term that keeps depth edges sharp, and mixes five training datasets, including one built from 3D films [18]. It is evaluated by **zero-shot cross-dataset transfer**, on datasets never seen in training [18].

*Online only: MiDaS (Ranftl et al., 2022) — https://blog.shyandram.dev/en/learn/deep-learning/depth-optical-flow/papers/midas-ranftl-et-al-2022/*

The snippet builds a toy depth map (a ground plane, a far wall and a box), simulates a relative model whose output is inverse depth with an unknown scale and shift, and compares three ways of reading it.

```
import numpy as np

rng = np.random.default_rng(1)
H, W, f, cam_h = 120, 160, 150.0, 1.5
v, u = np.mgrid[0:H, 0:W].astype(float)
Z = np.full((H, W), 30.0)                                    # far wall
below = v > H / 2 + 1
```

```
Z[below] = np.minimum(30.0, f * cam_h / (v[below] - H / 2))    # ground plane seen by a
 ↪  camera 1.5 m high
box = (np.abs(u - 50) < 15) & (v > 45) & (v < 80)
Z[box] = 5.0                                                   # an object 5 m away

# A "relative" network outputs inverse depth up to an unknown scale and shift, plus some
 ↪  noise.
pred = 0.37 / Z + 0.05 + rng.normal(0, 0.001, Z.shape)

def metrics(Zp, Zt):
    absrel = np.mean(np.abs(Zp - Zt) / Zt)
    rmse = np.sqrt(np.mean((Zp - Zt) ** 2))
    delta1 = np.mean(np.maximum(Zp / Zt, Zt / Zp) < 1.25)
    return f"AbsRel {absrel:6.3f}   RMSE {rmse:6.2f} m   delta<1.25 {100 * delta1:5.1f}%"

Z_raw = 1 / pred                                               # read the output as if it
 ↪  were metric
Z_med = Z_raw * np.median(Z) / np.median(Z_raw)                # median scaling (scale only)
A = np.c_[pred.ravel(), np.ones(pred.size)]                    # least squares for (s, t) in
 ↪  inverse depth
s, t = np.linalg.lstsq(A, (1 / Z).ravel(), rcond=None)[0]
Z_ssi = 1 / (s * pred + t)
print("raw output        ", metrics(Z_raw, Z))
print("median scaling    ", metrics(Z_med, Z))
print("scale+shift (LSQ) ", metrics(Z_ssi, Z), f"  s = {s:.3f}, t = {t:.4f}")
```

```
raw output         AbsRel  0.460   RMSE  10.46 m   delta<1.25  10.2%
median scaling     AbsRel  0.785   RMSE   7.24 m   delta<1.25  54.8%
scale+shift (LSQ)  AbsRel  0.043   RMSE   1.84 m   delta<1.25  99.5%   s = 2.699, t = -0.1348
```

These are toy numbers. Read as metric depth, the output is useless ($\delta < 1.25$ on 10% of the pixels). **Median scaling** multiplies the depth by one factor; it helps $\delta$ but makes AbsRel *worse*, because no single factor can undo a shift in inverse depth. The least-squares fit finds $s = 2.699$, $t = -0.1348$, almost exactly the simulated $1/Z = 2.703\,\hat{y} - 0.135$, and the map becomes nearly perfect. The same prediction can look poor or excellent depending on the alignment, so always check which one a paper used.

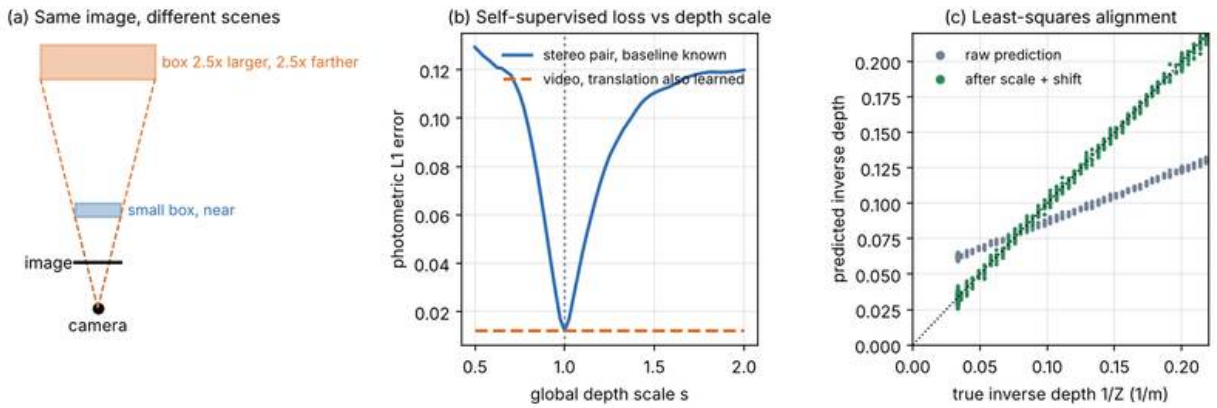


*Figure 5 — Scale ambiguity. (a) A box 2.5 times larger and farther gives the same image. (b) A self-supervised photometric loss pins the scale only when the baseline is known; if the camera translation is also learned, every scale fits equally well. (c) An affine-invariant prediction becomes accurate after a least-squares scale and shift in inverse depth (data from the snippet).*

### From relative to metric foundation models

Modern monocular models use standard dense-prediction networks; the progress came from training data and from how scale is handled. On the **relative** side, DPT [20] brought a Vision Transformer encoder to dense prediction, Depth Anything [19], [21] scaled training with tens of millions of pseudo-labelled images, and Marigold [22] turned an image diffusion model into a depth estimator. On the **metric** side, ZoeDepth [23] adds metric heads to a relative model, while Metric3D [24], [25], UniDepth [26] and Depth Pro [27] handle the camera explicitly, by mapping images to a canonical focal length or by estimating the camera from the image. The one idea to remember: relative depth is now robust across many kinds of images, and metric depth becomes usable once the model knows, or can infer, the camera. Dense pointmap models such as DUSt3R and VGGT, which predict geometry and cameras jointly, are covered in DLCV 11, Feed-forward reconstruction.

*Online only: Monocular depth foundation models: data and the camera — https://blog.shyandram.dev/en/learn/deep-learning/depth-optical-flow/notes/monocular-depth-foundation-models-data-and-the-camera/*

### Self-supervised depth from stereo pairs and video

**Plain version.** Instead of ground-truth depth, use a second view as the teacher. If the network's depth were right, you could warp the second view onto the first and get the same picture. The difference between the warped and the real image is the training signal.

### View synthesis as supervision

Garg et al. [28] trained a monocular network on stereo pairs with a known, small camera displacement: the predicted depth of the left image, together with the known baseline, warps the right image into the left view, and the photometric error of this reconstruction is the loss, with no depth labels at all [28]. Godard et al. [29] added a **left–right consistency** loss that makes the disparities predicted for the two views agree, which improved quality [29]. Zhou et al. [30] removed the second camera: from unlabelled monocular video they trained a depth network and a **pose network** together, with view synthesis between neighbouring frames as the only supervision [30].

For a target frame $I_t$ and a source frame $I_{t'}$, each target pixel $\mathbf{p}$ (in homogeneous coordinates) is mapped to the source by

$$\mathbf{p}' \sim K\, T_{t\to t'}\, D_t(\mathbf{p})\, K^{-1}\mathbf{p},$$

where $K$ is the camera intrinsic matrix, $D_t(\mathbf{p})$ is the predicted depth of the pixel, $T_{t\to t'}$ is the predicted relative camera pose (rotation and translation) and $\sim$ means equality up to the homogeneous scale. Sampling $I_{t'}$ at $\mathbf{p}'$ with a bilinear sampler (DLCV 09A.3) gives the synthesized image $I_{t'\to t}$; gradients flow through it into both networks.

### Monodepth2

Monodepth2 by Godard et al. [31] showed that a simple model with careful design choices beats more complex ones. Its photometric error between two images combines SSIM (DLCV 09, full-reference metrics) and L1:

$$\mathrm{pe}(I_a, I_b) = \frac{\alpha}{2}(1 - \mathrm{SSIM}(I_a, I_b)) + (1-\alpha)\|I_a - I_b\|_1, \qquad \alpha = 0.85,$$

where $I_a, I_b$ are the target image and a synthesized image, compared per pixel [31], [32]. Three ideas make it robust [31]:

1. **Per-pixel minimum reprojection.** With several source frames (previous and next), take the *minimum* error over sources instead of the average, $\mathcal{L}_p = \min_{t'} \mathrm{pe}(I_t, I_{t'\to t})$, because a pixel occluded in one source is usually visible in the other.
2. **Auto-masking.** Ignore pixels whose error *without* warping is already lower than with warping. This removes a static camera, and objects that move with the camera, which would otherwise teach the network "infinite depth" for cars driving ahead.
3. **Full-resolution multi-scale loss.** Upsample lower-resolution depth predictions to the input size before computing the error, which reduces artefacts.

An edge-aware smoothness term, weighted by $\lambda = 0.001$, discourages depth changes where the image itself is smooth [31].

### Scale, again

Trained from monocular video, a depth network never sees a metric quantity: depth and translation can be scaled together without changing $\mathbf{p}'$. Figure 5b shows it on the synthetic stereo scene: with the baseline fixed, the photometric loss has a sharp minimum at the true scale; if the translation scales with the depth, the loss is flat. This is why Monodepth2 reports monocular-trained results after per-image **median ground-truth scaling**, following SfMLearner, but not for stereo-trained ones, "as scale can be inferred from the known camera baseline" [31]. The other limits are those of matching itself: textureless and reflective surfaces, moving objects and occlusions.

## Optical flow as a learning problem

**Plain version.** Flow is stereo without the rulebook: the match can move in any direction, by a fraction of a pixel or by a hundred pixels, and objects move independently. Classical methods solved an energy for every image pair; deep methods learn to match and refine from synthetic videos.

### From energies to networks

The classical methods start from brightness constancy, $I_1(\mathbf{p}) = I_2(\mathbf{p} + \mathbf{F}(\mathbf{p}))$, which gives one equation for two unknowns per pixel, and add an assumption to resolve it: global smoothness of the flow field (Horn and Schunck [33]), constant flow in a small window solved by least squares (Lucas and Kanade [34]), or a local polynomial model of the image (Farnebäck [35]). DLCV 04B introduces these ideas and the aperture problem.

*Optional: Brightness constancy and the classical flow solvers (in DLCV 04B's Supplementary notes). The linearized flow equation, the Horn–Schunck energy and the polynomial expansion behind Farnebäck's method.*

Hur and Roth [36] describe the move to deep learning in two families: CNNs used as feature extractors inside classical energy formulations, and end-to-end networks that regress the flow directly. The second family won. Four milestones lead to RAFT:

- **FlowNet** [37] regressed flow end to end with a CNN. Lacking real training data, its authors built **Flying Chairs**, 22,872 image pairs of rendered 3D chair models moved by random affine transformations over photographs, and found that networks trained on it still generalized to Sintel and KITTI [37]. Its FlowNetC variant added a **correlation layer** that compares the two images' features at a range of displacements.
- **FlowNet 2.0** [38] stacked several FlowNets, each warping the second image with the current estimate, and showed that the order of the training datasets matters a lot [38].
- **SPyNet** [39] used a classical image pyramid: at each level one image is warped by the current flow and a small network predicts an update of less than a pixel [39].
- **PWC-Net** [40] combined **p**yramid, **w**arping and **c**ost volume on *learned* features, and is 17 times smaller than FlowNet2 [40].

### RAFT: all-pairs correlation and iterative updates

**Plain version.** RAFT compares every pixel of the first image with every pixel of the second, once. Then it starts from zero motion and, like a person adjusting a sketch, repeatedly looks at the match scores near its current guess and nudges the guess, the same small update applied again and again.

RAFT by Teed and Deng [41] has three parts [41]:

1. **Features.** A CNN encoder maps both images to features at 1/8 resolution with $D = 256$ channels; a separate context network encodes the first image only.
2. **All-pairs correlation.** The correlation volume holds the dot product of every feature pair,

$$C_{ijkl} = \sum_h g(I_1)_{ijh}\, g(I_2)_{klh},$$

where $g(I)_{ijh}$ is channel $h$ of the feature at row $i$, column $j$, and $C \in \mathbb{R}^{H\times W\times H\times W}$ at feature resolution. Average pooling of the last two dimensions with kernels 1, 2, 4 and 8 gives a four-level

**correlation pyramid**, which keeps full resolution for the first image and covers large displacements in the coarse levels. 3. **Iterative updates.** Starting from $\mathbf{f}_0 = \mathbf{0}$, a convolutional GRU (gated recurrent unit) repeatedly reads the correlation values in a small neighbourhood (radius $r = 4$) around the position each pixel currently points to, at all pyramid levels, together with the context features and the current flow, and outputs an update: $\mathbf{f}_{k+1} = \mathbf{f}_k + \Delta\mathbf{f}$. The weights are shared across iterations. A learned convex upsampling turns the 1/8-resolution flow into full resolution.

The loss supervises every iterate with exponentially increasing weight,

$$\mathcal{L} = \sum_{i=1}^{N} \gamma^{N-i} \, \|\mathbf{f}_{\text{gt}} - \mathbf{f}_i\|_1, \qquad \gamma = 0.8,$$

where $\mathbf{f}_i$ is the upsampled flow after update $i$, $N$ is the number of updates and $\mathbf{f}_{\text{gt}}$ the ground truth [41]. RAFT unrolls 12 updates in training and can run more at test time; the paper evaluates after 32 updates on Sintel and 24 on KITTI [41]. Training follows the standard schedule: Flying Chairs, then FlyingThings3D, then fine-tuning on the target benchmark [41].

Why did this design win? The all-pairs volume avoids a weakness of coarse-to-fine schemes, which can lose a small fast object at a coarse level for good [41]; the shared update behaves like a learned optimizer; and the recipe transferred to stereo [13] and scene flow [42].

*Online only: RAFT (Teed and Deng, 2020) — https://blog.shyandram.dev/en/learn/deep-learning/depth-optical-flow/papers/raft-teed-and-deng-2020/*

### After RAFT

Later models change one part of the recipe. GMFlow [43] and Unimatch [44] replace iterative lookups by **global matching** with Transformer features: every pixel is compared with all pixels of the other image and the flow is the softmax-weighted average position; Unimatch uses one model for flow, stereo and depth from posed images [44]. FlowFormer [45] encodes RAFT's cost volume with transformer layers. SEA-RAFT [46] keeps RAFT but adds a mixture-of-Laplace loss, a directly regressed initial flow and rigid-motion pre-training, for speed and better generalization [46].

### Code: measuring flow errors

The snippet builds a pair of frames with known flow: the background (the camera test image) rotates by 2° and zooms by 3% about the centre, and a textured patch moves by (14, 6) pixels on top of it. It runs OpenCV's Farnebäck flow with one and with four pyramid levels and reports the **end-point error** (EPE, the length of the error vector) and the **Fl** outlier rate used by KITTI (defined in Evaluation) on different kinds of pixels.

```
import numpy as np
import cv2
from scipy import ndimage as ndi
from skimage import color, data, transform
```

```
I0 = transform.resize(data.camera() / 255.0, (240, 320), anti_aliasing=True)
H, W = I0.shape
rows, cols = np.mgrid[0:H, 0:W].astype(float)
cy, cx = H / 2, W / 2

# Ground-truth motion from frame 0 to frame 1: the background rotates by 2 degrees and zooms
 ↪  by 3 %
# about the image centre; a 50 x 50 textured patch moves by (dx, dy) = (14, 6) pixels.
a, s = np.deg2rad(2.0), 1.03
bx = cx + s * (np.cos(a) * (cols - cx) - np.sin(a) * (rows - cy))     # where each background
 ↪  pixel goes
by = cy + s * (np.sin(a) * (cols - cx) + np.cos(a) * (rows - cy))
flow = np.stack([bx - cols, by - rows], -1)
patch = transform.resize(color.rgb2gray(data.astronaut())[30:230, 150:350], (50, 50),
 ↪  anti_aliasing=True)
obj0 = (rows >= 90) & (rows < 140) & (cols >= 120) & (cols < 170)
flow[obj0] = (14.0, 6.0)

# Render frame 1 by backward mapping (inverse rotation and zoom), then paste the moved patch.
xs = cx + (np.cos(a) * (cols - cx) + np.sin(a) * (rows - cy)) / s
ys = cy + (-np.sin(a) * (cols - cx) + np.cos(a) * (rows - cy)) / s
I1 = ndi.map_coordinates(I0, [ys, xs], order=3, mode="reflect")
I0 = I0.copy(); I0[90:140, 120:170] = patch
I1[96:146, 134:184] = patch
# occluded: background points of frame 0 that land under the patch in frame 1
occ = ~obj0 & (by >= 96) & (by < 146) & (bx >= 134) & (bx < 184)

def evaluate(est, mask):
    epe = np.linalg.norm(est - flow, axis=-1)[mask]
    mag = np.linalg.norm(flow, axis=-1)[mask]
    fl = np.mean((epe > 3) & (epe > 0.05 * mag))       # KITTI outlier: > 3 px and > 5 % of
     ↪  the true motion
    return f"EPE {epe.mean():5.2f} px  Fl {100 * fl:5.1f}%"

u8 = lambda im: (np.clip(im, 0, 1) * 255).round().astype(np.uint8)
inner = np.zeros((H, W), bool); inner[10:-10, 10:-10] = True
grad = ndi.gaussian_filter(np.hypot(ndi.sobel(I0, 0), ndi.sobel(I0, 1)), 4)
bg = inner & ~obj0 & ~occ
flat = grad < 0.05                                     # almost no texture (mostly the sky)
for levels in [1, 4]:
    est = cv2.calcOpticalFlowFarneback(u8(I0), u8(I1), None, pyr_scale=0.5, levels=levels,
     ↪  winsize=15,
                                       iterations=5, poly_n=7, poly_sigma=1.5, flags=0)
    print(f"Farneback, {levels} pyramid level(s):")
    for name, m in [("background, textured", bg & ~flat), ("background, flat", bg & flat),
                    ("moving patch", obj0), ("occluded", occ), ("all (Fl-all)", inner)]:
        print(f"   {name:21s} {evaluate(est, m)}")
print(f"true motion: background up to {np.linalg.norm(flow[~obj0], axis=-1).max():.1f} px,
 ↪  patch "
      f"{np.hypot(14, 6):.1f} px; occluded pixels: {occ.sum()}")
```

```
Farneback, 1 pyramid level(s):
   background, textured  EPE  0.87 px  Fl   9.3%
```

```
   background, flat      EPE  5.51 px  Fl  90.3%
   moving patch          EPE 18.24 px  Fl 100.0%
   occluded              EPE  5.07 px  Fl  68.0%
   all (Fl-all)          EPE  2.86 px  Fl  35.6%
Farneback, 4 pyramid level(s):
   background, textured  EPE  1.18 px  Fl  11.9%
   background, flat      EPE  5.52 px  Fl  90.5%
   moving patch          EPE  2.30 px  Fl  30.4%
   occluded              EPE 12.20 px  Fl  99.8%
   all (Fl-all)          EPE  2.55 px  Fl  35.2%
true motion: background up to 9.3 px, patch 15.2 px; occluded pixels: 871
```

These numbers are from this toy pair only. With one level, the patch's 15 px motion is out of reach and every patch pixel is an outlier; with four levels the patch is found (EPE 2.30 px), the coarse-to-fine idea of SPyNet and PWC-Net in classical form. Two failures remain. In flat regions (mostly the sky) 90% of the pixels are outliers: there is nothing to match. On **occluded** pixels, the strip the patch covers in frame 1, the pyramid makes things worse (EPE 12.2 px), because the coarse levels drag them along with the patch (Figure 6). Learned methods are trained to handle both cases from data, but neither disappears.

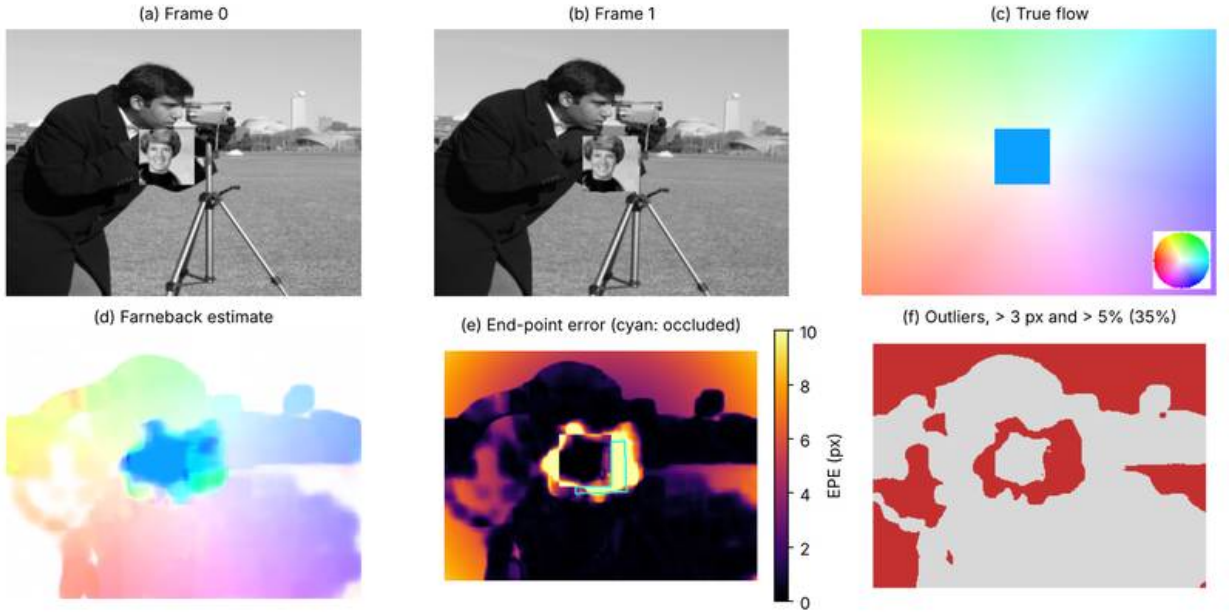



*Figure 6 — Farnebäck flow (4 pyramid levels) on the toy pair. Flow is drawn with hue for direction and saturation for length. Errors concentrate in textureless regions, around the moving patch and in the occluded strip (cyan). A 10 px border is not evaluated.*

### Scene flow in brief

**Plain version.** Optical flow says how a point moves in the image. Scene flow says how it moves in 3D: closer, farther, left or right, in metres.

**Scene flow** is the dense 3D motion field of the scene, a term that goes back to Vedula et al. [47]; optical flow is its projection onto the image. RAFT-3D states the task as: given a pair of stereo or RGB-D video frames, estimate the 3D motion of every pixel [42]. With a stereo camera it can be estimated from the disparity at two times plus the optical flow between them. The KITTI 2015 benchmark by Menze and Geiger [48] evaluates exactly these parts: D1 (disparity in the first frame), D2 (disparity in the second frame), Fl (optical flow) and SF, which counts a pixel as wrong if any of the three is wrong [49]. Menze and Geiger modelled the scene as a small number of rigidly moving objects, with each superpixel a 3D plane assigned to one of them [48]; RAFT-3D [42] instead estimates a dense field of rigid 3D motions with RAFT-style iterations and groups pixels softly into rigid objects with learned embeddings [42]. Dynamic reconstruction (DLCV 11) and point tracking (DLCV 07) are its long-range relatives.

### Evaluation

**Plain version.** All three tasks are scored per pixel against ground truth, with an average error and a "percentage of bad pixels". The details, especially how monocular predictions are aligned and which pixels count, change the numbers more than you would expect.

### Depth metrics

The standard set goes back to Eigen et al. [17]. With $y_i$ the predicted and $y_i^*$ the true depth at the $n$ evaluated pixels,

$$\mathrm{AbsRel} = \frac{1}{n}\sum_i \frac{|y_i - y_i^*|}{y_i^*}, \qquad \mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_i (y_i - y_i^*)^2},$$

$$\mathrm{SqRel} = \frac{1}{n}\sum_i \frac{(y_i - y_i^*)^2}{y_i^*}, \qquad \delta_k = \frac{1}{n}\Big|\Big\{i : \max\Big(\tfrac{y_i}{y_i^*}, \tfrac{y_i^*}{y_i}\Big) < 1.25^k\Big\}\Big|,$$

plus RMSE of $\log y$ [17]. AbsRel, SqRel and RMSE are errors (lower is better); $\delta_1, \delta_2, \delta_3$ with $k = 1, 2, 3$ are accuracies (higher is better). AbsRel weights near and far pixels by their relative error; RMSE is dominated by far pixels; $\delta_1$ counts pixels within 25% of the truth.

State three conventions with every result: the **alignment** (none for metric models, per-image median scaling for self-supervised monocular ones [31], least-squares scale and shift for affine-invariant ones [18]); the **depth cap** (80 m is standard on KITTI [31]); and the **split** (often that of Eigen et al. on KITTI [17], [31]). The alignment snippet showed how much the first one alone changes.

### Stereo metrics

- **EPE** (mean absolute disparity error, in pixels).
- **bad-τ**: percentage of pixels with a disparity error larger than $\tau$ pixels. Middlebury's Version 3 evaluation reports bad 2.0 by default, on all or only non-occluded pixels [3]; KITTI 2012 reports the percentage of erroneous pixels at thresholds from 2 to 5 pixels, in non-occluded areas (Out-Noc) and in total (Out-All) [50].
- **D1** (KITTI 2015): a pixel's disparity is correct if its error is below 3 px *or* below 5% of the true value; D1 is the percentage of outliers, reported over background, foreground or all pixels (D1-bg, D1-fg, D1-all) [49]. The relative clause keeps near objects, with large disparities, from being judged too harshly.

### Flow metrics

- **EPE**: the mean Euclidean distance between estimated and true flow vectors, $\frac{1}{n}\sum_i \|\mathbf{F}_i - \mathbf{F}_i^*\|_2$, where $\mathbf{F}_i$ and $\mathbf{F}_i^*$ are the estimated and true flow at pixel $i$; FlowNet calls it the standard error measure for optical flow and also trains with it [37]. Sintel results are reported as average EPE (often written AEPE).
- **Fl-all** (KITTI 2015): the percentage of pixels whose end-point error is at least 3 px *and* at least 5% of the true flow length, over all pixels with ground truth [49]. The snippet's `evaluate` function computes exactly this.

### Datasets and benchmarks

| Dataset | Task | What it contains | Ground truth |
|---|---|---|---|
| KITTI 2012 [51], [50] | stereo, flow | 194 training and 195 test pairs of real road scenes | sparse, from a laser scanner |
| KITTI 2015 [48], [49] | stereo, flow, scene flow | 200 training and 200 test scenes with moving cars | sparse, semi-automatic, with 3D CAD models fitted to moving vehicles |
| Middlebury 2014 [52] | stereo | 33 high-resolution indoor scenes; 20 used in the Version 3 evaluation (10 training, 10 test) | dense, subpixel-accurate |
| ETH3D [53] | stereo, multi-view stereo | indoor and outdoor scenes with DSLR images and low-resolution stereo videos; includes a low-resolution two-view benchmark | high-precision laser scans |
| MPI Sintel [54] | flow (also stereo, depth) | frames of an open-source animated film; "clean" and "final" passes, the final one with motion blur and atmospheric effects | dense, rendered |

| Dataset | Task | What it contains | Ground truth |
|---|---|---|---|
| Flying Chairs [37] | flow (training) | 22,872 synthetic image pairs | dense, rendered |
| FlyingThings3D [10] | stereo, flow, scene flow (training) | about 25,000 synthetic stereo frames at 960 × 540 | dense, rendered |
| NYU Depth v2 [55] | monocular depth (indoor) | Kinect video of 464 indoor scenes; 1,449 densely labelled RGB-D pairs | dense, from the Kinect depth camera, missing values filled in |

Sizes and ground-truth types come from the cited pages and papers; the description of Sintel's two passes is from [37], and KITTI 2015's semi-automatic ground truth from [49], [48]. The synthetic sets are used for pre-training, the real ones for fine-tuning and testing.

## Failure modes

**Plain version.** Every method in this tutorial relies on one of two things: something to match, or a prior learned from training data. Each failure below breaks one of them.

- **Textureless regions** (white walls, sky, road): flat matching costs (Figure 3b), filled by smoothness or learned context, both possibly wrong.
- **Repetitive patterns** (fences, tiles): several disparities fit equally well, as the stripes showed.
- **Reflective and transparent surfaces**: the matched content is a reflection or the scene behind. Tosi et al. count transparent and reflective objects among the four main open challenges of deep stereo, with domain shift, over-smoothing and asymmetric stereo pairs [9].
- **Occlusions**: no correct match exists; they need detection (left–right check) and filling, and in flow they are dragged along with neighbouring motion.
- **Thin structures and depth edges**: windows fatten foreground objects, smoothness erases poles and wires, and a soft argmin over a two-peaked cost lands between surfaces.
- **Large motion of small objects**: coarse-to-fine schemes can lose them; RAFT's all-pairs volume targets this [41].
- **Domain gap**: models pre-trained on synthetic data and fine-tuned on one benchmark degrade on new cameras, weather and scenes [9], [36].
- **Scale and the camera** (monocular): relative depth has no scale, and metric depth fails when the focal length is unknown or the image has been cropped unexpectedly.
- **Moving objects in self-supervision** break the static-scene assumption; auto-masking handles only the simplest cases [31]. Rajapaksha et al. add thin, small and distant objects to the open gaps of monocular depth [56].

## Modern view

### What the surveys say

Five reviews cover the ground of this tutorial. **Laga et al.** [8] survey deep stereo (two-view and multi-view) up to about 2019, split into methods that learn parts of the classical pipeline and end-

to-end pipelines, with datasets and supervision modes; monocular depth is outside their scope. **Poggi et al.** [15] stress that learning and stereo help each other: learning improves stereo, and stereo supervises monocular networks; they also cover confidence estimation and domain shift. **Tosi et al.** [9] review the 2020s: architectures grouped into cost volume aggregation, neural architecture search, iterative optimization, Vision Transformers and Markov random fields, and four open challenges (domain shift, over-smoothing, transparent and reflective objects, asymmetric stereo); in 2024 they noted that a stereo foundation model was still missing. **Rajapaksha et al.** [56] organize monocular depth by input and output, architecture and degree of supervision. **Hur and Roth** [36] trace optical flow from energy minimization to CNNs and name domain overfitting and complex training schedules as the main limits.

*Online only: Guide to the Modern view surveys — https://blog.shyandram.dev/en/learn/deep-learning/depth-optical-flow/papers/guide-to-the-modern-view-surveys/*

### The state of the art, 2023–2026

- **Foundation models for monocular depth.** Large-scale data (Depth Anything [19], [21]), generative priors (Marigold [22]) and explicit camera handling (Metric3D [24], [25], UniDepth [26], Depth Pro [27]) made zero-shot depth from arbitrary photos practical, first relative and now increasingly metric.
- **Iterative matching for stereo and flow.** The RAFT recipe of a correlation volume plus recurrent updates dominates both tasks [9]; recent variants add geometry volumes (IGEV-Stereo [14]), global matching with attention (GMFlow, Unimatch [43], [44]) or simpler training (SEA-RAFT [46]).
- **Zero-shot stereo.** FoundationStereo [16] answered the missing-foundation-model point of Tosi et al. by combining a million synthetic pairs with monocular priors.
- **Convergence with 3D reconstruction.** Feed-forward models that predict pointmaps and cameras (DLCV 11) blur the line between depth estimation, stereo and structure from motion.

### Open problems

1. **Metric depth in the wild**: unknown, cropped or unusual cameras, and a reliable uncertainty for each prediction.
2. **Non-Lambertian scenes**: glass, mirrors and water for both stereo and flow [9].
3. **Sharp boundaries and thin structures** without over-smoothing or "flying pixels" between surfaces.
4. **Temporal consistency**: per-frame depth flickers in video; flow and depth over long sequences should agree with each other.
5. **Evaluation**: sparse or semi-automatic ground truth, differing alignment conventions and benchmark saturation make small differences hard to interpret.

### Key takeaways

- Rectified stereo constrains matches to image rows; flow searches across time; monocular depth relies on learned or geometric assumptions.
- Depth sensitivity increases as disparity shrinks; calibration, baseline and sub-pixel precision limit usable range.
- Cost aggregation and regularization fill uncertain regions but can erase thin structures.
- Distinguish relative from metric depth and state any scale or shift alignment used in evaluation.
- Photometric supervision depends on visibility and scene assumptions; moving or occluded content needs special handling.
- Report error units, evaluated pixels, depth caps and splits; comparable model names do not imply comparable protocols.

### Exercises

1. **Choose a baseline.** A stereo camera with $f = 1000$ px must measure depth at 40 m with an error below 1 m, assuming a disparity error of 0.3 px. What is the smallest baseline? What disparity does a point at 40 m then have, and what happens at 80 m?

   **Hint**

   From $\delta Z \approx Z^2 \delta d/(fB) < 1$: $B > 1600 \cdot 0.3/1000 = 0.48$ m. The disparity at 40 m is $fB/Z = 12$ px. At 80 m the error is four times larger (about 4 m) and the disparity is 6 px.

2. **Window size and fattening.** In the block-matching snippet, change `win` to 3, 7 and 15 and compare bad-1 on the four regions. Why does a large window help in one place and hurt in another?

   **Hint**

   In our run, textureless bad-1 fell from 84.4% ($3 \times 3$) to 67.7% ($7 \times 7$) and 39.9% ($15 \times 15$), while the thin pole rose from 15.3% to 17.4% and 74.6%. A large window gathers texture from outside the flat patch, but near depth edges it mixes two surfaces and spreads the foreground disparity into the background ("foreground fattening"), which swallows the pole.

3. **Stripes and the period.** The stripes on the wall have a period of 0.15 world units, and the wall there is about 8.6 units away. Estimate their period in pixels (use $f = 320$ px), then explain why block matching gives wrong disparities there. Which change to the scene would make the stripes matchable, and which change to the algorithm might help?

   **Hint**

   The period is about $320 \cdot 0.15/8.6 \approx 5.6$ px, smaller than the true disparity there (about 9.3 px), so shifts that differ by one period fit equally well. A wider search does not help. Non-periodic texture would; a smoothness term that prefers the surrounding wall's disparity reduces the errors (in our run SGBM's bad-1 on the stripes was 53.3% against 89.6% for block matching) but does not remove them.

4. **Scale-invariant losses by hand.** Show that Eigen's error $D(y, y^*)$ does not change when $y$ is multiplied by a constant $c > 0$. Then show that MiDaS's least-squares $(s, t)$ makes $\mathcal{L}_{\text{ssi}}$ unchanged under $d \mapsto ad + b$ for any $a \neq 0$ and $b$.

   **Hint**

   $\log(cy_i) = \log c + \log y_i$, and $\alpha$ absorbs $-\log c$, so every residual is unchanged. For MiDaS, if $(s, t)$ is optimal for $d$, then $(s/a, t - sb/a)$ gives the same aligned values for $ad + b$.

5. **Pyramids and large motion.** In the flow snippet, move the patch by $(24, 10)$ instead of $(14, 6)$ and run with 1, 4 and 6 pyramid levels; then move it by only $(2, 1)$. Update the pasted position in frame 1 and the occlusion mask too. When does the pyramid help?

   **Hint**

   In our run, at $(24, 10)$ the patch was lost with every setting (EPE about 26 px), and 6 levels gave exactly the same result as 4, so the extra levels added nothing here. At $(2, 1)$ all three settings gave a patch EPE of 0.27 px. A pyramid helps only for motions that its coarse levels can bring within reach of the local model; beyond that, a different search is needed, such as RAFT's all-pairs correlation.

## References


1. R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed., Cambridge University Press, 2004. doi
2. D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International Journal of Computer Vision*, vol. 47, no. 1–3, pp. 7–42, 2002. doi
3. D. Scharstein, R. Szeliski and H. Hirschmüller, "Middlebury Stereo Evaluation — Version 3," official benchmark page (bad 2.0 metric, non-occluded and all masks). link ·stereo home page
4. R. Zabih and J. Woodfill, "Non-parametric local transforms for computing visual correspondence," in *Computer Vision — ECCV '94* (Lecture Notes in Computer Science), pp. 151–158, Springer, 1994. doi
5. H. Hirschmüller, "Stereo processing by semiglobal matching and mutual information," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 2, pp. 328–341, 2008. doi
6. J. Žbontar and Y. LeCun, "Stereo matching by training a convolutional neural network to compare image patches," *Journal of Machine Learning Research*, vol. 17, no. 65, pp. 1–32, 2016. JMLR ·arXiv
7. OpenCV contributors, "cv::StereoSGBM class documentation," OpenCV source, `modules/calib3d/include/opencv2/calib3d.hpp` (4.x branch). source
8. H. Laga, L. V. Jospin, F. Boussaid and M. Bennamoun, "A survey on deep learning techniques for stereo-based depth estimation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 4, pp. 1738–1764, 2022. doi ·arXiv
9. F. Tosi, L. Bartolomei and M. Poggi, "A survey on deep stereo matching in the twenties," *International Journal of Computer Vision*, vol. 133, no. 7, pp. 4245–4276, 2025. doi ·arXiv

10. N. Mayer, E. Ilg, P. Häusser, P. Fischer, D. Cremers, A. Dosovitskiy and T. Brox, "A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation," in *Proc. IEEE CVPR*, pp. 4040–4048, 2016. CVF ·doi
11. A. Kendall, H. Martirosyan, S. Dasgupta, P. Henry, R. Kennedy, A. Bachrach and A. Bry, "End-to-end learning of geometry and context for deep stereo regression," in *Proc. IEEE ICCV*, 2017. CVF
12. J.-R. Chang and Y.-S. Chen, "Pyramid stereo matching network," in *Proc. IEEE/CVF CVPR*, 2018. CVF
13. L. Lipson, Z. Teed and J. Deng, "RAFT-Stereo: Multilevel recurrent field transforms for stereo matching," in *Proc. International Conference on 3D Vision (3DV)*, pp. 218–227, 2021. doi ·arXiv
14. G. Xu, X. Wang, X. Ding and X. Yang, "Iterative geometry encoding volume for stereo matching," in *Proc. IEEE/CVF CVPR*, 2023. CVF
15. M. Poggi, F. Tosi, K. Batsos, P. Mordohai and S. Mattoccia, "On the synergies between machine learning and binocular stereo for depth estimation from images: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021. doi ·arXiv
16. B. Wen, M. Trepte, J. Aribido, J. Kautz, O. Gallo and S. Birchfield, "FoundationStereo: Zero-shot stereo matching," in *Proc. IEEE/CVF CVPR*, 2025. CVF ·arXiv
17. D. Eigen, C. Puhrsch and R. Fergus, "Depth map prediction from a single image using a multi-scale deep network," in *Proc. NeurIPS*, 2014. proceedings ·arXiv
18. R. Ranftl, K. Lasinger, D. Hafner, K. Schindler and V. Koltun, "Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 3, pp. 1623–1637, 2022. doi ·arXiv
19. L. Yang, B. Kang, Z. Huang, X. Xu, J. Feng and H. Zhao, "Depth Anything: Unleashing the power of large-scale unlabeled data," in *Proc. IEEE/CVF CVPR*, 2024. CVF ·arXiv
20. R. Ranftl, A. Bochkovskiy and V. Koltun, "Vision transformers for dense prediction," in *Proc. IEEE/CVF ICCV*, 2021. CVF
21. L. Yang, B. Kang, Z. Huang, Z. Zhao, X. Xu, J. Feng and H. Zhao, "Depth Anything V2," in *Proc. NeurIPS*, 2024. proceedings ·arXiv
22. B. Ke, A. Obukhov, S. Huang, N. Metzger, R. C. Daudt and K. Schindler, "Repurposing diffusion-based image generators for monocular depth estimation," in *Proc. IEEE/CVF CVPR*, 2024. CVF ·arXiv
23. S. F. Bhat, R. Birkl, D. Wofk, P. Wonka and M. Müller, "ZoeDepth: Zero-shot transfer by combining relative and metric depth," arXiv:2302.12288, 2023. arXiv
24. W. Yin, C. Zhang, H. Chen, Z. Cai, G. Yu, K. Wang, X. Chen and C. Shen, "Metric3D: Towards zero-shot metric 3D prediction from a single image," in *Proc. IEEE/CVF ICCV*, 2023. CVF · arXiv
25. M. Hu, W. Yin, C. Zhang, Z. Cai, X. Long, K. Wang, H. Chen, G. Yu, C. Shen and S. Shen, "Metric3D v2: A versatile monocular geometric foundation model for zero-shot metric depth and surface normal estimation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 12, pp. 10579–10596, 2024. doi ·arXiv
26. L. Piccinelli, Y.-H. Yang, C. Sakaridis, M. Segu, S. Li, L. Van Gool and F. Yu, "UniDepth: Universal monocular metric depth estimation," in *Proc. IEEE/CVF CVPR*, 2024. CVF ·arXiv
27. A. Bochkovskii, A. Delaunoy, H. Germain, M. Santos, Y. Zhou, S. R. Richter and V. Koltun, "Depth Pro: Sharp monocular metric depth in less than a second," in *Proc. ICLR*, 2025. Open-

Review ·arXiv
28. R. Garg, V. Kumar B.G., G. Carneiro and I. Reid, "Unsupervised CNN for single view depth estimation: Geometry to the rescue," in *Proc. ECCV* (Lecture Notes in Computer Science), pp. 740–756, 2016. doi ·arXiv
29. C. Godard, O. Mac Aodha and G. J. Brostow, "Unsupervised monocular depth estimation with left-right consistency," in *Proc. IEEE CVPR*, 2017. CVF
30. T. Zhou, M. Brown, N. Snavely and D. G. Lowe, "Unsupervised learning of depth and ego-motion from video," in *Proc. IEEE CVPR*, 2017. CVF
31. C. Godard, O. Mac Aodha, M. Firman and G. J. Brostow, "Digging into self-supervised monocular depth estimation," in *Proc. IEEE/CVF ICCV*, 2019. CVF ·arXiv
32. Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. doi
33. B. K. P. Horn and B. G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, no. 1–3, pp. 185–203, 1981. doi
34. B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, 1981. pdf
35. G. Farnebäck, "Two-frame motion estimation based on polynomial expansion," in *Image Analysis* (Lecture Notes in Computer Science), pp. 363–370, Springer, 2003. doi
36. J. Hur and S. Roth, "Optical flow estimation in the deep learning age," in *Modelling Human Motion*, N. Noceti, A. Sciutti and F. Rea, Eds., pp. 119–140, Springer, 2020. doi ·arXiv
37. A. Dosovitskiy, P. Fischer, E. Ilg, P. Häusser, C. Hazırbaş, V. Golkov, P. van der Smagt, D. Cremers and T. Brox, "FlowNet: Learning optical flow with convolutional networks," in *Proc. IEEE ICCV*, 2015. CVF ·arXiv
38. E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy and T. Brox, "FlowNet 2.0: Evolution of optical flow estimation with deep networks," in *Proc. IEEE CVPR*, 2017. CVF
39. A. Ranjan and M. J. Black, "Optical flow estimation using a spatial pyramid network," in *Proc. IEEE CVPR*, 2017. CVF
40. D. Sun, X. Yang, M.-Y. Liu and J. Kautz, "PWC-Net: CNNs for optical flow using pyramid, warping, and cost volume," in *Proc. IEEE/CVF CVPR*, 2018. CVF
41. Z. Teed and J. Deng, "RAFT: Recurrent all-pairs field transforms for optical flow," in *Proc. ECCV* (Lecture Notes in Computer Science), pp. 402–419, 2020. doi ·arXiv
42. Z. Teed and J. Deng, "RAFT-3D: Scene flow using rigid-motion embeddings," in *Proc. IEEE/CVF CVPR*, 2021. CVF ·arXiv
43. H. Xu, J. Zhang, J. Cai, H. Rezatofighi and D. Tao, "GMFlow: Learning optical flow via global matching," in *Proc. IEEE/CVF CVPR*, 2022. CVF
44. H. Xu, J. Zhang, J. Cai, H. Rezatofighi, F. Yu, D. Tao and A. Geiger, "Unifying flow, stereo and depth estimation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 11, pp. 13941–13958, 2023. doi ·arXiv
45. Z. Huang, X. Shi, C. Zhang, Q. Wang, K. C. Cheung, H. Qin, J. Dai and H. Li, "FlowFormer: A transformer architecture for optical flow," in *Proc. ECCV* (Lecture Notes in Computer Science), pp. 668–685, 2022. doi ·arXiv
46. Y. Wang, L. Lipson and J. Deng, "SEA-RAFT: Simple, efficient, accurate RAFT for optical flow," in *Proc. ECCV* (Lecture Notes in Computer Science), pp. 36–54, 2024. doi ·arXiv

47. S. Vedula, S. Baker, P. Rander, R. Collins and T. Kanade, “Three-dimensional scene flow,” in *Proc. IEEE ICCV*, vol. 2, pp. 722–729, 1999. doi
48. M. Menze and A. Geiger, “Object scene flow for autonomous vehicles,” in *Proc. IEEE CVPR*, pp. 3061–3070, 2015. CVF ·doi
49. A. Geiger, P. Lenz, C. Stiller and R. Urtasun, “Scene flow evaluation 2015 (stereo 2015, flow 2015, scene flow 2015),” The KITTI Vision Benchmark Suite, official benchmark page (dataset size, D1, D2, Fl and SF outlier definitions). link
50. A. Geiger, P. Lenz, C. Stiller and R. Urtasun, “Stereo evaluation 2012,” The KITTI Vision Benchmark Suite, official benchmark page (dataset size, Out-Noc and Out-All at 2–5 px). link
51. A. Geiger, P. Lenz and R. Urtasun, “Are we ready for autonomous driving? The KITTI vision benchmark suite,” in *Proc. IEEE CVPR*, pp. 3354–3361, 2012. doi
52. D. Scharstein, H. Hirschmüller, Y. Kitajima, G. Krathwohl, N. Nešić, X. Wang and P. Westling, “High-resolution stereo datasets with subpixel-accurate ground truth,” in *Pattern Recognition (Proc. GCPR)* (Lecture Notes in Computer Science), pp. 31–42, Springer, 2014. doi ·datasets
53. T. Schöps, J. L. Schönberger, S. Galliani, T. Sattler, K. Schindler, M. Pollefeys and A. Geiger, “A multi-view stereo benchmark with high-resolution images and multi-camera videos,” in *Proc. IEEE CVPR*, pp. 2538–2547, 2017. CVF ·datasets
54. D. J. Butler, J. Wulff, G. B. Stanley and M. J. Black, “A naturalistic open source movie for optical flow evaluation,” in *Proc. ECCV* (Lecture Notes in Computer Science), pp. 611–625, 2012. doi ·project page
55. N. Silberman, D. Hoiem, P. Kohli and R. Fergus, “Indoor segmentation and support inference from RGBD images,” in *Proc. ECCV* (Lecture Notes in Computer Science), pp. 746–760, 2012. doi ·dataset page
56. U. Rajapaksha, F. Sohel, H. Laga, D. Diepeveen and M. Bennamoun, “Deep learning-based depth estimation methods from monocular image and videos: A comprehensive survey,” *ACM Computing Surveys*, vol. 56, no. 12, 2024. doi ·arXiv

# DLCV 12 • Image Generation: From GANs to Text-to-Image Systems

How image generators are built and judged as vision systems: the task and what good means, the model families on one map, text encoders, cross-attention and guidance in text-to-image models, control, personalization and editing, autoregressive and unified multimodal generators, evaluation, data, risks and failure modes.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/image-generation/

## What you'll learn

- Define generation and editing tasks through their conditions and required outputs.
- Distinguish generator families from pixel, latent and token representations.
- Trace text encoding, conditioning, sampling and decoding.
- Compare spatial control, personalization and editing mechanisms.
- Evaluate quality, diversity, alignment, cost and data-related failure modes.

## The big picture

An image generator turns a request into a new picture. The request can be nothing at all ("any face"), a class label ("a goldfinch"), a sentence ("a red ball on the grass"), or another image plus an instruction ("make it winter"). The answer should look real, should vary from one try to the next, and should actually show what was asked.

> **Explain like I'm five**
>
> Imagine a sketch artist who works for the police. You describe a face; the artist starts with a rough smudge and keeps refining it, glancing back at your words after every stroke ("big eyes, short hair"). A generator does the same thing many times per second. The "words" are turned into numbers by a reading model, the "glancing back" is a step called cross-attention, and the "smudge-to-picture" refinement is the generator itself. Some artists work differently, placing one mosaic tile after another, but they all listen to the description while they draw.

**Why it matters.** Image generation went from small, blurry faces to photorealistic, prompt-following pictures in less than a decade, and it is now part of design tools, photo editors, data pipelines and restoration systems (DLCV 09B uses the same models to restore damaged photos). As a computer-vision **system**, a modern generator combines a language model, an image compressor, a generative network, a sampler and an evaluation pipeline. Each part can fail on its own, and each part has its own literature.

**Where this chapter sits.** The theory lives in the ML track, and this chapter does not repeat it. ML 07 derives VAEs, GANs, normalizing flows and the classic metrics (Inception Score, FID, precision and recall) with code. ML 08 derives diffusion, DDIM, classifier-free guidance, latent diffusion, diffusion transformers, flow matching and few-step samplers. ML 09 covers autoregressive models over pixels and tokens, and ML 10 covers CLIP and parameter-efficient fine-tuning. Here we take the **system view**: how these pieces are assembled into text-to-image products, how they are steered, how they are judged, and what can go wrong. The two code examples run in seconds on a CPU with NumPy and scikit-image, and use no network weights.

**Next.** DLCV 12A extends these systems from images to video.

## The task: what to generate, and what "good" means

**Plain version.** Generation produces outputs under conditions such as text, labels or an input image. Evaluate realism, diversity and adherence to those conditions separately.

### Six kinds of generation

All the tasks below are sampling from a conditional distribution $p(x \mid c)$, where $x$ is an image and $c$ is the condition; the unconditional case uses an empty condition $c = \emptyset$.

| Task | Condition $c$ | Output | Example method in this chapter |
|---|---|---|---|
| Unconditional | nothing | an image from the training domain | StyleGAN on faces [1] |
| Class-conditional | a class label | an image of that class | BigGAN [2], DiT [3] |
| Text-to-image | a text prompt | an image matching the prompt | Imagen [4], latent diffusion [5] |
| Image-to-image | an image (often plus text) | a new image that keeps its structure | SDEdit [6], ControlNet [7] |
| Editing | an image plus an edit request | the edited image | Prompt-to-Prompt [8], InstructPix2Pix [9] |
| Inpainting / outpainting | an image plus a mask | the image with the masked region filled | RePaint [10] |

Because the form is shared, one trained text-to-image model can serve most rows of the table: edits, masks and control images are extra conditions or a different starting point for the sampler.

### Three questions: fidelity, diversity, alignment

- **Fidelity (quality).** Does each sample look like a real image of the domain, without artifacts?
- **Diversity (coverage).** Do many samples together cover the variety of real images, or does the model repeat a few favorites (mode collapse)?

- **Alignment.** Does the sample show what the condition asked for: the right objects, counts, colors, positions and text?

Add a fourth, practical question: **cost**, the number of network evaluations and the seconds per image. The first three questions pull against each other. Most generators expose a dial that trades diversity for fidelity and alignment: the **truncation trick** of BigGAN shrinks the spread of the input noise [2], and **classifier-free guidance** in diffusion models pushes samples toward the condition [11]. Turning such a dial up makes images cleaner and more on-prompt, but more alike. Keep this trade-off in mind whenever you read a single "quality" number. The metrics that measure fidelity and diversity separately are derived in ML 07; this chapter adds the alignment side in the evaluation section below.

## The model families on one map

**Plain version.** GANs, diffusion, flow and autoregressive models learn different generation procedures. Separately choose whether the system represents images as pixels, continuous latents or tokens.

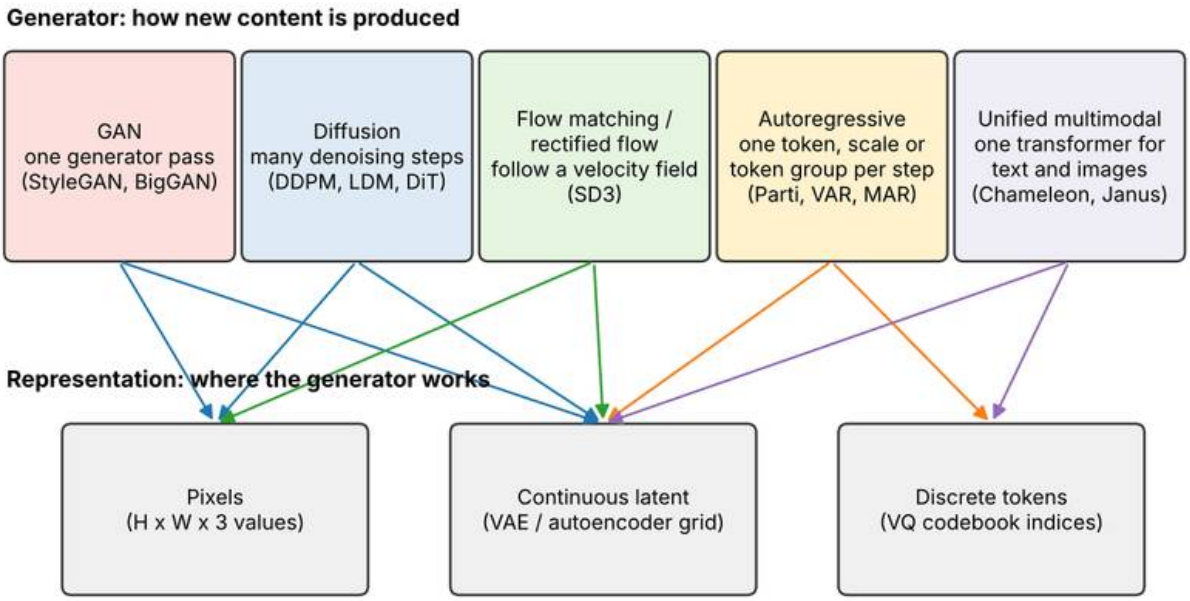



*Figure 1 — The families on one map. The top row is how new content is produced; the bottom row is where the generator works. Arrows show common pairings; the representation is a separate design choice from the generator.*

**GANs: one pass through a generator.** A generator maps random noise to an image, and a discriminator, trained at the same time, tries to tell generated images from real ones; the generator improves by fooling it [12]. Sampling is a single forward pass, so GANs are fast. StyleGAN [1] maps the noise to an intermediate "style" vector and injects it at every resolution of the generator,

which separates coarse attributes (pose, identity) from fine ones (freckles, hair) and allows scale-specific control. BigGAN [2] scaled class-conditional GANs on ImageNet and introduced the truncation trick mentioned above. Training is unstable and can miss modes, which made GANs hard to scale to open-ended prompts. Theory and code: ML 07, GANs.

**VAEs and tokenizers: compress first.** A variational autoencoder (VAE) learns an encoder to a compact latent code, a decoder back to pixels, and a simple prior over codes [13]. In today's systems the autoencoder's main job is **compression in front of another generator**: latent diffusion runs its denoiser on a continuous latent grid that is several times smaller than the image [5], and token-based models use a **vector-quantized** autoencoder (VQ-VAE [14], VQGAN [15]) that turns an image into a grid of integer indices from a learned codebook, like words from a dictionary. The decoder sets a ceiling: no generator can produce detail the decoder cannot reconstruct. Details: ML 07, VAEs and ML 09, visual tokens.

**Diffusion: many small denoising steps.** Train a network to remove a little Gaussian noise from a noisy image; to generate, start from pure noise and apply it many times [16]. Training is a stable regression, and with better architectures and classifier guidance diffusion models overtook GANs on ImageNet [17]. Running the same process on an autoencoder's latent grid, **latent diffusion**, made high-resolution text-to-image generation affordable [5]. Theory and code: ML 08 and its section on latent diffusion.

**Flow matching and rectified flow: follow a velocity field.** Draw a noise sample and a data sample, connect them with a straight line, and train a network to predict the direction of travel along such lines; to generate, start from noise and follow the predicted directions [18], [19]. It is closely related to diffusion (a different path between noise and data) and is now used at scale: Stable Diffusion 3 trains a large latent transformer with a rectified-flow objective [20]. Theory: ML 08, flow matching.

**Autoregressive and unified models: write the picture piece by piece.** Turn the image into a sequence of tokens and predict the next token (or the next group of tokens, or the next, sharper version of the whole token map) given everything generated so far. This is the language-model recipe applied to images, and its newest form is a single transformer that reads and writes both text and images. It gets its own section below.

| Family | How it samples | Sequential network calls | Main strength | Main thing to watch |
|---|---|---|---|---|
| GAN | one generator pass | one | speed, smooth editable latent space | training stability, mode coverage |
| Diffusion | repeated denoising from noise | tens to hundreds; fewer with distillation | quality, coverage, stable training | sampling cost |
| Flow matching | integrate a learned velocity | tens; fewer with distillation | simple objective, straighter paths | still needs several steps unless distilled |
| Autoregressive | predict tokens in some order | one per token, group or scale | reuses language-model machinery, exact likelihood of tokens | tokenizer ceiling, error accumulation |

Two points from the map are easy to forget. The generator and the representation are separate choices: a diffusion model can work on pixels (Imagen [4]) or on latents (latent diffusion [5]), and an autoregressive model can use discrete or continuous tokens [21]. And real systems mix families: an adversarial loss trains the tokenizer of VQGAN [15], and adversarial distillation turns a diffusion model into a one-to-four-step generator [22].

## The text-to-image system

**Plain version.** A latent text-to-image pipeline encodes the prompt, generates a conditioned latent representation and decodes pixels. The conditioning interface connects language to image features.

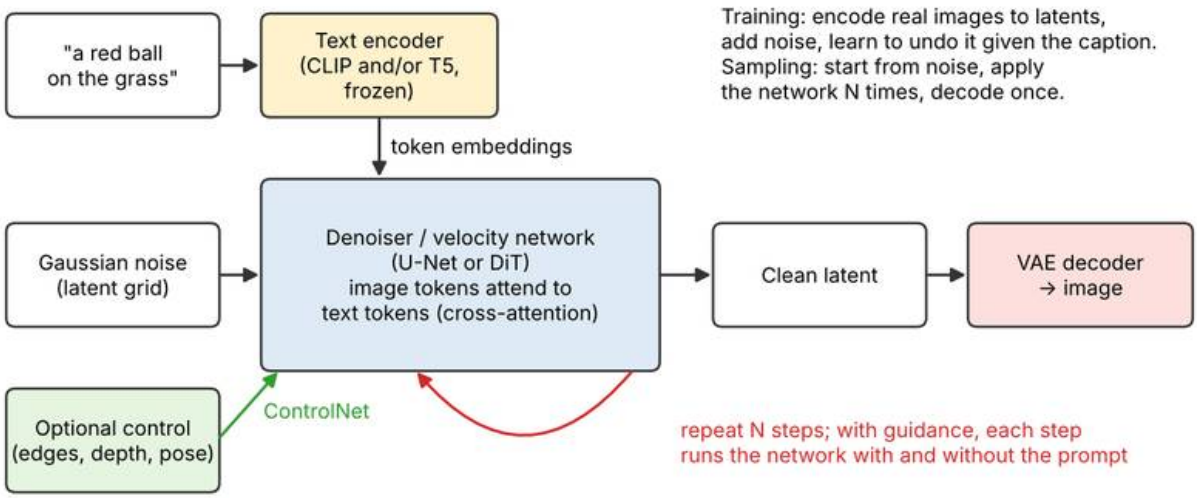



*Figure 2 — A latent text-to-image pipeline. The text encoder and the autoencoder are usually trained separately and frozen; only the denoiser (or velocity network) is trained on image–caption pairs. Sampling calls the denoiser N times and the decoder once.*

In training, each image of an image–caption pair is encoded to a latent, noised to a random level, and the denoiser learns to undo the noise **given the caption's embeddings**. At sampling time, the denoiser starts from pure noise and runs for $N$ steps, then the decoder produces pixels. The diffusion and sampling equations behind these two sentences are in ML 08; below we look at the parts that are specific to text.

### Text encoders: CLIP or a language model

**Plain version.** The text encoder decides what the generator can "understand". A model trained to match images and captions knows visual concepts; a large language model trained on text alone knows grammar, word order and spelling-related structure better. Many systems use both.

Two kinds of encoder dominate:

- **CLIP text encoders** are trained contrastively so that an image and its caption land near each other in a shared embedding space [23] (see ML 10). Their embeddings are visually grounded.
- **Language-model encoders** such as T5 [24] are trained on text only. Imagen's key finding was that a large **frozen** T5 encoder works surprisingly well, and that making the text encoder larger improved both fidelity and image–text alignment much more than making the image diffusion model larger [4].

Recent systems combine them. SDXL adds a second text encoder to the design of earlier Stable Diffusion models [25]. Stable Diffusion 3 uses three frozen encoders, two CLIP models and T5-XXL; its authors report that dropping T5 at inference time costs little on most prompts but clearly hurts prompts with long written text or many details [20]. A related finding concerns spelling: text encoders that see whole word pieces rather than characters ("character-blind") make it hard to draw a word letter by letter, and character-aware encoders render visual text much more accurately [26]. DALL·E 2 takes yet another route: a "prior" network first generates a CLIP **image** embedding from the caption, and a diffusion decoder then generates the image from that embedding [27].

### Cross-attention: how words reach pixels

**Plain version.** At every layer, each position of the image asks a question ("what should be here?") and looks over all the words of the prompt to find the ones that answer it. The answer it collects is a weighted mix of the words, with large weights on the words that match.

Latent diffusion introduced cross-attention layers into the U-Net denoiser for this purpose [5]. Let $\Phi \in \mathbb{R}^{N \times d_\phi}$ be the image features at one layer ($N$ spatial positions, each a vector of width $d_\phi$), and $E \in \mathbb{R}^{L \times d_e}$ the text-encoder outputs ($L$ tokens of width $d_e$). Learned matrices $W_Q \in \mathbb{R}^{d_\phi \times d}$, $W_K, W_V \in \mathbb{R}^{d_e \times d}$ project them to a common width $d$:

$$Q = \Phi W_Q, \qquad K = E W_K, \qquad V = E W_V, \qquad A = \mathrm{softmax}\Big(\frac{QK^\top}{\sqrt{d}}\Big), \qquad \mathrm{out} = AV.$$

Here $Q \in \mathbb{R}^{N \times d}$ holds one **query** per image position, $K, V \in \mathbb{R}^{L \times d}$ hold one **key** and one **value** per text token, the softmax is taken along each row, and $A \in \mathbb{R}^{N \times L}$ is the **attention map**: row $i$ is a probability distribution over the $L$ tokens that says how much position $i$ listens to each word. The output $\mathrm{out} \in \mathbb{R}^{N \times d}$ gives every image position a mix of word values; it is added back to the image features. Queries come from the image and keys and values come from the text, which is exactly what makes it *cross*-attention rather than self-attention (compare NLP 07). Reading column $j$ of $A$ as an image gives a heat map of **where word $j$ is being drawn**, a fact that Prompt-to-Prompt turned into an editing method [8].

Transformer backbones offer an alternative. Stable Diffusion 3's MM-DiT concatenates image tokens and text tokens into one sequence and lets them attend to each other in both directions, with separate weights for the two modalities [20]. The text then also "reads" the image, not only the other way round.

The toy below builds a $16 \times 16$ grid of image features in which a disk region resembles the words "red" and "ball" and the background resembles "grass", then runs one cross-attention step. The projection matrices are identities to keep it readable; in a trained model they are learned.

```
import numpy as np

rng = np.random.default_rng(0)
d = 32                                          # feature width
words = ["a", "red", "ball", "on", "grass"]  # the prompt, one text token per word
E = rng.normal(size=(len(words), d))         # text-encoder outputs, one row per word

# A 16 x 16 grid of image tokens: a disk ("the ball") on a background ("the grass").
yy, xx = np.mgrid[:16, :16]
ball = ((yy - 6) ** 2 + (xx - 9) ** 2 <= 9).ravel()   # 29 of the 256 positions
X = 1.0 * rng.normal(size=(256, d))                   # noisy image features
X[ball] += 0.5 * (E[1] + E[2])                        # ball region looks like "red ball"
X[~ball] += 0.5 * E[4]                                # background looks like "grass"

# Cross-attention: queries from the image, keys and values from the text.
# Real models learn W_q, W_k, W_v; identity matrices keep the toy readable.
Q, K, V = X, E, E
scores = Q @ K.T / np.sqrt(d)                         # 256 image tokens x 5 words
A = np.exp(scores - scores.max(axis=1, keepdims=True))
A /= A.sum(axis=1, keepdims=True)                     # softmax over the words
out = A @ V                                           # each image token gets a mix of word
 ↪  values

print("each row sums to 1:", np.allclose(A.sum(axis=1), 1), "| output shape:", out.shape)
for j, w in enumerate(words):
    print(f"{w:6s} mean weight inside the ball={A[ball, j].mean():.2f}  "
          f"outside={A[~ball, j].mean():.2f}")
```

It prints:

```
each row sums to 1: True | output shape: (256, 32)
a      mean weight inside the ball=0.02  outside=0.06
red    mean weight inside the ball=0.43  outside=0.07
ball   mean weight inside the ball=0.52  outside=0.04
on     mean weight inside the ball=0.02  outside=0.08
grass  mean weight inside the ball=0.01  outside=0.74
```

(These numbers come from our toy run, not from any trained model.) Inside the disk, almost all the attention goes to "red" and "ball"; outside, three quarters of it goes to "grass". The function words "a" and "on" get little weight anywhere. Figure 3 shows the five maps. Notice what the mechanism does **not** guarantee: "red" and "ball" share the same region because the features say so, and nothing in the attention formula forces a color word to attach to the right object. When two objects are present, a color can attend to the wrong one, which is one root of the attribute-binding failures discussed later [28].

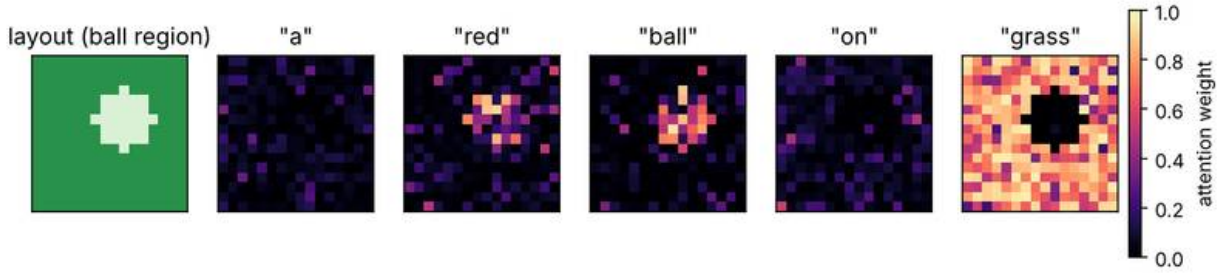



*Figure 3 — Cross-attention maps in the toy. Each map is one column of the attention matrix A reshaped to the image grid: where each word is being 'drawn'. Prompt-to-Prompt edits images by reusing such maps [8].*

### Guidance as used in practice

Classifier-free guidance (CFG) is derived and tested in ML 08; here is the reminder. During training the caption is dropped (replaced by an empty prompt) for a random fraction of examples, so one network learns both a conditional and an unconditional prediction [11]. At sampling time the two are combined, $\hat{\epsilon} = \epsilon_\emptyset + w(\epsilon_c - \epsilon_\emptyset)$, where $\epsilon_c$ and $\epsilon_\emptyset$ are the predictions with and without the prompt and $w$ is the guidance scale; $w > 1$ moves past the conditional prediction toward the prompt.

For a system builder, four practical facts matter:

- **It is a dial, not a measurement.** Higher $w$ usually gives more prompt-faithful, higher-contrast images with less variety; very high $w$ gives over-saturated, distorted images. Report the value and the convention (some code writes $1 + w$).
- **It doubles the work per step** unless the two predictions are batched, so a 30-step guided sampler costs about 60 network evaluations.
- **It also works for token models.** Autoregressive and masked token generators apply the same formula to next-token logits (ML 09).
- **Distilled models may bake it in.** Some distilled checkpoints are trained to reproduce guided outputs, so the guidance scale becomes an input or disappears; read the release documentation before comparing settings (ML 08).

### Backbones and some landmark systems

The denoiser was first a **U-Net** [29], a convolutional encoder–decoder with skip connections, with cross-attention layers inserted at several resolutions [5]. The **diffusion transformer** (**DiT**) replaces it with a Vision Transformer over latent patches and showed steady gains as transformer compute grew [3] (details in ML 08). Large recent systems mostly use transformers. The table lists a few systems whose design is documented in papers; it is a map of design choices, not a ranking.

| System | Where it generates | Generator and backbone | Text conditioning |
|---|---|---|---|
| Latent diffusion / Stable Diffusion [5] | autoencoder latent | diffusion, U-Net | cross-attention to a text encoder |
| DALL·E 2 [27] | pixels | diffusion decoder from a CLIP image embedding | a prior maps the caption to a CLIP image embedding |
| Imagen [4] | pixels, 64 → 256 → 1024 cascade | diffusion | frozen T5-XXL |
| Parti [30] | discrete tokens (ViT-VQGAN) | autoregressive encoder–decoder transformer | the text is the encoder's input |
| Muse [31] | discrete tokens | masked parallel decoding, transformer | embeddings from a pretrained language model |
| SDXL [25] | autoencoder latent | diffusion, larger U-Net, optional refiner | two text encoders |
| Stable Diffusion 3 [20] | autoencoder latent | rectified flow, MM-DiT transformer | two CLIP encoders and T5-XXL |

The resolutions of the Imagen cascade are from the project page listed with its reference.

*Online only: Esser et al., "Scaling Rectified Flow Transformers for High-Resolution Image Synthesis" (ICML 2024) — https://blog.shyandram.dev/en/learn/deep-learning/image-generation/papers/esser-et-al-scaling-rectified-flow-transformers-for-high-resolution-image-synthesis-icml-2024/*

## Control, personalization and editing

**Plain version.** A prompt is a weak steering wheel: words cannot pin down an exact pose, a particular person's dog, or "change only the hat". Three groups of methods add stronger steering: extra **control images** (edges, depth, pose), small **personalization** updates that teach the model a new concept, and **editing** procedures that change a given image while keeping the rest.

### Spatial control: ControlNet

ControlNet adds a spatial condition (an edge map, a depth map, a segmentation, a human pose) to a large pretrained text-to-image model without damaging it [7]. It **locks** the original network, makes a **trainable copy** of its encoding layers that receives the condition, and connects the copy back to the original through **zero convolutions**: $1 \times 1$ convolutions whose weights and biases start at zero. At the first training step the added branch outputs exactly zero, so the model behaves exactly like the original and learning starts from a working generator. The paper reports robust training with both small (under 50k) and large (over 1M) datasets [7].

*Online only: ControlNet's zero convolutions: why training starts from the original model — https://blog.shyandram.dev/en/learn/deep-learning/image-generation/notes/controlnets-zero-convolutions-why-training-starts-from-the-original-model/*

### Personalization: teaching the model a new concept

Three methods trade storage, speed and fidelity differently:

- **Textual inversion** keeps the whole model frozen and learns only a new word embedding (a new "pseudo-word") from 3–5 images of a concept, by minimizing the usual denoising loss on those images [32]. The stored result is a single vector, but its expressiveness is limited by what the frozen model can already draw.
- **DreamBooth** fine-tunes the whole model on a few images of a subject captioned with a rare identifier plus the class name, such as "a [V] dog", and adds a **class-specific prior-preservation loss** that keeps the model able to draw ordinary dogs; without it, the model drifts and loses diversity [33].
- **LoRA** (low-rank adaptation) freezes each weight matrix $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and learns an additive update $\Delta W = BA$ with $B \in \mathbb{R}^{d_{\text{out}} \times r}$, $A \in \mathbb{R}^{r \times d_{\text{in}}}$ and a small rank $r$ [34]. It was introduced for language models; the same update can be attached to the attention layers of a text-to-image denoiser, which makes a personalized style or subject a small file that can be switched on and off (ML 10, PEFT).

*Online only: Personalization objectives side by side — https://blog.shyandram.dev/en/learn/deep-learning/image-generation/notes/personalization-objectives-side-by-side/*

### Editing and inpainting

- **SDEdit** adds noise to an input image (for example a rough color sketch) up to an intermediate noise level, then runs the ordinary reverse process [6]. A low starting noise level keeps the input's layout and colors (faithful); a high one gives the model more freedom (realistic). One number sets the balance.
- **Prompt-to-Prompt** edits a generated image by changing the prompt ("a cat on a bike" → "a dog on a bike") while re-using the original cross-attention maps during sampling, so the layout stays the same and only the changed word's content changes [8]. Editing a *real* photo this way first requires finding a noise that reproduces it, for example with deterministic DDIM inversion (ML 08).
- **InstructPix2Pix** follows written instructions ("make it look like winter"). Its training data was generated automatically by combining a language model (GPT-3) and a text-to-image model (Stable Diffusion) into a large set of before-and-after editing examples; a conditional diffusion model trained on these pairs then edits in a single sampling run, without per-image fine-tuning or inversion [9].
- **RePaint** inpaints with an unconditional diffusion model: at every reverse step it replaces the known region with a correctly noised copy of the input and lets the model generate only the masked region [10].

A survey by Huang et al. organizes the many diffusion-based editing methods by learning strategy, input condition and task [35].

### Fast sampling, briefly

A text-to-image model is used interactively, so the number of network calls matters. The routes are covered in ML 08, few-step generation: deterministic samplers such as DDIM need fewer steps with the same network [36]; progressive distillation halves the step count repeatedly [37]; consistency models learn to jump from any point of a trajectory to its end [38]; and adversarial diffusion distillation (ADD) combines a diffusion teacher with an adversarial loss to sample a large model in 1–4 steps [22]. When you compare systems, count every network evaluation (both guidance passes included) plus the decoder.

## Autoregressive and unified multimodal generators

**Plain version.** An autoregressive generator writes an image the way a language model writes a sentence: one token, or one group of tokens, at a time, each conditioned on what came before. The newest step is to use **one** transformer for both text and images, so the same model can describe a picture, answer questions about it and draw a new one.

### The autoregressive line in one paragraph

The theory is in ML 09; here is the summary. A tokenizer (VQ-VAE or VQGAN, ML 09) turns the image into a grid of integers, and a transformer models them with the chain rule, trained by next-token cross-entropy with teacher forcing (ML 09). Raster-order models such as Parti [30] and LlamaGen [39] predict one token per step; masked models such as MaskGIT [40] and Muse [31] predict many tokens in parallel and refine them over a few rounds (ML 09); VAR predicts a whole, sharper token map per step, coarse to fine [41] (ML 09); and MAR drops the discrete codebook and models each continuous token with a small diffusion head [21] (ML 09). Sampling cost, KV caching and the comparison with diffusion are in ML 09.

### Autoregressive text-to-image in practice

From the system side, four facts are worth knowing.

- **The text enters as context, not through cross-attention to a separate denoiser.** Parti treats text-to-image as translation: a transformer encoder reads the prompt and a decoder writes image tokens; the paper scales this encoder–decoder model to 20 billion parameters and introduces the PartiPrompts benchmark of over 1,600 English prompts [30]. Muse instead feeds a text embedding from a pretrained large language model to a masked token transformer, and it supports inpainting, outpainting and mask-free editing without fine-tuning [31].
- **Language-model infrastructure transfers.** LlamaGen uses a plain Llama-style decoder; besides class-conditional models from 111M to 3.1B parameters it trains a 775M text-conditional model, and it reports that an LLM serving framework sped up image generation by 326%–414% [39].

- **The tokenizer is decisive.** MAGVIT-v2 argues that the tokenizer is the key component: with its new "lookup-free" quantizer, which lets the vocabulary grow large, a language model outperformed diffusion models on ImageNet when given the same training data, a similar model size and a similar training budget [42]. Small details such as text and thin lines are typical casualties of a coarse tokenizer (ML 09).
- **Guidance and sampling knobs carry over.** CFG on logits, temperature and top-$k$ sampling set the fidelity–diversity balance, and every guided step costs two forward passes.

### Unified multimodal models

**Plain version.** Instead of a text model that hands a prompt to a separate image model, a unified model puts text and image pieces into **one sequence** and trains one transformer on it. The design questions are: are image pieces discrete tokens or continuous vectors, is the image part generated by next-token prediction or by diffusion, and do understanding and generation share one image encoder?

The models below are documented in papers and illustrate the main design choices:

- **Chameleon** puts discrete image tokens and text tokens in one sequence from the start ("early fusion") and handles captioning, question answering and the generation of text, images and mixed documents with one model [43].
- **Emu3** trains one transformer from scratch with next-token prediction **only**, over discrete tokens of images, text and video [44].
- **Transfusion** keeps images continuous and trains one transformer with a next-token loss on text and a diffusion loss on image patches; its authors report that this scales better than quantizing images into tokens [45].
- **Show-o** combines autoregressive text modeling with discrete (masked) diffusion over image tokens [46].
- **Janus** shares one transformer but uses **separate** visual encoders for understanding and for generation, because the two tasks need different levels of image detail [47].

| Model | Image representation | Image objective | Visual encoders |
|---|---|---|---|
| Chameleon [43] | discrete tokens | next token | one, shared |
| Emu3 [44] | discrete tokens | next token | one, shared |
| Transfusion [45] | continuous patches | diffusion | one, shared |
| Show-o [46] | discrete tokens | masked (discrete) diffusion | one, shared |
| Janus [47] | discrete tokens for generation | next token | separate for understanding and generation |

*Online only: Unified multimodal models, one by one — https://blog.shyandram.dev/en/learn/deep-learning/image-generation/notes/unified-multimodal-models-one-by-one/*

What is still open: whether one model can be state-of-the-art at both understanding and generation at the same size, how much a shared visual tokenizer costs each task (the motivation for Janus), and how evaluation should compare a unified model with a specialized diffusion system.

Claims about scaling in these papers are each paper's own measurements, on its own data; they are not yet a settled law.

## Evaluating text-to-image models

**Plain version.** You need four kinds of evidence: do the images look real and varied (distribution metrics), do they match their prompts overall (alignment scores), do they get the details right (compositional checks), and do people like them (human preference). Each kind catches failures the others miss.

### Distribution metrics and their pitfalls

FID compares feature statistics of generated and real images; precision and recall separate fidelity from coverage. Both are defined and implemented in ML 07, and ML 08 covers the newer feature spaces. For text-to-image models the common protocol is a **zero-shot COCO FID**: generate images from COCO captions [48] with a model that was not trained on COCO and compare with the real COCO images, as Imagen did [4]. Five pitfalls matter in practice:

- **Finite-sample bias.** FID is biased for finite sample sets, and the bias depends on the model being evaluated, so even equal sample counts do not make a comparison fully fair; Chong and Forsyth propose extrapolating to an infinite sample count [49].
- **Preprocessing.** The resizing filter and JPEG compression used before feature extraction change FID substantially; Parmar et al. show this and provide a reference implementation [50].
- **What the features see.** Inception features are so close to ImageNet classes that matching the class histogram of generated images can lower FID without making images better [51].
- **Agreement with people.** In a large human study no common metric correlated strongly with human judgments, and FID under-rated diffusion models; the authors recommend richer feature extractors such as DINOv2 [52]. CMMD replaces Inception features and the Gaussian assumption with CLIP embeddings and a kernel distance [53].
- **No alignment.** FID does not look at the prompt at all: a model that ignores every prompt but draws beautiful COCO-like images can score well.

### Prompt alignment: CLIPScore

CLIPScore measures how well an image and a text agree in CLIP's embedding space [54]. With $\mathbf{c}$ the CLIP embedding of the text and $\mathbf{v}$ the CLIP embedding of the image,

$$\text{CLIP-S}(\mathbf{c}, \mathbf{v}) = w \cdot \max\left(\cos(\mathbf{c}, \mathbf{v}), 0\right),$$

where cos is the cosine similarity and $w$ is a constant that only rescales the range ($w = 2.5$ in the original paper, which proposed the score for image captioning). For text-to-image evaluation the score is averaged over prompt–image pairs. It is cheap and needs no reference image, but it inherits its CLIP's blind spots: an embedding that summarizes the whole image is weak at counts, spatial

relations and which attribute belongs to which object. The GenEval study found that its object-level checks agree with human judgments much better than CLIPScore on such compositional tasks [55].

### Compositional benchmarks

**Plain version.** Instead of one overall similarity, ask concrete yes-or-no questions about the image: is there a dog, are there exactly three, is the dog left of the cat, is the car red? A detector or a vision–language model answers them.

**GenEval** generates prompts from templates for six tasks (single object, two objects, counting, colors, position and attribute binding), detects the objects with an instance-segmentation model, classifies object colors with CLIP, and scores each image as correct or not [55]. **T2I-CompBench** provides 6,000 compositional prompts in three categories (attribute binding, object relationships and complex compositions) with metrics tailored to each [56]. Imagen's **DrawBench** [4] and Parti's **PartiPrompts** [30] are prompt sets for side-by-side human comparison. Object detection itself is the subject of DLCV 03; an open-vocabulary detector (DLCV 03) can check objects outside a fixed class list.

*Online only: Ghosh et al., "GenEval: An Object-Focused Framework for Evaluating Text-to-Image Alignment" (NeurIPS 2023) — https://blog.shyandram.dev/en/learn/deep-learning/image-generation/papers/ghosh-et-al-geneval-an-object-focused-framework-for-evaluating-text-to-image-alignment-neurips-2023/*

The toy below shows why object-level checks matter. It draws synthetic "generated images" made of colored disks and squares for the prompt "two red circles and a blue square", detects the shapes with connected components, and compares two scores: a **presence score** that asks which prompt words appear anywhere in the image (a deliberately simple stand-in for a global similarity score, not CLIP itself) and an **exact check** that requires every (color, shape) pair with the right count, as GenEval does.

```
import numpy as np
from skimage.draw import disk, rectangle
from skimage.measure import label, regionprops

COLORS = {"red": (0.85, 0.1, 0.1), "blue": (0.1, 0.2, 0.85)}

def render(objects, size=96):
    """Draw (color, shape) objects at fixed slots on a gray canvas: a stand-in for a
     ↪  generated image."""
    img = np.full((size, size, 3), 0.8)
    for k, (col, shape) in enumerate(objects):
        cy, cx = 24 + 48 * (k // 2), 24 + 48 * (k % 2)
        if shape == "circle":
            rr, cc = disk((cy, cx), 14)
        else:
            rr, cc = rectangle((cy - 12, cx - 12), extent=(24, 24))
        img[rr, cc] = COLORS[col]
    return img
```

```
def detect(img):
    """Toy 'detector': connected colored regions -> (color, shape) pairs."""
    found = []
    for region in regionprops(label(np.abs(img - 0.8).sum(-1) > 0.3)):
        r, g, b = img[tuple(region.coords.T)].mean(0)
        col = "red" if r > b else "blue"
        shape = "square" if region.extent > 0.9 else "circle"   # filled bbox fraction
        found.append((col, shape))
    return found

def presence_score(prompt, img):
    """Bag-of-words check: which prompt words appear anywhere (ignores counts and
     ↪  binding)."""
    words = {w for obj in prompt for w in obj}
    seen = {w for obj in detect(img) for w in obj}
    return len(words & seen) / len(words)

def exact_check(prompt, img):
    """GenEval-style check: every (color, shape) pair with the right count."""
    return sorted(detect(img)) == sorted(prompt)

prompt = [("red", "circle"), ("red", "circle"), ("blue", "square")]   # "two red circles and
 ↪  a blue square"
candidates = {
    "correct":          prompt,
    "wrong count":      prompt + [("red", "circle")],
    "colors swapped":   [("blue", "circle"), ("blue", "circle"), ("red", "square")],
    "object missing":   prompt[:2],
}
for name, objs in candidates.items():
    img = render(objs)
    print(f"{name:15s} presence score={presence_score(prompt, img):.2f}  "
          f"exact check={'pass' if exact_check(prompt, img) else 'fail'}")
```

It prints:

```
correct         presence score=1.00  exact check=pass
wrong count     presence score=1.00  exact check=fail
colors swapped  presence score=1.00  exact check=fail
object missing  presence score=0.50  exact check=fail
```

The presence score gives full marks to the wrong count and to the swapped colors, because every word of the prompt is present *somewhere*; only the exact check catches them. Real benchmarks replace our connected-component "detector" with a trained detector and a color classifier, and inherit their errors: the check is only as good as the detector, which is why GenEval reports its agreement with human annotators [55]. Figure 4 shows the four images.

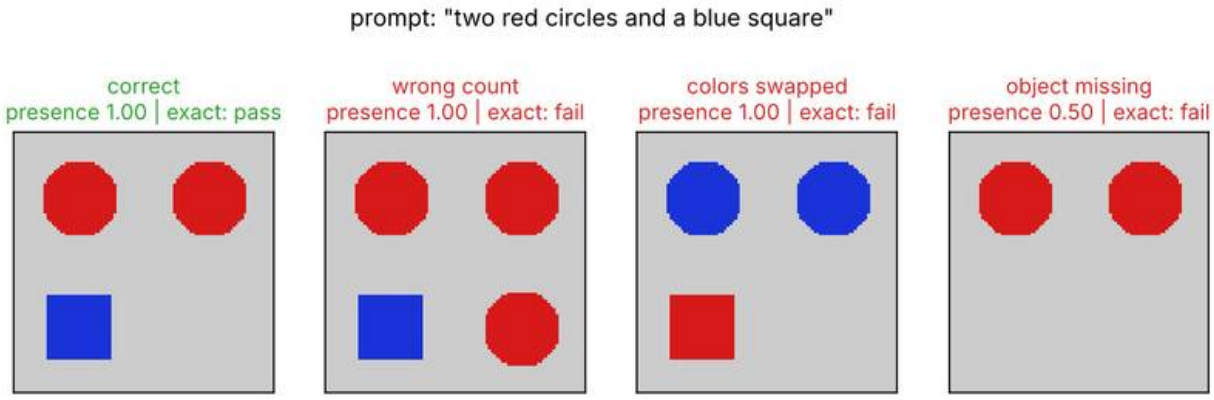



*Figure 4 — A word-presence score cannot see counting or binding errors; an object-level check can. Synthetic shapes stand in for generated images.*

### Human preference and holistic evaluation

People remain the reference. Because human studies are slow, several groups trained **preference models** that predict which of two images people prefer for a prompt: ImageReward was trained on 137k expert comparisons [57], and PickScore on Pick-a-Pic, an open dataset of real users' choices collected through a web app [58]. **HEIM** evaluates models along 12 aspects (including alignment, quality, aesthetics, originality, reasoning, knowledge, bias, toxicity, fairness, robustness, multilinguality and efficiency) over 62 scenarios and 26 models, and finds that no single model is best at everything [59]. A learned preference model is itself a model with biases; if a generator is tuned to maximize it, the score can rise faster than real quality.

**A reporting checklist.** For any claim about a text-to-image model, state: the prompts (and how they were sampled), the number of images per prompt and seeds, the sampler, the number of steps and guidance scale, the resolution, the metric implementations and feature extractors, and show uncurated samples. Report distribution, alignment, compositional and preference results side by side, and say which numbers you ran yourself and which were copied from papers.

## Data

**Plain version.** A generator can only learn what its training images and captions show, so the dataset decides its knowledge, its style and many of its biases. Large text-to-image datasets are scraped from the web, which raises questions of quality, copyright and consent.

- **Class-conditional benchmarks** use ImageNet [60]; text-to-image papers often evaluate zero-shot on COCO captions [48]; face generators use datasets such as FFHQ, the high-quality face dataset introduced with StyleGAN [1].
- **LAION-5B** is an openly released dataset of 5.85 billion CLIP-filtered image–text pairs, of which 2.32 billion have English text [61]. It distributes URLs and alt-text rather than the images them-

selves, and its datasheet states that LAION releases the metadata under CC-BY-4.0 but does not own the copyright of the images or text, that people depicted in the images may not have consented to their upload, and that the dataset should be used for academic research in its current form; it describes ways to request removal of links [61].
- **Content audits.** Birhane et al. examined the earlier LAION-400M dataset and found explicit and problematic image–text pairs, including pornography, malign stereotypes and slurs [62].
- **Captions are data too.** Stable Diffusion 3 trains on a 50/50 mix of original captions and synthetic captions written by a vision–language model, and filters its data for sexual content, low aesthetic scores and near-duplicates (to reduce regurgitation of training images) [20].

## Risks and responsible use

**Plain version.** A model that can draw anything can also copy, stereotype and deceive. The practical responses are to measure these behaviors, to reduce them in the data and the model, and to label generated images so that viewers can check where they came from.

### Memorization and copying

Carlini et al. showed that diffusion models memorize individual training images and can emit them: with a generate-and-filter pipeline they extracted over a thousand training examples, from photos of individual people to trademarked logos, and they found diffusion models less private than GANs in their experiments [63]. Somepalli et al. built image-retrieval tools to compare generated images with training images and found cases where models, including Stable Diffusion, copied content from their training data [64]. Deduplicating training data [20] and checking generated images against the training set are the standard first defenses.

### Bias

Bianchi et al. found that ordinary prompts (traits, occupations, objects) produce and amplify demographic stereotypes, and that neither counter-stereotypical prompts nor system guardrails removed them in their tests [65]. Luccioni et al. proposed a way to measure social bias in generated people by comparing the variation caused by gender and ethnicity words with the variation caused by professions, and found under-representation of marginalized identities in the three systems they tested [66]. Bias evaluation needs its own prompts and analysis; overall quality metrics do not reveal it. See also ML 10, data provenance, bias and privacy.

### Deepfakes, provenance and watermarks

Two complementary tools exist for labeling generated images:

- **Signed provenance (C2PA).** The C2PA specification defines a "manifest" of provenance claims (who made the asset and how) that is cryptographically signed and bound to the asset, either

by a cryptographic hash (a hard binding) or by a fingerprint or invisible watermark (a soft binding) [67]. A valid signature shows that the claims were not altered; it does not prove that the content is true, and a manifest stored in metadata can simply be removed.
- **Invisible watermarks.** Tree-Ring watermarking hides a pattern in the Fourier transform of the **initial noise** of a diffusion sampler and detects it by inverting the sampler [68]. Stable Signature fine-tunes the latent decoder so that every generated image carries a binary signature that a pretrained extractor can read [69]. Classical watermarking is covered in IP 08.

Neither is a complete answer. Zhao et al. prove and show that pixel-level invisible watermarks can be removed by a "regeneration attack" (add noise, then reconstruct with a generative model) [70], and Saberi et al. show a fundamental trade-off for low-perturbation watermarks between failing to detect watermarked images and falsely flagging real ones, and demonstrate spoofing attacks that make real images look watermarked [71]. Treat provenance and watermarks as evidence to weigh, not as proof.

*Online only: Hard bindings, soft bindings and watermarks — https://blog.shyandram.dev/en/learn/deep-learning/image-generation/notes/hard-bindings-soft-bindings-and-watermarks/*

## Failure modes

**Plain version.** Today's generators are excellent at "a photo of X" and still unreliable at the precise parts of a request: how many, which one has which color, where things are, what the text says. Knowing the typical failures tells you what to test.

- **Counting.** "Four apples" often gives three or five. GenEval's counting task (2, 3 or 4 objects) measures it directly [55].
- **Spatial relations.** "A cup to the left of a laptop" is often drawn in the wrong arrangement; in GenEval's original study the best model produced the correct position in only 15% of images [55].
- **Attribute binding and leakage.** In "a red car and a blue bench" the colors can swap between objects or leak onto the background; GenEval's best model at the time got attribute binding right in 35% of images [55]. Attend-and-Excite also documents **catastrophic neglect**, where one of the subjects in the prompt is not drawn at all, and steers the cross-attention maps at sampling time so that every subject token gets attention [28].
- **Text rendering.** Misspelled signs and labels are common; character-blind text encoders are one documented cause [26], and Stable Diffusion 3's ablation shows that the T5 encoder matters most for prompts with written text [20].
- **Hands.** Generated hands often have the wrong number of fingers or irregular shapes; HandRefiner repairs them afterwards by inpainting conditioned on a reconstructed hand mesh [72]. Note that object-level benchmarks cannot check this: GenEval's authors point out that their detector can count people but not fingers [55].
- **Copying of content and style.** A prompt that names a famous image, product or artist can pull out a near-copy of training content [64], [63].

**How to test.** Build a small prompt set per failure type, generate several images per prompt with

fixed seeds, check them with an object-level tool where possible and by eye where not, and track the failure rate across model versions.

## Modern view

**Surveys worth reading.**

- **Zhang et al. [73]** (2023) is a self-contained review of text-to-image diffusion models: a short introduction to diffusion, the background of text-conditioned synthesis, the pioneering systems and their improvements, applications beyond images (video, other modalities, text-guided editing) and open challenges.
- **Croitoru et al. [74]** (IEEE TPAMI 2023) survey diffusion models in vision, grouped by formulation (DDPM, score networks, SDEs) and by application (generation, super-resolution, inpainting, editing, translation, segmentation).
- **Huang et al. [35]** (IEEE TPAMI 2025) classify diffusion-based image editing by learning strategy, user-input condition and editing task, give special attention to inpainting and outpainting, and propose a benchmark (EditEval) with a metric scored by a large multimodal model.
- **Xiong et al. [75]** (TMLR 2025) survey autoregressive vision models, split by representation into pixel-based, token-based and scale-based models, and discuss their links to other generative families and their uses in image, video, 3-D and multimodal generation.

**State of the art, 2023–2026.** (1) **Latent transformers with flow objectives and several text encoders** became the standard large-scale recipe, as in Stable Diffusion 3 [20], building on DiT [3]. (2) **Autoregressive models returned** through better tokenizers [42], next-scale prediction [41] and continuous tokens [21]. (3) **Unified multimodal models** that understand and generate with one transformer appeared in quick succession [43], [45], [46], [47], [44]. (4) **Few-step generation** made interactive use practical [22], [38]. (5) **Evaluation moved beyond FID** toward compositional checks [55], [56], learned preference models [57], [58] and holistic benchmarks [59]. Recent work also revisits pixel-space generation and generation in representation spaces; see ML 08, pixel space, latent space and representation autoencoders.

**Open problems.** Reliable counting, spatial reasoning and attribute binding; legible text in any language; evaluation metrics that agree with people at the level of single images and cannot be gamed; preventing memorization without losing quality; measuring and reducing bias; provenance that survives editing; consent and licensing for training data; and the cost of high-resolution and video generation.

*Online only: Latent Diffusion: compress perception before learning the image distribution — https://blog.shyandram.dev/en/learn/deep-learning/image-generation/papers/latent-diffusion-compress-perception-before-learning-the-image-distribution/; Esser et al., "Scaling Rectified Flow Transformers for High-Resolution Image Synthesis" (ICML 2024) — https://blog.shyandram.dev/en/learn/deep-learning/image-generation/papers/esser-et-al-scaling-rectified-flow-transformers-for-high-resolution-image-synthesis-icml-2024/*

## Key takeaways

- Define the condition and output constraints before choosing a generator.
- Generator objectives and image representations are separate choices; systems can combine families.
- Inspect how text or spatial conditions enter the model, and what guidance changes at inference.
- Attention maps show internal interactions, not a guarantee of correct object–attribute binding.
- Control, personalization and editing impose different data and compute requirements.
- Combine distribution, alignment, composition and human-preference evaluations; report the sampling protocol and cost.
- Inspect memorization, bias and provenance separately from visual quality.

## Exercises

1. **Map the tasks.** For each task in the table of six kinds of generation, write down what the condition $c$ is as a tensor (shape and type) for a latent text-to-image model, and which part of the pipeline in Figure 2 receives it.

   **Hint**

   The prompt becomes $L$ token embeddings that enter through cross-attention. A class label can be written as a text prompt or as an embedding added to the time embedding. For image-to-image and editing, the input image is encoded to a latent and either noised to an intermediate level (SDEdit) or concatenated to the denoiser's input channels (InstructPix2Pix); a control image enters through ControlNet's trainable copy; a mask for inpainting decides which latent positions are regenerated.

2. **Neglect in the toy.** In the cross-attention snippet, add a sixth word "dog" whose embedding is a fresh random vector that no image region resembles. What does its attention map look like, and what does it do to the weights of the other words? Then make the ball region resemble only "ball" (not "red"). Which map changes?

   **Hint**

   The "dog" column gets small, unstructured weights everywhere, because no query matches its key: nothing tells the generator where a dog should go, which is the situation Attend-and-Excite fixes by strengthening a neglected token's attention during sampling. The softmax still sums to one per row, so the other words lose a little weight. Removing "red" from the ball features makes the "red" map dark: a color word attends only where the features already suggest it.

3. **Break the metric.** Extend the compositional-check snippet with a candidate that has the right objects but in an image where the blue square overlaps one red circle. What does the toy detector report, and what would a real detector-based benchmark need to handle it?

   **Hint**

Overlapping shapes merge into one connected component, so the toy detector sees fewer objects and the wrong shape; the exact check fails even though a person might accept the image. Real benchmarks use instance segmentation, which can separate overlapping objects, and GenEval adds rules for overlapping boxes when judging positions. Every automatic check has such blind spots, which is why benchmarks report agreement with human annotators.

4. **Design an evaluation.** You must decide whether a new model is better than the current one at "product photos with a short brand name on the package". Write an evaluation plan: prompts, number of images, automatic metrics, human study, and what result would change your decision.

**Hint**

Use a fixed prompt set that varies product type, text length and background; several seeds per prompt; an OCR check of the rendered text against the requested string; an object-level check of the product type; FID or CMMD against real product photos for realism (with a fixed implementation and sample count); and a blind side-by-side human comparison on alignment and quality. Decide in advance which difference you would act on, and include a memorization check against the training set for brand logos.

5. **Provenance under editing.** An image carries a C2PA manifest with a hard binding and was generated by a model with a Stable Signature watermark. It is then cropped, re-saved as JPEG and uploaded to a site that strips metadata. Which evidence of its origin is likely to survive, and why?

**Hint**

The manifest is lost with the metadata, and even if it were kept, the hash of the hard binding would no longer match the cropped, re-encoded pixels. A soft binding or a robust invisible watermark can survive moderate cropping and compression (Stable Signature reports detection after strong cropping), but a regeneration attack can remove pixel-level watermarks. The honest answer is "maybe the watermark, not the manifest", which is why the two are designed to work together.

## References

1. T. Karras, S. Laine and T. Aila, "A Style-Based Generator Architecture for Generative Adversarial Networks," *CVPR*, 2019. arXiv
2. A. Brock, J. Donahue and K. Simonyan, "Large Scale GAN Training for High Fidelity Natural Image Synthesis," *ICLR*, 2019. arXiv
3. W. Peebles and S. Xie, "Scalable Diffusion Models with Transformers," *ICCV*, 2023. CVF
4. C. Saharia, W. Chan, S. Saxena et al., "Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding," *NeurIPS*, 2022. arXiv ·project page
5. R. Rombach, A. Blattmann, D. Lorenz, P. Esser and B. Ommer, "High-Resolution Image Synthesis with Latent Diffusion Models," *CVPR*, 2022. CVF

6. C. Meng, Y. He, Y. Song et al., “SDEdit: Guided Image Synthesis and Editing with Stochastic Differential Equations,” *ICLR*, 2022. arXiv
7. L. Zhang, A. Rao and M. Agrawala, “Adding Conditional Control to Text-to-Image Diffusion Models,” *ICCV*, 2023. CVF
8. A. Hertz, R. Mokady, J. Tenenbaum et al., “Prompt-to-Prompt Image Editing with Cross-Attention Control,” *ICLR*, 2023. arXiv
9. T. Brooks, A. Holynski and A. A. Efros, “InstructPix2Pix: Learning to Follow Image Editing Instructions,” *CVPR*, 2023. CVF
10. A. Lugmayr, M. Danelljan, A. Romero et al., “RePaint: Inpainting using Denoising Diffusion Probabilistic Models,” *CVPR*, 2022. CVF
11. J. Ho and T. Salimans, “Classifier-Free Diffusion Guidance,” NeurIPS 2021 Workshop on Deep Generative Models and Downstream Applications; arXiv:2207.12598, 2022. arXiv
12. I. J. Goodfellow, J. Pouget-Abadie, M. Mirza et al., “Generative Adversarial Nets,” *NeurIPS*, 2014. arXiv
13. D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” *ICLR*, 2014. arXiv
14. A. van den Oord, O. Vinyals and K. Kavukcuoglu, “Neural Discrete Representation Learning,” *NeurIPS*, 2017. arXiv
15. P. Esser, R. Rombach and B. Ommer, “Taming Transformers for High-Resolution Image Synthesis,” *CVPR*, 2021. CVF
16. J. Ho, A. Jain and P. Abbeel, “Denoising Diffusion Probabilistic Models,” *NeurIPS*, 2020. arXiv
17. P. Dhariwal and A. Nichol, “Diffusion Models Beat GANs on Image Synthesis,” *NeurIPS*, 2021. arXiv
18. Y. Lipman, R. T. Q. Chen, H. Ben-Hamu, M. Nickel and M. Le, “Flow Matching for Generative Modeling,” *ICLR*, 2023. arXiv
19. X. Liu, C. Gong and Q. Liu, “Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow,” *ICLR*, 2023. arXiv
20. P. Esser, S. Kulal, A. Blattmann et al., “Scaling Rectified Flow Transformers for High-Resolution Image Synthesis,” *ICML*, PMLR 235, 2024. arXiv
21. T. Li, Y. Tian, H. Li, M. Deng and K. He, “Autoregressive Image Generation without Vector Quantization,” *NeurIPS*, 2024. arXiv
22. A. Sauer, D. Lorenz, A. Blattmann and R. Rombach, “Adversarial Diffusion Distillation,” *ECCV*, LNCS, pp. 87–103, 2024. doi ·arXiv
23. A. Radford, J. W. Kim, C. Hallacy et al., “Learning Transferable Visual Models From Natural Language Supervision,” *ICML*, PMLR 139, 2021. arXiv
24. C. Raffel, N. Shazeer, A. Roberts et al., “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer,” *Journal of Machine Learning Research*, vol. 21, 2020. arXiv
25. D. Podell, Z. English, K. Lacey et al., “SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis,” *ICLR*, 2024. arXiv
26. R. Liu, D. Garrette, C. Saharia et al., “Character-Aware Models Improve Visual Text Rendering,” *ACL*, 2023. ACL Anthology
27. A. Ramesh, P. Dhariwal, A. Nichol, C. Chu and M. Chen, “Hierarchical Text-Conditional Image Generation with CLIP Latents,” arXiv:2204.06125, 2022. arXiv
28. H. Chefer, Y. Alaluf, Y. Vinker, L. Wolf and D. Cohen-Or, “Attend-and-Excite: Attention-Based Semantic Guidance for Text-to-Image Diffusion Models,” *ACM Transactions on Graphics*, vol. 42,

no. 4 (SIGGRAPH), 2023. doi ·arXiv
29. O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," *MICCAI*, 2015. arXiv
30. J. Yu, Y. Xu, J. Y. Koh et al., "Scaling Autoregressive Models for Content-Rich Text-to-Image Generation," *Transactions on Machine Learning Research*, 2022. arXiv
31. H. Chang, H. Zhang, J. Barber et al., "Muse: Text-To-Image Generation via Masked Generative Transformers," *ICML*, PMLR 202, 2023. arXiv
32. R. Gal, Y. Alaluf, Y. Atzmon et al., "An Image is Worth One Word: Personalizing Text-to-Image Generation using Textual Inversion," *ICLR*, 2023. arXiv
33. N. Ruiz, Y. Li, V. Jampani, Y. Pritch, M. Rubinstein and K. Aberman, "DreamBooth: Fine Tuning Text-to-Image Diffusion Models for Subject-Driven Generation," *CVPR*, 2023. CVF
34. E. J. Hu, Y. Shen, P. Wallis et al., "LoRA: Low-Rank Adaptation of Large Language Models," *ICLR*, 2022. arXiv
35. Y. Huang, J. Huang, Y. Liu et al., "Diffusion Model-Based Image Editing: A Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2025. doi ·arXiv
36. J. Song, C. Meng and S. Ermon, "Denoising Diffusion Implicit Models," *ICLR*, 2021. arXiv
37. T. Salimans and J. Ho, "Progressive Distillation for Fast Sampling of Diffusion Models," *ICLR*, 2022. arXiv
38. Y. Song, P. Dhariwal, M. Chen and I. Sutskever, "Consistency Models," *ICML*, PMLR 202, 2023. arXiv
39. P. Sun, Y. Jiang, S. Chen et al., "Autoregressive Model Beats Diffusion: Llama for Scalable Image Generation," arXiv:2406.06525, 2024. arXiv
40. H. Chang, H. Zhang, L. Jiang, C. Liu and W. T. Freeman, "MaskGIT: Masked Generative Image Transformer," *CVPR*, 2022. CVF
41. K. Tian, Y. Jiang, Z. Yuan, B. Peng and L. Wang, "Visual Autoregressive Modeling: Scalable Image Generation via Next-Scale Prediction," *NeurIPS*, 2024. arXiv
42. L. Yu, J. Lezama, N. B. Gundavarapu et al., "Language Model Beats Diffusion — Tokenizer is Key to Visual Generation," *ICLR*, 2024. arXiv
43. Chameleon Team, "Chameleon: Mixed-Modal Early-Fusion Foundation Models," arXiv:2405.09818, 2024. arXiv
44. X. Wang, Y. Cui, J. Wang et al., "Multimodal learning with next-token prediction for large multimodal models," *Nature*, vol. 650, pp. 327–333, 2026 (preprint "Emu3: Next-Token Prediction is All You Need," 2024). doi ·arXiv
45. C. Zhou, L. Yu, A. Babu et al., "Transfusion: Predict the Next Token and Diffuse Images with One Multi-Modal Model," *ICLR*, 2025. arXiv
46. J. Xie, W. Mao, Z. Bai et al., "Show-o: One Single Transformer to Unify Multimodal Understanding and Generation," *ICLR*, 2025. arXiv
47. C. Wu, X. Chen, Z. Wu et al., "Janus: Decoupling Visual Encoding for Unified Multimodal Understanding and Generation," *CVPR*, 2025. CVF
48. T.-Y. Lin, M. Maire, S. Belongie et al., "Microsoft COCO: Common Objects in Context," *ECCV*, 2014. arXiv
49. M. J. Chong and D. Forsyth, "Effectively Unbiased FID and Inception Score and Where to Find Them," *CVPR*, 2020. arXiv
50. G. Parmar, R. Zhang and J.-Y. Zhu, "On Aliased Resizing and Surprising Subtleties in GAN

Evaluation," *CVPR*, 2022. CVF
51. T. Kynkäänniemi, T. Karras, M. Aittala, T. Aila and J. Lehtinen, "The Role of ImageNet Classes in Fréchet Inception Distance," *ICLR*, 2023. arXiv
52. G. Stein, J. C. Cresswell, R. Hosseinzadeh et al., "Exposing Flaws of Generative Model Evaluation Metrics and Their Unfair Treatment of Diffusion Models," *NeurIPS*, 2023. arXiv
53. S. Jayasumana, S. Ramalingam, A. Veit, D. Glasner, A. Chakrabarti and S. Kumar, "Rethinking FID: Towards a Better Evaluation Metric for Image Generation," *CVPR*, 2024. CVF
54. J. Hessel, A. Holtzman, M. Forbes, R. Le Bras and Y. Choi, "CLIPScore: A Reference-free Evaluation Metric for Image Captioning," *EMNLP*, 2021. ACL Anthology
55. D. Ghosh, H. Hajishirzi and L. Schmidt, "GenEval: An Object-Focused Framework for Evaluating Text-to-Image Alignment," *NeurIPS Datasets and Benchmarks Track*, 2023. arXiv
56. K. Huang, K. Sun, E. Xie, Z. Li and X. Liu, "T2I-CompBench: A Comprehensive Benchmark for Open-world Compositional Text-to-image Generation," *NeurIPS Datasets and Benchmarks Track*, 2023. proceedings
57. J. Xu, X. Liu, Y. Wu et al., "ImageReward: Learning and Evaluating Human Preferences for Text-to-Image Generation," *NeurIPS*, 2023. arXiv
58. Y. Kirstain, A. Polyak, U. Singer et al., "Pick-a-Pic: An Open Dataset of User Preferences for Text-to-Image Generation," *NeurIPS*, 2023. arXiv
59. T. Lee, M. Yasunaga, C. Meng et al., "Holistic Evaluation of Text-To-Image Models," *NeurIPS Datasets and Benchmarks Track*, 2023. arXiv
60. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," *CVPR*, 2009. doi
61. C. Schuhmann, R. Beaumont, R. Vencu et al., "LAION-5B: An Open Large-Scale Dataset for Training Next Generation Image-Text Models," *NeurIPS Datasets and Benchmarks Track*, 2022. arXiv
62. A. Birhane, V. U. Prabhu and E. Kahembwe, "Multimodal datasets: misogyny, pornography, and malignant stereotypes," arXiv:2110.01963, 2021. arXiv
63. N. Carlini, J. Hayes, M. Nasr et al., "Extracting Training Data from Diffusion Models," *USENIX Security Symposium*, 2023. USENIX ·arXiv
64. G. Somepalli, V. Singla, M. Goldblum, J. Geiping and T. Goldstein, "Diffusion Art or Digital Forgery? Investigating Data Replication in Diffusion Models," *CVPR*, 2023. CVF
65. F. Bianchi, P. Kalluri, E. Durmus et al., "Easily Accessible Text-to-Image Generation Amplifies Demographic Stereotypes at Large Scale," *ACM FAccT*, 2023. doi
66. A. S. Luccioni, C. Akiki, M. Mitchell and Y. Jernite, "Stable Bias: Analyzing Societal Representations in Diffusion Models," *NeurIPS Datasets and Benchmarks Track*, 2023. arXiv
67. Coalition for Content Provenance and Authenticity (C2PA), "C2PA Technical Specification," version 2.1, 2024. specification
68. Y. Wen, J. Kirchenbauer, J. Geiping and T. Goldstein, "Tree-Ring Watermarks: Fingerprints for Diffusion Images that are Invisible and Robust," *NeurIPS*, 2023. arXiv
69. P. Fernandez, G. Couairon, H. Jégou, M. Douze and T. Furon, "The Stable Signature: Rooting Watermarks in Latent Diffusion Models," *ICCV*, 2023. CVF
70. X. Zhao, K. Zhang, Z. Su et al., "Invisible Image Watermarks Are Provably Removable Using Generative AI," *NeurIPS*, 2024. arXiv
71. M. Saberi, V. S. Sadasivan, K. Rezaei et al., "Robustness of AI-Image Detectors: Fundamental

Limits and Practical Attacks," *ICLR*, 2024. arXiv
72. W. Lu, Y. Xu, J. Zhang, C. Wang and D. Tao, "HandRefiner: Refining Malformed Hands in Generated Images by Diffusion-based Conditional Inpainting," *ACM Multimedia*, 2024, pp. 7085–7093. doi
73. C. Zhang, C. Zhang, M. Zhang, I. S. Kweon and J. Kim, "Text-to-image Diffusion Models in Generative AI: A Survey," arXiv:2303.07909, 2023. arXiv
74. F.-A. Croitoru, V. Hondru, R. T. Ionescu and M. Shah, "Diffusion Models in Vision: A Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023. doi
75. J. Xiong, G. Liu, L. Huang et al., "Autoregressive Models in Vision: A Survey," *Transactions on Machine Learning Research*, 2025. arXiv

# DLCV 12A • Video Generation

How video generators are built and judged: video autoencoders and tokenizers, factorized and full spatiotemporal attention, text-to-video and image-to-video, camera and motion control, long videos, temporal consistency, autoregressive versus diffusion generation, FVD and benchmark suites, data, risks and failure modes.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/video-generation/

## What you'll learn

- Separate frame quality, motion quality and temporal consistency.
- Understand spatial/temporal compression and token-budget tradeoffs.
- Compare temporal extensions of image models with video-native attention.
- Trace text, image, camera and motion conditions through long-video generation.
- Evaluate joint-clip and streaming models with complementary metrics and human studies.

## The big picture

A video generator makes a short moving picture from a request: a sentence ("a dog runs along a beach at sunset"), a still image to bring to life, a camera path, or the first seconds of a clip to continue. The result must look good in every frame, the motion must be believable, and things must stay the same thing from one frame to the next.

> **Explain like I'm five**
>
> Think of a flipbook. If you draw every page separately from memory, the dog's spots move around, its ears change size and the whole book seems to shimmer when you flip it. A good flipbook artist first sketches the rough motion through all the pages at once, then adds the details so that each spot stays on the same part of the dog as it runs. A video generator does the same thing: it plans many frames together, in a compressed sketch space, so that the details move with the objects instead of jumping around.

**Why it matters.** Video generation is used in creative tools, for animating photos, for generating training data and simulation scenes, and as a candidate "world simulator" for robots and vehicles. It is also the most expensive kind of visual generation: a five-second clip contains dozens of images that must all agree. Most of the engineering is about one trade-off: modeling all frames jointly, for consistency, without paying for every pixel of every frame to interact with every other.

**Where this chapter sits.** The image side is covered elsewhere and is not repeated. DLCV 12 explains text encoders, cross-attention, guidance in practice, control and editing, text-to-image evaluation and risks. ML 08 derives diffusion, classifier-free guidance, latent diffusion, diffusion transformers, flow matching and few-step samplers, and its section Video adds temporal compression and a consistency problem is the short preview that this chapter expands. ML 09 covers autoregressive generation and visual tokenizers. DLCV 13 discusses action-conditioned

video prediction, large generative world models and whether realistic video implies physical understanding. DLCV 09A.3 introduces optical flow, warping and temporal-consistency measures. Here we focus on what is specific to **generating** video. All code runs in seconds on a CPU with NumPy, SciPy and scikit-image, without network weights.

### What makes video harder than images

**Plain version.** Video generation must produce good frames, plausible motion and consistent identities. Independent high-quality frames can still form a flickering video.

### The tasks

Every task below samples a video $x \in \mathbb{R}^{F \times H \times W \times 3}$ (with $F$ frames of height $H$ and width $W$, three color channels) from a conditional distribution $p(x \mid c)$; the tasks differ only in the condition $c$.

| Task | Condition $c$ | Examples in this chapter |
|---|---|---|
| Text-to-video (T2V) | a text prompt | Make-A-Video [1], CogVideoX [2], Wan [3] |
| Image-to-video (I2V) | a first frame, often plus a prompt | Stable Video Diffusion [4], DynamiCrafter [5] |
| Video extension or prediction | earlier frames of the same clip | VDM [6], Video LDM [7], StreamingT2V [8] |
| Frame interpolation | keyframes to fill in between | Make-A-Video [1], Stable Video Diffusion [4] |
| Video editing | an input video plus an edit request | Tune-A-Video [9], Text2Video-Zero [10] |
| Controlled generation | a camera path or object trajectories | MotionCtrl [11], CameraCtrl [12], DragNUWA [13] |

As with images (DLCV 12), one trained model can often serve several rows: a first frame, earlier frames or keyframes are extra conditions, and masking decides which frames are generated.

### Three demands on every clip

- **Per-frame quality.** Each frame on its own should look like a real image: sharp, free of artifacts, well composed.
- **Motion quality.** The clip should move, and the motion should be plausible: speed, direction and deformation should look like the real thing. A clip that barely moves is a failure even if every frame is perfect.
- **Temporal consistency.** What is the same object should stay the same object. Three failures have names. **Flicker** is fast, frame-to-frame change of brightness or fine texture that no real motion explains. **Identity drift** is a slow change of a subject's appearance (a face, a logo, the pattern on a shirt) over the clip. A failure of **object permanence** is an object that disappears, changes or reappears wrongly after being hidden.

The demands pull against each other. The easiest way to remove flicker is to generate almost no motion; the easiest way to get dramatic motion is to let details change freely. Evaluation, later in this chapter, has to check all three together.

### The cost

A video is large. A clip of $F = 81$ frames at $480 \times 832$ pixels (about five seconds at 16 frames per second) holds $81 \cdot 480 \cdot 832 \cdot 3 \approx 9.7 \times 10^7$ numbers, the size of about 81 images. A generator also cannot simply treat the frames one after another: consistency requires information to flow between distant frames. With attention, the cost of letting every position see every other position grows with the **square** of the number of positions. Two design levers control it: **compress** the video before generating (the next section), and **restrict** which positions attend to which (the section after it).

## Compressing video: 3D VAEs and video tokenizers

**Plain version.** Video autoencoders compress space and time to reduce generation cost. Their reconstruction quality limits recoverable detail and motion.

### Why not compress each frame with an image autoencoder?

The first latent video models reused the autoencoder of an image latent diffusion model [14] and encoded every frame separately. The Video LDM authors report that this causes flickering, because the image autoencoder was trained on images only and does not know that consecutive frames should decode consistently; they fine-tuned the **decoder** with added temporal layers and a 3D-convolutional discriminator on video, and found this step critical for good results [7]. Stable Video Diffusion also builds on a pretrained image latent model and compresses only in space [4], [3]. A frame-wise autoencoder also keeps one latent frame per video frame, so it does nothing for the cost in time.

### 3D causal VAEs

A **3D VAE** uses convolutions over time as well as space, so it can downsample both [2]. Most recent systems make it **causal in time**: every temporal convolution is padded only at the beginning of the clip, so a latent frame depends only on the current and earlier video frames, never on later ones [15], [2]. A useful consequence is that the very first frame is encoded on its own. The usual frame layout is therefore "$1+T$": the first frame becomes the first latent frame, and every following group of $f_t$ frames becomes one more latent frame. A still image is then just a one-frame video, which lets one autoencoder, and one generator, train on images and videos together [15], [16], [3].

Let a clip have $F = 1 + T$ frames of size $H \times W$, compressed by a factor $f_t$ in time and $f_s$ along each spatial axis into $C$ latent channels. The latent grid and the number of transformer tokens are

$$T' = 1 + \frac{T}{f_t}, \qquad H' = \frac{H}{f_s}, \qquad W' = \frac{W}{f_s}, \qquad L = T' \cdot \frac{H'}{p} \cdot \frac{W'}{p},$$

where $T'$, $H'$ and $W'$ are the numbers of latent frames, rows and columns, $p$ is the side of the square patch that the transformer groups into one token (often $p = 2$), and $L$ is the sequence length. Wan's report gives exactly this count, $L = (1+T/4)\cdot H/16\cdot W/16$, for its 4×8×8 autoencoder and $2 \times 2$ patches [3]. Two numbers summarize a design: the **compression of numbers**, $\frac{3FHW}{CT'H'W'}$, and the **pixels per token**, about $f_t f_s^2 p^2$. Because the latent has more channels than the 3 of RGB, a video VAE compresses tokens much more than it compresses numbers.

Figure 1 shows the arithmetic for a 17-frame clip.

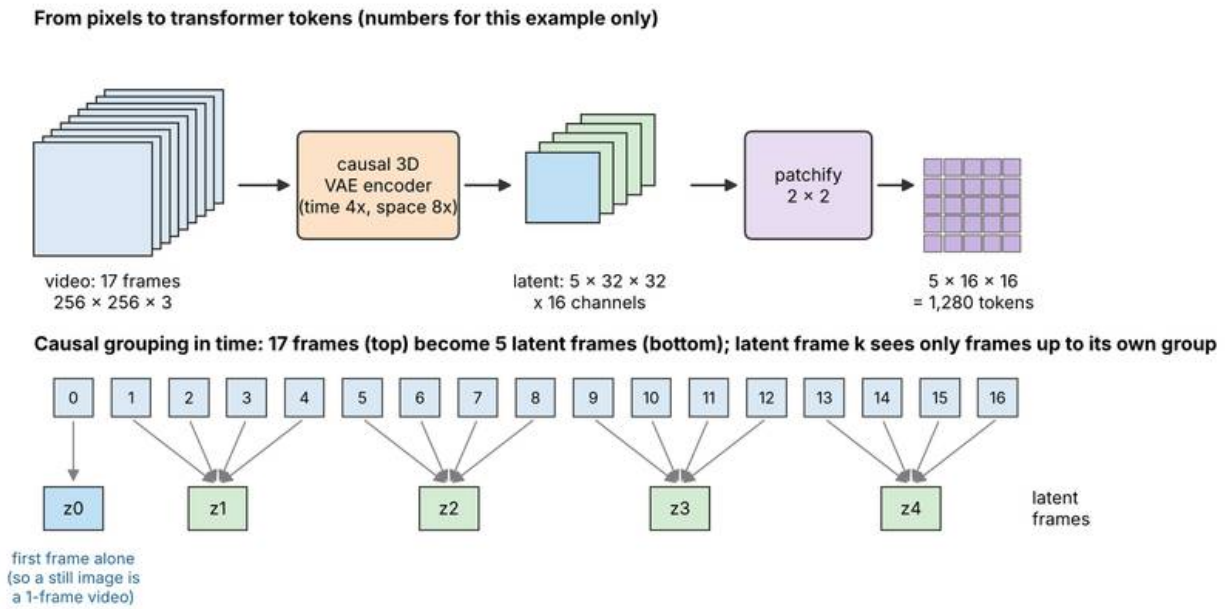



*Figure 1 — A causal video VAE in numbers. The first frame is encoded alone; each later latent frame summarizes four frames and sees no future frames. A 2 × 2 patchify step then sets the transformer's sequence length.*

Published systems document their factors. The table lists a few (a map of choices, not a ranking).

| System | Compression (time × height × width) | Latent channels | Notes |
|---|---|---|---|
| Stable Video Diffusion [4] | 1 × 8 × 8 | 4 | image autoencoder, no temporal compression (factors as listed in [3]) |
| CogVideoX [2] | 4 × 8 × 8 | 16 | 3D causal VAE |
| HunyuanVideo [16] | 4 × 8 × 8 | 16 | causal 3D VAE, $(T+1)$-frame input |
| Wan [3] | 4 × 8 × 8 | 16 | causal; first frame compressed only in space |

| System | Compression (time × height × width) | Latent channels | Notes |
|---|---|---|---|
| Movie Gen [17] | 8 × 8 × 8 | 16 | temporal autoencoder inflated from an image autoencoder |
| LTX-Video [18] | 8 × 32 × 32 | 128 | about 1:192 overall; no separate patchify step |
| MAGVIT-v2 in VideoPoet [19] | 17 frames of 128 × 128 → 5 × 16 × 16 discrete tokens | vocabulary of $2^{18}$ | discrete tokens for a language model |

LTX-Video frames the comparison in pixels per token: a 4 × 8 × 8 autoencoder followed by 2 × 2 patches gives 1,024 pixels per token, while its own design gives 8,192 [18]. Pushing compression further has a price. CogVideoX's ablation found that a 8 × 16 × 16 setting made the generator's training converge with great difficulty even with more latent channels [2], and LTX-Video asks its decoder to perform the final denoising step in pixel space to recover fine detail lost by the strong compression [18].

*Online only: Causal 3D convolutions and the 1 + T frame layout — https://blog.shyandram.dev/en/learn/deep-learning/video-generation/notes/causal-3d-convolutions-and-the-1–t-frame-layout/*

### Continuous latents or discrete tokens

Like image models (DLCV 12), video generators either work on a **continuous** latent grid (a VAE, used by diffusion and flow models) or on **discrete tokens** from a learned codebook (used by masked and autoregressive token models). MAGVIT introduced a 3D tokenizer that quantizes a video into spatiotemporal tokens for masked generation [20]. MAGVIT-v2 replaced it with a causal 3D CNN tokenizer and a **lookup-free quantizer** whose vocabulary can grow large, and with it a language model outperformed diffusion models on image and video benchmarks in the authors' tests [15]; codebooks and their problems are covered in ML 09.

### Code: latent size and attention cost

The snippet computes the latent grid, the number of tokens and the number of query–key pairs in one attention layer for an 81-frame, 480×832 clip, for three autoencoders. The pair counts anticipate the next section: "full" lets every token attend to every token, "factorized" splits attention into a spatial part within each latent frame and a temporal part at each position.

```
def latent_grid(frames, height, width, ft, fs, channels=16):
    """Latent shape of a causal video VAE: the first frame alone, then groups of ft
     ↪  frames."""
    t = 1 + (frames - 1) // ft
    return t, height // fs, width // fs, channels

def attention_pairs(t, s):
    """Query-key pairs in one attention layer over t latent frames of s tokens each."""
```

```
    full = (t * s) ** 2                 # every token attends to every token
    factorized = t * s**2 + s * t**2    # spatial attention per frame + temporal attention
     ↪  per position
    return full, factorized

frames, H, W, patch = 81, 480, 832, 2   # about 5 s at 16 fps; DiT patches of 2 x 2 latent
 ↪  positions
pixels = frames * H * W * 3
print(f"pixel values: {pixels:,}")
for name, ft, fs, c in [("image VAE, every frame", 1, 8, 4),
                        ("video VAE 4x8x8", 4, 8, 16),
                        ("video VAE 8x16x16", 8, 16, 64)]:
    t, h, w, c = latent_grid(frames, H, W, ft, fs, c)
    s = (h // patch) * (w // patch)     # tokens per latent frame after patchify
    full, fact = attention_pairs(t, s)
    print(f"{name:23s} latent {t}x{h}x{w}x{c} ({pixels / (t*h*w*c):5.1f}x fewer numbers)  "
          f"tokens {t*s:7,}  full {full:.1e}  factorized {fact:.1e}  ratio {full/fact:4.0f}")
```

It prints:

```
pixel values: 97,044,480
image VAE, every frame  latent 81x60x104x4 ( 48.0x fewer numbers)  tokens 126,360  full
 ↪  1.6e+10  factorized 2.1e+08  ratio   77
video VAE 4x8x8         latent 21x60x104x16 ( 46.3x fewer numbers)  tokens  32,760  full
 ↪  1.1e+09  factorized 5.2e+07  ratio   21
video VAE 8x16x16       latent 11x30x52x64 ( 88.4x fewer numbers)  tokens   4,290  full
 ↪  1.8e+07  factorized 1.7e+06  ratio   11
```

(All numbers are our own arithmetic; the $8 \times 16 \times 16$ row with 64 channels is a hypothetical configuration, not a specific model.) Three lessons. First, the $4 \times 8 \times 8$ video VAE keeps about as many numbers as the frame-wise image VAE, but it cuts the **tokens** by a factor of four, and the attention pairs by a factor of sixteen, because each latent position carries more channels. Second, the 32,760 tokens match Wan's formula for this clip size. Third, full attention over 32,760 tokens means about $10^9$ query–key pairs per layer, which is why the choice between full and factorized attention, and the compression factor, decide what a model can afford.

### Architectures: from inflated image models to video transformers

**Plain version.** One route adds temporal layers to an image model; another trains a video Transformer. Both must decide how information crosses space and time.

#### Inflating a pretrained image model

The idea is to reuse what an image model already knows about appearance and to learn only motion. Video Diffusion Models (VDM) turned the image diffusion U-Net into a **3D U-Net factorized over space and time**: every $3 \times 3$ convolution became a $1 \times 3 \times 3$ convolution that acts on

each frame separately, each spatial attention block kept attending within a frame, and a new temporal attention block after it attended over frames at each spatial position [6]. Because temporal attention can be switched off, the same network can be trained on single images and on videos at once, and the authors found this joint training important for sample quality [6].

Later systems keep the pretrained image layers and insert new temporal layers that are **initialized to do nothing**, so the inflated model starts exactly as the image model. Let $h$ be the features of one block, reshaped so that frames are treated as a batch of images. The inflated block computes

$$h_{\text{out}} = h + Z(\,\text{TempAttn}(h)),$$

where TempAttn attends along the frame axis at each spatial position and $Z$ is a linear projection whose weights start at zero, so that $h_{\text{out}} = h$ at the start of training (the same idea as ControlNet's zero convolutions in DLCV 12). Training then moves the model gradually from "a batch of independent images" toward "a video". Make-A-Video [1], Video LDM [7], AnimateDiff [21] and Open-Sora [22] all use a version of this idea: identity-initialized layers, a learned mixing weight or zero-initialized projections. Make-A-Video also shows the payoff: it learns appearance and its link to text from image–text pairs and only motion from unlabeled video, so it needs no paired text–video data [1].

*Online only: Four ways to start a video model from an image model — https://blog.shyandram.dev/en/learn/deep-learning/video-generation/notes/four-ways-to-start-a-video-model-from-an-image-model/*

### Factorized or full spatiotemporal attention

**Plain version.** A token can look at every other token in the clip (full attention), or first at the tokens of its own frame and then at the tokens in the same place in other frames (factorized attention). Factorized attention is much cheaper; full attention can follow an object that moves far between frames.

Let a clip have $T$ latent frames of $S$ tokens each, so $N = TS$ tokens in total. One full attention layer compares $N^2 = T^2S^2$ query–key pairs. Factorized attention runs spatial attention inside each frame ($T$ problems of size $S$, so $TS^2$ pairs) and temporal attention at each position ($S$ problems of size $T$, so $ST^2$ pairs). The ratio is

$$\frac{T^2S^2}{TS^2 + ST^2} = \frac{TS}{S + T},$$

which is close to $T$ when $S \gg T$, as in the snippet above (ratio 21 for 21 latent frames). The factorized pattern comes from video understanding: TimeSformer compared self-attention schemes and found "divided" space–time attention best among those it tried for video classification [23], and ViViT proposed several variants that factorize the spatial and temporal dimensions [24]. Latte compared four such decompositions inside a latent video diffusion transformer (interleaved spatial and temporal blocks, late fusion, and two ways of splitting attention inside a block) and found interleaved spatial and temporal transformer blocks performed best in its experiments [25].

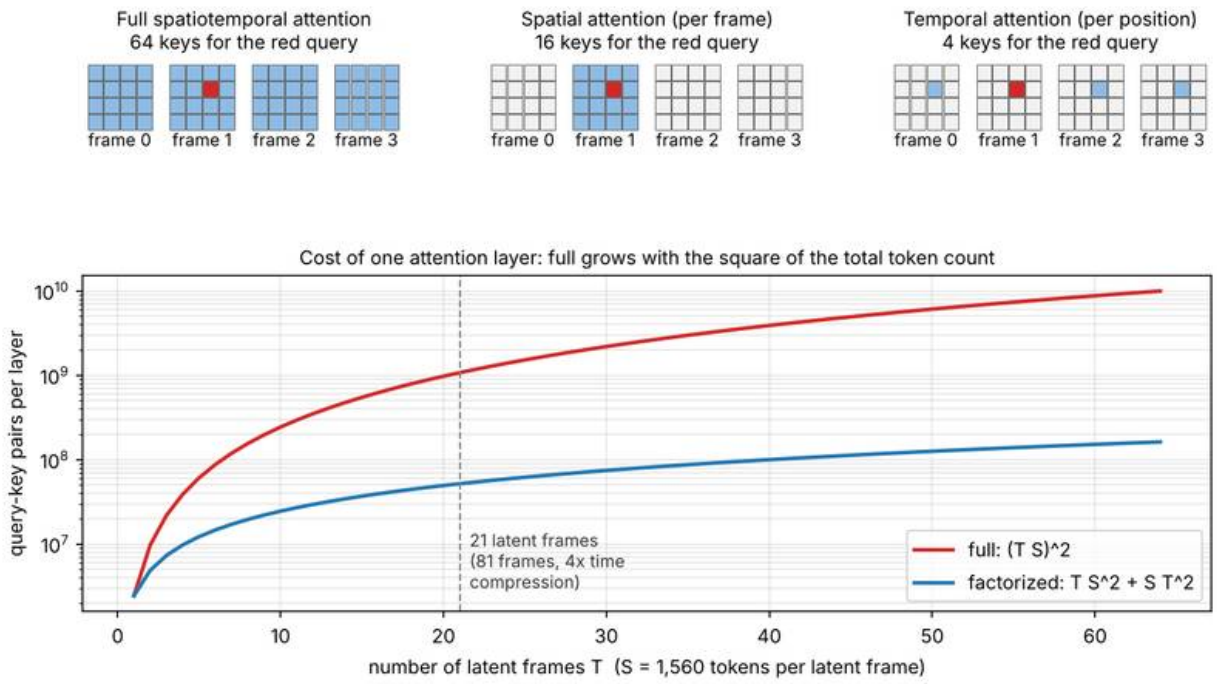



*Figure 2 — Which keys a query sees under full, spatial and temporal attention, and what that costs. Factorized attention sends information between distant positions in different frames only indirectly, through several layers.*

The case for full attention is about motion. CogVideoX argues that with separated spatial and temporal attention, an object that moves far between adjacent frames cannot attend directly to itself in the next frame; information must travel indirectly through other patches, which makes consistency of large motions hard to learn; it therefore uses full 3D attention over text and video tokens together [2]. HunyuanVideo chose full attention, citing better results than divided attention, a single design for images and videos, and the reuse of acceleration methods developed for language models [16]. Wan also uses full spatiotemporal attention [3]; Open-Sora Plan moved from a factorized "2+1D" denoiser to 3D full attention and added a cheaper sparse attention operation [26]; and LTX-Video uses its very high compression precisely so that full attention stays affordable [18]. The practical rule: full attention is preferred when the token count allows it, and the autoencoder is designed to make the token count allow it.

### Cascades or one model for the whole clip

**Cascades** generate a short, small, low-frame-rate video and then improve it with separate models. Imagen Video uses seven sub-models (one base model, three spatial super-resolution and three temporal super-resolution models) to reach 128 frames of 1280 × 768 at 24 frames per second, about 5.3 seconds [27]. Make-A-Video combines a base model, a frame-interpolation network and two super-resolution models [1]; Video LDM generates sparse keyframes and fills in frames with an interpolation model [7]. A cascade keeps each stage small, but the base stage fixes the

global motion from few frames. Lumiere argued that generating distant keyframes and then interpolating makes global temporal consistency difficult, and introduced a **Space-Time U-Net** that downsamples and upsamples in time as well as space and generates the full-frame-rate clip in one pass [28]. Large latent transformers with temporal compression take the same single-stage route; Movie Gen notes that its 8× temporal compression lets it forgo the frame-interpolation models of earlier work [17].

### A map of landmark systems

The table lists systems whose design is documented in a paper or technical report. It is a map of design choices, not a ranking; Figure 3 places them in time.

| System | Generator and backbone | Where it generates | Note |
|---|---|---|---|
| VDM [6] | diffusion, factorized 3D U-Net | pixels | joint image–video training |
| Make-A-Video [1] | diffusion, inflated U-Net, cascade | pixels | no paired text–video data |
| Imagen Video [27] | diffusion, cascade of seven models | pixels | spatial and temporal super-resolution |
| Video LDM [7] | diffusion, frozen image LDM plus temporal layers | image-autoencoder latent | fine-tuned temporal decoder |
| Stable Video Diffusion [4] | diffusion, inflated latent U-Net | image-autoencoder latent | staged training with data curation |
| Lumiere [28] | diffusion, Space-Time U-Net | pixels | whole clip in one pass |
| Latte [25] | diffusion transformer, factorized | latent | compares four factorizations |
| CogVideoX [2] | diffusion transformer, full 3D attention | 3D causal VAE latent | text and video tokens concatenated |
| Open-Sora [22] | diffusion transformer, factorized (STDiT) | 3D autoencoder latent | fully open training code and weights |
| Open-Sora Plan [26] | diffusion transformer, 3D attention with a cheaper sparse variant (Skiparse) | wavelet-flow VAE latent | open project with data pipeline |
| HunyuanVideo [16] | flow-matching transformer, full attention | causal 3D VAE latent | over 13 billion parameters |
| Movie Gen [17] | flow-matching transformer | temporal-autoencoder latent | 30B parameters, synchronized audio |
| Wan [3] | flow-matching diffusion transformer, full attention | causal 3D VAE latent | 1.3B and 14B models |
| LTX-Video [18] | latent diffusion transformer, full attention | highly compressed latent | decoder does the last denoising step |
| MAGVIT [20], VideoPoet [19] | masked or autoregressive token transformer | discrete tokens | see the section on autoregressive generation |

Flow matching and diffusion transformers are derived in ML 08 and ML 08, diffusion transformers [29], [30]. Parameter counts above are the ones stated in the papers.

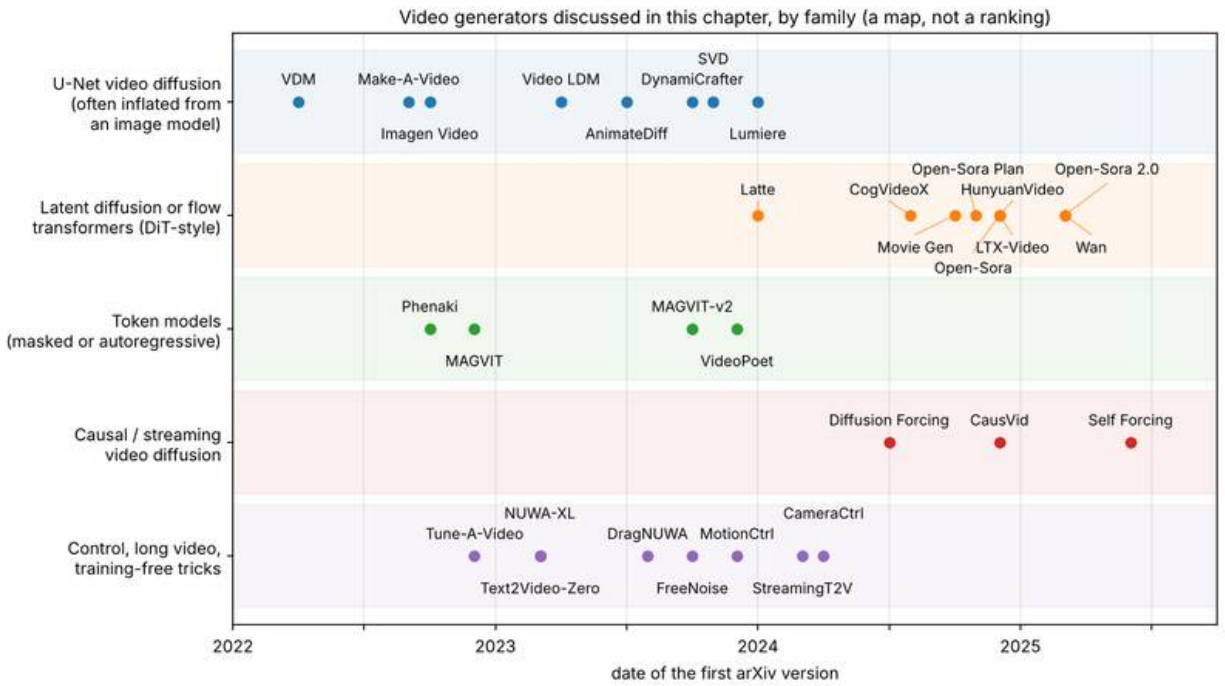



*Figure 3 — The systems in this chapter by family and by the date of their first arXiv version. The center of gravity moved from inflated U-Nets (2022–2023) to latent transformers with full attention (2024–2025), with token models and causal diffusion as parallel lines.*

### Conditioning: text-to-video and image-to-video

**Plain version.** The prompt enters a video model the same way it enters an image model, through attention to text-encoder outputs. A first frame for image-to-video enters as an extra input that every generated frame can see. The difference from images is that the condition must hold for the whole clip, and that "how much it moves" is itself something to control.

### Text-to-video

Text encoders, cross-attention and joint text–image attention are explained in DLCV 12. Video systems use the same mechanisms. Wan embeds text with **cross-attention** from video tokens to text-encoder outputs [3]. CogVideoX concatenates text and video tokens in one sequence and uses an **expert adaptive LayerNorm** with separate modulation for the two modalities, because their feature spaces and numerical scales differ [2]. HunyuanVideo processes video and text tokens in separate streams first and then concatenates them for joint processing ("dual-stream to single-stream"), with a multimodal language model as its main text encoder [16]. Captions must describe motion and events, not only appearance (see Data).

Make-A-Video also feeds the **frame rate** to the model, as data augmentation and as a control of motion speed [1]. Guidance works as for images (DLCV 12).

### Image-to-video

Image-to-video ("animate this photo") must keep the input frame's content while inventing motion. Three documented designs:

- **Stable Video Diffusion** replaces the text embedding with the image embedding of the conditioning frame from a pretrained image–text model, and concatenates a noise-augmented copy of the frame to the denoiser input along the channel axis, repeated for every frame. The authors found that a constant guidance scale caused either drift away from the conditioning frame or oversaturation, and instead increased the guidance scale linearly across frames [4].
- **DynamiCrafter** projects the image into a text-aligned context space with a query transformer, so that the video model can attend to it like a prompt, and also concatenates the full image with the initial noise to preserve visual detail [5].
- **HunyuanVideo** uses the latent of the reference image directly as the first latent frame [16], a natural choice when the causal VAE encodes the first frame alone.

A common failure of image-to-video is **too little motion**: the model can score well on fidelity to the input by barely changing it. StreamingT2V reports that competing image-to-video models tend toward "video stagnation" when used repeatedly to extend a clip [8].

### Camera and motion control

**Plain version.** A prompt such as "the camera slowly orbits the statue" is a weak way to describe motion. Control methods give the generator the motion directly: a camera path (where the camera is in each frame) or trajectories (where an object or a dragged point should go). The challenge is to obey the motion without changing the content.

- **Camera motion as a small adapter.** Stable Video Diffusion trained separate camera-motion LoRA modules for horizontal movement, zooming and a static camera on top of its base model [4]; AnimateDiff's **MotionLoRA** adapts its motion module to new motion patterns such as shot types at a low cost [21].
- **Camera poses as a dense input. CameraCtrl** trains a plug-and-play camera module on top of a video diffusion model, leaving the base model untouched. Instead of feeding raw camera matrices, it represents each frame's camera as a per-pixel **Plücker embedding**, a 6-number description of the viewing ray through each pixel, which ties camera geometry to image positions; the authors found that training videos with diverse camera paths and an appearance similar to the base model's data gave the best balance of control and generalization [12].
- **Camera and objects together. MotionCtrl** controls camera motion and object motion independently, with camera poses and object trajectories as conditions that carry no appearance information [11]. **DragNUWA** combines text, an image and trajectories for fine-grained control of open-domain video [13].

A camera path is the same pose sequence that 3D reconstruction estimates (DLCV 11). Following it does not make the generated scene geometrically consistent, and a control signal is only as reliable as the poses or tracks it was trained on.

*Online only: Camera poses as Plücker embeddings — https://blog.shyandram.dev/en/learn/deep-learning/video-generation/notes/camera-poses-as-plücker-embeddings/*

### Long videos

**Plain version.** Models are trained on short clips, a few seconds long, because memory grows with length. To make a longer video you must either keep extending a clip piece by piece, which tends to drift, or plan the whole video coarsely first and fill in the details, which needs long training videos. Either way, keeping the story and the identities straight over a minute is still hard.

Four strategies are documented:

1. **Extend autoregressively, chunk by chunk.** Generate a clip, then generate the next clip conditioned on the last frames of the previous one. VDM showed that simply replacing the known frames during sampling gives continuations that look good alone but do not match the earlier frames, and proposed **reconstruction guidance**, which adds a guidance term from the model's reconstruction of the conditioning frames [6]. Video LDM trains prediction models that receive a few context frames through a mask [7]. **StreamingT2V** adds a short-term memory (attention to features of the previous chunk) for smooth transitions and a long-term memory (features of the first chunk) so the model does not forget the initial scene [8].
2. **Plan coarse, then fill in. NUWA-XL** applies a global diffusion model to generate keyframes over the entire time range and local diffusion models that recursively fill in frames between them, which lets it train directly on long videos and generate segments in parallel [31].
3. **Change the noise, not the model. FreeNoise** extends a pretrained short-clip model without tuning: it reschedules a sequence of initial noises with long-range correlation and applies temporal attention over windows, and adds a motion-injection method for prompts that change over time [32].
4. **Variable-length generation from a story. Phenaki** compresses video into discrete tokens with a tokenizer that is causal in time, so it handles variable lengths, and generates long videos from a sequence of prompts that change over time [33].

The central difficulty is **error accumulation**: each chunk is conditioned on generated, slightly wrong frames, and errors compound, exactly as in the rollouts of a learned world model (DLCV 13). The causal-diffusion methods in the section on autoregressive generation address the same problem from the training side.

*Online only: Reconstruction guidance for extending a video — https://blog.shyandram.dev/en/learn/deep-learning/video-generation/notes/reconstruction-guidance-for-extending-a-video/*

### Temporal consistency

**Plain version.** A consistent video keeps each detail attached to the thing it belongs to: when the dog moves, its spots move with it; when the camera pans, the texture of the wall slides along. The simplest test is to undo the motion and check that nothing else changed. The trap is that a video that does not move at all passes this test perfectly.

### Where inconsistency comes from

A generator invents detail that the condition does not specify. If that invented detail is drawn afresh for every frame, it flickers. If it is fixed to the screen instead of to the objects, it "sticks" while the scene moves under it. Consistency means the invented detail must be **propagated along the motion**. Architectures do this through temporal layers and joint generation of all frames; several training-free methods enforce it in pretrained image models. Text2Video-Zero makes the self-attention of every frame attend to the **first frame** (cross-frame attention), to preserve the appearance and identity of the foreground object, and adds motion dynamics to the latent codes to keep the background consistent [10]. Make-A-Video uses the **same noise initialization** for every frame in its final per-frame super-resolution stage, to encourage consistent invented detail across frames [1]. Tune-A-Video adds a spatiotemporal attention mechanism to an image model and tunes it on a single video [9].

### Measuring consistency

Three measures are common, and each is defined in terms of quantities you can compute from the generated video alone:

- **Frame difference** $D = \frac{1}{(F-1)\,P} \sum_{t=2}^{F} \|x_t - x_{t-1}\|_1$, where $x_t$ is frame $t$ and $P$ the number of pixel values per frame. It is cheap but mixes real motion with flicker; VBench measures temporal flickering this way only on static scenes, where all change is flicker [34].
- **Warping error** $E = \frac{1}{(F-1)\,P} \sum_{t=2}^{F} \|x_t - \mathcal{W}(x_{t-1}, u_t)\|_1$, where $u_t$ is the motion (optical flow) from frame $t-1$ to frame $t$ and $\mathcal{W}$ warps the previous frame along it. It removes the real motion before comparing; flow, warping and its masked form are explained in DLCV 09A.3. For a generated video the flow must be **estimated from the video itself**.
- **Feature similarity over time.** VBench's subject consistency compares the features of each frame with those of the first and the previous frame, $S = \frac{1}{F-1} \sum_{t=2}^{F} \frac{1}{2}(\langle d_1, d_t \rangle + \langle d_{t-1}, d_t \rangle)$, where $d_t$ is a unit-length feature vector of frame $t$ from a self-supervised image model and $\langle \cdot, \cdot \rangle$ is the dot product (cosine similarity) [34].

All three reward a still video. For that reason VBench pairs them with a **dynamic degree** dimension that measures the amount of motion with an optical-flow network [34].

### Code: flicker, sticking and freezing

The toy builds a "coarse video" that pans an image 2 pixels per frame, as a prompt might request, and adds invented fine detail in four ways: new noise every frame, the same noise fixed on the screen, the same noise moved along with the scene, and a frozen video that repeats the first frame. It estimates each video's own motion with phase correlation and reports the three quantities.

```
import numpy as np
from scipy import ndimage as ndi
```

```
from skimage import data, transform
from skimage.registration import phase_cross_correlation

rng = np.random.default_rng(0)
T, v = 16, 2                                  # 16 frames; the prompt asks for a pan of 2 px
 ↪  per frame
scene = transform.resize(data.camera() / 255.0, (128, 128), anti_aliasing=True)
content = np.stack([np.roll(scene, v * t, axis=1) for t in range(T)])   # the "coarse" video

def detail():                                 # invented fine texture, as a generator would
 ↪  add
    n = ndi.gaussian_filter(rng.standard_normal(scene.shape), 0.7)
    return 0.04 * n / n.std()

d0 = detail()
videos = {
    "independent noise": content + np.stack([detail() for _ in range(T)]),  # new detail
     ↪  every frame
    "shared noise":      content + d0,                                      # same detail,
     ↪  fixed on screen
    "propagated noise":  content + np.stack([np.roll(d0, v * t, axis=1) for t in range(T)]),
     ↪  # detail moves with the scene
}
videos["frozen"] = np.repeat(videos["propagated noise"][:1], T, axis=0)      # a still image
 ↪  repeated

c = (slice(None), slice(8, -8), slice(8, -8))  # ignore borders (np.roll wraps around)
def measure(vid):
    shifts = [phase_cross_correlation(a, b, upsample_factor=10)[0][1] for a, b in
     ↪  zip(vid[1:], vid[:-1])]
    flicker = np.abs(np.diff(vid, axis=0))[c].mean()                         #
     ↪  frame-difference energy
    warped = np.stack([np.roll(f, int(round(s)), axis=1) for f, s in zip(vid[:-1], shifts)])
    warp_err = np.abs(vid[1:] - warped)[c].mean()                            # error after
     ↪  the video's own motion
    return flicker, warp_err, np.mean(np.abs(shifts))

for name, vid in videos.items():
    f, w, m = measure(vid)
    print(f"{name:18s} frame difference {f:.4f}   warping error {w:.4f}   estimated motion
     ↪  {m:.2f} px/frame")
```

It prints:

```
independent noise  frame difference 0.0808   warping error 0.0446   estimated motion 2.00
 ↪  px/frame
shared noise       frame difference 0.0528   warping error 0.0419   estimated motion 2.01
 ↪  px/frame
propagated noise   frame difference 0.0796   warping error 0.0000   estimated motion 2.00
 ↪  px/frame
frozen             frame difference 0.0000   warping error 0.0000   estimated motion 0.00
 ↪  px/frame
```

(All numbers are from our toy run.) Read the four lines together:

- **Frame difference alone is misleading.** It rates the screen-fixed noise (0.0528) as *more* stable than the correct, propagated detail (0.0796), because the propagated detail moves, and it gives the frozen video a perfect zero.
- **Warping error separates the cases that move.** After undoing the estimated motion, only the propagated detail leaves nothing behind; both new-every-frame noise and screen-fixed noise remain visible.
- **A motion measure is needed as well.** The frozen video also has zero warping error. Only the estimated motion (0.00 instead of the requested 2 pixels per frame) exposes it, which is the role of VBench's dynamic degree.

Figure 4 shows the same effects as motion-compensated x–t slices: correct, propagated detail forms straight vertical lines.

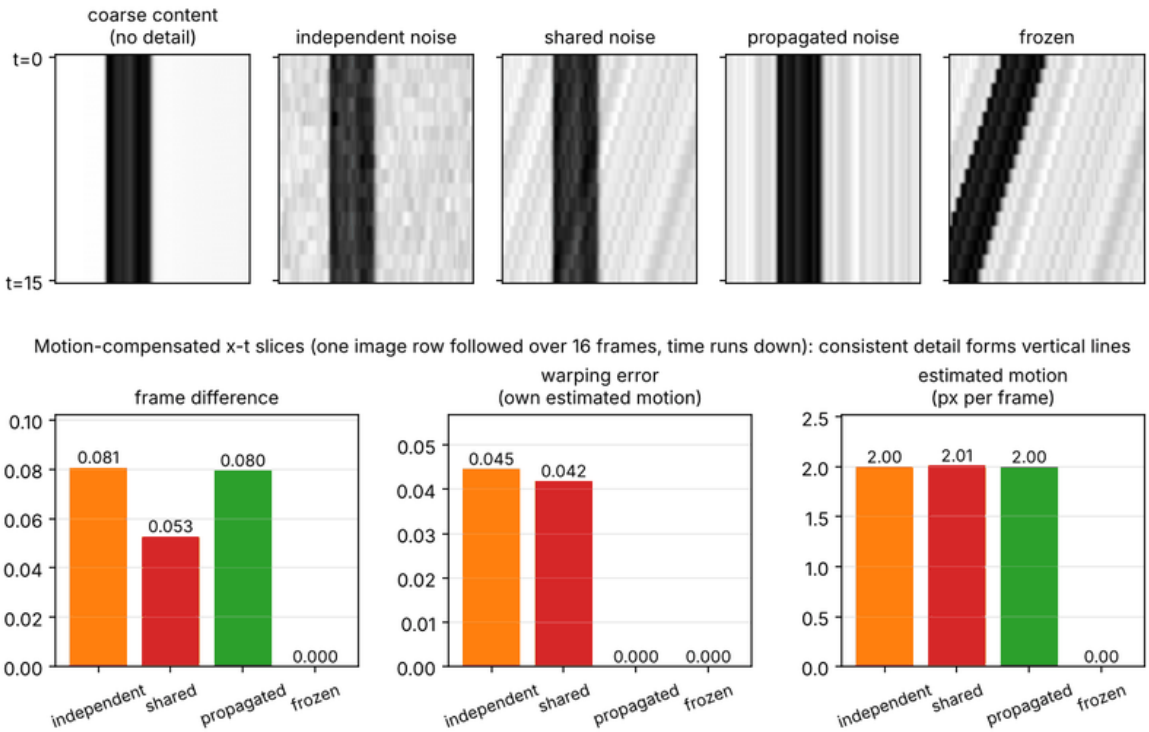



*Figure 4 — The consistency toy. Undoing the requested motion turns correct detail into vertical lines; flicker shows as speckle, screen-fixed detail as slanted streaks, and a frozen video as a slanted band. No single measure ranks all four correctly.*

### Autoregressive and diffusion video generation

**Plain version.** There are two ways to produce a clip. A **diffusion** (or flow) model refines all frames of a fixed-length clip together, from noise to video, so every frame can see the future as

well as the past. An **autoregressive** model writes the video piece by piece, each piece conditioned on what came before, like a language model writing text; it can go on as long as you like and can stream, but early mistakes stay. Recent hybrids try to get both.

### Token models: masked and autoregressive

With a discrete video tokenizer, video generation becomes sequence modeling (ML 09). **MAGVIT** uses masked token modeling, predicting many tokens in parallel, and supports many generation tasks (such as frame prediction, interpolation and inpainting) with a single model [20]. **Phenaki** generates its causal video tokens with a bidirectional masked transformer conditioned on text [33]. **VideoPoet** is a decoder-only transformer that processes text, image, video and audio tokens; MAGVIT-v2 turns 17 frames of $128 \times 128$ video at 8 frames per second into a $5 \times 16 \times 16$ grid, that is 1,280 tokens from a vocabulary of $2^{18}$, and the model is trained like a language model, with pretraining on a mixture of generative tasks followed by task-specific adaptation [19]. Token models reuse language-model infrastructure (KV caching, ML 09) and can extend a clip naturally, but every generated token is a commitment, and errors accumulate (ML 09).

### Diffusion: the whole clip at once

A standard video diffusion transformer denoises all latent frames jointly with **bidirectional** attention. CausVid states the consequence plainly: generating a single frame requires processing the entire sequence, including the future, which makes such models hard to use interactively [35]. Their strengths are the ones that made diffusion dominant for images (stable training, high quality, guidance), plus global consistency within the clip. Their limits are a fixed maximum length and high latency.

### Hybrids: causal, streamable diffusion

- **Diffusion Forcing** trains a diffusion model to denoise a sequence of tokens with **independent noise levels per token**, and uses it as a causal next-token model that generates future frames without fully diffusing past ones; this allows rolling out video beyond the training horizon, where its baselines diverge [36].
- **CausVid** adapts a pretrained bidirectional video diffusion transformer into a causal, autoregressive one that generates frames on the fly, distills its 50-step sampler into a 4-step one with a bidirectional teacher supervising the causal student, and uses KV caching for streaming generation; the authors report that this mitigates error accumulation and allows long videos despite training on short clips [35].
- **Self Forcing** attacks **exposure bias** directly: during training, each frame is generated conditioned on the model's own previous outputs (an autoregressive rollout with KV caching), not on ground-truth frames, and a loss is applied to the whole generated video; it reports real-time streaming generation with sub-second latency on a single GPU [37].

| Aspect | Autoregressive or causal | Bidirectional diffusion |
|---|---|---|
| Length | open-ended, chunk by chunk | fixed by the training clip length, unless extended |
| Latency | first frames available early, can stream | whole clip must be denoised first |
| Future information | none (causal) | every frame sees past and future |
| Main risk | error accumulation, exposure bias | memory and compute for long clips |
| Typical use | interactive generation, world models, live video | offline text-to-video and image-to-video |

The same causal-versus-bidirectional distinction matters for world models, where actions arrive one step at a time (DLCV 13). Remember also, from ML 08, that a causal **autoencoder** does not make a generator autoregressive: the generator's attention and sampling decide that.

## Evaluating video generators

**Plain version.** You need evidence of four things: that the videos as a set look like real videos (distribution metrics such as FVD), that they match their prompts, that their motion and consistency are good (dimension-by-dimension suites such as VBench), and that people prefer them (human studies). The standard distribution metric turns out to look mostly at single frames, so it can miss exactly the temporal problems this chapter is about.

### Fréchet Video Distance

**FVD** extends FID (ML 08) to video [38]. Each video, real or generated, is passed through a pretrained video network, an I3D action-recognition model [39] trained on Kinetics-400 [40], and its output (the logits) is taken as a feature vector. A Gaussian is fitted to each set of features, and

$$\mathrm{FVD} = \|\mu_r - \mu_g\|_2^2 + \mathrm{Tr}\big(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}\big),$$

where $\mu_r, \Sigma_r$ are the mean and covariance of the real-video features, $\mu_g, \Sigma_g$ those of the generated videos, Tr is the trace and $(\cdot)^{1/2}$ the matrix square root. Lower is better. The original paper introduced FVD with a human study in which it correlated well with judgments of generated videos [38]. Class-conditional and unconditional benchmarks often report FVD on UCF-101, a dataset of 101 human-action classes [41], [1], [25].

**Its main limit: content bias.** Ge et al. found that FVD increases only slightly under large temporal corruption at matched frame quality, and that a well-chosen subset of generated videos **without motion** can lower FVD drastically [42]. They trace the bias to the features: an action classifier trained on Kinetics learns object and scene appearance, because many actions can be recognized from one frame; features from a self-supervised video model, VideoMAE v2 [43], were much less biased [42]. The pitfalls of FID also apply (DLCV 12), plus video-specific ones: clip length, frame rate and resizing to a low, square resolution [42].

*Online only: Ge et al., "On the Content Bias in Fréchet Video Distance" (CVPR 2024) — https://blog.*

*shyandram.dev/en/learn/deep-learning/video-generation/papers/ge-et-al-on-the-content-bias-in-fréchet-video-distance-cvpr-2024/*

### Code: a Fréchet distance that cannot see time

The toy makes "real" clips of a blob moving in a straight line and two corrupted sets: one with **blurred frames** (worse frames, correct motion) and one with **shuffled frames** (perfect frames, broken motion). It computes a Fréchet distance with two hand-made feature extractors: one that averages per-frame appearance statistics over time, like an extractor that mostly sees content, and one that adds two temporal statistics.

```
import numpy as np
from scipy import linalg, ndimage as ndi

rng = np.random.default_rng(0)
T, S = 16, 32
yy, xx = np.mgrid[:S, :S]

def real_videos(n):
    """Toy 'real' clips: a bright blob moving in a straight line, plus mild sensor noise."""
    vids = np.empty((n, T, S, S))
    for i in range(n):
        p, vel, r = rng.uniform(10, 22, 2), rng.uniform(-0.7, 0.7, 2), rng.uniform(2.5, 4)
        for t in range(T):
            cy, cx = p + vel * t
            vids[i, t] = np.exp(-((yy - cy) ** 2 + (xx - cx) ** 2) / (2 * r**2))
    return vids + 0.02 * rng.standard_normal(vids.shape)

def frame_stats(f):                       # appearance of one frame
    gy, gx = np.gradient(f)
    return [f.mean(), f.std(), f.max(), np.abs(gx).mean() + np.abs(gy).mean()]

def appearance_features(v):               # frame statistics averaged over time: blind to
 ↪  frame order
    return np.mean([frame_stats(f) for f in v], axis=0)

def motion_features(v):                   # add how the content changes from frame to frame
    d = np.abs(np.diff(v, axis=0)).mean(axis=(1, 2))
    return np.concatenate([appearance_features(v), [d.mean(), d.std()]])

def frechet(a, b):                        # Fréchet distance between Gaussian fits of two
 ↪  feature sets
    mu1, mu2 = a.mean(0), b.mean(0)
    s1, s2 = np.cov(a, rowvar=False), np.cov(b, rowvar=False)
    covmean = linalg.sqrtm(s1 @ s2).real
    return float(((mu1 - mu2) ** 2).sum() + np.trace(s1 + s2 - 2 * covmean))

real, other_real = real_videos(400), real_videos(400)
blurred = ndi.gaussian_filter(other_real, sigma=(0, 0, 0.7, 0.7))          # worse frames,
 ↪  same motion
shuffled = np.stack([v[rng.permutation(T)] for v in other_real])          # perfect frames,
 ↪  broken motion
```

```
for name, feats in [("appearance features", appearance_features), ("+ motion features",
 ↪  motion_features)]:
    f = lambda vids: np.stack([feats(v) for v in vids])
    ref = f(real)
    scale = ref.std(0)                    # put all features on a comparable scale
    fd = {k: frechet(ref / scale, f(v) / scale) for k, v in
          [("second real set", other_real), ("blurred frames", blurred), ("shuffled frames",
           ↪  shuffled)]}
    print(f"{name:20s} " + "   ".join(f"{k} {d:6.2f}" for k, d in fd.items()))
```

It prints:

```
appearance features  second real set   0.01   blurred frames 115.10   shuffled frames   0.01
+ motion features    second real set   0.07   blurred frames 141.87   shuffled frames
 ↪  12397.17
```

(All numbers are from our toy run, with toy features, not I3D.) With appearance-only features, the shuffled set, whose motion is destroyed, is exactly as close to the real set as a second real set (0.01), while mild blur is heavily penalized. Adding two temporal statistics makes the shuffled set by far the worst. The real FVD is not this extreme, since I3D does see some motion, but this is the shape of the content bias that Ge et al. measured [42]: **the metric is only as temporal as its features**. Figure 5 shows the clips and the distances.

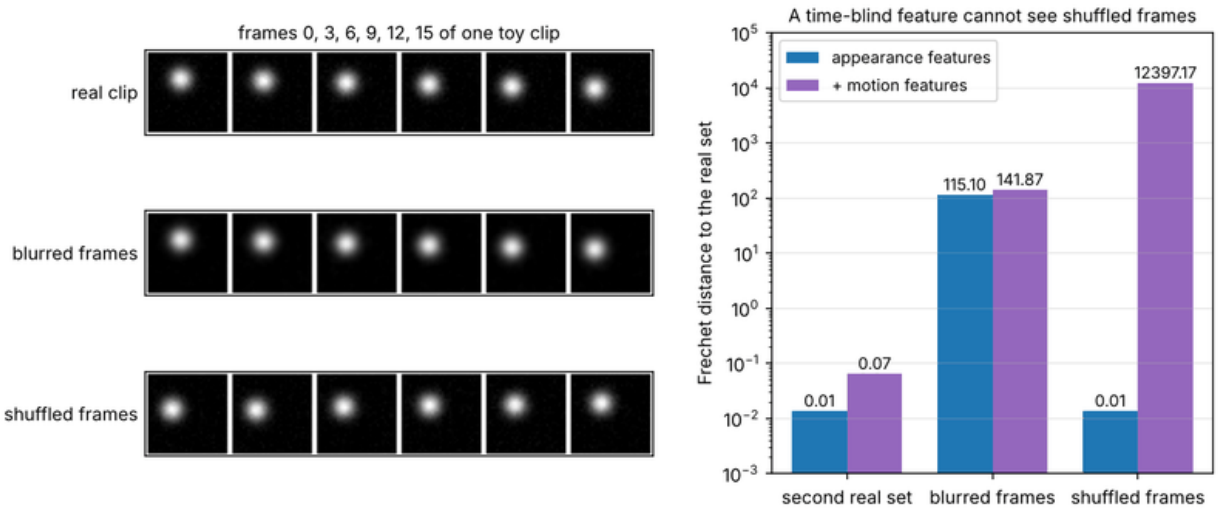



*Figure 5 — A Fréchet distance computed on time-blind features cannot tell a shuffled clip from a real one. Toy data and toy features.*

### Benchmark suites and human evaluation

**Plain version.** Instead of one number, modern benchmarks ask many narrow questions, each with its own prompts and its own automatic check, and confirm that the checks agree with people.

- **VBench** splits "video generation quality" into 16 dimensions, each with tailored prompts and an evaluation method: temporal quality (subject consistency, background consistency, temporal flickering, motion smoothness, dynamic degree), frame-wise quality (aesthetic and imaging quality), semantics (object class, multiple objects, human action, color, spatial relationship, scene), style (appearance and temporal style) and overall consistency; it provides human preference annotations for each dimension to validate the automatic scores [34]. **VBench++** extends it to image-to-video, with an image suite of adaptive aspect ratio, and to trustworthiness [44].
- **EvalCrafter** builds 700 prompts from an analysis of real user data, evaluates visual quality, content quality, motion quality and text–video alignment with 17 objective metrics, and fits coefficients so that the combined score agrees with users' opinions [45].
- **Human evaluation** remains the reference. Stable Video Diffusion compares its training choices with human preference studies [4]; HunyuanVideo reports evaluations by professionals [16]; Movie Gen evaluates with pairwise A/B tests by expert annotators along text alignment, visual quality, and realness and aesthetics, over a public set of 1,003 prompts, and reports that FVD and Inception Score did not correlate with its human evaluation of video quality [17]. OpenSora 2.0 reports both human evaluation and VBench scores [46].

Physical plausibility needs its own tests; DLCV 13 describes benchmarks that found physical understanding unrelated to visual realism.

**A reporting checklist.** State the prompts and how they were chosen, the clip length, frame rate and resolution, the number of samples and seeds, the sampler, steps and guidance (including any schedule across frames), the feature extractor and preprocessing of every distribution metric, the dimension suite and its version, and the human-study protocol (pairwise or absolute, number of annotators, majority rule). Report a temporal measure **together with** a motion measure and a per-frame quality measure, and show uncurated clips.

## Data

**Plain version.** Video models learn appearance, motion and the link to language from their training clips. Video with good captions is much scarcer than captioned images, so systems combine image and video data and spend great effort on filtering and recaptioning clips.

- **Images plus videos.** VDM [6], Phenaki [33] and many later systems train jointly on images and videos; Make-A-Video learns text–visual links only from image–text pairs [1]; the causal "1 + $T$" autoencoders make images one-frame videos [16], [3].
- **Web video collections.** WebVid, web videos with weak captions, was introduced with the Frozen in Time retrieval model (WebVid-2M in the paper; a 10M version was released later, according to its arXiv record) [47], and Make-A-Video was trained on WebVid-10M [1]. **Panda-**

**70M** curates 3.8M high-resolution videos, splits them into semantically consistent clips and captions them with several cross-modality teacher models plus a retrieval model that selects the best caption, giving 70M clips with captions [48].

- **Curation matters.** Stable Video Diffusion's Large Video Dataset starts with 580M annotated clips, forming 212 years of content, then removes nearly static clips with optical-flow scores, clips with much written text by OCR, and clips with low aesthetic scores; models trained on curated subsets were preferred by people over models trained on uncurated data [4]. CogVideoX describes its own video-captioning model as a key part of its data pipeline [2].
- **Evaluation datasets.** UCF-101 [41] and Kinetics [40] are human-action datasets that supply FVD benchmarks and FVD's feature network.

*Online only: Blattmann et al., "Stable Video Diffusion: Scaling Latent Video Diffusion Models to Large Datasets" (2023) — https://blog.shyandram.dev/en/learn/deep-learning/video-generation/papers/blattmann-et-al-stable-video-diffusion-scaling-latent-video-diffusion-models-to-large-datasets-2023/*

Web video raises the same questions of copyright, consent and content safety as web images; see DLCV 12, Data.

### Risks and responsible use

**Plain version.** Realistic generated video can put words in someone's mouth or show events that never happened. The defenses are the same three as for images (detect fakes, label generated content, and limit misuse at the source), but video is harder: it is re-encoded on every upload, and a few convincing seconds are enough to mislead.

- **Deepfakes and detection.** FaceForensics++ is a benchmark for detecting facial manipulations (four manipulation methods at random compression levels and sizes, over 1.8 million manipulated images, with a hidden test set); its authors found that domain-specific knowledge improves detection even under strong compression and that the detectors outperform human observers on their benchmark [49]. Detectors trained on one generation method can fail on newer ones, so detection is an arms race.
- **Watermarks for video. Video Seal** jointly trains a watermark embedder and extractor with transformations such as video codecs applied in between, and propagates the watermark over time so that not every high-resolution frame has to be watermarked [50]. Signed provenance manifests, invisible watermarks and their failure modes are explained in DLCV 12, Deepfakes, provenance and watermarks and its note Hard bindings, soft bindings and watermarks; the same lessons apply to video, where re-encoding and cropping are routine.
- **Bias, memorization and consent** follow from the data as for images (DLCV 12, Risks); people appear in web video far more often than in many image collections, which raises the stakes of consent.
- **False authority.** A generated video that looks physically right is easily taken as evidence. Treat generated clips as illustrations, not records, and see DLCV 13 on why realism is not physical correctness.

### Failure modes

**Plain version.** Today's video generators are good at short, single-shot clips of common scenes. They still fail at keeping things the same over time, at physics, at fine-grained motion and at anything long.

- **Flicker and texture sticking.** Fine detail changes from frame to frame or stays fixed while the scene moves (Figure 4).
- **Identity drift and morphing.** Faces, text and patterns change slowly; two objects merge or one object splits.
- **Object permanence.** Objects vanish when occluded or come back different.
- **Too little motion.** Near-static clips that score well on consistency; dynamic-degree measures exist because of this [34], and image-to-video extension can stagnate [8].
- **Implausible physics and dynamics.** Motion looks fluid but is physically wrong (see DLCV 13).
- **Drift in long videos.** Error accumulation when extending clip by clip [35], [37]; forgetting of the initial scene [8].
- **Weak control.** Camera paths that are only roughly followed, trajectories that change object shape; MotionCtrl's design goal of appearance-free motion conditions targets exactly this [11].
- **Inherited image failures.** Counting, attribute binding, text rendering and hands fail as in images (DLCV 12, Failure modes), now with the extra chance of changing over time.

**How to test.** Build small prompt sets per failure type (fast motion, occlusion, a person turning around, written text, a requested camera move), generate several seeds per prompt, measure consistency **and** motion, inspect motion-compensated x–t slices for flicker and sticking, and watch the clips; a frame grid hides most temporal failures.

### Modern view

**Surveys worth reading.**

- **Melnik et al. [51]** (TMLR 2024) survey video diffusion models: the mathematical formulation, architectural choices, methods for temporal consistency, a taxonomy of applications by input modality (text, images, video, audio), training and evaluation practice with datasets and metrics, and open challenges such as long videos and computational cost.
- **Xing et al. [52]** (2023) review video diffusion models in three areas, video generation, video editing and other video understanding tasks, with further categories in each, and discuss challenges and future trends.
- **Liu et al. [53]** (2024) review OpenAI's Sora, introduced in February 2024, from public technical reports and reverse engineering: background, related technologies, applications and limitations such as safe and unbiased generation. Because it relies on reverse engineering, read its technical claims as informed hypotheses.

**Historical models versus available services.** The papers here explain architectural milestones, not a current purchasing shortlist. As checked on 5 November 2026, OpenAI states that Sora's

web/app experiences ended on 26 April 2026 and its API ended on 24 September 2026 [54]. The independent Open-Sora projects are different projects. For current deployment choices and the checks that matter to a shot, see the video-generation selection guide.

**State of the art, 2023–2026.** Five trends stand out. (1) **Latent transformers with full attention and causal 3D VAEs** became the dominant recipe for large open models (CogVideoX [2], HunyuanVideo [16], Wan [3]), with flow-matching objectives in several of them [16], [17], [3]. (2) **Open models report closing much of the gap** with closed systems in their own human evaluations and VBench scores, and training cost became a research topic: Open-Sora 2.0 reports training a commercial-level model for about $200k [46]. (3) **Higher compression** reduces tokens so that full attention stays affordable, with the decoder taking over more of the detail [18]. (4) **Causal, few-step, streaming generation** brought interactive and real-time use [35], [37], [36]. (5) **Evaluation moved beyond FVD** toward dimension suites [34], [44], [45] and large human studies [17], with a recognized need for temporally sensitive features [42]. Synchronized audio generation also appeared [17], [19].

**Open problems.** Consistency and identity over minutes rather than seconds; physically correct dynamics and interaction (DLCV 13); precise, disentangled control of camera, objects and timing; metrics that measure motion and consistency without rewarding stillness and that agree with people; the cost of training and sampling long, high-resolution clips; caption quality for events and actions; and provenance that survives re-encoding.

## Key takeaways

- A sharp frame is not enough: motion and identity must remain coherent across time.
- Compression reduces generation cost but limits recoverable detail and temporal information.
- Temporal layers, factorized attention and full space–time attention offer different communication patterns and costs.
- State what is conditioned: text, the first frame, camera pose or motion trajectory.
- Chunk extension and causal generation enable longer outputs but must manage accumulated errors.
- Evaluate appearance, motion, temporal consistency and preference together; report data curation, duration and full cost.

## Exercises

1. **Token budgets.** A model uses a causal $4\times8\times8$ VAE and $2\times2$ patches. How many tokens does a 121-frame clip at $720\times1280$ produce? How many query–key pairs does one full attention layer have, and by what factor does factorized attention reduce it? Which single change, doubling the temporal compression or doubling the patch size, reduces the full-attention cost more?

   **Hint**

   $T' = 1 + 120/4 = 31$, and each latent frame has $(720/16)\cdot(1280/16) = 45\cdot80 = 3600$ tokens, so $L = 111{,}600$ tokens and about $1.25\times10^{10}$ pairs. The factorized ratio is $TS/(T+S) = 31\cdot3600/3631 \approx 30.7$.

Doubling $f_t$ gives $T' = 16$ (pairs shrink about 3.8×); doubling $p$ divides the tokens per frame by 4 (pairs shrink 16×), at the cost of coarser spatial tokens. Use the snippet's functions to check.

2. **Break the consistency measures.** In the consistency toy, add a fifth video in which the propagated detail is shifted by 2 pixels per frame but the coarse content does **not** move (a "moving texture on a still scene"). What do the three measures report, and which of them would you trust to detect this failure?

**Hint**

Phase correlation now sees two competing motions: the texture moves, the content does not. Depending on their relative strength, the estimated shift is 0 or 2, and whichever layer is not compensated leaves a warping error. A single global motion estimate is not enough for real videos with several moving parts; dense optical flow with occlusion masks is used in practice (DLCV 09A.3).

3. **Content bias in your own features.** Modify the Fréchet toy so that the "generated" set has correct frames but the blob moves twice as fast as in the real set. Which feature extractor detects it? Then design a third feature that would also detect a blob that moves at the right speed but reverses direction halfway.

**Hint**

Speed changes the mean frame difference, so only the motion features detect it. A reversal keeps the mean absolute difference similar; a feature that compares the direction of motion over time (for example the correlation between the first and second halves of the centroid trajectory, or the net displacement divided by the path length) is needed. Real metrics face the same question: which temporal properties do the features encode?

4. **Choose an architecture.** You must build an image-to-video model for product photos (a slow 3-second camera orbit around an object) with a small video dataset and a strong pretrained image model. Choose between inflating the image model with factorized temporal layers and training a full-attention video transformer from scratch, and say how you would add camera control.

**Hint**

With little video and a strong image model, inflation with zero- or identity-initialized temporal layers reuses appearance knowledge and needs less data; camera motion is slow, so factorized attention's weakness with large motion matters less. Condition on the photo (concatenated latent plus an image embedding, as in Stable Video Diffusion) and add a camera module with per-pixel Plücker embeddings, trained on clips with diverse, known camera paths. Evaluate identity preservation of the product across frames, camera-path accuracy and a dynamic-degree measure.

5. **Evaluation plan.** A team claims its new model has "better temporal consistency" because its VBench subject-consistency score is higher. List three alternative explanations and the additional evidence you would ask for.

**Hint**

The model might move less (check dynamic degree and motion smoothness), might generate simpler or shorter clips (check prompts, clip length and frame rate), or might differ in frame quality that affects the feature extractor (check frame-wise quality). Ask for the full dimension table with the same VBench version, uncurated clips per prompt, a human pairwise study on consistency with motion held comparable, and FVD with a temporally sensitive feature extractor.

## References


1. U. Singer, A. Polyak, T. Hayes et al., "Make-A-Video: Text-to-Video Generation without Text-Video Data," arXiv:2209.14792, 2022. arXiv
2. Z. Yang, J. Teng, W. Zheng et al., "CogVideoX: Text-to-Video Diffusion Models with An Expert Transformer," *ICLR*, 2025. arXiv
3. Team Wan, A. Wang, B. Ai, B. Wen et al., "Wan: Open and Advanced Large-Scale Video Generative Models," arXiv:2503.20314, 2025. arXiv
4. A. Blattmann, T. Dockhorn, S. Kulal et al., "Stable Video Diffusion: Scaling Latent Video Diffusion Models to Large Datasets," arXiv:2311.15127, 2023. arXiv
5. J. Xing, M. Xia, Y. Zhang et al., "DynamiCrafter: Animating Open-Domain Images with Video Diffusion Priors," *ECCV*, LNCS, pp. 399–417, 2024. doi ·arXiv
6. J. Ho, T. Salimans, A. Gritsenko, W. Chan, M. Norouzi and D. J. Fleet, "Video Diffusion Models," *NeurIPS*, 2022. proceedings ·arXiv
7. A. Blattmann, R. Rombach, H. Ling et al., "Align your Latents: High-Resolution Video Synthesis with Latent Diffusion Models," *CVPR*, 2023. CVF
8. R. Henschel, L. Khachatryan, H. Poghosyan et al., "StreamingT2V: Consistent, Dynamic, and Extendable Long Video Generation from Text," *CVPR*, 2025. CVF
9. J. Z. Wu, Y. Ge, X. Wang et al., "Tune-A-Video: One-Shot Tuning of Image Diffusion Models for Text-to-Video Generation," *ICCV*, 2023. CVF
10. L. Khachatryan, A. Movsisyan, V. Tadevosyan et al., "Text2Video-Zero: Text-to-Image Diffusion Models are Zero-Shot Video Generators," *ICCV*, 2023. CVF
11. Z. Wang, Z. Yuan, X. Wang et al., "MotionCtrl: A Unified and Flexible Motion Controller for Video Generation," *SIGGRAPH 2024 Conference Papers*, 2024. doi ·arXiv
12. H. He, Y. Xu, Y. Guo et al., "CameraCtrl: Enabling Camera Control for Text-to-Video Generation," *ICLR*, 2025. arXiv
13. S. Yin, C. Wu, J. Liang et al., "DragNUWA: Fine-grained Control in Video Generation by Integrating Text, Image, and Trajectory," arXiv:2308.08089, 2023. arXiv
14. R. Rombach, A. Blattmann, D. Lorenz, P. Esser and B. Ommer, "High-Resolution Image Synthesis with Latent Diffusion Models," *CVPR*, 2022. CVF
15. L. Yu, J. Lezama, N. B. Gundavarapu et al., "Language Model Beats Diffusion — Tokenizer is Key to Visual Generation," *ICLR*, 2024. arXiv
16. W. Kong, Q. Tian, Z. Zhang et al., "HunyuanVideo: A Systematic Framework For Large Video Generative Models," arXiv:2412.03603, 2024. arXiv
17. A. Polyak, A. Zohar, A. Brown et al., "Movie Gen: A Cast of Media Foundation Models," arXiv:2410.13720, 2024. arXiv

18. Y. HaCohen, N. Chiprut, B. Brazowski et al., “LTX-Video: Realtime Video Latent Diffusion,” arXiv:2501.00103, 2024. arXiv
19. D. Kondratyuk, L. Yu, X. Gu et al., “VideoPoet: A Large Language Model for Zero-Shot Video Generation,” *ICML*, PMLR 235, 2024. PMLR ·arXiv
20. L. Yu, Y. Cheng, K. Sohn et al., “MAGVIT: Masked Generative Video Transformer,” *CVPR*, 2023. CVF
21. Y. Guo, C. Yang, A. Rao et al., “AnimateDiff: Animate Your Personalized Text-to-Image Diffusion Models without Specific Tuning,” *ICLR*, 2024. arXiv
22. Z. Zheng, X. Peng, T. Yang et al., “Open-Sora: Democratizing Efficient Video Production for All,” arXiv:2412.20404, 2024. arXiv
23. G. Bertasius, H. Wang and L. Torresani, “Is Space-Time Attention All You Need for Video Understanding?,” *ICML*, PMLR 139, 2021. PMLR
24. A. Arnab, M. Dehghani, G. Heigold, C. Sun, M. Lučić and C. Schmid, “ViViT: A Video Vision Transformer,” *ICCV*, 2021. CVF
25. X. Ma, Y. Wang, X. Chen et al., “Latte: Latent Diffusion Transformer for Video Generation,” *Transactions on Machine Learning Research*, 2025. arXiv
26. B. Lin, Y. Ge, X. Cheng et al., “Open-Sora Plan: Open-Source Large Video Generation Model,” arXiv:2412.00131, 2024. arXiv
27. J. Ho, W. Chan, C. Saharia et al., “Imagen Video: High Definition Video Generation with Diffusion Models,” arXiv:2210.02303, 2022. arXiv
28. O. Bar-Tal, H. Chefer, O. Tov et al., “Lumiere: A Space-Time Diffusion Model for Video Generation,” *SIGGRAPH Asia 2024 Conference Papers*, 2024. doi ·arXiv
29. Y. Lipman, R. T. Q. Chen, H. Ben-Hamu, M. Nickel and M. Le, “Flow Matching for Generative Modeling,” arXiv:2210.02747, 2022. arXiv
30. W. Peebles and S. Xie, “Scalable Diffusion Models with Transformers,” *ICCV*, 2023. CVF
31. S. Yin, C. Wu, H. Yang et al., “NUWA-XL: Diffusion over Diffusion for eXtremely Long Video Generation,” *ACL*, pp. 1309–1320, 2023. doi ·arXiv
32. H. Qiu, M. Xia, Y. Zhang et al., “FreeNoise: Tuning-Free Longer Video Diffusion via Noise Rescheduling,” *ICLR*, 2024. arXiv
33. R. Villegas, M. Babaeizadeh, P.-J. Kindermans et al., “Phenaki: Variable Length Video Generation From Open Domain Textual Description,” arXiv:2210.02399, 2022. arXiv
34. Z. Huang, Y. He, J. Yu et al., “VBench: Comprehensive Benchmark Suite for Video Generative Models,” *CVPR*, 2024. CVF
35. T. Yin, Q. Zhang, R. Zhang et al., “From Slow Bidirectional to Fast Autoregressive Video Diffusion Models,” *CVPR*, 2025. CVF
36. B. Chen, D. M. Monso, Y. Du, M. Simchowitz, R. Tedrake and V. Sitzmann, “Diffusion Forcing: Next-token Prediction Meets Full-Sequence Diffusion,” *NeurIPS*, 2024. proceedings ·arXiv
37. X. Huang, Z. Li, G. He, M. Zhou and E. Shechtman, “Self Forcing: Bridging the Train-Test Gap in Autoregressive Video Diffusion,” *NeurIPS*, 2025. arXiv
38. T. Unterthiner, S. van Steenkiste, K. Kurach, R. Marinier, M. Michalski and S. Gelly, “Towards Accurate Generative Models of Video: A New Metric & Challenges,” arXiv:1812.01717, 2018. arXiv
39. J. Carreira and A. Zisserman, “Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset,” *CVPR*, 2017. CVF

40. W. Kay, J. Carreira, K. Simonyan et al., “The Kinetics Human Action Video Dataset,” arXiv: 1705.06950, 2017. arXiv
41. K. Soomro, A. R. Zamir and M. Shah, “UCF101: A Dataset of 101 Human Actions Classes From Videos in The Wild,” technical report CRCV-TR-12-01, arXiv:1212.0402, 2012. arXiv
42. S. Ge, A. Mahapatra, G. Parmar, J.-Y. Zhu and J.-B. Huang, “On the Content Bias in Fréchet Video Distance,” *CVPR*, 2024. CVF
43. L. Wang, B. Huang, Z. Zhao et al., “VideoMAE V2: Scaling Video Masked Autoencoders With Dual Masking,” *CVPR*, 2023. CVF
44. Z. Huang, F. Zhang, X. Xu et al., “VBench++: Comprehensive and Versatile Benchmark Suite for Video Generative Models,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 48, no. 3, pp. 3268–3285, 2026. doi ·arXiv
45. Y. Liu, X. Cun, X. Liu et al., “EvalCrafter: Benchmarking and Evaluating Large Video Generation Models,” *CVPR*, 2024. CVF
46. Z. Zheng, X. Peng, Y. Lou et al., “Open-Sora 2.0: Training a Commercial-Level Video Generation Model in $200k,” arXiv:2503.09642, 2025. arXiv
47. M. Bain, A. Nagrani, G. Varol and A. Zisserman, “Frozen in Time: A Joint Video and Image Encoder for End-to-End Retrieval,” *ICCV*, 2021. CVF ·arXiv
48. T.-S. Chen, A. Siarohin, W. Menapace et al., “Panda-70M: Captioning 70M Videos with Multiple Cross-Modality Teachers,” *CVPR*, 2024. CVF
49. A. Rössler, D. Cozzolino, L. Verdoliva, C. Riess, J. Thies and M. Nießner, “FaceForensics++: Learning to Detect Manipulated Facial Images,” *ICCV*, 2019. doi ·arXiv
50. P. Fernandez, H. Elsahar, I. Z. Yalniz and A. Mourachko, “Video Seal: Open and Efficient Video Watermarking,” arXiv:2412.09492, 2024. arXiv
51. A. Melnik, M. Ljubljanac, C. Lu, Q. Yan, W. Ren and H. Ritter, “Video Diffusion Models: A Survey,” *Transactions on Machine Learning Research*, 2024. arXiv
52. Z. Xing, Q. Feng, H. Chen et al., “A Survey on Video Diffusion Models,” arXiv:2310.10647, 2023. arXiv
53. Y. Liu, K. Zhang, Y. Li et al., “Sora: A Review on Background, Technology, Limitations, and Opportunities of Large Vision Models,” arXiv:2402.17177, 2024. arXiv
54. OpenAI, “What to know about the Sora discontinuation,” official service notice, 2026, accessed 5 November 2026. Notice.

# DLCV 12B • One-Step Generation

Learn what a one-step generator must predict, compare trajectory distillation, consistency, reflow and distribution matching, and measure the speed–quality trade with reproducible examples.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/one-step-generation/

## What you'll learn

- Explain why one local denoising direction may not reach a distant endpoint.
- Compare endpoint regression, consistency, reflow and distribution matching.
- Distinguish matching teacher trajectories from matching output distributions.
- Understand step-size-conditioned shortcuts and average-velocity targets.
- Report network evaluations, complete latency, alignment and diversity.

## The big picture

> **Explain like I'm five**
>
> A drawing teacher normally gives you many small instructions: move here, look again, correct the curve, add a detail. If you keep only the first instruction and make it much bigger, the drawing goes wrong. A fast student needs a different lesson: learn the whole move, learn where the teacher will finish, or learn how a finished drawing should look. Those lessons can all produce a picture quickly, but they are different kinds of learning.

A diffusion denoiser usually predicts a local correction at a particular noise level. A one-step generator must turn a random input into a complete sample. It therefore needs a **finite change**, not merely a more confident local correction. This chapter covers diffusion distillation, consistency models, rectified flows and shortcut/average-velocity models together. Some borrow a pretrained teacher; others train a generator directly. "One-step" describes inference, not a single common training algorithm.

Read ML 08: Diffusion Models and Flow Matching first for noising schedules, prediction targets and probability-flow equations. DLCV 12: Image Generation supplies the image-generation pipeline and its evaluation context. Here the main question is narrower: **what must change when the iterative part is compressed into one or a few calls?** Super-resolution applies these ideas to an observed image rather than unconditional noise; that application has a separate one-step SR chapter.

For a first pass, read the numerical example, the family table and the evaluation checklist. Then choose a detailed paper introduction. The equations distinguish training objectives; understanding every derivation is not required to understand the differences between the families.

### Why taking one large step is different

**Plain version.** A local tangent gives the current direction, not the displacement to a distant endpoint. Accurate local predictions can still yield a poor one-step jump.

Write a probability-flow or flow-matching ordinary differential equation as

$$\frac{dx_t}{dt} = v(x_t, t), \qquad x_{t+h} \approx x_t + h\, v(x_t, t).$$

Here $x_t \in \mathbb{R}^d$ is the current image or latent vector, $t$ is continuous time, $v$ the instantaneous velocity, and $h$ a signed time increment. The approximation on the right is an Euler step. For a smooth field, its local error depends on how $v$ changes with time and position along the path. Making $h$ large discards those changes. A more accurate neural estimate of the same instantaneous field does not automatically repair this integration error.

There are three separate error sources: **model error**, because a learned field differs from its target; **numerical error**, because a finite-step solver approximates the learned continuous dynamics; and **distribution mismatch**, because the terminal noise, conditioning or resolution at inference differs from training. Improving one does not eliminate the others. Probability-flow theory [1] describes an ideal connection between marginal distributions; it is not a certificate that any finite neural sampler exactly follows it.

The same caution applies to flow matching [2]. Even when training pairs are interpolated along straight lines, the marginal velocity at a location averages directions from several possible pairs. Its resulting trajectories can bend. The straightness of the sampled training interpolation is therefore not the same as straightness of the learned marginal flow.

### A toy where the score is known exactly

The following experiment removes neural-network error. Its target is an equally weighted mixture of four 2-D Gaussians. After adding Gaussian noise of standard deviation $\sigma$, each component variance is $a^2 + \sigma^2$, where $a = 0.12$ is the component standard deviation. The exact score is

$$s(x, \sigma) = -\frac{1}{a^2 + \sigma^2} \sum_{j=1}^{4} w_j(x, \sigma)(x - \mu_j).$$

Here $\mu_j$ is component $j$'s center and $w_j$ is its normalized posterior mixture weight at $x$. For this additive-noise family, the probability-flow equation parameterized by noise scale is $dx/d\sigma = -\sigma s(x, \sigma)$ [3]. We integrate from large to small $\sigma$.

The initial samples are drawn from the **exact noisy mixture**, not assumed to be standard Gaussian. This makes the numerical comparison controlled. A real generator must also handle how well its chosen starting distribution matches the training boundary. We compare coarse Euler endpoints with an accurate adaptive ODE solution from the same initial points; the error is not an image-quality metric or a trained-model benchmark.

```
import numpy as np
```

```
from scipy.integrate import solve_ivp

centers = np.array([[-1.7, 0.], [1.7, 0.], [0., -1.7], [0., 1.7]])
base_var = 0.12**2
sigma_start, sigma_end = 3.0, 0.04

def score(x, sigma):
    """Exact score of the equally weighted Gaussian mixture after Gaussian noising."""
    var = base_var + sigma**2
    delta = x[:, None, :] - centers[None, :, :]
    logits = -(delta**2).sum(-1) / (2 * var)
    weights = np.exp(logits - logits.max(1, keepdims=True))
    weights /= weights.sum(1, keepdims=True)
    return -(weights[:, :, None] * delta).sum(1) / var

def euler(initial, steps):
    x = initial.copy()
    grid = np.linspace(sigma_start, sigma_end, steps + 1)
    for s0, s1 in zip(grid[:-1], grid[1:]):
        x += (s1 - s0) * (-s0 * score(x, s0))
    return x

rng = np.random.default_rng(7)
initial = centers[rng.integers(4, size=512)] + np.sqrt(base_var + sigma_start**2) *
↪  rng.normal(size=(512, 2))
shape = initial.shape
solution = solve_ivp(lambda sigma, flat: (-sigma * score(flat.reshape(shape),
↪  sigma)).ravel(),
                     (sigma_start, sigma_end), initial.ravel(), rtol=1e-9, atol=1e-10,
                      ↪  dense_output=True)
reference = solution.y[:, -1].reshape(shape)
for steps in [1, 2, 4, 8, 16, 32, 64]:
    estimate = euler(initial, steps)
    rmse = np.sqrt(np.mean((estimate - reference)**2))
    print(f'{steps:2d} evaluations: endpoint RMSE {rmse:.4f}')
```

```
 1 evaluations: endpoint RMSE 0.8098
 2 evaluations: endpoint RMSE 0.5686
 4 evaluations: endpoint RMSE 0.3011
 8 evaluations: endpoint RMSE 0.1224
16 evaluations: endpoint RMSE 0.0316
32 evaluations: endpoint RMSE 0.0164
64 evaluations: endpoint RMSE 0.0085
```

These are our seeded numerical results. One Euler evaluation has a large endpoint error despite using the exact score. More evaluations follow the curved flow more accurately. The adaptive reference is itself a numerical solution, with tight tolerances; it is not an analytic closed-form transport map.

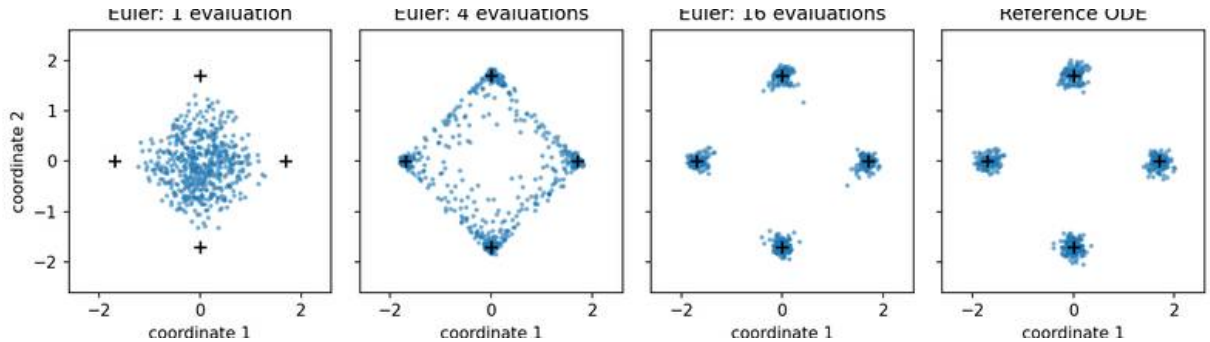



*Figure 1 — Reducing evaluations without changing the prediction target introduces integration error, even with an exact score. These synthetic samples are our experiment, not outputs from an image model.*

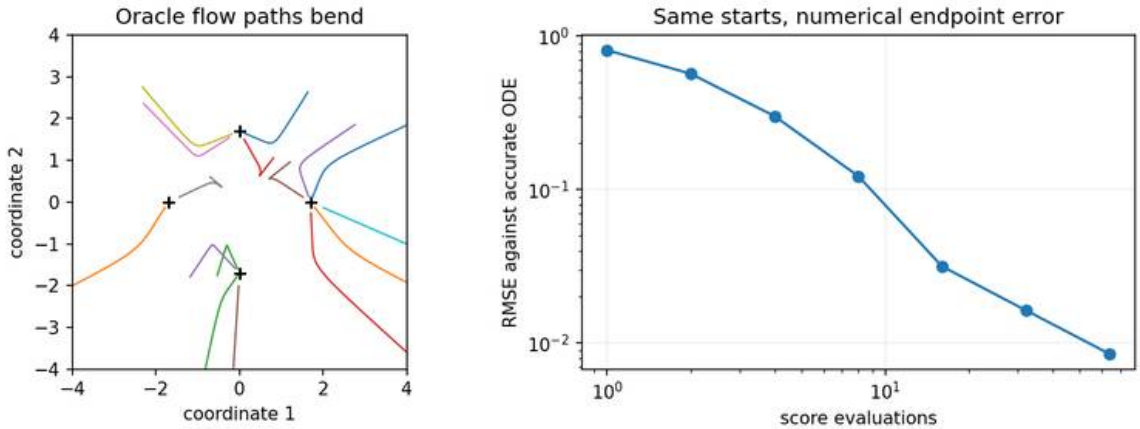



*Figure 2 — A tangent and a finite displacement are different. Distillation and large-jump objectives try to learn information that the one-step Euler baseline does not have.*

## A map of acceleration methods

**Plain version.** Separate faster integration of an existing model from training a new endpoint, jump or distribution predictor.

| Family | What is learned or changed | Needs a separate teacher? | Main constraint |
|---|---|---|---|
| Faster solver | A better numerical update for an existing field | Uses the existing field | Few-step accuracy still depends on the field and schedule |
| Progressive/trajectory distillation | Several teacher moves compressed into one student move | Yes | Student follows the selected teacher mapping |
| Consistency models | The same clean endpoint from different locations on one path | Distillation: yes; direct training: no | Accurate endpoint agreement across noise levels |

| Family | What is learned or changed | Needs a separate teacher? | Main constraint |
|---|---|---|---|
| Reflow and endpoint distillation | A more regular noise–image coupling, then a direct map | Usually a pretrained model for large image applications | Straightening and map approximation are separate stages |
| Adversarial distillation | Realistic outputs plus teacher-derived guidance | Yes in ADD/LADD | Discriminator stability and diversity |
| Distribution matching | Student output distribution agrees with a teacher-defined target | Yes | An auxiliary student score must track a changing distribution |
| Shortcut / average-velocity learning | Finite-interval moves conditioned on their interval | Can train from scratch | Self-target accuracy, time conditioning and boundary behavior |

The table follows the methodological separation in diffusion-distillation surveys [4], with direct-training families added because this chapter's scope is one-step generation rather than distillation alone. Categories overlap. DMD2 combines distribution matching and an adversarial objective; InstaFlow combines reflow and endpoint distillation. A model's training losses are more informative than assigning it one permanent label.

DDIM [5] and DPM-Solver [6] are useful baselines before training a student. They reduce discretization error per evaluation but do not, by themselves, learn a complete endpoint map. A comparison with an inefficient sampler can exaggerate a distilled model's speed advantage. Conversely, a solver that works well at moderate step counts may still fail when forced into one evaluation.

### Compressing teacher trajectories

**Plain version.** A teacher assigns a destination to each noisy input. Endpoint regression trains the student to reproduce that chosen mapping directly.

### Progressive distillation

Progressive Distillation [7] repeatedly trains one student step to match two deterministic teacher steps. After training, the student becomes the teacher and the sampling budget is halved again. This gives a curriculum of larger jumps instead of asking a fresh network to approximate the whole map immediately.

Let a deterministic teacher take $z_t$ through an intermediate time to $z_s$. A DDIM-style single update based on clean prediction $\hat{x}$ is

$$z_s = \frac{\sigma_s}{\sigma_t} z_t + \left( \alpha_s - \frac{\sigma_s}{\sigma_t} \alpha_t \right) \hat{x}.$$

Here $z_t, z_s$ are noisy states, and $\alpha_t, \sigma_t$ are the signal and noise schedule coefficients. Solving this affine equation for $\hat{x}$ gives the student target that reproduces the teacher's two-step endpoint. The target depends on the selected teacher mapping and starting state. It differs from ordinary clean-image denoising, whose squared-error optimum can average several compatible clean images.

This approach transfers work to training: many teacher evaluations produce targets while inference becomes cheaper. It can accumulate approximation error across stages. A student may need a particular schedule and output parameterization, so loading it into an arbitrary sampler is not equivalent to the trained procedure.

*Online only: Progressive Distillation: two teacher moves in one student move (ICLR 2022) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-generation/papers/progressive-distillation-two-teacher-moves-in-one-student-move-iclr-2022/*

### Coupling is part of the target

There are many valid maps from a noise distribution to an image distribution. A trajectory teacher chooses one **coupling**: which output belongs to which random input. Matching that pair is stronger than matching only the output distribution.

A small example makes the distinction concrete. Consider a random input $z$ whose sign can select one of two target modes. The maps $G_1(z) = \text{sign}(z)$ and $G_2(z) = -\text{sign}(z)$ have the same balanced output distribution when $z$ is symmetric. They have opposite samplewise correspondence. A pairwise regression loss strongly penalizes replacing one with the other even though neither changes the marginal distribution.

This is why one-step generation is not intrinsically "too deterministic." A deterministic map of a **random latent input** can generate diverse outputs. Collapse occurs when the map ignores parts of that input or covers too few target modes. Averaging endpoints from unrelated random seeds during regression creates a different problem: the average can lie between modes and be an implausible sample.

### Endpoint agreement with consistency models

**Plain version.** Instead of learning the next small correction, learn where a path ends. Two states on the same path should give the same answer.

A consistency function satisfies

$$f(x_t, t) = f(x_s, s) = x_\varepsilon$$

for states $x_t, x_s$ on a shared probability-flow trajectory, with $x_\varepsilon$ its near-clean endpoint at a small positive boundary time $\varepsilon$. A neural parameterization enforces the boundary condition through

$$f_\theta(x, t) = c_{\text{skip}}(t)x + c_{\text{out}}(t)F_\theta(x, t),$$

where $F_\theta$ is a learned network, $c_{\text{skip}}(\varepsilon) = 1$, and $c_{\text{out}}(\varepsilon) = 0$. Thus the boundary returns its input instead of allowing the same constant output everywhere. Boundary anchoring and proper training pairs matter: "make two predictions agree" alone has trivial solutions.

**Consistency distillation** uses a pretrained score model and a numerical solver to construct neighboring path states. The student output at the noisier state matches a stopped-gradient target-model output at the cleaner state. **Consistency training** learns from data without a separately

pretrained diffusion teacher, using nearby noise levels and a shared-noise construction. They share the endpoint interpretation but differ in their supervision [8]. Later improvements change weighting, noise-level sampling and target-model updates [9]; the original configuration should not be treated as the final form of every consistency model.

One-step sampling evaluates $f_\theta$ at a noisy start. Few-step sampling alternates endpoint prediction with adding controlled noise at a lower level and predicting again. This can improve quality, but it is not simply repeating an identical deterministic map on its own output. A checkpoint trained for particular noise levels needs the corresponding sampling schedule.

*Online only: Consistency Models: the same endpoint along a path (ICML 2023) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-generation/papers/consistency-models-the-same-endpoint-along-a-path-icml-2023/*

### Latent consistency and guidance

Latent Consistency Models (LCM) apply consistency distillation in a pretrained image autoencoder's latent space [10], [11]. This reduces spatial computation but retains the cost and reconstruction limits of encoding/decoding. The denoiser becomes an endpoint predictor for the guided latent probability flow.

Classifier-free guidance normally combines conditional and unconditional network predictions. LCM conditions its student on guidance strength and distills the guided flow, so the chosen behavior can be produced without two separate denoiser predictions at each student step. The guidance range seen during training is part of the model's usable interface. A guidance number in a distilled pipeline is not automatically interchangeable with the same number in the original pipeline: conventions and parameterizations differ.

LCM's skipping-step construction pairs states farther apart during training than an immediately adjacent grid interval. This improves the usefulness of the consistency target for finite jumps. It still relies on the teacher and training schedule, and one-step outputs can be worse than two- or four-step outputs. "Supports one-step inference" is not a promise that one step is always the best operating point.

*Online only: Latent Consistency Models: endpoints in a guided latent flow (2023) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-generation/papers/latent-consistency-models-endpoints-in-a-guided-latent-flow-2023/*

### Straightening before distilling

**Plain version.** A student learns a long move more easily if the teacher's route is less tangled. Reflow changes the route before endpoint regression compresses it.

Rectified Flow [12] trains an instantaneous velocity using interpolated endpoint pairs. A reflow round samples a noise point $z$, follows the current flow to $x = \Phi(z)$, and trains another flow on the **coupled** pairs $(z, x)$ instead of independently pairing arbitrary noise and data. The resulting interpolation target is

$$x_t = (1-t)z + tx, \qquad \mathcal{L}_{\text{reflow}} = \mathbb{E}\|v_\theta(x_t, t) - (x - z)\|_2^2.$$

Here time increases from noise at 0 to data at 1; $v_\theta$ is instantaneous velocity, and $\Phi$ is the earlier model's sampled endpoint map. This time direction differs from the noise-scale variable in the earlier toy. Writing the convention explicitly prevents a silent sign reversal.

Ideal reflow has useful distribution-preservation and transport-cost properties under its assumptions. A finite neural fit and approximate endpoint sampler introduce error. One reflow round does not guarantee globally straight learned trajectories. A network can still need several Euler evaluations after reflow; InstaFlow's experiments distinguish this from subsequent one-step endpoint distillation [13].

The endpoint student can minimize $d(G_\theta(z, c), \Phi_{\text{reflow}}(z, c))$, where $c$ is a text condition and $d$ a selected reconstruction/perceptual distance. This objective preserves the reflow teacher's pairing. Reflow can change that pairing while preserving the ideal output marginal; endpoint regression then approximates the changed map. They solve related but different problems.

*Online only: InstaFlow: reflow, then endpoint distillation (ICLR 2024) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-generation/papers/instaflow-reflow-then-endpoint-distillation-iclr-2024/*

### Adding a realism signal

**Plain version.** Matching a teacher endpoint can reproduce its errors. A discriminator supplies a second signal: does the student output look like the target data? This signal helps appearance but needs careful stabilization and evaluation of diversity.

Adversarial Diffusion Distillation (ADD) combines a pretrained student, a frozen diffusion teacher and a discriminator [14]. The student generates from selected noisy inputs. A diffusion-derived distillation term pulls its output toward a teacher denoising target, while an adversarial term rewards realistic appearance in pretrained feature space. The teacher and discriminator are training-time tools; neither must execute as an iterative teacher sampler during deployment.

For a generic adversarial formulation, a generator loss contains $-\mathbb{E}[D_\phi(G_\theta(z, c), c)]$, where $D_\phi$ gives a realness score and $G_\theta$ the student output. The discriminator is trained to distinguish real and generated samples under its own loss and regularization. This scalar is a learned training signal, not a calibrated probability that an image is truthful or prompt-correct.

ADD supports refinement across a small set of trained noise levels. LADD moves adversarial distillation into latent diffusion feature representations, reducing the need to decode pixels and run a separate large image feature encoder during training [15]. The complete computational trade includes the teacher, discriminator and feature extraction during training, as well as the final decoder during inference.

Adversarial and diffusion losses do not guarantee diversity automatically. Inspect whether different random seeds yield different plausible compositions, whether uncommon prompt concepts survive, and whether apparent sharpness comes from repeated texture. A low FID can coexist with poor conditional coverage. Training stability, real-data access and how prompts condition the discriminator belong in any comparison.

*Online only: Adversarial Diffusion Distillation: a teacher and a discriminator (ECCV 2024) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-generation/papers/adversarial-diffusion-distillation-a-teacher-and-a-discriminator-eccv-2024/*

### Matching distributions rather than individual endpoints

**Plain version.** A student can be rewarded for producing the right collection of images without being forced to copy the teacher's exact image for every noise seed. That freedom can help, but the distribution of the student must be estimated while it keeps changing.

DMD learns a one-step generator and uses two diffusion score estimates [16]. One fixed model describes the teacher-defined target distribution. A second model learns the distribution of the student's current outputs. At a common noise level, their score difference supplies a direction for moving generated samples toward the target.

Let $x = G_\theta(z, c)$ and $x_t = \alpha_t x + \sigma_t \epsilon$. A representative update has the form

$$\nabla_\theta \mathcal{L} \approx \mathbb{E}\left[ w(t)\alpha_t \left( s_{\text{student}}(x_t, t, c) - s_{\text{target}}(x_t, t, c) \right) \frac{\partial G_\theta(z, c)}{\partial \theta} \right].$$

Here $z$ is a random generator input, $c$ the condition, $\epsilon$ fresh Gaussian noise, $\alpha_t, \sigma_t$ the noising coefficients, $s = \nabla_x \log p_t(x \mid c)$ a score, and $w(t)$ a training weight. The equation is shorthand for the vector–Jacobian product back to generator parameters. Score estimates and time weighting make this a practical approximation; it is not an exact, directly evaluated image-density KL divergence.

Why include the student score? Following only a target score can push every sample toward already attractive modes, reducing diversity. The difference accounts for where the student already has probability mass. Why add noise? Clean generated and target distributions may have little overlapping support, while noising supplies a smoother comparison space. Why maintain a second diffusion network? The student output distribution changes after each generator update, so its score is not supplied by the fixed teacher.

Original DMD also uses a regression loss on teacher-generated noise–image pairs to stabilize learning and preserve coverage. This means the original method is not completely free of trajectory targets. SwiftBrush uses variational score-distillation ideas for text-to-image generation, while SiD derives score-identity-based objectives [17], [18]. These related methods should be compared through their actual objectives, weighting and data requirements rather than treated as identical implementations of one equation.

### What DMD2 changes

DMD2 removes the offline endpoint-regression dataset and stabilizes the changing student-score estimate by updating it more often than the generator [19]. It also adds adversarial learning from real images to address errors in a teacher-defined target. The word "real" in a teacher-score diagram should therefore be read carefully: the fixed teacher's score is an approximation, not direct access to the true data density.

For multi-step students, DMD2 trains on intermediate inputs produced by the student's own sampling procedure. This reduces mismatch between teacher-noised training inputs and states the student actually visits. The method's four-step large-model experiments are not evidence that an identically configured one-step large model has the same score. Steps, backbone, guidance and benchmark protocol must stay attached to every table row.

*Online only: DMD: matching a teacher-defined distribution (CVPR 2024) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-generation/papers/dmd-matching-a-teacher-defined-distribution-cvpr-2024/; DMD2: tracking the student and using real data (NeurIPS 2024) — https://blog.shyandram.dev/en/learn/deep-learning/one-step-generation/papers/dmd2-tracking-the-student-and-using-real-data-neurips-2024/*

### Learning finite jumps directly

**Plain version.** A network can learn "move this far in time" rather than only "move in this direction now." This changes its input and its target, so direct training can support large jumps without a separate teacher checkpoint.

### Shortcut models

A shortcut model predicts a normalized displacement $s_\theta(x, t, d)$ for a requested interval $d$ [20]. Its update is $x' = x + d\, s_\theta(x, t, d)$. At $d = 0$, training is grounded in a flow-matching velocity target. For longer intervals, one jump should agree with two consecutive shorter jumps:

$$s(x, t, 2d) = \tfrac{1}{2}\left[s(x, t, d) + s(x + d\, s(x, t, d), t + d, d)\right].$$

Here $t$ is the current time, $d$ the shorter interval, and both terms use the same conditional generation task. The second term is evaluated at the **updated position and time**. A stopped-gradient target built from the shorter jumps trains the longer jump. Replacing the updated time by the original time would be a different objective.

Flow supervision prevents arbitrary self-consistency from being the only signal. The combined method can train in one phase, mixing local supervised targets with bootstrapped finite jumps. "No separate teacher" does not mean no self-targets, no training compute or no approximation error. Large jumps remain limited by network capacity and the accuracy of their bootstrapped targets.

### Average velocity with MeanFlow

MeanFlow distinguishes an interval's average velocity from its endpoint's instantaneous velocity [21]. In a data-to-noise time convention, define

$$u(z_t, r, t) = \frac{1}{t - r} \int_r^t v(z_\tau, \tau)\, d\tau, \qquad z_r = z_t - (t - r) u(z_t, r, t).$$

Here $r < t$, $z_t$ is a point on the flow, $v$ instantaneous velocity and $u$ average velocity along the same trajectory. The integral is not approximated by calling $v$ once. A learned $u$ aims to represent its effect directly.

Differentiating the defining relation along the path gives

$$u = v - (t - r)\,(\partial_t u + (\nabla_z u)v)\,.$$

The derivative is total along the changing state, with $r$ held fixed. The training construction uses a Jacobian–vector product and a stopped-gradient target, so automatic differentiation can form the derivative without materializing a full image-sized Jacobian. The direct-training formulation replaces unavailable marginal velocity targets through its flow-matching construction. This identity is more informative than calling the method "Euler with bigger steps": the learned quantity is different.

Both source time and destination time must be represented. At $r = t$ the average velocity approaches the instantaneous velocity. At the full noise-to-data interval, a single average-velocity evaluation supplies the intended whole displacement. Exact identities motivate the objective, while a finite trained network only approximates them. The paper's JVP ablation demonstrates how sensitive this construction is to differentiating with respect to the correct variables.

### Evaluating the complete trade-off

**Plain version.** A fast result is useful only if it still does the intended task. Evaluate the generated distribution, the conditioning, diversity and total cost together.

### Steps, evaluations and time

A sampling step is a state update. A network function evaluation (NFE) is a network call. A solver can use multiple calls per update, and ordinary classifier-free guidance can require conditional and unconditional predictions. Batching those predictions may produce one API call but still process twice the examples. Report the convention explicitly; "one call" and "one example's worth of denoiser compute" are not always the same.

Complete latency includes text encoding, condition extraction, denoising/generation, decoding and any postprocessing. Separate initialization and model loading from steady-state inference. Fix hardware, precision, resolution, batch size and synchronization; otherwise a claimed speedup can mix method improvements with system changes. Memory use also matters: a single large generator may exceed the deployment budget even though it runs only once.

If guidance has been baked into the student, count its absence as part of the trained system's speed advantage, but also report the usable guidance range. If a prompt embedding is cached, say so. Comparing cached student text processing with uncached teacher processing does not measure only distillation.

### Quality, alignment and diversity

FID compares means and covariances of image-feature distributions [22]. It does not compare corresponding image pairs and does not directly assess whether a requested relation is satisfied. The feature extractor, reference split, sample count and resizing affect the score. Small prompt subsets are not interchangeable with larger evaluation sets; FID-5k and FID-30k are different protocols.

CLIP-based similarity scores compare image and text embeddings [23]. Implementations differ in normalization, scaling and backbone. A score reported as 0.3 in one table and 30 in another may reflect scaling rather than a hundredfold quality difference. GenEval tests object, count, color and relational requirements using a structured evaluation protocol [24]. Such tests complement, rather than replace, human judgments and distribution metrics.

Diversity should be tested across prompts and across seeds for one prompt. A generator can cover many broad classes but repeatedly draw one composition for each condition. Conversely, a model can have high visual variation because it ignores the prompt. Evaluate the two together. Retaining a random input does not prove that the learned map uses it effectively.

### Reading benchmark tables safely

A method may be tested on CIFAR-10, class-conditional ImageNet or text-conditioned COCO prompts [25], [26], [27]. Their scores answer different questions. A smaller-resolution unconditional result does not establish high-resolution prompt-following quality. Nor does a four-step result establish a one-step operating point.

For a useful comparison, record: the target task; backbone and initialization; training data and teacher; resolution; guidance convention; NFE; full latency and memory; distribution quality; conditional alignment; diversity; and uncertainty from finite samples. Use each paper's own controlled ablations for claims about individual components. Cross-paper tables can show reported operating points, but cannot isolate causality when data, architecture and training budgets differ.

### Failure modes and choosing a family

**Plain version.** Acceleration moves work and assumptions; it does not remove them. Choose the objective according to what must be preserved.

- **Need the same endpoint for a given seed?** Trajectory or endpoint regression is a natural starting point. It imposes a specific coupling and can copy teacher artifacts.
- **Need a flexible step budget?** Consistency, shortcut and interval-conditioned models explicitly support more than one budget, but require the trained sampling procedure.
- **Need a new output distribution rather than exact teacher pairing?** Distribution matching provides that freedom. Watch auxiliary-score lag, mode dropping and training stability.
- **Need strong perceptual appearance?** Adversarial terms can help, but evaluate diversity and conditioning rather than judging sharpness alone.

- **Need no pretrained teacher?** Direct consistency, shortcut or average-velocity training can fit that constraint. Compare complete training cost; removing a teacher checkpoint is not the same as making training inexpensive.

A one-step checkpoint can inherit the teacher's biases, lose rare concepts, overfit an evaluation prompt set, or fail when controls are changed after distillation. Distribution objectives and self-targets have their own failure modes: inaccurate student scores, collapsed targets, poor boundary conditioning and train–test input mismatch. A model trained on a narrow time or guidance range should not be treated as a universal replacement for its teacher.

### Modern view

**Plain version.** The field increasingly asks which object should be learned for a small inference budget, rather than assuming every improvement is a better approximation of the same denoising step.

Fan et al.'s distillation survey [4] organizes methods into fidelity/output objectives, trajectory methods and adversarial methods. Its taxonomy is useful for identifying what supervises a student and which parts of a teacher are retained. It does not imply that the categories are disjoint: DMD2 uses both distribution and adversarial signals. The broader diffusion survey by Yang et al. [28] places efficient sampling alongside model formulation and applications. Lipman et al.'s flow-matching guide [2] supplies a complementary transport view: local velocity, endpoint coupling and numerical integration are separate design choices.

From 2023 through 2025, consistency distillation, reflow, adversarial distillation and distribution matching produced different routes to few-step image generation. Shortcut models and MeanFlow show a separate direct-training route to finite jumps [20], [21]. Their class-conditional benchmark results should not be silently generalized to large text-to-image systems. The 2026 DMAD preprint explores adversarial distribution matching as an alternative to maintaining a separately updated student-score model [29]; its broader visual-generation setting and preprint status matter when comparing it with established image benchmarks.

Open problems include preserving rare modes and compositional control, maintaining editable conditioning after distillation, measuring quality beyond feature-distribution summaries, reducing training memory and compute, and transferring a one-step design across resolutions or domains. Speed alone is not a complete objective. For restoration, a further requirement appears: generated details should remain consistent with a particular observation. The one-step SR chapter applies this chapter's ideas under that additional constraint.

### Key takeaways

- A finite jump requires an endpoint or displacement target, not just a local direction used once.
- Separate solver error, learned-model error and train–test mismatch.
- Trajectory regression preserves a chosen coupling; distribution matching can learn a different one.

- Consistency needs boundary anchors; reflow changes coupling before compression into fewer steps.
- Distribution, adversarial and average-velocity objectives provide different supervision and estimation challenges.
- Compare full pipeline latency and network evaluations alongside quality, prompt adherence and diversity.

## Exercises

1. **Oracle versus solver error.** In the mixture example, compare a nonuniform noise-scale grid with the uniform grid at the same number of score evaluations. Where should more points be placed, and does that choice depend on the starting scale?

   **Hint**

   Inspect where the trajectories bend and where component assignments become sharp. Compare endpoint RMSE from the same initial samples. A better grid can improve a finite-step solver without changing its score. Avoid selecting the grid using a separate set of initial points and then calling its improvement universal.

2. **Endpoint or distribution?** For symmetric Gaussian $z$, compare $G_1(z) = \operatorname{sign}(z)$ and $G_2(z) = -\operatorname{sign}(z)$. Compute their expected squared pairwise difference and their output probabilities. Which objective can distinguish them?

   **Hint**

   Except on a probability-zero boundary, their values differ by 2, so the squared pairwise difference is 4. Both assign probability one half to each output. An ideal marginal-distribution comparison cannot distinguish them; a paired regression target can.

3. **A broken shortcut.** In the two-short-jump equation, replace the second time $t + d$ by $t$. Explain why this is not the same consistency relation, even if the network receives the updated state.

   **Hint**

   The vector field depends on time as well as position. For the scalar field $v(x, t) = t$, the two smaller Euler directions differ solely because time advances. A network asked for the original time on the second jump is answering the wrong interval query.

4. **Count the whole pipeline.** A teacher uses 20 updates with two denoiser predictions per update. A student uses four updates with baked-in guidance. Both decode once, and each extracts a text embedding once. State the denoiser-evaluation ratio and explain why the wall-clock speedup need not equal it.

**Hint**

The denoiser evaluation counts are 40 and 4, a ratio of 10. Shared text encoding and decoding, batch efficiency, precision and memory behavior change the measured runtime ratio. Include those fixed costs in a simple timing model before predicting a deployment speedup.

5. **Diagnose a suspicious improvement.** A student has lower FID but worse prompt-relation accuracy and nearly identical outputs across seeds. Propose three checks before accepting that it preserves its teacher's generation quality.

**Hint**

Inspect per-prompt seed diversity, evaluate object and relation constraints on a held-out prompt set, and repeat distribution evaluation with a matched sample count and reference split. A favorable aggregate feature statistic does not settle conditional coverage.

## References


1. Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, B. Poole, "Score-Based Generative Modeling through Stochastic Differential Equations," *ICLR*, 2021. paper
2. Y. Lipman, M. Havasi, P. Holderrieth, et al., "Flow Matching Guide and Code," *arXiv tutorial*, 2024. paper
3. T. Karras, M. Aittala, T. Aila, S. Laine, "Elucidating the Design Space of Diffusion-Based Generative Models," *NeurIPS*, 2022. source
4. X. Fan, Z. Wu, H. Wu, L. Cao, "A Survey on Pre-Trained Diffusion Model Distillations," *arXiv preprint*, *version 2*, 2025. paper
5. J. Song, C. Meng, S. Ermon, "Denoising Diffusion Implicit Models," *ICLR*, 2021. paper
6. C. Lu, Y. Zhou, F. Bao, J. Chen, C. Li, J. Zhu, "DPM-Solver: A Fast ODE Solver for Diffusion Probabilistic Model Sampling in Around 10 Steps," *NeurIPS*, 2022. paper
7. T. Salimans, J. Ho, "Progressive Distillation for Fast Sampling of Diffusion Models," *ICLR*, 2022. paper
8. Y. Song, P. Dhariwal, M. Chen, I. Sutskever, "Consistency Models," *ICML*, 2023. paper
9. Y. Song, P. Dhariwal, "Improved Techniques for Training Consistency Models," *ICLR*, 2024. paper
10. S. Luo, Y. Tan, L. Huang, J. Li, H. Zhao, "Latent Consistency Models: Synthesizing High-Resolution Images with Few-Step Inference," *arXiv preprint*, 2023. paper
11. R. Rombach, A. Blattmann, D. Lorenz, P. Esser, B. Ommer, "High-Resolution Image Synthesis with Latent Diffusion Models," *CVPR*, 2022. source
12. X. Liu, C. Gong, Q. Liu, "Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow," *ICLR*, 2023. paper
13. X. Liu, X. Zhang, J. Ma, J. Peng, Q. Liu, "InstaFlow: One Step is Enough for High-Quality Diffusion-Based Text-to-Image Generation," *ICLR*, 2024. paper
14. A. Sauer, D. Lorenz, A. Blattmann, R. Rombach, "Adversarial Diffusion Distillation," *ECCV*, 2024. paper

15. A. Sauer, F. Boesel, T. Dockhorn, A. Blattmann, P. Esser, R. Rombach, "Fast High-Resolution Image Synthesis with Latent Adversarial Diffusion Distillation," *SIGGRAPH Asia Conference Papers*, 2024. paper ·published version
16. T. Yin, M. Gharbi, R. Zhang, E. Shechtman, F. Durand, W. T. Freeman, T. Park, "One-step Diffusion with Distribution Matching Distillation," *CVPR*, 2024. paper
17. T. H. Nguyen, A. Tran, "SwiftBrush: One-Step Text-to-Image Diffusion Model with Variational Score Distillation," *CVPR*, 2024. paper
18. M. Zhou, H. Zheng, Z. Wang, M. Yin, H. Huang, "Score identity Distillation: Exponentially Fast Distillation of Pretrained Diffusion Models for One-Step Generation," *ICML*, 2024. paper
19. T. Yin, M. Gharbi, T. Park, R. Zhang, E. Shechtman, F. Durand, W. T. Freeman, "Improved Distribution Matching Distillation for Fast Image Synthesis," *NeurIPS*, 2024. paper
20. K. Frans, D. Hafner, S. Levine, P. Abbeel, "One Step Diffusion via Shortcut Models," *ICLR*, 2025. paper
21. Z. Geng, M. Deng, X. Bai, J. Z. Kolter, K. He, "Mean Flows for One-step Generative Modeling," *NeurIPS*, 2025. paper
22. M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, S. Hochreiter, "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium," *NeurIPS*, 2017. source
23. A. Radford, J. W. Kim, C. Hallacy, et al., "Learning Transferable Visual Models From Natural Language Supervision," *ICML*, 2021. paper
24. D. Ghosh, H. Hajishirzi, L. Schmidt, "GenEval: An Object-Focused Framework for Evaluating Text-to-Image Alignment," *NeurIPS Datasets and Benchmarks*, 2023. paper
25. A. Krizhevsky, V. Nair and G. Hinton, "CIFAR-10 and CIFAR-100 datasets," official dataset page; companion technical report: A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," 2009. dataset and report
26. J. Deng, W. Dong, R. Socher, L. Li, Kai Li, Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," *CVPR*, 2009. doi
27. T. Lin, M. Maire, S. Belongie, et al., "Microsoft COCO: Common Objects in Context," *ECCV*, 2014. paper
28. L. Yang, Z. Zhang, Y. Song, et al., "Diffusion Models: A Comprehensive Survey of Methods and Applications," *ACM Computing Surveys*, 2023. paper
29. Z. Yu, J. Yuan, H. Yang, et al., "DMAD: Distribution Matching as Adversarial Distillation for Fast Visual Generation," *arXiv preprint*, 2026. paper
30. R. Zhang, P. Isola, A. A. Efros, E. Shechtman and O. Wang, "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric," in *Proc. IEEE/CVF CVPR*, 2018, pp. 586–595. doi ·arXiv
31. C. Schuhmann, R. Beaumont, R. Vencu, et al., "LAION-5B: An open large-scale dataset for training next generation image-text models," *NeurIPS Datasets and Benchmarks*, 2022. paper

# DLCV 13 • World Models: Video, Latent Dynamics and Planning

What a world model is, how model-based reinforcement learning uses one (World Models, the Dreamer line, MuZero), why video generators are proposed as world simulators, how latent dynamics and JEPA-style predictive representations work, and how to evaluate all of them, with complete CPU experiments.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/world-models/

## What you'll learn

- Distinguish observations, hidden states, beliefs and actions.
- Compare online planning, imagined policy learning and value-equivalent search.
- Separate plausible video prediction from useful action-conditioned simulation.
- Understand recurrent and feature-predictive latent dynamics.
- Measure multi-step errors and detect planners exploiting model mistakes.

## The big picture

Tracking asks where an object went. A world model asks what could happen next, and especially what would happen **if an agent took a particular action**. A robot approaching a cup needs more than an object label: it needs to predict how its gripper and the cup will move, which outcomes are uncertain, and which action is likely to reach its goal.

> **Explain like I'm five**
>
> Before pushing a toy car, you imagine where it will go. You can imagine a gentle push and a strong push, pick the better one, push, and then look again to see what really happened. A world model is a learned version of that imagination. A planner is the part that tries out pushes in imagination and picks one. Looking again after each push is what keeps small imagination mistakes from piling up.

The term "world model" covers several research traditions. Ding et al. organize the literature around two functions: building internal representations that capture how the world works, and predicting future states to simulate and guide decisions [1]. This chapter uses an operational definition: **a world model is a learned model that represents enough of an environment to predict relevant changes under specified conditions, usually including the agent's own actions.** Its usefulness depends on the task. A model for rendering a scene, a model for learning representations, and a model for controlling a robot may need very different information.

## Observations, states, beliefs and actions

**Plain version.** An image reveals only part of the environment. Memory or belief must account for hidden objects and unobserved motion.

Let $s_t$ denote the underlying environment state at time $t$, $o_t$ the observation and $a_t$ the executed action:

$$s_{t+1} \sim p(s_{t+1} \mid s_t, a_t), \qquad o_t \sim p(o_t \mid s_t).$$

The first distribution is the **transition** (dynamics) model; the second is the **observation** model. The state might contain object positions, velocities and contact conditions. An image shows only a projection, with occlusions and sensor noise; Szeliski covers the geometry and motion side of this observation process [3]. Two identical images can have different futures: a ball at the same position might be moving left or right.

A learned model therefore builds a representation from history:

$$z_t = E_\theta(o_{0:t}, a_{0:t-1}), \qquad p_\theta(z_{t+1} \mid z_t, a_t),$$

where $E_\theta$ is an encoder with parameters $\theta$, $o_{0:t}$ is the sequence of observations up to $t$, and $p_\theta(z_{t+1} \mid z_t, a_t)$ is a learned latent transition. An encoder does not automatically make $z_t$ a sufficient state. If it discards velocity, hidden objects or a relevant material property, the transition model cannot recover that information. Recurrent memory or a short video context helps, but must be tested under the actual occlusions and delays.

When the state is only partially observed, the principled object is a **belief**, a probability distribution over possible states given the history. Bishop's treatment of sequential latent-variable models (hidden Markov models and linear dynamical systems) gives the probabilistic background [2]. A compact latent vector can approximate useful aspects of this distribution; it need not map one-to-one onto human-readable physical variables.

### A prediction factorization

Predicting the future splits into three questions: where are we now, how do things change under the actions, and what would the camera see?

For a proposed action sequence, the probability of future observations factorizes as

$$p_\theta(o_{t+1:t+H} \mid o_{\le t}, a_{0:t+H-1}) = \int p_\theta(z_t \mid o_{\le t}, a_{<t}) \prod_{k=t}^{t+H-1} p_\theta(z_{k+1} \mid z_k, a_k)\, p_\theta(o_{k+1} \mid z_{k+1})\, dz_{t:t+H},$$

where $H$ is the prediction horizon. The three factors are **inference** (what is the current latent state given the past?), **transition** (how does it evolve under the actions?) and **rendering** (what would the camera see?). A planner may work directly with $z$ and skip the pixel decoder if its goal and cost are defined in latent space. Much of the design space in this chapter consists of choosing which of these factors to learn, and how.

## Model-based reinforcement learning

**Plain version.** A learned dynamics model lets an agent test actions in simulation, potentially reducing real trials when its predictions are useful.

### The setting

An RL agent interacts with an environment over discrete steps. At each step it observes, chooses $a_t$ from a **policy** $\pi(a_t \mid s_t)$, receives a reward $r_t$, and the environment moves on. The goal is to maximize the expected **return**

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k},$$

where $\gamma \in [0, 1)$ is a discount factor that weights near rewards more than distant ones. The **value** $V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s]$ is the expected return from state $s$ when following $\pi$.

**Model-free** methods learn a policy or value directly from real transitions. **Model-based** methods also learn a model of transitions and rewards and use it to reduce real interaction. The idea is old: Sutton's Dyna architecture (1991) already combined learning from real experience with planning on experience simulated by a learned model, using the same value updates for both [4]. Moerland et al. survey the field and organize it along two axes: how the dynamics model is learned (handling stochasticity, uncertainty, partial observability, temporal abstraction) and how planning and learning are integrated (where to start planning, how much budget to give it, how to plan, and how to feed the results back into acting and learning) [5].

### Three ways to use a model

Once a model is learned, there are three main ways to turn its predictions into decisions.

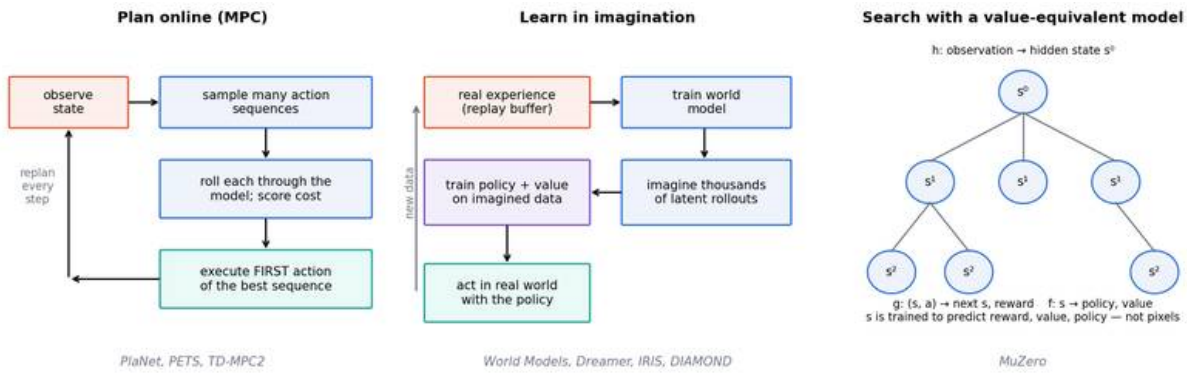



*Figure 1 — Three ways to use a learned model. The same model can serve more than one role; the difference is where the decision is computed.*

1. **Plan online (model-predictive control).** At every real step, search for a good action sequence by simulating candidates in the model, execute only the first action, observe, and repeat. PlaNet [7], PETS [29] and TD-MPC2 [11] work this way.
2. **Learn in imagination.** Train a policy (and usually a value function) on trajectories generated by the model, then act with that policy without searching at run time. World Models [6], the Dreamer line [8], [9], [10], IRIS [12] and DIAMOND [13] work this way.
3. **Search with a value-equivalent model.** Learn a model that predicts only what planning needs (rewards, values, policy), not observations, and run tree search in its latent space. MuZero [14] is the canonical example.

All three share one danger: the model is wrong somewhere, and an optimizer is very good at finding exactly those places. We return to this in the section on uncertainty and exploitation.

### Why errors compound

Each imagined step starts from the previous imagined step, so small prediction mistakes feed into the next prediction and can grow with every step.

Suppose the learned transition $\hat{f}$ has one-step error at most $\varepsilon$, and the true dynamics $f$ are Lipschitz with constant $L$ (a small change in state changes the next state by at most $L$ times as much). Starting from the same state and applying the same actions, the gap $e_k$ between predicted and true states after $k$ steps obeys $e_{k+1} \le L e_k + \varepsilon$, so

$$e_H \le \varepsilon \sum_{k=0}^{H-1} L^k = \varepsilon \, \frac{L^H - 1}{L - 1} \quad (L \neq 1).$$

For $L > 1$ the bound grows exponentially with the horizon $H$; for $L < 1$ it saturates at $\varepsilon/(1 - L)$. This is a worst-case bound, but it explains two common design choices: replanning often (MPC) and using short model rollouts. Janner et al. made the second idea concrete: their MBPO algorithm branches short model rollouts from real states rather than imagining long trajectories from the start, and they argue both theoretically and empirically that this keeps the benefit of model data while limiting its bias [30].

## World Models: compress, predict, control

**Plain version.** Encode observations, predict changes in that representation, and train a controller using the learned dynamics.

Ha and Schmidhuber's *World Models* (published at NeurIPS 2018 as "Recurrent World Models Facilitate Policy Evolution", with an interactive online version) made the idea popular in deep learning [6]. Their agent has three parts:

- **V (vision):** a variational autoencoder that compresses each $64 \times 64$ frame into a small latent vector $z_t$.

- **M (memory):** a recurrent network with a mixture-density output that predicts a distribution over $z_{t+1}$ given $z_t$, the action and its hidden state $h_t$.
- **C (controller):** a deliberately tiny linear policy $a_t = W_c[z_t; h_t] + b_c$, trained with an evolution strategy rather than backpropagation.

In a car-racing task, the controller using V and M features solved the task; in a VizDoom task, the controller was trained entirely inside the environment simulated by M and then transferred to the real game [6]. The design separates *perception*, *prediction* and *decision*, and shows that a controller can be very small when the representation already carries the relevant dynamics. It also showed the central risk: a controller trained inside an imperfect dream can learn to exploit the dream's mistakes.

## Latent dynamics: from PlaNet to the Dreamer line

**Plain version.** Instead of predicting future images, predict a compact hidden state and only decode images when training. A hidden state that mixes a deterministic memory with a random part can represent both what the agent remembers and what it is unsure about.

### Recurrent state-space models

Hafner et al.'s PlaNet learned environment dynamics from images and chose actions by fast online planning in latent space; its key modeling idea is a latent dynamics model with both deterministic and stochastic components, the **recurrent state-space model** (RSSM) [7]. The RSSM keeps a deterministic memory $h_t$ and a stochastic state $z_t$:

$$h_t = f_\theta(h_{t-1}, z_{t-1}, a_{t-1}), \qquad z_t \sim p_\theta(z_t \mid h_t) \ \text{(prior)}, \qquad z_t \sim q_\theta(z_t \mid h_t, o_t) \ \text{(posterior)}.$$

Here $f_\theta$ is a recurrent cell, the **prior** predicts $z_t$ without seeing the current image (this is what is used when imagining), and the **posterior** also sees the image $o_t$ (used during training and when filtering real observations). Heads read the model state $(h_t, z_t)$ to predict the image $\hat{o}_t$, the reward $\hat{r}_t$ and a continuation flag (whether the episode goes on).

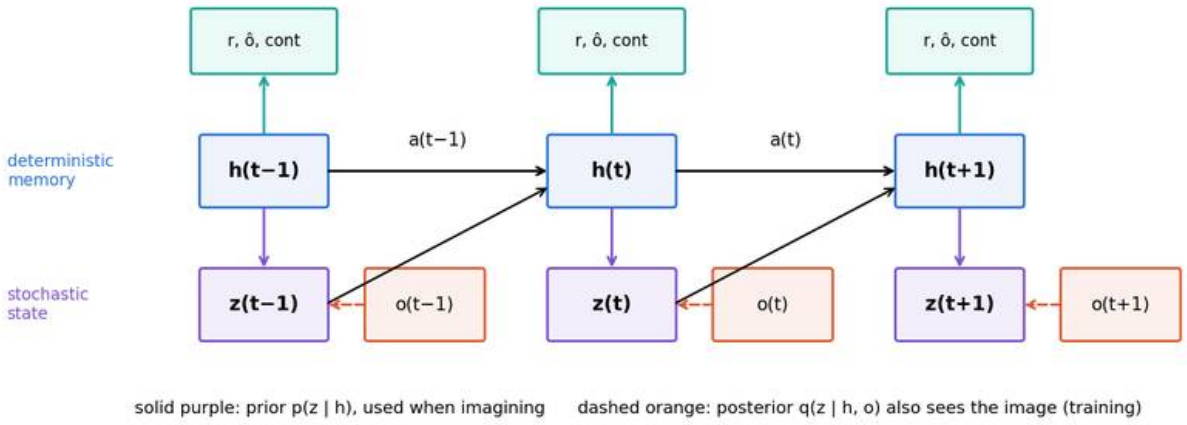



*Figure 2 — An unrolled RSSM. Imagination uses only the prior path (solid); training additionally conditions z on the real image (dashed).*

Training maximizes a variational lower bound on the likelihood of observed sequences. A simplified per-step loss is

$$\mathcal{L}_t = -\log p_\theta(o_t \mid h_t, z_t) - \log p_\theta(r_t \mid h_t, z_t) + \beta\, D_{\mathrm{KL}}(q_\theta(z_t \mid h_t, o_t) \,\|\, p_\theta(z_t \mid h_t)),$$

with $\beta > 0$ weighting the KL term. The reconstruction terms make the state informative; the KL term teaches the prior to predict what the posterior will see, which is exactly the skill needed for imagination. Real implementations add loss balancing and other stabilizers, so treat this as a teaching outline, not the exact loss of any one paper.

*Online only: The ELBO and the reparameterization trick: training a latent state — https://blog.shyandram.dev/en/learn/deep-learning/world-models/notes/the-elbo-and-the-reparameterization-trick-training-a-latent-state/*

### Dreamer: learning behavior in imagination

Dreamer (2020) kept the RSSM but replaced online planning with an actor–critic trained entirely on imagined latent trajectories. Its key move was to propagate analytic gradients of learned state values back through trajectories imagined in the compact latent space [8]. Starting from latent states inferred from real data, the agent imagines $H$ steps with its actor, scores them with a learned critic, and trains the critic on bootstrapped targets such as the $\lambda$-return

$$V_t^\lambda = \hat{r}_t + \gamma\, \hat{c}_t \big[ (1-\lambda)\, v_\psi(s_{t+1}) + \lambda\, V_{t+1}^\lambda \big],$$

where $\hat{r}_t$ and $\hat{c}_t$ are the predicted reward and continuation, $v_\psi$ is the critic, and $\lambda \in [0,1]$ trades off one-step bootstrapping ($\lambda = 0$) against longer imagined returns ($\lambda = 1$). The actor is updated to increase these values.

The line continued:

- **DreamerV2** (2021) switched to *discrete* (categorical) latent variables and became the first agent to reach human-level performance on the 55-game Atari benchmark by learning behaviors inside a separately trained world model [9].
- **DreamerV3** (Nature, 2025) used robustness techniques based on normalization, balancing and transformations so that a single configuration outperformed specialized methods across more than 150 tasks, and was the first algorithm to collect diamonds in Minecraft from scratch without human data or curricula [10].

Two related lines show that the RSSM is one choice among several. **TD-MPC2** plans in the latent space of an *implicit*, decoder-free world model (no image reconstruction), reported strong results across 104 online RL tasks with a single set of hyperparameters, and trained a single 317M-parameter agent on 80 tasks [11]. **IRIS** replaced the RSSM with a discrete autoencoder plus an autoregressive Transformer over image tokens and learned from the equivalent of two hours of Atari gameplay [12]. **DIAMOND** went further toward pixels: it trains the agent inside a *diffusion* world model, arguing that discrete latents can drop visual details that matter for control, and also demonstrated the model as an interactive neural game engine for a 3D game [13].

## MuZero: a model that predicts only what planning needs

**Plain version.** You do not have to predict what the screen will look like to play a good game. You only need to predict how good positions are, which moves look promising, and what rewards you will get. MuZero learns exactly that, and then searches.

AlphaZero-style tree search needs a simulator of the game rules. MuZero removes that requirement by learning a model that, applied repeatedly, predicts the quantities most directly relevant to planning: reward, action-selection policy and value [14]. It has three functions:

$$s^0 = h_\theta(o_1, \dots, o_t), \qquad (r^k, s^k) = g_\theta(s^{k-1}, a^k), \qquad (\mathbf{p}^k, v^k) = f_\theta(s^k),$$

where $h_\theta$ (representation) encodes the observation history into a hidden state $s^0$, $g_\theta$ (dynamics) maps a hidden state and a hypothetical action $a^k$ to a predicted reward $r^k$ and next hidden state, and $f_\theta$ (prediction) outputs a policy vector $\mathbf{p}^k$ and a value $v^k$. A Monte Carlo tree search unrolls $g_\theta$ to choose actions. Training unrolls the model for $K$ steps along real trajectories and minimizes

$$\mathcal{L} = \sum_{k=0}^{K} \Big[ \ell^r(u_{t+k}, r^k) + \ell^v(z_{t+k}, v^k) + \ell^p(\boldsymbol{\pi}_{t+k}, \mathbf{p}^k) \Big] + c\|\theta\|^2,$$

where $u$ are observed rewards, $z$ are value targets (bootstrapped returns), $\boldsymbol{\pi}$ are the visit-count distributions produced by the search, $\ell^r, \ell^v, \ell^p$ are per-quantity losses and $c$ weights an L2 penalty.

The hidden state $s^k$ is never asked to reconstruct an observation. It only has to be **value-equivalent**: planning with it must give the same rewards, values and policies as planning in the real environment would. With this design MuZero reached a new state of the art on 57 Atari games and, without being given the rules, matched AlphaZero's superhuman performance in Go, chess and shogi [14]. The lesson for vision is that a world model's target should be chosen by its use; predicting every pixel is one option, not a requirement.

## Video prediction and generation as world simulators

**Plain version.** If a model can generate realistic future video, perhaps it has learned how the world works. Sometimes it has; often it has only learned what videos usually look like. Testing which is the case is the hard part.

### Action-conditioned video prediction

Finn et al. trained an **action-conditioned video prediction** model on a dataset of 59,000 robot pushing interactions; by predicting pixel motion rather than raw pixels, the model generalized partly to unseen objects, and conditioning on different future actions produced a "visual imagination" of different futures [15]. This is a world model in the most literal sense: image in, action in, image out.

### Why squared-error predictors blur

Real futures are uncertain. Suppose that from the same past, a ball goes left with probability $\frac{1}{2}$ and right with probability $\frac{1}{2}$. For squared error, the best single prediction is the **conditional mean**:

$$\hat{o}^\star = \arg\min_{\hat{o}} \; \mathbb{E}[\|o_{t+1} - \hat{o}\|^2 \mid o_{\le t}] = \mathbb{E}[o_{t+1} \mid o_{\le t}],$$

which here is two half-bright balls, an image that never happens. A generative model that **samples** one of the two futures produces a realistic image but, on average, a *higher* squared error.

```
import numpy as np

x = np.arange(64)
def frame(center):                      # a 1-D "image" with one bright ball
    return np.exp(-0.5 * ((x - center) / 2.0) ** 2)

# Same past, two equally likely futures: the ball bounces left or right.
futures = np.stack([frame(20), frame(44)])
mean_pred = futures.mean(axis=0)        # what an MSE-trained predictor converges to
sample_pred = futures[0]                # what a good generative sampler might output

def expected_mse(pred):                 # average over the two possible futures
    return np.mean([(pred - f) ** 2 for f in futures])

print(f"peak brightness: true future 1.00, mean prediction {mean_pred.max():.2f}")
print(f"expected MSE of mean prediction:   {expected_mse(mean_pred):.4f}")
print(f"expected MSE of a sharp sample:    {expected_mse(sample_pred):.4f}")
```

Output:

```
peak brightness: true future 1.00, mean prediction 0.50
```

```
expected MSE of mean prediction:   0.0277
expected MSE of a sharp sample:    0.0554
```

The sharp sample has exactly twice the expected error of the blurry mean. This is why pixel MSE rewards blur, why stochastic video models were introduced, and why evaluating a world model with a single squared-error number can prefer the wrong model.

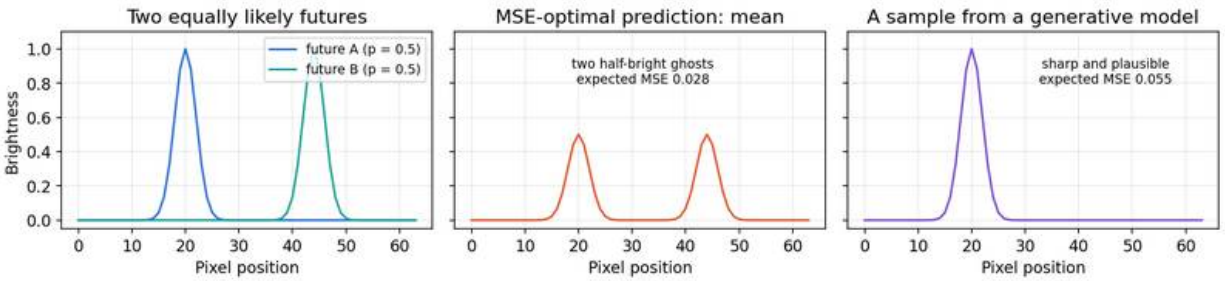



*Figure 3 — Output of the code above. The MSE-optimal prediction averages incompatible futures; a sample is plausible but scores worse on MSE.*

Babaeizadeh et al.'s SV2P addressed this by predicting a different possible future for each sample of a latent variable, and found substantially better predictions than the same model without stochasticity [16]. Modern video generators use the same principle at a much larger scale, with autoregressive token models or diffusion and flow models (see ML 08 and ML 09).

### Large generative world models

Several large models now generate video conditioned on actions, text or actions inferred from the video itself, and are presented as world simulators.

- **GAIA-1** casts driving world modeling as next-token prediction over discrete tokens of video, text and action, producing realistic driving scenarios with control over ego-vehicle behavior and scene features [17].
- **UniSim** learns a "universal simulator" of real-world interaction by combining datasets that are rich along different axes (objects in images, dense actions in robot data, motion in navigation data); it simulates both high-level instructions and low-level controls, and policies trained purely in it were deployed zero-shot on real robots [18].
- **Genie** is a generative interactive environment trained without action labels on unlabelled Internet video. It consists of a video tokenizer, an autoregressive dynamics model and a *latent action model* that infers actions from changes between frames, so users can act frame by frame in generated worlds even though no action labels were used in training [19]. Such latent actions need not correspond to calibrated physical actuators.
- **Genie 3**, announced by Google DeepMind in August 2025, generates worlds from text that can be navigated in real time at 24 frames per second and 720p, keeping consistency for a few minutes. The announcement lists limitations: a constrained action space, difficulty modeling

several independent agents, inaccurate real-world locations, and interaction limited to minutes rather than hours [20].
- **Cosmos** is NVIDIA's open platform of "world foundation models": pretrained video models, video tokenizers, a data-curation pipeline and post-training examples meant to be fine-tuned into task-specific world models for robots and vehicles [21].

### Does realistic video mean physical understanding?

Two careful studies say: not necessarily. Motamed et al.'s **Physics-IQ** benchmark tests fluid dynamics, optics, solid mechanics, magnetism and thermodynamics; across current video models (including Sora, Runway, Pika, Lumiere, Stable Video Diffusion and VideoPoet), physical understanding was severely limited and unrelated to visual realism [22]. Kang et al. trained diffusion video models on a 2D simulator governed by classical mechanics and found perfect in-distribution generalization, measurable scaling for combinatorial generalization, but failure out of distribution; the models behaved as if retrieving the closest training example rather than abstracting a rule [23].

For robot control, also ask what an "action" means: force, joint target, gripper command, camera movement or an inferred token. A wrong unit, timing offset or control frequency changes the conditional problem even if the video looks convincing.

## Predictive representations: the JEPA family

**Plain version.** Predicting every pixel wastes effort on details that do not matter, like leaf texture. Instead, predict an abstract *description* of the missing or future part. If the description is good, the prediction is useful for planning.

A **joint-embedding predictive architecture** (JEPA) encodes a visible *context* $x$ and a hidden *target* $y$ with encoders, and trains a predictor to map the context embedding to the target embedding:

$$\mathcal{L} = \| P_\phi(E_\theta(x),\, m) - \mathrm{sg}[\bar{E}(y)] \|^2,$$

where $E_\theta$ is the context encoder, $\bar{E}$ the target encoder, $P_\phi$ a predictor conditioned on information $m$ about where the target is (for example its position in the image or time), and sg (stop-gradient) blocks gradients into the target branch. Nothing is reconstructed in pixel space. The danger is **collapse**: if both encoders output a constant, the loss is zero. Asymmetric designs (a predictor on one side only, stop-gradient, and a target encoder that is not trained by the same gradient) are used to prevent it; ML 06 discusses these self-supervised mechanisms in more detail.

The JEPA line began as representation learning, on images (I-JEPA [24]) and then on video (V-JEPA [25]), and V-JEPA 2.1 [27] later refined its dense features. Two models show what turns it into a world model: action-conditioned prediction, and planning toward a goal in feature space.

- **V-JEPA 2** pretrains an action-free JEPA on over one million hours of Internet video, then post-trains a latent **action-conditioned** world model, V-JEPA 2-AC, on less than 62 hours of unlabeled robot video; planning toward image goals with it enabled zero-shot pick-and-place on

Franka arms in two labs without data from those environments, task-specific training or reward [26]. The split between passive video and action data makes the observation/action distinction of this chapter concrete.

- **DINO-WM** shows the same idea with an off-the-shelf encoder: it predicts future DINOv2 patch features from offline trajectories and plans by optimizing action sequences toward the features of a goal image, without reconstructing pixels or learning a reward model [28].

*Online only: More of the JEPA line: I-JEPA, V-JEPA and V-JEPA 2.1 — https://blog.shyandram.dev/en/learn/deep-learning/world-models/notes/more-of-the-jepa-line-i-jepa-v-jepa-and-v-jepa-21/*

Goals in feature space need care. A distance $\|z - z_{\text{goal}}\|^2$ rewards only what the representation preserves. If the representation ignores gripper closure, a planner may reach a visually similar state without actually grasping the object. Better feature quality alone does not establish that an encoder predicts the consequences of arbitrary actions.

### How the families relate

The families differ mainly in what they predict, and each prediction target raises its own question to test.

| Family | Prediction target | Main benefit | Question to test |
|---|---|---|---|
| Latent dynamics (RSSM, TD-MPC2) | compact state, reward, value | cheap, repeated rollouts | does the latent retain task-relevant dynamics? |
| Value-equivalent (MuZero) | reward, value, policy only | no effort on irrelevant detail | does it transfer when the reward changes? |
| Autoregressive video (GAIA-1, Genie, IRIS) | next frame or token | explicit sequential factorization | do errors drift over long rollouts? |
| Diffusion/flow video (DIAMOND, Cosmos) | jointly generated frames or chunks | expressive, uncertain futures | can actions steer outcomes at control latency? |
| Predictive embeddings (JEPA, DINO-WM) | target features | structure without pixel reconstruction | are actions represented and goals meaningful? |

These are compatible choices. A Transformer can parameterize any of them; a latent world model can decode through diffusion; a video generator can use an autoregressive outer loop and a diffusion inner loop. Architecture, prediction target, temporal factorization and training objective are separate design decisions.

## Learning dynamics from data

**Plain version.** A dynamics model is only as good as its data: frames, actions and rewards must stay in order and correctly aligned, and test episodes must be truly separate from training ones.

Training examples should retain ordering and action alignment:

$$(o_t, a_t, o_{t+1}, r_t, \text{done}_{t+1}),$$

where $\text{done}_{t+1}$ marks the end of an episode. Record whether each action happens before or after each camera timestamp. Split by whole episodes, environment instances and relevant subjects; randomly splitting adjacent frames leaks nearly identical scenes and makes a transition predictor look better than it is (see ML 11 for leakage in general).

The data-collection policy matters. A model trained on slow, gentle interactions may fail for abrupt motion. Observational correlations are especially problematic when actions are absent: a model may learn that a cup moves when a hand approaches, yet have no evidence about what alternative pushes would do. This is the difference between passive video and interventional data.

A **one-step loss** measures prediction from real history. A **multi-step loss** unrolls the model's own predictions and penalizes their accumulated error. Unrolling changes training difficulty and gradient paths; it does not remove distribution shift outside the observed interactions.

## Complete CPU experiment: fit, predict, plan

**Plain version.** We now run the whole loop on a toy system: collect data, fit a model, check its predictions one step and many steps ahead, and plan with it.

We use a two-dimensional state $x_t = (\text{position}, \text{velocity})$, a scalar action $u_t$ and a damped linear system:

$$x_{t+1} = Ax_t + Bu_t + \epsilon_t,$$

where $A$ is the $2 \times 2$ state-transition matrix, $B$ the action-input vector and $\epsilon_t$ small Gaussian noise. The system is deliberately fully observed. It isolates dynamics and planning before adding image encoders and partial observability. Training and test episodes are generated independently. We fit only on training transitions and evaluate both one-step prediction and open-loop rollouts with the same held-out actions.

The planner uses the **cross-entropy method** (CEM): sample candidate action sequences from a Gaussian, keep the lowest-cost "elite" sequences under the model, refit the Gaussian's mean and spread to them, and repeat. It executes one action, receives a fresh real state and replans. This is model-predictive control.

*Online only: The cross-entropy method: planning by sampling and refitting — https://blog.shyandram.dev/en/learn/deep-learning/world-models/notes/the-cross-entropy-method-planning-by-sampling-and-refitting/*

```
import numpy as np

rng = np.random.default_rng(7)
A = np.array([[1.0, 0.1], [0.0, 0.94]])
B = np.array([0.005, 0.1])

def collect(n, steps=40):
    x = np.zeros((n, steps + 1, 2))
    x[:, 0] = rng.normal(size=(n, 2)) * [0.8, 0.3]
    u = rng.uniform(-1, 1, size=(n, steps))
```

```
    for t in range(steps):
        x[:, t + 1] = (x[:, t] @ A.T + u[:, t, None] * B
                       + rng.normal(scale=0.002, size=(n, 2)))
    return x, u

train, train_u = collect(120)
test, test_u = collect(80)
phi = np.column_stack([train[:, :-1].reshape(-1, 2), train_u.ravel()])
target = train[:, 1:].reshape(-1, 2)
W = np.linalg.lstsq(phi, target, rcond=None)[0]
W_no_action = np.linalg.lstsq(phi[:, :2], target, rcond=None)[0]

def predict(x, u):
    return np.concatenate([x, np.asarray(u)[..., None]], axis=-1) @ W

one_step = predict(test[:, :-1], test_u)
rollout = np.zeros_like(test)
rollout[:, 0] = test[:, 0]
for t in range(test_u.shape[1]):
    rollout[:, t + 1] = predict(rollout[:, t], test_u[:, t])
rmse = lambda a, b: np.sqrt(np.mean((a - b) ** 2))
print("One-step:", rmse(one_step, test[:, 1:]))
print("Without actions:", rmse(test[:, :-1] @ W_no_action, test[:, 1:]))
print("Final-horizon rollout:", rmse(rollout[:, -1], test[:, -1]))

def plan(x, mean, horizon=20, population=256):
    std = np.ones(horizon)
    for _ in range(4):
        candidates = np.clip(rng.normal(mean, std, (population, horizon)), -1, 1)
        imagined = np.broadcast_to(x, (population, 2)).copy()
        costs = np.zeros(population)
        for t in range(horizon):
            imagined = predict(imagined, candidates[:, t])
            costs += (imagined[:, 0] ** 2 + 0.15 * imagined[:, 1] ** 2
                      + 0.02 * candidates[:, t] ** 2)
        costs += 20 * imagined[:, 0] ** 2
        elite = candidates[np.argsort(costs)[:32]]
        mean = elite.mean(axis=0)
        std = elite.std(axis=0) + 0.05
    return mean

real = np.array([1.5, 0.0])
mean = np.zeros(20)
for _ in range(40):
    sequence = plan(real, mean)
    real = A @ real + B * sequence[0]  # real system; never used inside plan()
    mean = np.r_[sequence[1:], 0.0]
print("Controlled final state:", real)
assert rmse(one_step, test[:, 1:]) < 0.01
assert abs(real[0]) < 0.1
```

Output:

```
One-step: 0.0019866110218121832
```

```
Without actions: 0.04106321931941812
Final-horizon rollout: 0.013772759824274517
Controlled final state: [-0.00453985 -0.0210205 ]
```

The one-step RMSE (0.0020) is at the noise level; dropping the action input raises it twentyfold (0.0411), so the action genuinely matters. The open-loop rollout error at horizon 40 (0.0138) is about seven times the one-step error even for this nearly perfect linear model, a mild case of the compounding derived above. MPC drives the position from 1.5 to about −0.0045, compared with 1.5 under zero action. These are synthetic results from this script, not published comparisons.

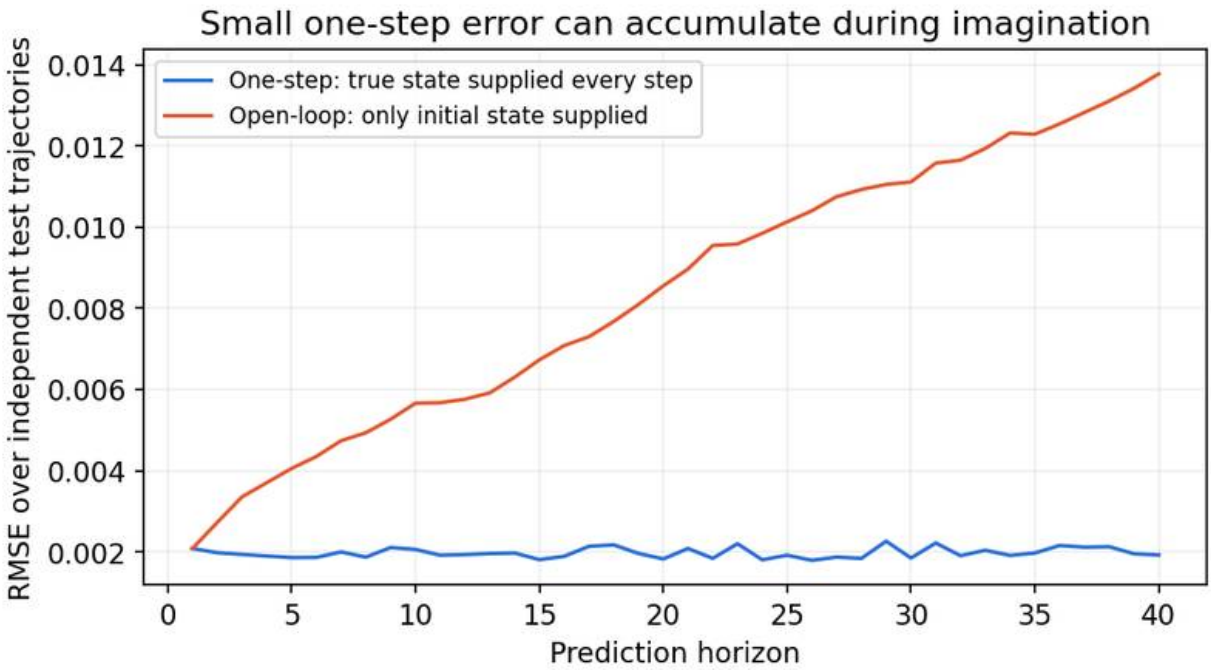



*Figure 4 — Independent synthetic test episodes. One-step evaluation receives the true state each time; open-loop evaluation receives only the initial state and the held-out actions.*

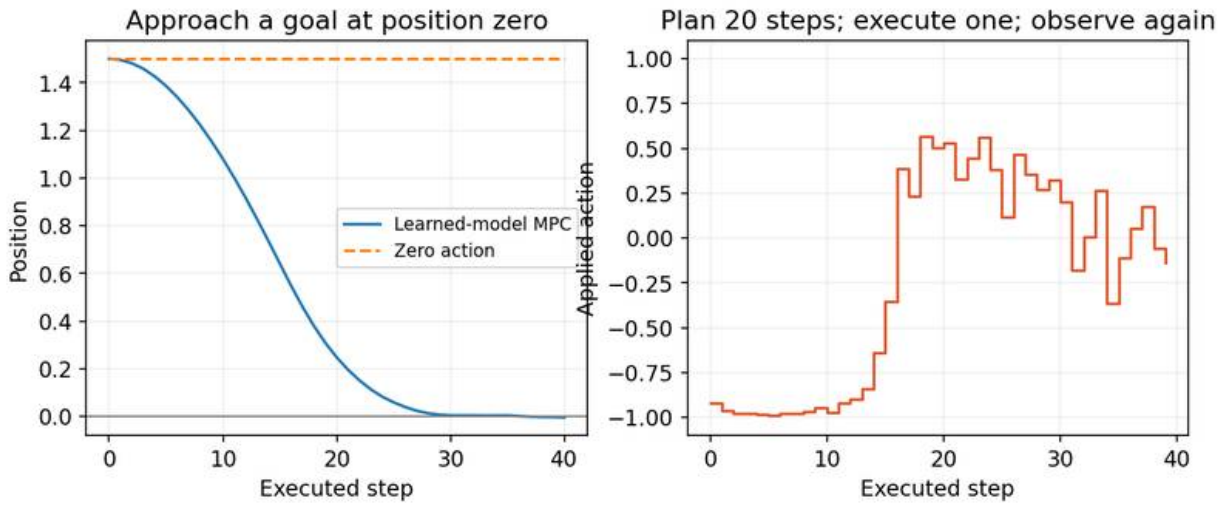



*Figure 5 — CEM plans with the learned transition model. Only its first action is executed in the synthetic environment before replanning from a new observation.*

Replacing $x_t$ with an image embedding does not reproduce Dreamer or V-JEPA 2-AC: it adds an encoder-learning problem, latent uncertainty and the need for a valid goal or reward.

## Planning, uncertainty and model exploitation

**Plain version.** A planner looks for the action the model likes best. If the model is wrong somewhere, the planner will happily go there. Knowing *where* the model is unsure is what keeps it honest.

A finite-horizon planning objective can be written as

$$a^*_{t:t+H-1} = \arg\min_{a_{t:t+H-1}} \mathbb{E}_\theta\Big[\sum_{k=1}^{H} c(z_{t+k}, a_{t+k-1}) + \lambda_{\mathrm{T}}\, c_{\mathrm{T}}(z_{t+H})\Big],$$

where $c$ is a per-step cost, $c_{\mathrm{T}}$ a terminal cost weighted by $\lambda_{\mathrm{T}}$, and the expectation is over the model's predicted latent states $z$. **Model-predictive control** executes the first action, observes again and replans. Feedback corrects drift, but cannot rescue an action whose immediate consequence was badly modeled.

### Two kinds of uncertainty

**Aleatoric** uncertainty is inherent randomness (a coin flip, the ball that may go left or right); more data does not remove it. **Epistemic** uncertainty comes from lack of data (the model has never seen this situation); more data does remove it. Sampling a generative model can represent aleatoric uncertainty, but sample diversity is not proof of calibration. Epistemic uncertainty is commonly approximated with **ensembles**: several models trained on bootstrap resamples or with different initializations agree where data is dense and tend to disagree where it is not. Chua et al.'s

PETS combined probabilistic networks (for aleatoric noise) in an ensemble (for epistemic uncertainty) with sampling-based uncertainty propagation and CEM planning, matching the final performance of strong model-free methods on several benchmarks with far fewer samples [29].

### Code: a planner exploiting a learned model

The agent has logged data only for actions in $[-1, 1]$, but its planner may choose any action in $[-3, 3]$.

```
import numpy as np
from sklearn.neural_network import MLPRegressor

rng = np.random.default_rng(0)
true_reward = lambda a: np.sin(3 * a) * np.exp(-0.3 * a ** 2)   # unknown to the agent

# Logged interactions only cover actions in [-1, 1].
a_data = rng.uniform(-1, 1, 200)
r_data = true_reward(a_data) + 0.05 * rng.normal(size=200)

ensemble = []
for k in range(5):                        # bootstrap + different initialization
    idx = rng.integers(0, 200, 200)
    m = MLPRegressor(hidden_layer_sizes=(64, 64), max_iter=3000, random_state=k)
    ensemble.append(m.fit(a_data[idx, None], r_data[idx]))

grid = np.linspace(-3, 3, 601)
preds = np.stack([m.predict(grid[:, None]) for m in ensemble])
mean, std = preds.mean(0), preds.std(0)

for name, score in [("single model", preds[0]),
                    ("ensemble mean", mean),
                    ("pessimistic: mean - 2 std", mean - 2 * std)]:
    a_star = grid[np.argmax(score)]       # the "planner": pick the best-looking action
    print(f"{name:26s} picks a = {a_star:+.2f}, predicted {score.max():.2f}, "
          f"true {true_reward(a_star):.2f}")
print(f"mean std inside data {std[np.abs(grid) <= 1].mean():.3f}, "
      f"outside {std[np.abs(grid) > 1.5].mean():.3f}")
```

Output:

```
single model               picks a = -3.00, predicted 1.79, true -0.03
ensemble mean              picks a = -3.00, predicted 1.58, true -0.03
pessimistic: mean - 2 std  picks a = +0.43, predicted 0.87, true 0.91
mean std inside data 0.025, outside 0.321
```

Both the single model and the ensemble mean extrapolate a rising line outside the data, and the planner runs to the edge of the action space, where the true reward is about zero. Ensemble disagreement is thirteen times larger outside the data than inside, and a *pessimistic* score (mean

minus two standard deviations) keeps the planner inside the data, where it finds an action with true reward 0.91. Restricting actions, shortening horizons, penalizing disagreement and testing with real feedback are the practical responses. Agreement among ensemble members is not proof of correctness, though: all five can share the same bias.

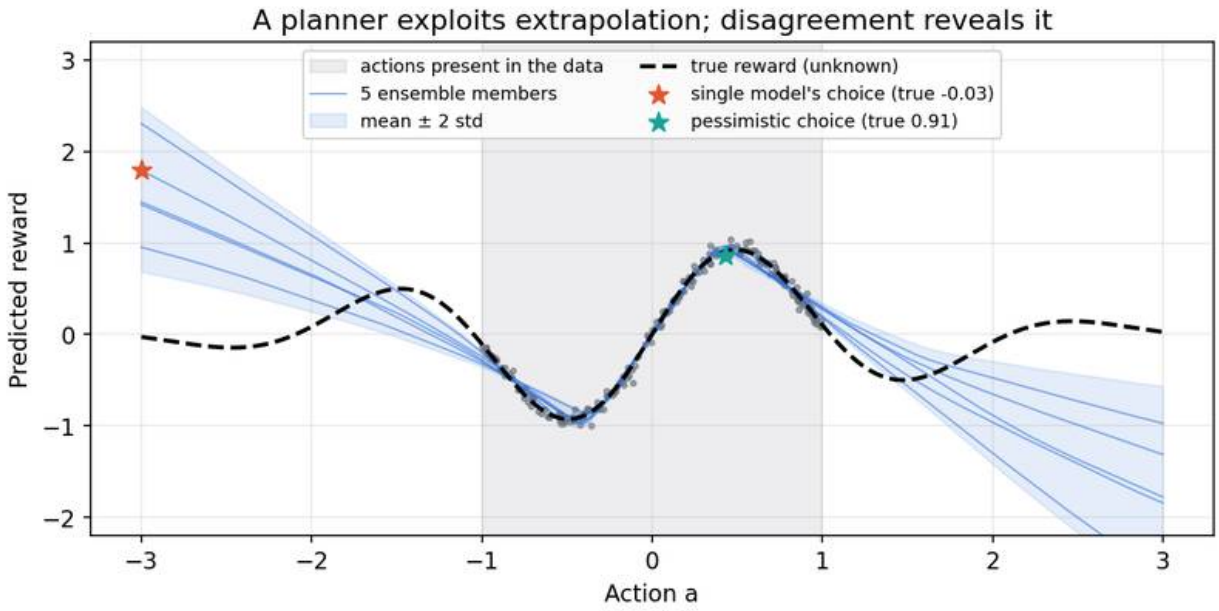



*Figure 6 — Output of the code above. Optimizing a learned model drives the planner to where the model is least reliable; ensemble spread exposes those regions.*

## How to evaluate a world model

**Plain version.** No single number says whether a world model is good. Ask several questions, each with its own measurement and its own control.

| Question | Measurement | Important control |
|---|---|---|
| Does it predict known dynamics? | one-step and horizon-indexed state or feature error | independent episodes; fixed action sequence |
| Does appearance stay coherent? | reconstruction and video-quality metrics, identity consistency | compare camera motion and occlusions |
| Do actions matter correctly? | matched initial context with different specified actions | real action-conditioned reference trajectories |
| Are physical constraints respected? | collisions, object permanence, conservation; benchmarks such as Physics-IQ [22] | task-specific tolerances; out-of-distribution cases [23] |
| Does it help decisions? | success rate, return, interventions, control latency | test in the real or a trusted environment |

Pixel fidelity and decision usefulness can disagree, as the blur example showed. An averaged

prediction can have low squared error while showing an impossible scene; an accurate latent predictor can omit visually irrelevant texture. Report the target, horizon, conditions and output resolution with every metric.

For comparison, include a persistence baseline (predict "nothing changes"), an action-free predictor and a simple task-appropriate dynamics model. Evaluate held-out objects, viewpoints, action ranges and environment properties separately. Report closed-loop control results in addition to imagined ones, and account for encoder and sampler latency: a planner that needs ten seconds per action may be useless for a robot running at 10 Hz.

## Modern view

**Plain version.** Three lines of work now meet under the name "world model": model-based reinforcement learning, video generators reframed as simulators, and predictive representations. The open questions are mostly about whether their predictions are reliable enough to act on.

**Surveys.** Moerland et al. give the most systematic treatment of model-based RL: dynamics-model learning (stochasticity, uncertainty, partial observability, temporal abstraction), a taxonomy of how planning and learning are integrated, and implicit (value-equivalent, MuZero-style) approaches [5]. Ding et al. (ACM Computing Surveys) organize the broader "world model" literature by function, understanding the world versus predicting its future, and review applications in generative games, autonomous driving, robotics and social simulation [1]. Zhu et al. survey "general world models" from the video-generation side, covering video generators, autonomous-driving world models and world models inside autonomous agents, and discuss their limitations [31]. Guan et al. give an initial survey of world models specifically for autonomous driving [32].

**State of the art, 2023–2026.**

- *Model-based RL matured.* DreamerV3 showed a single configuration working across more than 150 tasks and was published in Nature in 2025 [10]; TD-MPC2 showed decoder-free latent planning scaling to a multi-task 317M-parameter agent [11]; IRIS and DIAMOND showed Transformer and diffusion world models competitive on the Atari 100k benchmark [12], [13].
- *Video generators were reframed as simulators.* GAIA-1, UniSim, Genie, Genie 3 and Cosmos trained large models on video with actions, text or inferred latent actions [17], [18], [19], [20], [21]. The strongest evidence that they help agents is UniSim's zero-shot transfer of policies trained inside the simulator [18]; the strongest caution is that realism does not imply physics [22], [23].
- *Predictive representations moved toward control.* The JEPA line went from images (I-JEPA) to video (V-JEPA) to action-conditioned planning on real robots (V-JEPA 2-AC) and denser features (V-JEPA 2.1) [24], [25], [26], [27]; DINO-WM showed that frozen self-supervised features can already support zero-shot planning [28].

**Open problems.**

- *Long-horizon consistency.* Errors compound; generated worlds drift; object identity and off-screen state are hard to keep for minutes, let alone hours [20].

- *Causal and physical correctness.* Models trained on passive video learn correlations; out-of-distribution physics fails [22], [23]. Interventional (action) data is scarce and expensive.
- *Action interfaces.* Latent actions inferred from video, robot joint commands and text instructions are different things; mapping between them reliably is unsolved.
- *Exploitation and uncertainty.* Planners and policies trained in imagination exploit model errors; calibrated epistemic uncertainty for large video models is largely missing.
- *Evaluation.* There is no agreed benchmark that ties prediction quality to downstream decision quality across domains; most papers still report either video metrics or task returns, rarely both.
- *Cost.* Sampling a large video model can be far slower than real time, which conflicts with closed-loop control.

## Key takeaways

- The prediction target and action interface determine what a world model can support.
- Partial observations require memory or belief estimation; an image is not the full state.
- Planning, imagined policy training and value-equivalent search use learned dynamics differently.
- Pixel realism does not establish physical correctness; feature prediction also needs meaningful targets and anti-collapse mechanisms.
- Long rollouts compound errors, and optimization can exploit them; check uncertainty and replan.
- Evaluate held-out predictions, controlled action effects and behavior in the real environment.

## Exercises

1. **Action ablation.** In the CPU experiment, remove the action column. Compare one-step error, horizon-40 error and control behavior. Why can a short-term action-free predictor still look reasonable?

   **Hint**

   Imagined actions no longer affect the action-free predictor, so a planner cannot rank their consequences at all: every candidate gets the same predicted cost. Over one step the damped dynamics change the state only slightly, so persistence-like predictions still score moderately well, which is why one-step error alone is a weak test.

2. **Partial observations.** Hide velocity and fit position-to-position dynamics. Then use the last two positions and the previous action as features. When does this history help, and where does observation noise hurt?

   **Hint**

   The difference of two positions estimates velocity (a finite difference), so the two-position model recovers most of the state. Noise in each position is amplified when divided by the time step, so the benefit shrinks as observation noise grows.

3. **Planning horizon.** Compare horizons 5, 20 and 40, adjusting the initial mean to the chosen length. Measure final distance, action effort and runtime across several independent seeds.

**Hint**

Keep the cost definition fixed and count all model calls. Very short horizons cannot "see" that braking early avoids overshoot; very long horizons cost more and accumulate more model error, although for this linear system the error is small.

4. **Domain shift.** Collect training episodes with velocity damping 0.94, then execute control in an environment with damping 0.75. Compare open-loop execution of a plan with replanning at every step. Identify an action range where the training data provide weak coverage.

**Hint**

Separate the learned dynamics (fitted on 0.94) from the true execution dynamics (0.75); do not refit on evaluation transitions. Replanning uses fresh observations, so it partially corrects the biased model; open-loop execution cannot.

5. **Mean versus sample.** Extend the blur example to three equally likely futures at positions 10, 32 and 54. Compute the expected MSE of the mean prediction and of a single sharp sample. Then propose a metric that would prefer the sample.

**Hint**

The mean now has three one-third-height bumps; the expected MSEs are about 0.037 (mean) and 0.074 (sample), again a factor of two. That factor is general: if $X$ and $Y$ are independent draws from the same distribution, $\mathbb{E}\|X-Y\|^2 = 2\,\mathbb{E}\|X-\mathbb{E}X\|^2$, so a perfect sampler always has twice the MSE of the mean. A distribution-level metric (for example, comparing sets of generated samples with sets of real futures, as FID-style metrics do for images) or a best-of-$N$ error over several samples would reward sharp, plausible predictions.

6. **Pessimism trade-off.** In the ensemble code, replace the factor 2 in "mean − 2 std" by 0, 0.5, 1 and 4. Plot the chosen action and its true reward for each. What goes wrong when the factor is too large?

**Hint**

At 0 the planner exploits extrapolation; as the factor grows it stays inside the data. Too much pessimism makes the planner avoid any action with even slight disagreement, so it may miss the best action inside the data and, in an RL setting, stop exploring.

## References


1. J. Ding et al., "Understanding world or predicting future? A comprehensive survey of world models," *ACM Computing Surveys*, vol. 58, no. 3, 2025. doi arXiv
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006, ch. 13 "Sequential Data". book
3. R. Szeliski, *Computer Vision: Algorithms and Applications*, 2nd ed., Springer, 2022. book
4. R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, vol. 2, no. 4, 1991. doi
5. T. M. Moerland, J. Broekens, A. Plaat and C. M. Jonker, "Model-based reinforcement learning: a survey," *Foundations and Trends in Machine Learning*, vol. 16, no. 1, 2023. doi arXiv
6. D. Ha and J. Schmidhuber, "Recurrent world models facilitate policy evolution," NeurIPS, 2018; interactive version "World Models," arXiv:1803.10122. paper interactive
7. D. Hafner et al., "Learning latent dynamics for planning from pixels," ICML, 2019. paper
8. D. Hafner, T. Lillicrap, J. Ba and M. Norouzi, "Dream to control: learning behaviors by latent imagination," ICLR, 2020. arXiv
9. D. Hafner, T. Lillicrap, M. Norouzi and J. Ba, "Mastering Atari with discrete world models," ICLR, 2021. arXiv
10. D. Hafner, J. Pasukonis, J. Ba and T. Lillicrap, "Mastering diverse control tasks through world models," *Nature*, vol. 640, 2025. doi arXiv
11. N. Hansen, H. Su and X. Wang, "TD-MPC2: scalable, robust world models for continuous control," ICLR, 2024. arXiv
12. V. Micheli, E. Alonso and F. Fleuret, "Transformers are sample-efficient world models," ICLR, 2023. arXiv
13. E. Alonso et al., "Diffusion for world modeling: visual details matter in Atari," NeurIPS, 2024. arXiv
14. J. Schrittwieser et al., "Mastering Atari, Go, chess and shogi by planning with a learned model," *Nature*, vol. 588, 2020. doi arXiv
15. C. Finn, I. Goodfellow and S. Levine, "Unsupervised learning for physical interaction through video prediction," NIPS, 2016. arXiv
16. M. Babaeizadeh, C. Finn, D. Erhan, R. H. Campbell and S. Levine, "Stochastic variational video prediction," arXiv:1710.11252, 2017. arXiv
17. A. Hu et al., "GAIA-1: a generative world model for autonomous driving," technical report, arXiv:2309.17080, 2023. arXiv
18. S. Yang et al., "Learning interactive real-world simulators," arXiv:2310.06114, 2023. arXiv
19. J. Bruce et al., "Genie: generative interactive environments," arXiv:2402.15391, 2024. arXiv
20. Google DeepMind, "Genie 3: a new frontier for world models," blog post, August 5, 2025. blog
21. NVIDIA (N. Agarwal et al.), "Cosmos world foundation model platform for physical AI," arXiv:2501.03575, 2025. arXiv
22. S. Motamed, L. Culp, K. Swersky, P. Jaini and R. Geirhos, "Do generative video models understand physical principles?," arXiv:2501.09038, 2025. arXiv
23. B. Kang et al., "How far is video generation from world model: a physical law perspective," ICML, 2025. arXiv
24. M. Assran et al., "Self-supervised learning from images with a joint-embedding predictive

architecture," CVPR, 2023. doi arXiv
25. A. Bardes et al., "Revisiting feature prediction for learning visual representations from video," arXiv:2404.08471, 2024. arXiv
26. M. Assran et al., "V-JEPA 2: self-supervised video models enable understanding, prediction and planning," arXiv:2506.09985, 2025. arXiv
27. L. Mur-Labadia et al., "V-JEPA 2.1: unlocking dense features in video self-supervised learning," arXiv:2603.14482, 2026. arXiv
28. G. Zhou, H. Pan, Y. LeCun and L. Pinto, "DINO-WM: world models on pre-trained visual features enable zero-shot planning," arXiv:2411.04983, 2024. arXiv
29. K. Chua, R. Calandra, R. McAllister and S. Levine, "Deep reinforcement learning in a handful of trials using probabilistic dynamics models," NeurIPS, 2018. arXiv
30. M. Janner, J. Fu, M. Zhang and S. Levine, "When to trust your model: model-based policy optimization," NeurIPS, 2019. arXiv
31. Z. Zhu et al., "Is Sora a world simulator? A comprehensive survey on general world models and beyond," arXiv:2405.03520, 2024. arXiv
32. Y. Guan et al., "World models for autonomous driving: an initial survey," arXiv:2403.02622, 2024. arXiv
33. D. P. Kingma and M. Welling, "Auto-encoding variational Bayes," *International Conference on Learning Representations* (*ICLR*), 2014; arXiv:1312.6114. arXiv
34. D. J. Rezende, S. Mohamed and D. Wierstra, "Stochastic backpropagation and approximate inference in deep generative models," ICML, 2014. paper
35. R. Y. Rubinstein, "The cross-entropy method for combinatorial and continuous optimization," *Methodology and Computing in Applied Probability*, vol. 1, no. 2, 1999. doi
36. P.-T. de Boer, D. P. Kroese, S. Mannor and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of Operations Research*, vol. 134, no. 1, 2005. doi

# DLCV 14 • Robustness: Distribution Shift, Adversarial Examples and Anomalies

Why vision models fail outside their test set and what to do about it: kinds of distribution shift and the benchmarks that measure them, augmentation, pre-training, test-time and domain adaptation; adversarial threat models, FGSM, PGD, adversarial training, honest evaluation and certified robustness; out-of-distribution and industrial anomaly detection with their scores, metrics and pitfalls.

Online edition: https://blog.shyandram.dev/en/learn/deep-learning/robustness/

## What you'll learn

- Distinguish distribution shift, adversarial perturbations and unfamiliar inputs.
- Match shift interventions to their data assumptions.
- Specify threat models, implement attacks and interpret robustness certificates.
- Compare output- and feature-based OOD scores and patch-based anomaly detection.
- Audit contamination, test-set tuning and misleading aggregate metrics.

## The big picture

A model that scores 98% on its test set has only been checked on data that looks like its training data. Deployed models meet other data: photos from a new camera, images edited by an attacker, and objects that were never part of the label set. Robustness is the study of what happens then, and of how to make the answer less alarming.

> **Explain like I'm five**
>
> Think of a student who learned everything from one textbook. Three things can go wrong in the exam. First, the exam is printed in a smudged, faded font: the questions are the same kind, but they look different (**distribution shift**). Second, a prankster changes a few letters in a question so that the student misreads it, choosing exactly the letters that confuse this student most (**adversarial example**). Third, a question from a completely different subject appears: the right answer is "this is not my subject", but the student writes an answer anyway, confidently (**out-of-distribution input**). A good student copes with the smudges, is hard to trick, and knows when to say "I don't know".

**Why it matters.** The three failures cause different damage. Shift silently lowers accuracy in the field, so a model that looked finished keeps making more mistakes after release. Adversarial examples are a security problem: any system whose decision is worth manipulating (content filters, identity checks, traffic-sign recognition) will be attacked. Unanswerable inputs produce

confident nonsense, which is the most dangerous output a safety-critical system can give. The fixes differ too, so the first practical skill is to name which of the three problems you face.

**Where this chapter sits.** DLCV 02 already introduces ImageNet-C, -R and -A, the one-step FGSM attack, shortcut learning and calibration with temperature scaling; here we build on them rather than repeat them. ML 11 defines covariate, label and concept shift with importance weights and covers monitoring after deployment. PR 01 derives the reject option, and PR 10 separates selective classification, open-set recognition and anomaly detection on a small digits project. ML 01 and ML 06A cover reconstruction-based anomaly scores, and ML 10 explains why fine-tuning can erode a foundation model's robustness. All code in this chapter runs in seconds on a CPU with NumPy, SciPy, scikit-image and scikit-learn; the "networks" are small multilayer perceptrons on 8×8 digits, so every number we print is from a toy and is marked as such.

## Three robustness problems, one map

**Plain version.** Diagnose whether a failure comes from changed data, an intentional perturbation or input outside the task. Each calls for a different evaluation.

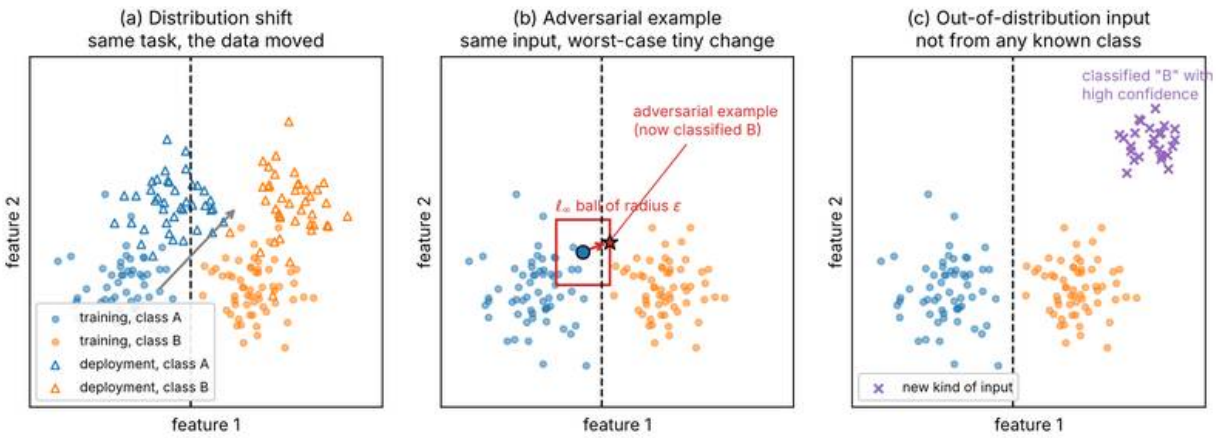



*Figure 1 — The three robustness problems on one synthetic two-class problem. (a) Distribution shift: deployment data (triangles) moved away from the training data, and a boundary that was fine for training data now makes errors. (b) Adversarial example: a worst-case point inside a small box around a correct input crosses the boundary. (c) Out-of-distribution input: points from a kind of input never seen in training still receive a confident class label.*

**Precise version.** Let a classifier $f$ be trained on samples from a distribution $p_{\text{train}}(\mathbf{x}, y)$ over images $\mathbf{x}$ and labels $y \in \mathcal{Y} = \{1, \dots, K\}$. Write $\text{acc}_p(f) = \Pr_{(\mathbf{x},y)\sim p}[f(\mathbf{x}) = y]$ for its accuracy on a distribution $p$.

1. **Distribution shift.** Test data come from $p_{\text{test}} \neq p_{\text{train}}$, but the label set $\mathcal{Y}$ is the same. The question is how much $\text{acc}_{p_{\text{test}}}(f)$ falls below $\text{acc}_{p_{\text{train}}}(f)$, and how to reduce the fall.
2. **Adversarial robustness.** For each test pair $(\mathbf{x}, y)$, an adversary may replace $\mathbf{x}$ by any $\mathbf{x}'$ in an allowed set $\mathcal{S}(\mathbf{x})$, usually a small ball around $\mathbf{x}$. The model is robust at $(\mathbf{x}, y)$ if $f(\mathbf{x}') = y$ for

**every** $\mathbf{x}' \in \mathcal{S}(\mathbf{x})$. The question is about the worst case, not the average.

3. **OOD and anomaly detection.** Some test inputs have no correct label in $\mathcal{Y}$ (a new object category), or come from a region of input space the training data never covered (a defective part, a corrupted frame). We want a score $s(\mathbf{x})$ that is high for in-distribution (ID) inputs and low for these, so that the system can abstain or raise an alarm.

| | Distribution shift | Adversarial example | OOD / anomaly |
|---|---|---|---|
| What changed | the data distribution | one input, on purpose | the input is outside the task |
| Who causes it | the world (camera, place, time) | an adversary with a goal | the world (new classes, defects) |
| What we want | keep accuracy | keep the label under every allowed change | flag the input, do not answer |
| Typical metric | accuracy on shifted test sets | robust accuracy at a budget $\varepsilon$ | AUROC, FPR at 95% TPR |
| In this chapter | Distribution shift | Adversarial examples | OOD and anomaly detection |

The three overlap at the edges. A heavy corruption can make an input effectively OOD; an adversarial example is a shift chosen by an opponent; and ImageNet-A, a set of natural photos selected because they fool a classifier [1], sits between shift and adversarial examples. Still, the table shows why one robustness number can never cover all three: they ask different questions and need different test data.

## Distribution shift

**Plain version.** A familiar task can arrive through new cameras, lighting, styles or locations. Accuracy and confidence may change differently under that shift.

### Kinds of shift

The usual vocabulary splits the joint distribution in two ways, $p(\mathbf{x}, y) = p(y \mid \mathbf{x})\, p(\mathbf{x}) = p(\mathbf{x} \mid y)\, p(y)$, and names the shift by which factor changes:

- **Covariate shift:** $p(\mathbf{x})$ changes, $p(y \mid \mathbf{x})$ does not. A new scanner, fog, a different hospital's staining.
- **Label (prior) shift:** $p(y)$ changes, $p(\mathbf{x} \mid y)$ does not. A rare disease becomes common during an outbreak.
- **Concept shift:** $p(y \mid \mathbf{x})$ itself changes. The definition of a class is revised.

ML 11 develops these with importance weights and the label-shift correction, so we do not repeat them. Two more terms are common in vision. **Domain shift** is covariate shift between named data sources (synthetic versus real, one hospital versus another), and **subpopulation shift** keeps the same kinds of images but changes how often each group appears; WILDS contains both [2]. The survey by Yang et al. adds a distinction that becomes important later in this chapter: a **covariate**

**shift** changes how inputs look while the labels stay valid, whereas a **semantic shift** introduces inputs whose labels are outside $\mathcal{Y}$ [3]. We want models to be *robust* to covariate shift and to *detect* semantic shift.

### Benchmarks that measure robustness to shift

**Plain version.** Because the real deployment data are rarely available in advance, researchers built test sets that change one thing at a time, or that collect naturally different photos of the same classes.

The main families, all labelled with ImageNet classes unless noted:

- **Synthetic corruptions. ImageNet-C** applies noise, blur, weather and digital corruptions at five severities, and its authors also proposed ImageNet-P for perturbation sequences [4]. The details and the mean corruption error are in DLCV 02.
- **Renditions and style. ImageNet-R** collects art, cartoons, sculptures and other renditions; the same paper introduced further real-world shift datasets and found that methods that helped with texture and local image statistics did not help with others, such as geographic change [5].
- **Natural adversarial images. ImageNet-A** keeps unmodified photos that a ResNet-50 got wrong; they transfer to other models, showing shared weaknesses [1].
- **Controlled collection. ObjectNet** photographs objects with randomized backgrounds, rotations and viewpoints, as a test set only, with no training split [6].
- **Re-collection. ImageNetV2** rebuilt the test set by following the original collection process; accuracy dropped although the procedure was copied as closely as possible [7] (see ML 11).
- **In the wild. WILDS** gathers ten datasets with shifts that arose in real applications, such as tissue slides from different hospitals and camera traps at new locations, and reports that standard training gives much lower out-of-distribution than in-distribution performance, with existing robustness methods not closing the gap [2].

**How to read the numbers.** A model with higher clean accuracy usually also has higher shifted accuracy, so "it scores better on ImageNet-R" may only mean "it is a better model". Taori et al. evaluated 204 ImageNet models in 213 test conditions and proposed **effective robustness**: accuracy on the shifted set *beyond* what a fitted trend over standard models predicts from the clean accuracy [8]. Miller et al. found that, for many shifts, shifted and clean accuracy lie close to a straight line across architectures and training choices ("accuracy on the line"), with weaker correlation for some synthetic shifts and for one medical dataset [9]. The consequence for practice: compare robustness interventions at matched clean accuracy, or report both numbers side by side.

*Online only: Effective robustness and accuracy on the line — https://blog.shyandram.dev/en/learn/deep-learning/robustness/notes/effective-robustness-and-accuracy-on-the-line/*

### Code: corruptions, augmentation and test-time normalization

The toy trains a small MLP on scikit-learn's 8×8 digits and corrupts the *test* images with three corruptions at severities 1 to 5. The network standardizes each pixel with statistics computed on the

training set, the way a batch-normalization layer uses running statistics at test time. We compare three models: the plain one; one trained with noise augmentation (a copy of every training image with random-severity noise); and the plain model with **test-batch normalization**, which recomputes the per-pixel mean and standard deviation on the corrupted test batch instead of using the training statistics.

```
import numpy as np
from scipy.ndimage import gaussian_filter
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

X, y = load_digits(return_X_y=True)
X = X / 16.0                                          # 8x8 images, pixels in [0, 1]
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3, stratify=y, random_state=0)
rng = np.random.default_rng(0)

corruptions = {                                       # severity s = 1..5
    "noise":    lambda X, s: np.clip(X + rng.normal(0, 0.1 * s, X.shape), 0, 1),
    "blur":     lambda X, s: gaussian_filter(X.reshape(-1, 8, 8), (0, 0.25 * s, 0.25 *
     ↪  s)).reshape(-1, 64),
    "contrast": lambda X, s: 0.5 + (X - 0.5) * (1 - 0.15 * s),
}

def norm(X, ref):                                     # standardize each pixel with ref's
 ↪  statistics
    return (X - ref.mean(0)) / (ref.std(0) + 0.05)

def fit(X, y):
    return MLPClassifier(hidden_layer_sizes=(64,), alpha=1e-3, max_iter=400,
     ↪  random_state=0).fit(norm(X, Xtr), y)

plain = fit(Xtr, ytr)
s_rand = rng.integers(1, 6, (len(Xtr), 1))            # random noise severity per training
 ↪  image
augmented = fit(np.vstack([Xtr, corruptions["noise"](Xtr, s_rand)]), np.r_[ytr, ytr])

for name, corrupt in corruptions.items():
    rows = {"plain": [], "noise-aug": [], "test-norm": []}
    for s in range(6):
        Xs = Xte if s == 0 else corrupt(Xte, s)
        rows["plain"].append(plain.score(norm(Xs, Xtr), yte))
        rows["noise-aug"].append(augmented.score(norm(Xs, Xtr), yte))
        rows["test-norm"].append(plain.score(norm(Xs, Xs), yte))   # statistics of the test
         ↪  batch
    for k, v in rows.items():
        print(f"{name:8s} {k:9s}", " ".join(f"{a:.2f}" for a in v))
```

It prints the test accuracy at severities 0 (clean) to 5:

```
noise    plain     0.98 0.94 0.75 0.54 0.41 0.31
```

```
noise     noise-aug 0.97 0.96 0.93 0.82 0.74 0.62
noise     test-norm 0.98 0.97 0.90 0.83 0.73 0.61
blur      plain     0.98 0.98 0.96 0.82 0.46 0.21
blur      noise-aug 0.97 0.97 0.97 0.90 0.78 0.58
blur      test-norm 0.98 0.98 0.97 0.95 0.89 0.81
contrast  plain     0.98 0.92 0.46 0.20 0.19 0.19
contrast  noise-aug 0.97 0.98 0.96 0.93 0.84 0.60
contrast  test-norm 0.98 0.98 0.98 0.98 0.98 0.98
```

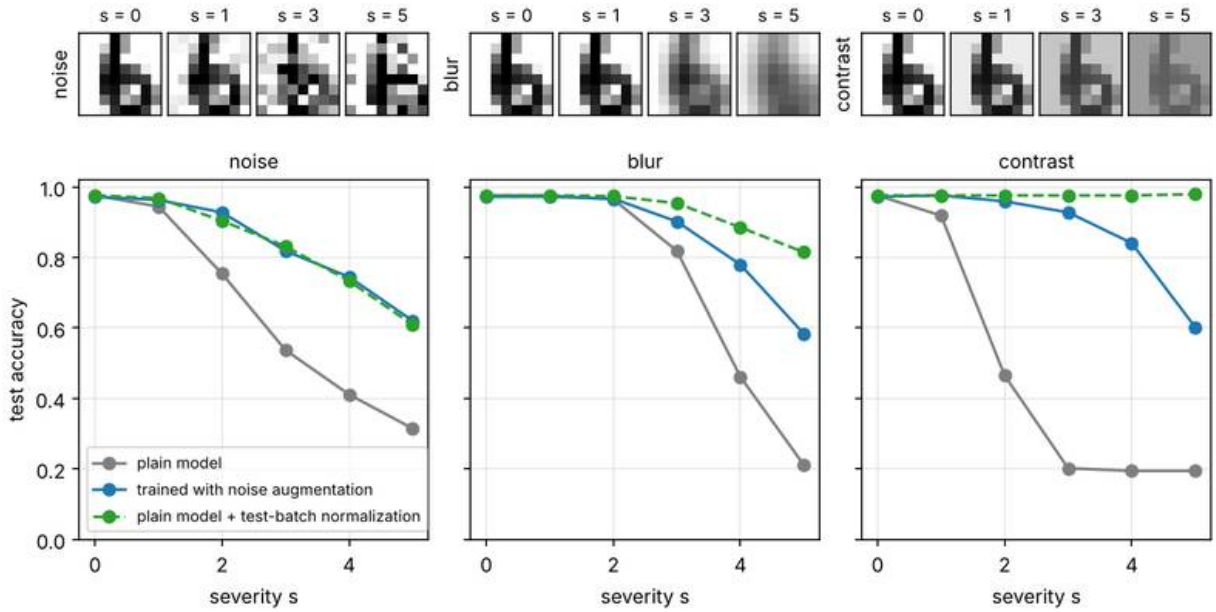



*Figure 2 — Accuracy of a small MLP on corrupted digits (our toy run). Top: one test image per corruption and severity. Bottom: plain model, model trained with noise augmentation, and plain model with per-pixel statistics recomputed on the corrupted test batch.*

(All numbers are from our toy run.) Four lessons carry over to real networks:

- **Mild-looking changes are enough.** A contrast reduction to 70% of the original range (severity 2) halves the plain model's accuracy, although a person reads the digits easily.
- **Augmentation helps, and not only for the corruption it used.** Training with noise also helped with blur and contrast here, which is the hope behind diverse augmentation. It cost a little clean accuracy (0.98 to 0.97), and real benchmarks show that the transfer to *other* shifts is far from guaranteed [5], [8].
- **Re-estimating normalization statistics is a cheap and strong fix when its assumption holds.** The contrast corruption is an affine change of every pixel, which per-pixel standardization undoes exactly, so accuracy stays at 0.98. Noise changes the statistics in a way standardization cannot undo, so the gain is smaller.
- **Test-batch normalization needs a batch.** It used all 540 test images of one corruption at once. With a few images, or with a batch that mixes several conditions, the statistics are wrong (Exercise 1).

### What helps, and what does not

**Plain version.** Interventions include diverse training, broader pretraining, test-time adaptation, target-domain adaptation and multi-domain training. Choose according to available data and deployment constraints.

**Data augmentation.** Augmentation makes the training distribution wider so that the test distribution is more likely to fall inside it. **AugMix** mixes several chains of random augmentations of one image, and adds a Jensen–Shannon consistency loss that asks the network to give similar predictions for the original and the augmented versions [10]. Its authors deliberately removed operations that overlap with ImageNet-C corruptions, so the benchmark measures transfer to unseen corruptions rather than memorization of the test corruptions [10]. Stylized training, which replaces image textures with painting styles, made a ResNet-50 rely more on shape and improved robustness to many distortions [11]. The caution comes from the ImageNet-R study: augmentation and larger models helped on some real-world shifts (style, blur) but not on others (geography), and no evaluated method consistently improved robustness across all of them [5]. General augmentation recipes are in DLCV 02.

**Pre-training on more diverse data.** In the testbed of Taori et al., most robustness interventions gave little or no effective robustness on natural shifts; the main exception was training on larger and more diverse data [8]. Zero-shot CLIP models, trained on image–text pairs from the web, improved effective robustness by a large amount on natural ImageNet shifts, but the CLIP paper also showed that the benefit almost vanished when a classifier was fitted on ImageNet in a fully supervised way [12]. Fang et al. tested five possible causes (dataset size, training distribution, language supervision during training, language supervision at test time, contrastive loss) and found that the **more diverse training distribution** was the main cause [13]. Fine-tuning can therefore erode robustness; WiSE-FT averages the weights of the zero-shot and fine-tuned models to keep most of both [14] (see also LP-FT in ML 10).

**Test-time adaptation (TTA).** If unlabeled test images arrive in batches, the model can adjust itself before predicting. The simplest form is the one in our toy: Schneider et al. replaced the batch-normalization statistics of 25 pretrained models with statistics of the corrupted test images and improved corruption robustness consistently, even when adapting with a single sample [15]. **TENT** goes one step further: it re-estimates the normalization statistics *and* updates the channel-wise affine parameters of the normalization layers online, by minimizing the entropy of the model's own predictions on each test batch [16]. TTA assumes a stream of test data from a reasonably stable shift; it adds computation at test time and can drift if the stream changes.

*Online only: Test-time entropy minimization — https://blog.shyandram.dev/en/learn/deep-learning/robustness/notes/test-time-entropy-minimization/*

**Unsupervised domain adaptation (UDA).** If unlabeled images from the target domain are available *during training*, the model can be trained to work there without target labels. **Domain-adversarial training (DANN)** adds a small domain classifier on top of the features and a **gradient-reversal layer** between them, so that the features become good for the label task on the labelled source domain and, at the same time, useless for telling source from target [17]. The idea rests on domain-adaptation theory: if a classifier cannot tell the domains apart in feature

space, a good source classifier should transfer.

*Online only: Domain-adversarial training and the gradient-reversal layer — https://blog.shyandram.dev/en/learn/deep-learning/robustness/notes/domain-adversarial-training-and-the-gradient-reversal-layer/*

**Domain generalization (DG).** Without any target data, the model is trained on several source domains in the hope that it works on an unseen one. The survey by Zhou et al. groups DG methods into domain alignment, meta-learning, data augmentation and ensemble learning, among others [18]. Gulrajani and Lopez-Paz then reran many DG algorithms in one testbed, DomainBed, with equal tuning effort and explicit model-selection rules, and found that plain empirical risk minimization (ERM, ordinary training on the pooled sources) with a strong network and augmentation matched or beat them [19]. Their key point is methodological: a DG method is incomplete without a rule for choosing hyperparameters that does not peek at the test domain.

*Online only: Gulrajani and Lopez-Paz, "In Search of Lost Domain Generalization" (ICLR 2021) — https://blog.shyandram.dev/en/learn/deep-learning/robustness/papers/gulrajani-and-lopez-paz-in-search-of-lost-domain-generalization-iclr-2021/*

**What does not help, or not reliably.** Robustness on synthetic corruptions was only weakly predictive of robustness on natural shifts in the testbed of Taori et al., and the same held for $\ell_p$ adversarial robustness [8]. A method tuned on one benchmark often helps only that benchmark [5]. Architecture changes at fixed data have mattered less than the data itself [8], [13]. The practical rule follows: test on several kinds of shift, including at least one real shift from your own deployment.

### Confidence under shift

**Plain version.** Under shift, a model is wrong more often, but it is usually not *less confident* by the same amount, so its probabilities stop meaning what they say.

Ovadia et al. benchmarked uncertainty methods under dataset shift on several tasks and found that calibration degrades as shift grows, that post-hoc calibration such as temperature scaling fitted on in-distribution validation data falls short under shift, and that methods that average over several models (such as deep ensembles) held up best [20]. Temperature scaling is explained in DLCV 02, and ML 11 shows on a toy how a fitted temperature stops working under shift. The practical consequence: a confidence threshold chosen on clean validation data will accept more wrong answers in the field, so monitor the distribution of confidences after deployment (ML 11).

## Adversarial examples

**Plain version.** For almost any correctly classified image, there is a nearby image, often indistinguishable to a person, that the network classifies wrongly. An attacker finds it by asking the network itself which direction to move each pixel.

### What an adversarial example is

Szegedy et al. reported that deep networks can be made to misclassify an image by an imperceptible perturbation found by maximizing the network's prediction error, and that the same perturbation often fools a different network trained on a different subset of the data [21]. Goodfellow et al. explained the effect by the near-linearity of networks in high dimensions and introduced the one-step fast gradient sign method [22], which DLCV 02 derives and tests. This section adds the parts needed to *evaluate* and *improve* robustness: threat models, the multi-step attack, adversarial training, honest evaluation and certificates.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

### Threat models

**Plain version.** "Robust" means nothing until you say robust to what: how much may the attacker change, and what does the attacker know?

A **threat model** fixes three things.

**1. The allowed perturbations.** Most work uses an $\ell_p$ ball of radius $\varepsilon$ around the clean image $\mathbf{x} \in [0,1]^d$:

$$\mathcal{S}_p(\mathbf{x}) = \{\mathbf{x}' \in [0,1]^d : \|\mathbf{x}' - \mathbf{x}\|_p \le \varepsilon\}, \qquad \|\boldsymbol{\delta}\|_\infty = \max_i |\delta_i|, \qquad \|\boldsymbol{\delta}\|_2 = \Big(\sum_i \delta_i^2\Big)^{1/2},$$

where $d$ is the number of pixel values, $\boldsymbol{\delta} = \mathbf{x}' - \mathbf{x}$ is the perturbation and $\delta_i$ its $i$-th entry. The $\ell_\infty$ ball lets every pixel change by at most $\varepsilon$ (a small change everywhere); the $\ell_2$ ball bounds the total energy of the change; the $\ell_0$ "norm" counts how many pixels change (a few pixels, changed a lot). On CIFAR-10, $\varepsilon = 8/255$ in $\ell_\infty$ is a standard budget: Madry et al. trained and attacked with $\varepsilon = 8$ on the 0–255 pixel scale [23], and RobustBench keeps leaderboards for $\ell_\infty$ and $\ell_2$ threat models and for common corruptions [24]. A small $\ell_p$ norm is a convenient proxy for "imperceptible", not a definition of it.

**2. The attacker's knowledge.** In a **white-box** attack, the attacker knows the model and its weights and can compute gradients. In a **black-box** attack, the attacker only queries the model. Papernot et al. showed a practical black-box attack: label inputs by querying the target model, train a local substitute model on them, craft adversarial examples on the substitute, and rely on **transfer** to the target [25]. Other black-box attacks estimate the decision boundary from queries alone. Robustness claims should be made against white-box attacks, because what one attacker cannot do today, a better-informed one may do tomorrow.

**3. The goal.** An **untargeted** attack wants any wrong label; a **targeted** attack wants a specific one (a stop sign read as a speed limit). Targeted attacks are harder, so the two give different numbers.

### FGSM and PGD, precisely

**Plain version.** FGSM takes one big step uphill on the loss. PGD takes many small steps uphill and, after each step, pulls the image back into the allowed ball. PGD finds stronger adversarial examples and is the standard first-order attack.

Let $\mathcal{L}(\boldsymbol{\theta}, \mathbf{x}, y)$ be the cross-entropy loss of the network with parameters $\boldsymbol{\theta}$ on input $\mathbf{x}$ with label $y$. **FGSM** [22] for the $\ell_\infty$ ball is

$$\mathbf{x}_{\text{adv}} = \Pi_{[0,1]^d}(\mathbf{x} + \varepsilon \ \text{sign}(\nabla_{\mathbf{x}} \mathcal{L}(\boldsymbol{\theta}, \mathbf{x}, y))),$$

where $\nabla_{\mathbf{x}} \mathcal{L}$ is the gradient of the loss with respect to the input pixels, sign is applied to each entry (giving $-1$, $0$ or $+1$), and $\Pi_{[0,1]^d}$ clips each pixel to the valid range. Moving every pixel by exactly $\pm\varepsilon$ in the direction that increases the loss is the best a *linear* approximation of the loss can do inside the $\ell_\infty$ ball.

**Projected gradient descent (PGD)** [23] repeats a smaller step $T$ times, starting from a random point in the ball:

$$\mathbf{x}^{(0)} = \mathbf{x} + \mathbf{u}, \quad \mathbf{u} \sim \mathcal{U}[-\varepsilon, \varepsilon]^d, \qquad \mathbf{x}^{(t+1)} = \Pi_{\mathcal{S}(\mathbf{x})}(\mathbf{x}^{(t)} + \alpha \ \text{sign}(\nabla_{\mathbf{x}} \mathcal{L}(\boldsymbol{\theta}, \mathbf{x}^{(t)}, y))),$$

where $\mathbf{u}$ is uniform noise in the ball (the random start), $\alpha$ is the step size, $t = 0, \dots, T-1$ counts the steps, and $\Pi_{\mathcal{S}(\mathbf{x})}$ is the projection onto the allowed set. For the $\ell_\infty$ ball intersected with the valid pixel range, the projection is two clips: first $x_i' \leftarrow \min(\max(x_i', x_i - \varepsilon), x_i + \varepsilon)$, then $x_i' \leftarrow \min(\max(x_i', 0), 1)$. The final iterate $\mathbf{x}^{(T)}$ is the adversarial example. PGD is "gradient *ascent* on the loss"; the name comes from optimization, where the same algorithm minimizes. Because the loss surface is not concave, the random start and a few restarts help; Madry et al. observed that the losses reached from many random starts were well concentrated, which supported PGD as a reliable first-order adversary [23].

A second common loss is the **margin loss** of Carlini and Wagner, which directly maximizes the gap between the largest wrong logit and the correct logit instead of the cross-entropy [26]. It matters when the cross-entropy saturates, and it is one reason a serious evaluation runs several attacks.

*Online only: Attack settings that decide an evaluation — https://blog.shyandram.dev/en/learn/deep-learning/robustness/notes/attack-settings-that-decide-an-evaluation/*

### Adversarial training

**Plain version.** If the model fails on the worst nearby image, train it on the worst nearby image. During training, attack every batch with PGD and fit the attacked images instead of the clean ones.

Madry et al. wrote robust training as a min-max (saddle point) problem [23]:

$$\min_{\boldsymbol{\theta}} \ \rho(\boldsymbol{\theta}), \qquad \rho(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}\Big[\max_{\boldsymbol{\delta}\in\mathcal{S}} \mathcal{L}(\boldsymbol{\theta}, \mathbf{x} + \boldsymbol{\delta}, y)\Big],$$

where $\mathcal{D}$ is the data distribution, $\mathcal{S}$ the set of allowed perturbations (for example, $\|\delta\|_\infty \le \varepsilon$), the inner maximum is the attacker's best perturbation and the outer minimum is the defender's training. In practice, the inner maximum is approximated by PGD with a few steps and the outer minimum by ordinary stochastic gradient descent on the resulting adversarial batch.

What it costs:

- **Compute.** Every training step needs a $T$-step attack, so training is roughly $T + 1$ times as expensive.
- **Capacity.** Madry et al. found that robust classification needs larger networks than clean classification; small networks trained against PGD could collapse to predicting one class, and training against one-step FGSM overfitted to its narrow set of perturbations and did not give robustness to PGD at large $\varepsilon$ [23].
- **Clean accuracy.** Robust models are usually less accurate on clean data. Tsipras et al. proved that this trade-off exists in a simple, natural setting and argued that robust models learn different features, which happen to align better with human perception [27]. **TRADES** splits the robust error into the natural error plus a boundary error and trains with a loss that trades the two explicitly through one weight [28].

*Online only: Madry et al., "Towards Deep Learning Models Resistant to Adversarial Attacks" (ICLR 2018) — https://blog.shyandram.dev/en/learn/deep-learning/robustness/papers/madry-et-al-towards-deep-learning-models-resistant-to-adversarial-attacks-iclr-2018/*

### Code: FGSM, PGD and adversarial training on digits

NumPy computes the input gradient of a one-hidden-layer MLP by hand: for softmax outputs $\mathbf{p}$ and a one-hot label $\mathbf{e}_y$, the gradient of the cross-entropy with respect to the logits is $\mathbf{p} - \mathbf{e}_y$, which we pass back through the output layer, the ReLU and the first layer. Both models get the same 50 passes over the training data; the robust one is fitted on PGD examples ($\varepsilon = 0.15$, 5 steps) made against its current weights.

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

X, y = load_digits(return_X_y=True)
X = X / 16.0                                          # pixels in [0, 1]
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3, stratify=y, random_state=0)
rng = np.random.default_rng(0)

def input_grad(m, X, y):
    """Gradient of the cross-entropy loss with respect to the input pixels."""
    (W1, W2), (b1, b2) = m.coefs_, m.intercepts_
    h = np.maximum(X @ W1 + b1, 0)                    # hidden ReLU layer
    z = h @ W2 + b2                                   # logits
    g = np.exp(z - z.max(1, keepdims=True)); g /= g.sum(1, keepdims=True)
    g[np.arange(len(y)), y] -= 1                      # dL/dz = softmax(z) - one-hot(y)
```

```
    return ((g @ W2.T) * (h > 0)) @ W1.T             # back through ReLU and first layer

def fgsm(m, X, y, eps):
    return np.clip(X + eps * np.sign(input_grad(m, X, y)), 0, 1)

def pgd(m, X, y, eps, steps=10):
    alpha = 2.5 * eps / steps                        # step size
    Xa = np.clip(X + rng.uniform(-eps, eps, X.shape), 0, 1)        # random start in the ball
    for _ in range(steps):
        Xa = Xa + alpha * np.sign(input_grad(m, Xa, y))             # ascent step on the loss
        Xa = np.clip(np.clip(Xa, X - eps, X + eps), 0, 1)           # project: eps-ball, then
         ↪  [0, 1]
    return Xa

def train(adversarial, epochs=50, eps=0.15):
    m = MLPClassifier(hidden_layer_sizes=(128,), random_state=0)
    m.partial_fit(Xtr, ytr, classes=np.arange(10))    # first pass on clean images
    for _ in range(epochs - 1):                       # Madry et al.: fit the worst case
     ↪  found by PGD
        m.partial_fit(pgd(m, Xtr, ytr, eps, steps=5) if adversarial else Xtr, ytr)
    return m

standard, robust = train(False), train(True)
for eps in [0, 0.05, 0.1, 0.15, 0.2]:
    acc = [m.score(atk(m, Xte, yte, eps), yte) for m in (standard, robust) for atk in (fgsm,
     ↪  pgd)]
    print(f"eps={eps:.2f}  standard: FGSM {acc[0]:.3f} PGD {acc[1]:.3f}"
          f"   adversarially trained: FGSM {acc[2]:.3f} PGD {acc[3]:.3f}")
```

It prints (about 10–20 seconds on a laptop CPU):

```
eps=0.00  standard: FGSM 0.967 PGD 0.967   adversarially trained: FGSM 0.956 PGD 0.956
eps=0.05  standard: FGSM 0.861 PGD 0.854   adversarially trained: FGSM 0.891 PGD 0.889
eps=0.10  standard: FGSM 0.574 PGD 0.528   adversarially trained: FGSM 0.757 PGD 0.746
eps=0.15  standard: FGSM 0.237 PGD 0.178   adversarially trained: FGSM 0.626 PGD 0.593
eps=0.20  standard: FGSM 0.057 PGD 0.007   adversarially trained: FGSM 0.415 PGD 0.361
```

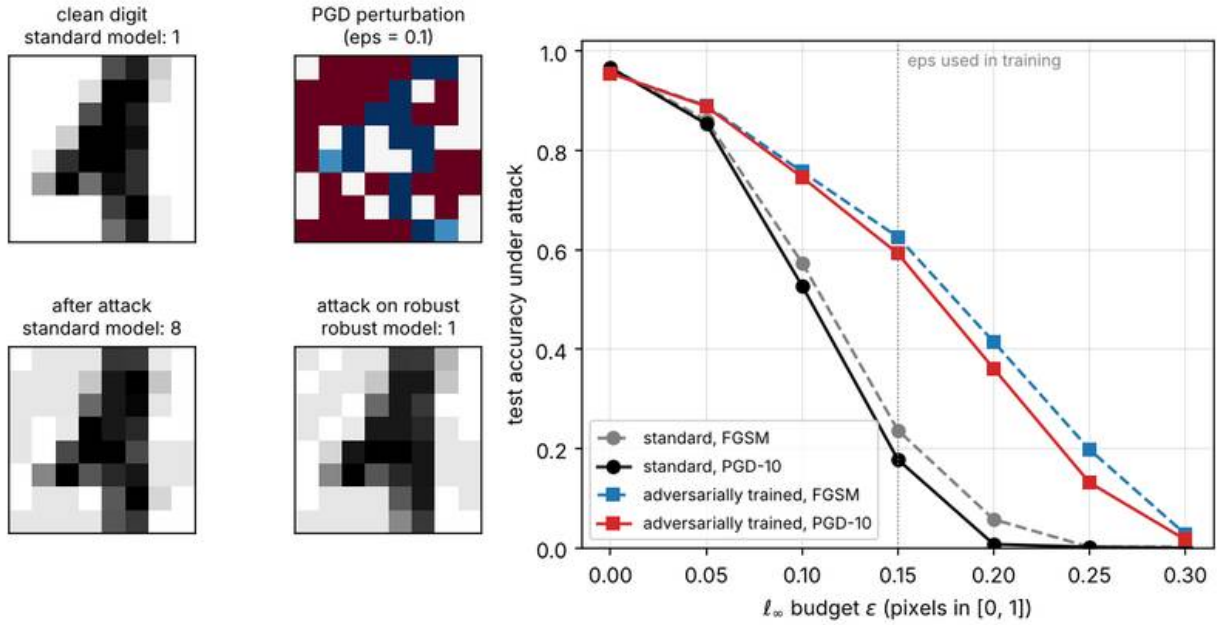



*Figure 3 — FGSM and PGD on two MLPs (our toy run). Left: a PGD perturbation with $\varepsilon = 0.1$ changes the standard model's answer from 1 to 8 but not the adversarially trained model's. Right: accuracy under attack against the budget $\varepsilon$; PGD is always at least as strong as FGSM, and adversarial training raises robust accuracy at every $\varepsilon$ at a small cost in clean accuracy.*

(All numbers are from our toy run.) Read the table in three directions:

- **Down a column:** accuracy falls quickly with $\varepsilon$. At $\varepsilon = 0.1$, each pixel may change by 1.6 of the dataset's 16 gray levels, and the standard model drops from 0.967 to 0.528 under PGD.
- **Across an attack pair:** PGD is never weaker than FGSM, and the gap grows with $\varepsilon$ (0.057 versus 0.007 at $\varepsilon = 0.2$ for the standard model). An evaluation with FGSM alone overstates robustness.
- **Across models:** adversarial training raises PGD accuracy at $\varepsilon = 0.15$ from 0.178 to 0.593, and costs about one point of clean accuracy (0.967 to 0.956), a small version of the trade-off. Beyond the training budget, robustness fades (0.131 at $\varepsilon = 0.25$ in the figure): robustness is guaranteed for nothing outside the threat model it was trained for.

### Evaluating robustness honestly

**Plain version.** It is easy to build a "defense" that only breaks the attacker's tools, for example by making gradients useless, while the model stays just as easy to fool. The robustness then exists only on paper, until someone uses a better attack.

Athalye, Carlini and Wagner named this **obfuscated gradients**, a kind of gradient masking, and described three types [29]: **shattered gradients** (non-differentiable or numerically unstable operations, such as quantizing the input, give gradients that are zero or wrong); **stochastic gradients** (random input transformations at test time make a single gradient misleading); and **vanishing**

**or exploding gradients** (defenses that run an iterative procedure, such as projecting the input onto a learned manifold, behave like an extremely deep network). For each type they gave a counter-attack: **BPDA** (backward pass differentiable approximation) runs the defense normally in the forward pass but replaces it by a differentiable approximation, often the identity, in the backward pass; **EOT** (expectation over transformation) averages gradients over the random transformations; and **reparameterization** changes variables so that the iterative step disappears from the gradient path. In a case study of nine white-box defenses accepted at ICLR 2018, seven relied on obfuscated gradients, and their attacks fully circumvented six and partially one [29].

The same paper lists warning signs that an evaluation is broken: one-step attacks work better than iterative ones; black-box attacks work better than white-box ones; unbounded attacks do not reach 100% success; random sampling finds adversarial examples that gradient attacks miss; and increasing the budget does not increase success [29]. Our toy passes the checks it can: PGD beats FGSM, and accuracy reaches zero as $\varepsilon$ grows.

To make evaluations less dependent on the evaluator's care, Croce and Hein proposed **AutoAttack**, a parameter-free ensemble of two improved PGD variants (with an automatic step-size schedule and a different loss) and two complementary existing attacks; applied to over 50 published defenses, it gave lower robust accuracy than reported in all but one case [30]. **RobustBench** builds a public leaderboard on AutoAttack with rules on allowed models, and invites adaptive attacks where AutoAttack may overestimate robustness [24]. A reported robust accuracy should therefore state the threat model, $\varepsilon$, the attacks (at least AutoAttack or an adaptive attack designed for the defense), the number of steps and restarts, and the test set.

*Online only: Athalye et al., "Obfuscated Gradients Give a False Sense of Security" (ICML 2018) — https://blog.shyandram.dev/en/learn/deep-learning/robustness/papers/athalye-et-al-obfuscated-gradients-give-a-false-sense-of-security-icml-2018/*

### Certified robustness

**Plain version.** An attack can only show that a model is *not* robust at a point. A certificate proves that *no* perturbation within a radius changes the answer, whatever attack is used.

**Randomized smoothing** [31] was, according to its authors, the only certified defense shown to be feasible on ImageNet at the time, and it remains the standard reference point. Take any base classifier $f$ and define a smoothed classifier $g$ that returns the class $f$ predicts most often under Gaussian noise:

$$g(\mathbf{x}) = \arg\max_{c \in \mathcal{Y}} \Pr_{\boldsymbol{\eta} \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})} \left[ f(\mathbf{x} + \boldsymbol{\eta}) = c \right],$$

where $\boldsymbol{\eta}$ is isotropic Gaussian noise with standard deviation $\sigma$ per pixel and $\mathbf{I}$ is the identity matrix. Cohen, Rosenfeld and Kolter proved that if the top class $c_A$ has probability at least $\underline{p_A}$ under the noise and every other class at most $\overline{p_B}$, then $g$ predicts $c_A$ for every perturbation $\delta$ with

$$\|\delta\|_2 < R = \frac{\sigma}{2} \left( \Phi^{-1}(\underline{p_A}) - \Phi^{-1}(\overline{p_B}) \right),$$

where $\Phi^{-1}$ is the inverse of the standard normal cumulative distribution function [31]. In practice,

$\underline{p_A}$ is a lower confidence bound estimated from Monte Carlo samples, and the method abstains when it cannot certify. The guarantee holds for the $\ell_2$ threat model, with a chosen failure probability, and for the smoothed classifier $g$, not for $f$. Larger $\sigma$ allows larger radii but lowers clean accuracy, the same trade-off in another form.

*Online only: Randomized smoothing in practice — https://blog.shyandram.dev/en/learn/deep-learning/robustness/notes/randomized-smoothing-in-practice/*

### Physical and patch attacks

**Plain version.** An attack does not have to be invisible or digital. A printed sticker can fool a camera.

Eykholt et al. generated perturbations that survive changes of viewpoint, distance and lighting, printed them as black-and-white stickers on a real stop sign, and caused a road-sign classifier to output the attacker's target class in most lab images and in many frames of a drive-by video [32]. Brown et al. made **adversarial patches**: universal, printable image regions that, placed anywhere in a scene, make a classifier report a chosen class [33]. Such attacks leave the $\ell_p$ threat model: they are large but localized. They matter for any camera-based system, and they connect to the presentation attacks on face recognition discussed in DLCV 02A.

## Out-of-distribution and anomaly detection

**Plain version.** A classifier always picks one of its known classes, even for a photo of something it has never seen. OOD detection adds a second output, a score that says "this input looks like my training data" or "it does not", so the system can refuse to answer.

### Which task are you solving?

Several closely related tasks go by different names. Yang et al. place them in one framework, **generalized OOD detection**, and distinguish them by what kind of shift must be detected and by whether ID data has one class or many [3]:

| Task | In-distribution data | Must flag | Typical example |
|---|---|---|---|
| **OOD detection** | a $K$-class training set | inputs from other classes (semantic shift), while still classifying ID inputs | a pet classifier sees a car |
| **Open-set recognition** | $K$ known classes | unknown classes, while classifying known ones | PR 10, DLCV 02A |
| **Sensory anomaly detection** | normal images of one kind | inputs that look different (covariate shift) | a scratched part on a production line |
| **Semantic anomaly / novelty detection** | one or a few classes | inputs from any other class | a new species in camera-trap images |

OOD detection and open-set recognition differ mainly in their history and benchmarks; the survey treats open-set recognition as multi-class novelty detection [3]. All of them differ from the **reject option** of PR 01, which abstains on *ambiguous* inputs between known classes rather than on *novel* ones.

### Scores

**Plain version.** Each detector boils down to one number per input. Output-based scores ask how confident the classifier is; feature-based scores ask how far the input's features lie from the training features.

Let $\mathbf{z}(\mathbf{x}) \in \mathbb{R}^K$ be the logits of a trained classifier, $\mathbf{h}(\mathbf{x}) \in \mathbb{R}^m$ its penultimate features (the input to the last linear layer), and adopt the convention that a **larger score means more in-distribution**. Four standard post-hoc scores, which need no retraining:

**Maximum softmax probability (MSP)** [34]:

$$s_{\text{MSP}}(\mathbf{x}) = \max_k \frac{e^{z_k(\mathbf{x})}}{\sum_{j=1}^{K} e^{z_j(\mathbf{x})}},$$

where $z_k$ is the $k$-th logit. Hendrycks and Gimpel showed that correctly classified examples tend to have higher MSP than misclassified and OOD ones, which made MSP the standard baseline [34].

**Energy score** [35]:

$$E(\mathbf{x}) = -T \log \sum_{k=1}^{K} e^{z_k(\mathbf{x})/T}, \qquad s_{\text{energy}}(\mathbf{x}) = -E(\mathbf{x}),$$

where $T > 0$ is a temperature (usually $T = 1$). The negative energy is a log-sum-exp of the logits, a smooth maximum that keeps the *size* of the logits, which the softmax normalizes away. Liu et al. argued that energy is better aligned with the input density than the softmax confidence, and reported lower FPR at 95% TPR than MSP on their CIFAR benchmarks [35].

**Mahalanobis distance** [36]: fit one Gaussian per class to the training features, with class means $\boldsymbol{\mu}_c$ and one shared ("tied") covariance $\boldsymbol{\Sigma}$, and score

$$s_{\text{Maha}}(\mathbf{x}) = -\min_c \, (\mathbf{h}(\mathbf{x}) - \boldsymbol{\mu}_c)^{\top} \boldsymbol{\Sigma}^{-1} (\mathbf{h}(\mathbf{x}) - \boldsymbol{\mu}_c),$$

where the minimum runs over the $K$ classes. This is the linear discriminant analysis of PR 01 placed on learned features. Lee et al. also added a small input perturbation and combined several layers; we use the last hidden layer only.

**k-nearest-neighbour (kNN) distance** [37]: normalize features to unit length, $\hat{\mathbf{h}} = \mathbf{h}/\|\mathbf{h}\|_2$, and score

$$s_{\text{kNN}}(\mathbf{x}) = -\|\hat{\mathbf{h}}(\mathbf{x}) - \hat{\mathbf{h}}_{(k)}\|_2,$$

where $\hat{\mathbf{h}}_{(k)}$ is the $k$-th nearest normalized training feature to $\hat{\mathbf{h}}(\mathbf{x})$. Sun et al. found that this non-parametric score, with no Gaussian assumption, works well and that feature normalization is critical [37].

These scores leave the classifier unchanged. Training-time methods change it: **Outlier Exposure** trains the classifier to give uniform (low-confidence) predictions on a large auxiliary set of outlier images, which improved detection of unseen OOD data in the original experiments [38].

### Metrics

**Plain version.** Treat detection as a binary decision, ID versus OOD, and ask how well the score separates the two groups, either over all thresholds or at one threshold that keeps most ID inputs.

Treat ID test inputs as positives and OOD inputs as negatives, and accept an input when $s(\mathbf{x}) \geq \tau$. Then $\mathrm{TPR}(\tau)$ is the fraction of ID inputs accepted and $\mathrm{FPR}(\tau)$ the fraction of OOD inputs accepted. Three metrics are standard [35]:

- **AUROC**, the area under the curve of TPR against FPR as $\tau$ varies. It equals the probability that a random ID input gets a higher score than a random OOD input; 0.5 is chance, 1 is perfect separation.
- **FPR@95TPR** $= \mathrm{FPR}(\tau_{95})$, where $\tau_{95}$ is the threshold with $\mathrm{TPR}(\tau_{95}) = 0.95$: the fraction of OOD inputs still accepted when 95% of ID inputs are kept. Lower is better. It describes one operating point, the kind a deployed system actually uses.
- **AUPR**, the area under the precision–recall curve, reported with ID inputs as the positive class (AUPR-In) or with OOD inputs as the positive class (AUPR-Out) [34]. It depends on the ratio of ID to OOD test inputs, which should therefore be reported.

ROC and precision–recall curves themselves are explained in DLCV 02.

### Code: four scores, near and far OOD

ID data are the digits 0–4. **Near-OOD** inputs are the unseen digits 5–9: the same kind of image, new classes. **Far-OOD** inputs are 8×8 patches of photographs from scikit-image, rescaled to the same pixel range: a different kind of image altogether. The classifier never sees either during training.

```
import numpy as np
from scipy.special import logsumexp, softmax
from skimage import data, transform
from sklearn.datasets import load_digits
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

X, y = load_digits(return_X_y=True)
X = X / 16.0
known = y < 5                                        # in-distribution (ID): digits 0-4
```

```
Xtr, Xid, ytr, yid = train_test_split(X[known], y[known], test_size=0.3, stratify=y[known],
 ↪  random_state=0)
X_near = X[~known]                                      # near-OOD: unseen digits 5-9
rng = np.random.default_rng(0)
photos = [transform.resize(im / 255.0, (256, 256)) for im in (data.camera(), data.moon(),
 ↪  data.coins(), data.text())]
X_far = []                                              # far-OOD: 8x8 patches of photographs
for _ in range(600):
    im = photos[rng.integers(4)]; r, c = rng.integers(0, 224, 2)
    p = transform.resize(im[r:r + 32, c:c + 32], (8, 8), anti_aliasing=True)
    X_far.append(((p - p.min()) / (np.ptp(p) + 1e-8)).ravel())
X_far = np.array(X_far)

m = MLPClassifier(hidden_layer_sizes=(64,), alpha=1e-3, max_iter=500,
 ↪  random_state=0).fit(Xtr, ytr)
(W1, W2), (b1, b2) = m.coefs_, m.intercepts_
feat = lambda X: np.maximum(X @ W1 + b1, 0)             # penultimate (hidden) features
unit = lambda F: F / (np.linalg.norm(F, axis=1, keepdims=True) + 1e-12)
F = feat(Xtr)
mu = np.stack([F[ytr == c].mean(0) for c in range(5)])           # class means
cov = sum(np.cov(F[ytr == c].T, bias=True) * np.sum(ytr == c) for c in range(5)) / len(F)
P = np.linalg.inv(cov + 1e-3 * np.eye(len(cov)))                # shared precision matrix

def scores(X):                                          # every score: larger = "more
 ↪  in-distribution"
    H = feat(X); z = H @ W2 + b2; d = H[:, None] - mu[None]
    return {"MSP": softmax(z, axis=1).max(1),
            "energy": logsumexp(z, axis=1),             # = -E(x) with temperature 1
            "Mahalanobis": -np.einsum("ncd,de,nce->nc", d, P, d).min(1),
            "kNN": -np.sort(np.linalg.norm(unit(H)[:, None] - unit(F)[None], axis=2), 1)[:,
             ↪  9]}

s_id = scores(Xid)
print(f"ID test accuracy {m.score(Xid, yid):.3f}")
for name, X_ood in [("near", X_near), ("far", X_far)]:
    s_ood = scores(X_ood)
    for k in s_id:
        auroc = roc_auc_score(np.r_[np.ones(len(Xid)), np.zeros(len(X_ood))], np.r_[s_id[k],
         ↪  s_ood[k]])
        fpr95 = np.mean(s_ood[k] >= np.quantile(s_id[k], 0.05))  # OOD accepted when 95% of
         ↪  ID is kept
        print(f"{name:4s} {k:11s} AUROC {auroc:.3f}  FPR@95TPR {fpr95:.3f}")
```

It prints:

```
ID test accuracy 1.000
near MSP         AUROC 0.941  FPR@95TPR 0.291
near energy      AUROC 0.955  FPR@95TPR 0.258
near Mahalanobis AUROC 0.961  FPR@95TPR 0.209
near kNN         AUROC 0.970  FPR@95TPR 0.152
far  MSP         AUROC 0.870  FPR@95TPR 0.448
far  energy      AUROC 0.811  FPR@95TPR 0.660
far  Mahalanobis AUROC 1.000  FPR@95TPR 0.000
```

```
far  kNN         AUROC 0.995  FPR@95TPR 0.010
```

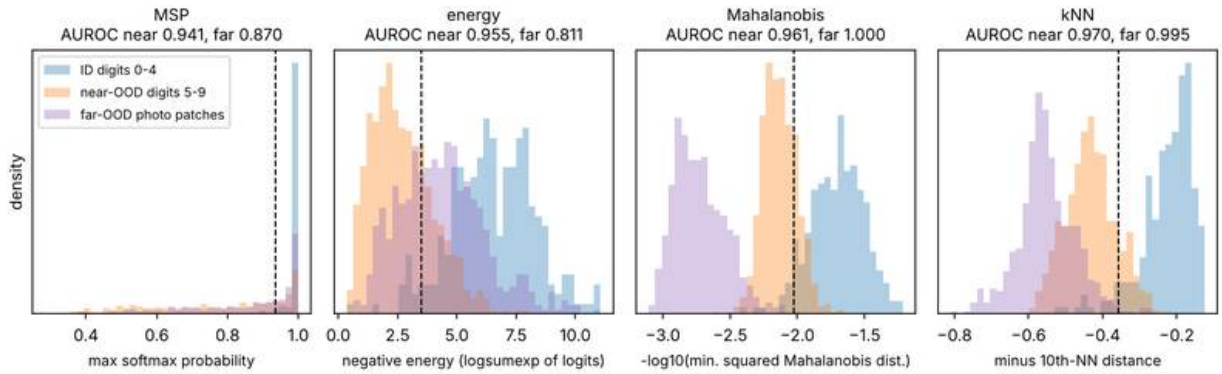



*Figure 4 — Score distributions for ID digits 0–4, near-OOD digits 5–9 and far-OOD photograph patches (our toy run). The dashed line is the threshold that accepts 95% of ID test inputs; everything to its right is accepted. The Mahalanobis panel uses a logarithmic scale for display only, which does not change any ranking.*

(All numbers are from our toy run.) Three lessons:

- **"Far" in content is not "easy" for every score.** The photo patches are nothing like digits, yet MSP accepts 45% and energy 66% of them at the 95%-TPR threshold. The classifier maps them confidently onto some digit, because confidence measures distance from the decision boundary, not distance from the data. Feature-distance scores see that the patches' features are far from all training features and reject almost all of them.
- **The ranking of scores changes with the OOD set.** Energy beats MSP on the unseen digits but loses to it on the patches. A large benchmark study, OpenOOD v1.5, likewise found no single method that wins across datasets [39].
- **AUROC and FPR@95TPR disagree in size.** An AUROC of 0.941 sounds good, but at a deployable threshold MSP still accepts 29% of unseen digits.

PR 10 runs a related open-set experiment with held-out digits on a different classifier and shows how a proxy-unknown set can mislead (PR 10).

### Industrial anomaly detection

**Plain version.** On a production line, defects are rare and varied, so you cannot collect examples of every defect. Instead, you learn only what *normal* parts look like, patch by patch, and flag any patch that looks unlike every normal patch you have seen.

**MVTec AD** made this setting a standard benchmark: 5354 high-resolution images of object and texture categories, defect-free images for training, test images with more than 70 types of defects (scratches, dents, contaminations, structural changes), and pixel-precise masks of the defect regions [40]. Methods are scored at two levels: **image-level** (is this part defective?) and **pixel-level**

(where is the defect?), usually with AUROC; region-overlap measures such as AU-PRO are also used, for example by MVTec AD 2 [41]. This is *sensory* anomaly detection in the terms of the table above.

**PatchCore** [42] is a strong, simple method built on the kNN idea. It passes defect-free training images through a network pre-trained on ImageNet and keeps **mid-level patch features** (the last layers are too specific to ImageNet classes and too coarse for small defects), each averaged over a small neighbourhood for context. All these features go into a **memory bank**, which **greedy coreset subsampling** shrinks to a small subset that still covers rare normal patterns. At test time, each patch's **distance to its nearest neighbour** in the memory bank forms the anomaly map, and the map's maximum (slightly re-weighted) is the image score. The method was designed for the "cold-start" setting in which only nominal (defect-free) images are available, and it reached state-of-the-art detection and localization on MVTec AD [42].

*Online only: PatchCore step by step — https://blog.shyandram.dev/en/learn/deep-learning/robustness/notes/patchcore-step-by-step/*

### Code: a patch memory bank on a brick wall

Without network weights we use raw 7×7 pixel patches as features, which keeps the mechanics of PatchCore (memory bank, coreset, nearest-neighbour distance map, max-pooled image score) and drops the learned features. "Products" are 64×64 crops of scikit-image's brick-wall texture: normal training crops from the top half of the image, test crops from the bottom half, and half of the test crops receive a thin synthetic scratch.

```
import numpy as np
from skimage import data, draw, transform
from skimage.util import view_as_windows
from sklearn.metrics import roc_auc_score
from sklearn.neighbors import NearestNeighbors

rng = np.random.default_rng(0)
tex = transform.rescale(data.brick() / 255.0, 0.5)   # 256x256 brick-wall "product surface"
S, P = 64, 7                                          # image size, patch size

def crop(top):                                        # normal images: top half; test images:
 ↪  bottom half
    r = rng.integers(0, 128 - S) + (0 if top else 128); c = rng.integers(0, 256 - S)
    return tex[r:r + S, c:c + S].copy()

def scratch(im):                                      # a thin line, 0.5 brighter or darker
    r, c = rng.integers(16, S - 16, 2); a = rng.uniform(0, np.pi); L = rng.integers(8, 16)
    rr, cc = draw.line(r, c, int(r + L * np.sin(a)), int(c + L * np.cos(a)))
    mask = np.zeros_like(im, bool); mask[rr, cc] = True
    im[mask] = np.clip(im[mask] + rng.choice([-0.5, 0.5]), 0, 1)
    return im, mask

def patches(im, step=1):                              # one 7x7 patch centred on each pixel
    return view_as_windows(np.pad(im, P // 2, mode="reflect"), (P, P), step).reshape(-1, P *
     ↪  P)
```

```
def greedy_coreset(M, n):                              # keep n patches that cover the bank well
    keep, d = [0], np.linalg.norm(M - M[0], axis=1)
    for _ in range(n - 1):
        i = int(d.argmax()); keep.append(i)
        d = np.minimum(d, np.linalg.norm(M - M[i], axis=1))
    return M[keep]

bank = np.vstack([patches(crop(top=True), step=2) for _ in range(16)])
memory = NearestNeighbors(n_neighbors=1).fit(greedy_coreset(bank, len(bank) // 10))

is_defect, img_score, pix_score, pix_true = [], [], [], []
for k in range(40):
    im, mask = scratch(crop(top=False)) if k % 2 else (crop(top=False), np.zeros((S, S), bool))
    amap = memory.kneighbors(patches(im))[0].reshape(S, S)   # nearest-neighbour distance per pixel
    is_defect.append(k % 2); img_score.append(amap.max())
    pix_score.append(amap.ravel()); pix_true.append(mask.ravel())
print(f"memory bank {len(bank)} -> {len(bank) // 10} patches")
print(f"image-level AUROC {roc_auc_score(is_defect, img_score):.3f}")
print(f"pixel-level AUROC {roc_auc_score(np.concatenate(pix_true), np.concatenate(pix_score)):.3f}")
```

It prints:

```
memory bank 16384 -> 1638 patches
image-level AUROC 0.955
pixel-level AUROC 0.996
```

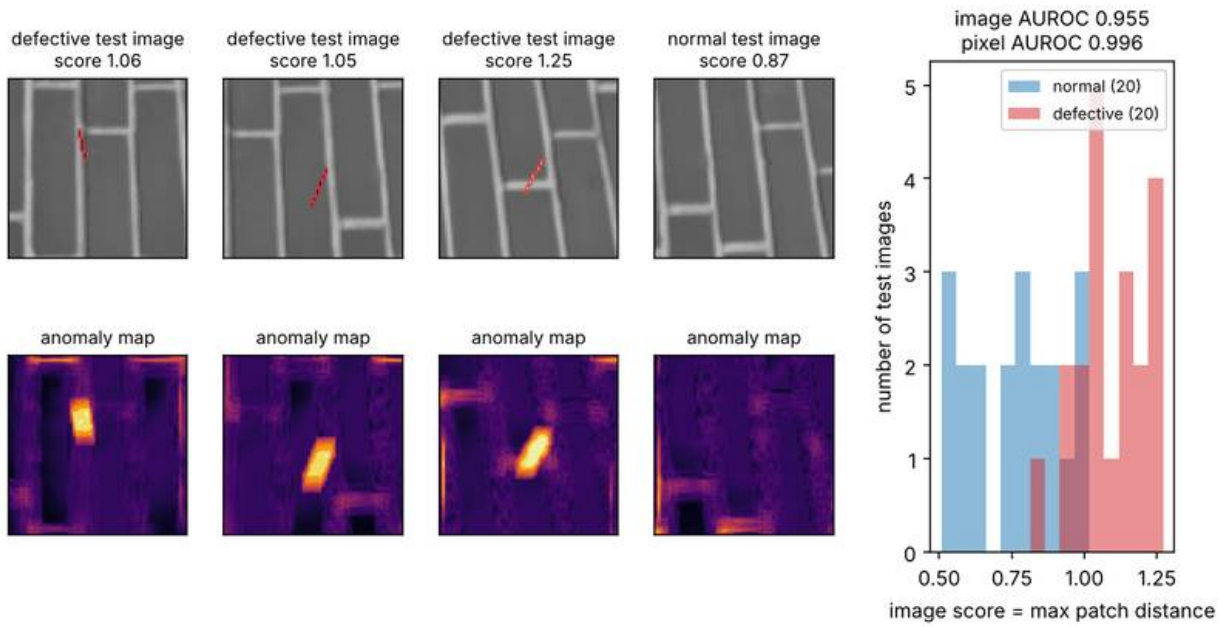



*Figure 5 — A patch memory bank on brick-wall crops (our toy run). Top: test images (red outline: the synthetic scratch). Bottom: nearest-neighbour distance maps. Right: image scores (maximum of the map) for 20 normal and 20 defective test images.*

(All numbers are from our toy run.) The pixel-level AUROC is almost perfect, but the image-level AUROC is lower: one image score is the *maximum* over 4096 patch distances, so a single unusual but normal patch (a mortar corner that the memory bank lacks) raises a normal image's score as much as a real defect does. More normal training images reduce such false alarms; fainter defects increase them (Exercise 4). In the anomaly maps, mortar edges glow a little: they are the rarest normal structures, which is exactly why PatchCore keeps a coreset that covers them instead of a random subsample.

### Pitfalls in OOD and anomaly evaluation

**Plain version.** OOD numbers depend more on *which* OOD data was chosen than on the method, and many published OOD test sets are flawed. Read every OOD result together with its test sets.

- **Near- versus far-OOD.** OpenOOD groups OOD test sets into near-OOD (similar to the ID data) and far-OOD (clearly different), and finds a large gap in detection performance between the two groups for all detectors [39]. Our toy shows that the label can mislead: for output-based scores, "far" photo patches were harder than "near" digits.
- **Contaminated OOD test sets.** Bitterwolf et al. found that many OOD test sets used with ImageNet-1K contain a large share of images that actually show ID classes, in some cases more than half, which distorts the evaluation; they built NINCO, an OOD test set with every image checked to be free of ID objects [43].
- **Pre-training overlap.** A detector built on a model pre-trained on web-scale data may have seen

the "unknown" classes during pre-training. Then the test is no longer about the unknown, and the setting must be described accordingly.
- **Thresholds and model selection.** Like any hyperparameter, the threshold and the choice of score must be set on validation data. Tuning them on the OOD test set (or on a proxy that is too similar to it) gives the same optimism as the oracle model selection that DomainBed warns against [19].
- **Covariate shift inside the ID set.** In deployment, an ID image may arrive blurred or under new lighting. OpenOOD v1.5 evaluates "full-spectrum" detection, in which covariate-shifted ID images must be *accepted* while semantic OOD images are rejected, and finds it a significant challenge for all current approaches [39]. This is where the first and third problems of this chapter meet.

## A robustness evaluation plan

**Plain version.** Before claiming a model is robust, write down what it is robust *to*, test that with the strongest tools available, and report the result next to the clean number.

1. **Name the problem.** Shift (which kind?), adversary (which threat model?), or unknown inputs (which kind of novelty?). If several, evaluate each separately.
2. **Build the test data first.** At least one real shifted set from the expected deployment (new site, device or time), synthetic corruptions as a cheap supplement, and a held-out OOD set that is checked for ID contamination.
3. **Use strong evaluation.** For adversarial claims: white-box attacks, PGD with restarts and AutoAttack or an adaptive attack, the warning-sign checks above, and the exact $\varepsilon$ and norm. For shift: effective robustness or matched clean accuracy. For OOD: AUROC and FPR@95TPR on near- and far-OOD sets.
4. **Keep the test data out of the loop.** Choose hyperparameters, thresholds and checkpoints on validation data from the training distribution, or state clearly that an oracle was used.
5. **Report the trade-offs.** Clean accuracy, robust accuracy, calibration, and compute cost together; a gain in one is not free.
6. **Monitor after release.** Track input statistics and confidence distributions, as in ML 11, and in multi-sensor systems the sensor faults listed in DLCV 03B.

## Modern view

**Surveys worth reading.**

- **Yang, Zhou, Li and Liu [3]** (IJCV 2024) unify anomaly detection, novelty detection, open-set recognition, OOD detection and outlier detection as sub-tasks of *generalized OOD detection*, organized by whether covariate or semantic shift must be detected and by whether ID data has one or several classes; they then review each area, with most detail on OOD detection methods (classification-based, density-based and distance-based). Read it first to sort out the vocabulary.

- **Zhou et al. [18]** (TPAMI) define domain generalization, relate it to domain adaptation and transfer learning, and review methods grouped into domain alignment, meta-learning, data augmentation, ensemble learning and others, together with theory and applications beyond vision.
- **Liang, He and Tan [44]** (IJCV) categorize test-time adaptation by the form of test data into test-time domain adaptation, test-time batch adaptation and online test-time adaptation, with a taxonomy of algorithms in each group and a discussion of open challenges.
- **Akhtar, Mian, Kardan and Shah [45]** (IEEE Access 2021) review adversarial attacks and defenses in computer vision since 2018: attacks on classifiers, attacks beyond classification, physical-world attacks, theories of why adversarial examples exist, and defenses grouped into model modification, input modification and external detectors, with certified defenses treated separately.
- **Pang, Shen, Cao and van den Hengel [46]** (ACM Computing Surveys 2021) organize deep anomaly detection into three frameworks: deep learning for generic feature extraction, learning representations of normality, and end-to-end anomaly score learning, with eleven finer categories, and discuss each one's assumptions, advantages and disadvantages.
- **Liu et al. [47]** (Machine Intelligence Research 2024) review industrial image anomaly detection by network architecture, level of supervision, loss function, metrics and datasets, and propose a setting closer to real manufacturing.

**The takeaways they share.** Distribution shift, adversarial robustness and OOD detection grew as separate communities with separate benchmarks, and the surveys repeatedly find that benchmark choice, model selection and evaluation protocol drive results as much as methods do. Simple baselines (ERM with good training, MSP or kNN scores, adversarial training with PGD) remain hard to beat when compared fairly.

**State of the art, 2023–2026.** Five trends are visible. (1) **Foundation models changed the starting point.** Large, diverse pre-training gives much of the natural-shift robustness that specialized methods could not [12], [13], and robust fine-tuning such as weight interpolation tries to keep it [14]; OOD benchmarks now include CLIP and DINOv2 backbones [39]. (2) **Generated data for adversarial training.** Training on images from a modern diffusion model, without external real data, set new results on RobustBench for CIFAR-10 and CIFAR-100 [48]; robust accuracy on these benchmarks still lies far below clean accuracy. (3) **Better OOD evaluation.** Unified benchmarks with near- and far-OOD splits and full-spectrum settings [49], [39] and cleaned test sets [43] replaced ad hoc comparisons. (4) **Zero- and few-shot anomaly detection.** WinCLIP uses CLIP with prompt ensembles over state words and window-level features to classify and segment anomalies with no or very few normal images [50]. (5) **Harder industrial benchmarks.** Because results on MVTec AD saturated, MVTec AD 2 adds transparent and overlapping objects, unusual illumination, high variance of normal data, very small defects and test scenarios with lighting changes, and reports that state-of-the-art methods remain far from solving it [41].

**Open problems.** Robustness to natural shifts that no benchmark anticipated; adversarial robustness beyond small $\ell_p$ balls (patches, physical changes, semantic edits) and at the scale of foundation models; certificates that are tight at useful radii for large images; OOD detection when the backbone may have seen everything during pre-training; detectors that accept covariate-shifted ID inputs while rejecting semantic novelty; and a common evaluation protocol that tests all three

problems of this chapter on the same model.

## Key takeaways

- Shift, adversarial robustness and OOD detection require different test conditions and metrics.
- State available target data before comparing adaptation with generalization.
- Adversarial claims require a threat model, attack access and perturbation budget.
- Strong or adaptive attacks help expose gradient masking; certificates cover only their stated assumptions.
- Evaluate OOD scores on multiple near/far sets and check contamination; no score is universally best.
- For anomaly detection, distinguish image-level decisions from localization and verify the normal-data assumptions.
- Compare clean performance, robustness, operating thresholds and cost under one explicit protocol.

## Exercises

1. **Test-batch normalization with small or mixed batches.** In the corruption snippet, apply test-batch normalization to batches of 8 test images instead of all 540, and then to one batch that mixes all three corruptions at severity 3. How does accuracy change, compared with the plain model and the noise-augmented one? What does this say about deploying test-time adaptation on a camera that switches between day and night?

   **Hint**

   Split `Xs` into chunks with `np.array_split(Xs, len(Xs) // 8)`, normalize each chunk with its own statistics, and concatenate the predictions. In our run at severity 3, batches of 8 kept part of the gain (noise: 0.86 with the full batch, 0.74 with batches of 8, 0.56 for the plain model; blur: 0.95, 0.87, 0.82). A batch mixing the three corruptions still helped here (0.91 against 0.55 for the plain model), because all three corruptions change the per-pixel statistics in a similar direction (less contrast, more spread); do not expect that in general. Small batches give noisy statistics, and a batch that mixes very different conditions (day and night) gives statistics that fit neither, so real TTA methods need reasonably large batches from one condition, or a way to detect a change in the stream and reset.

2. **A defense that only breaks the gradient.** Add a "defense" to the adversarial snippet that rounds every input pixel to the nearest multiple of 0.25 before the standard model sees it, and recompute the input gradient through the rounding (it is zero almost everywhere). What do FGSM and PGD report? Then attack again with BPDA, treating the rounding as the identity in the backward pass only. Which warning signs from the evaluation section did the first result show?

   **Hint**

With a zero gradient, `np.sign` returns zeros and PGD never moves from its random start. In our run the "defended" model kept 0.96 accuracy under this PGD at $\varepsilon = 0.15$, while random $\pm\varepsilon$ sign noise did better as an attack (0.93): random sampling beating a gradient attack is one of the warning signs. With BPDA, use `input_grad(m, quantize(Xa), y)` as the gradient at `Xa`. At $\varepsilon = 0.1$ the attack is weaker than on the undefended model (0.79 against 0.53), because rounding to steps of 0.25 undoes perturbations smaller than half a step; at $\varepsilon = 0.15$ BPDA drives accuracy to 0.01, below the undefended model's 0.18. This is the shattered-gradient case of Athalye et al.

3. **Accuracy–robustness trade-off.** Train the robust model with training budgets $\varepsilon \in \{0.05, 0.1, 0.15, 0.2\}$ and plot clean accuracy against PGD accuracy at $\varepsilon = 0.1$. Where is the knee of the curve? What changes if you increase the hidden layer from 128 to 512 units?

   **Hint**

   In our runs with 128 hidden units, clean accuracy fell from 0.972 to 0.941 as the training budget grew from 0.05 to 0.2, while PGD accuracy at $\varepsilon = 0.1$ rose from 0.706 to 0.754 (at 0.1 and 0.15) and then dropped to 0.733: training far beyond the test budget costs clean accuracy without further gains. With 512 hidden units, both numbers improved at every budget (for example 0.980 clean and 0.806 robust at a training budget of 0.1), in line with Madry et al.'s finding that robustness needs capacity. Use the same number of epochs and the same evaluation attack for every setting; exact values change a little with the random seed.

4. **Faint defects and more normal data.** In the brick-wall snippet, change the scratch strength from 0.5 to 0.35, and then also increase the number of normal training crops from 16 to 32. Report image- and pixel-level AUROC in each case, and explain the difference between the two metrics.

   **Hint**

   In our runs, the image-level AUROC dropped clearly at strength 0.35 with 16 crops (to about 0.70) and recovered partly with 32 crops (to about 0.88), while the pixel-level AUROC stayed above 0.99 in both cases. Image-level scores take the maximum over 4096 patches, so they are hurt by the rarest normal patches; more normal data puts more of those patches into the memory bank. Pixel-level AUROC pools millions of pixels, almost all normal, and ranks most scratch pixels above them even when a few normal patches are high. Expect some variation with the random seed, because there are only 40 test images.

5. **Choose the right tool.** For each scenario, say which of the three robustness problems it is, which evaluation you would run first, and which method you would try first: (a) a skin-lesion classifier moved to a clinic with a different dermatoscope; (b) a content filter that users deliberately try to bypass with small image edits; (c) a wildlife camera-trap classifier that starts seeing a species not in its label set; (d) a weld-inspection camera with no images of defective welds.

   **Hint**

(a) Covariate (domain) shift: collect a labelled test set from the new clinic; try normalization-statistics adaptation or fine-tuning on a few labelled target images, and compare with a more diverse pre-trained backbone at matched clean accuracy. (b) Adversarial, with an explicit threat model (which edits are allowed?); evaluate with white-box and transfer attacks, and try adversarial training; for patch-like edits, $\ell_p$ robustness is not enough. (c) Semantic shift: OOD or open-set detection; evaluate AUROC and FPR@95TPR with held-out species, using feature-distance scores on a good backbone. (d) Sensory anomaly detection: a patch memory bank on normal welds; evaluate with a small set of real defects collected later, at image and pixel level.

## References


1. D. Hendrycks, K. Zhao, S. Basart, J. Steinhardt and D. Song, "Natural Adversarial Examples," *CVPR*, 2021. CVF
2. P. W. Koh, S. Sagawa, H. Marklund, S. M. Xie, M. Zhang, A. Balsubramani et al., "WILDS: A Benchmark of in-the-Wild Distribution Shifts," *ICML*, PMLR 139, 2021. PMLR
3. J. Yang, K. Zhou, Y. Li and Z. Liu, "Generalized Out-of-Distribution Detection: A Survey," *International Journal of Computer Vision*, vol. 132, no. 12, pp. 5635–5662, 2024. doi ·arXiv
4. D. Hendrycks and T. Dietterich, "Benchmarking Neural Network Robustness to Common Corruptions and Perturbations," *ICLR*, 2019. arXiv
5. D. Hendrycks, S. Basart, N. Mu, S. Kadavath, F. Wang, E. Dorundo et al., "The Many Faces of Robustness: A Critical Analysis of Out-of-Distribution Generalization," *ICCV*, 2021. CVF
6. A. Barbu, D. Mayo, J. Alverio, W. Luo, C. Wang, D. Gutfreund, J. Tenenbaum and B. Katz, "ObjectNet: A large-scale bias-controlled dataset for pushing the limits of object recognition models," *NeurIPS*, 2019. proceedings
7. B. Recht, R. Roelofs, L. Schmidt and V. Shankar, "Do ImageNet Classifiers Generalize to ImageNet?," *ICML*, PMLR 97, 2019. PMLR
8. R. Taori, A. Dave, V. Shankar, N. Carlini, B. Recht and L. Schmidt, "Measuring Robustness to Natural Distribution Shifts in Image Classification," *NeurIPS*, 2020. proceedings
9. J. Miller, R. Taori, A. Raghunathan, S. Sagawa, P. W. Koh, V. Shankar, P. Liang, Y. Carmon and L. Schmidt, "Accuracy on the Line: on the Strong Correlation Between Out-of-Distribution and In-Distribution Generalization," *ICML*, PMLR 139, 2021. PMLR
10. D. Hendrycks, N. Mu, E. D. Cubuk, B. Zoph, J. Gilmer and B. Lakshminarayanan, "AugMix: A Simple Data Processing Method to Improve Robustness and Uncertainty," *ICLR*, 2020. arXiv
11. R. Geirhos, P. Rubisch, C. Michaelis, M. Bethge, F. A. Wichmann and W. Brendel, "ImageNet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness," *ICLR*, 2019. arXiv
12. A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal et al., "Learning Transferable Visual Models From Natural Language Supervision," *ICML*, PMLR 139, 2021. PMLR
13. A. Fang, G. Ilharco, M. Wortsman, Y. Wan, V. Shankar, A. Dave and L. Schmidt, "Data Determines Distributional Robustness in Contrastive Language Image Pre-training (CLIP)," *ICML*, PMLR 162, 2022. PMLR
14. M. Wortsman, G. Ilharco, J. W. Kim, M. Li, S. Kornblith, R. Roelofs et al., "Robust Fine-Tuning of Zero-Shot Models," *CVPR*, 2022. CVF

15. S. Schneider, E. Rusak, L. Eck, O. Bringmann, W. Brendel and M. Bethge, “Improving robustness against common corruptions by covariate shift adaptation,” *NeurIPS*, 2020. proceedings
16. D. Wang, E. Shelhamer, S. Liu, B. Olshausen and T. Darrell, “Tent: Fully Test-Time Adaptation by Entropy Minimization,” *ICLR*, 2021. arXiv
17. Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand and V. Lempitsky, “Domain-Adversarial Training of Neural Networks,” *Journal of Machine Learning Research*, vol. 17, no. 59, pp. 1–35, 2016. JMLR
18. K. Zhou, Z. Liu, Y. Qiao, T. Xiang and C. C. Loy, “Domain Generalization: A Survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022. doi ·arXiv
19. I. Gulrajani and D. Lopez-Paz, “In Search of Lost Domain Generalization,” *ICLR*, 2021. arXiv
20. Y. Ovadia, E. Fertig, J. Ren, Z. Nado, D. Sculley, S. Nowozin, J. V. Dillon, B. Lakshminarayanan and J. Snoek, “Can you trust your model’s uncertainty? Evaluating predictive uncertainty under dataset shift,” *NeurIPS*, 2019. proceedings
21. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow and R. Fergus, “Intriguing properties of neural networks,” *ICLR*, 2014. arXiv
22. I. J. Goodfellow, J. Shlens and C. Szegedy, “Explaining and Harnessing Adversarial Examples,” *ICLR*, 2015. arXiv
23. A. Madry, A. Makelov, L. Schmidt, D. Tsipras and A. Vladu, “Towards Deep Learning Models Resistant to Adversarial Attacks,” *ICLR*, 2018. arXiv
24. F. Croce, M. Andriushchenko, V. Sehwag, E. Debenedetti, N. Flammarion, M. Chiang, P. Mittal and M. Hein, “RobustBench: a standardized adversarial robustness benchmark,” *NeurIPS Datasets and Benchmarks Track*, 2021. proceedings ·leaderboard
25. N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik and A. Swami, “Practical Black-Box Attacks against Machine Learning,” *ACM Asia Conference on Computer and Communications Security* (*AsiaCCS*), pp. 506–519, 2017. doi
26. N. Carlini and D. Wagner, “Towards Evaluating the Robustness of Neural Networks,” *IEEE Symposium on Security and Privacy* (*SP*), pp. 39–57, 2017. doi
27. D. Tsipras, S. Santurkar, L. Engstrom, A. Turner and A. Madry, “Robustness May Be at Odds with Accuracy,” *ICLR*, 2019. arXiv
28. H. Zhang, Y. Yu, J. Jiao, E. P. Xing, L. El Ghaoui and M. I. Jordan, “Theoretically Principled Trade-off between Robustness and Accuracy,” *ICML*, PMLR 97, 2019. PMLR
29. A. Athalye, N. Carlini and D. Wagner, “Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples,” *ICML*, PMLR 80, 2018. PMLR
30. F. Croce and M. Hein, “Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks,” *ICML*, PMLR 119, 2020. PMLR
31. J. Cohen, E. Rosenfeld and J. Z. Kolter, “Certified Adversarial Robustness via Randomized Smoothing,” *ICML*, PMLR 97, 2019. PMLR
32. K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno and D. Song, “Robust Physical-World Attacks on Deep Learning Visual Classification,” *CVPR*, 2018. CVF
33. T. B. Brown, D. Mané, A. Roy, M. Abadi and J. Gilmer, “Adversarial Patch,” arXiv:1712.09665, 2017. arXiv
34. D. Hendrycks and K. Gimpel, “A Baseline for Detecting Misclassified and Out-of-Distribution Examples in Neural Networks,” *ICLR*, 2017. arXiv

35. W. Liu, X. Wang, J. D. Owens and Y. Li, “Energy-based Out-of-distribution Detection,” *NeurIPS*, 2020. proceedings
36. K. Lee, K. Lee, H. Lee and J. Shin, “A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks,” *NeurIPS*, 2018. proceedings
37. Y. Sun, Y. Ming, X. Zhu and Y. Li, “Out-of-Distribution Detection with Deep Nearest Neighbors,” *ICML*, PMLR 162, 2022. PMLR
38. D. Hendrycks, M. Mazeika and T. Dietterich, “Deep Anomaly Detection with Outlier Exposure,” *ICLR*, 2019. arXiv
39. J. Zhang, J. Yang, P. Wang, H. Wang, Y. Lin, H. Zhang et al., “OpenOOD v1.5: Enhanced Benchmark for Out-of-Distribution Detection,” arXiv:2306.09301, 2023. arXiv
40. P. Bergmann, M. Fauser, D. Sattlegger and C. Steger, “MVTec AD — A Comprehensive Real-World Dataset for Unsupervised Anomaly Detection,” *CVPR*, 2019. CVF
41. L. Heckler-Kram, J.-H. Neudeck, U. Scheler, R. König and C. Steger, “The MVTec AD 2 Dataset: Advanced Scenarios for Unsupervised Anomaly Detection,” *International Journal of Computer Vision*, vol. 134, no. 4, 2026. doi ·arXiv
42. K. Roth, L. Pemula, J. Zepeda, B. Schölkopf, T. Brox and P. Gehler, “Towards Total Recall in Industrial Anomaly Detection,” *CVPR*, 2022. CVF
43. J. Bitterwolf, M. Müller and M. Hein, “In or Out? Fixing ImageNet Out-of-Distribution Detection Evaluation,” *ICML*, PMLR 202, 2023. PMLR
44. J. Liang, R. He and T. Tan, “A Comprehensive Survey on Test-Time Adaptation Under Distribution Shifts,” *International Journal of Computer Vision*, vol. 133, no. 1, pp. 31–64, 2025. doi · arXiv
45. N. Akhtar, A. Mian, N. Kardan and M. Shah, “Advances in Adversarial Attacks and Defenses in Computer Vision: A Survey,” *IEEE Access*, vol. 9, pp. 155161–155196, 2021. doi ·arXiv
46. G. Pang, C. Shen, L. Cao and A. van den Hengel, “Deep Learning for Anomaly Detection: A Review,” *ACM Computing Surveys*, vol. 54, no. 2, pp. 1–38, 2021. doi ·arXiv
47. J. Liu, G. Xie, J. Wang, S. Li, C. Wang, F. Zheng and Y. Jin, “Deep Industrial Image Anomaly Detection: A Survey,” *Machine Intelligence Research*, vol. 21, no. 1, pp. 104–135, 2024. doi ·arXiv
48. Z. Wang, T. Pang, C. Du, M. Lin, W. Liu and S. Yan, “Better Diffusion Models Further Improve Adversarial Training,” *ICML*, PMLR 202, 2023. PMLR
49. J. Yang, P. Wang, D. Zou, Z. Zhou, K. Ding, W. Peng et al., “OpenOOD: Benchmarking Generalized Out-of-Distribution Detection,” *NeurIPS Datasets and Benchmarks Track*, 2022. proceedings
50. J. Jeong, Y. Zou, T. Kim, D. Zhang, A. Ravichandran and O. Dabeer, “WinCLIP: Zero-/Few-Shot Anomaly Classification and Segmentation,” *CVPR*, 2023. CVF