ARTIFICIAL NEURAL NETWORKS

— A VISUAL, HANDS-ON COURSE —

From the single neuron to transformers and diffusion — the companion book to the **ANN Studio** app.

COMES WITH A **FREE** COMPANION APP

Play
Interactive simulations

Labs
Build, train & visualize networks

Learn
Clear explanations with visuals

Practice
Quizzes, exercises & challenges

FOR STUDENTS, AND EVERY CURIOUS LEARNER

A simple, rigorous, and practical path to **AI**, **ANNs**, and **Deep Learning**.

DR. ISMAEL ABDULRAHMAN

Artificial Neural Networks:

From the single neuron to transformers and diffusion

Dr. Ismael Khorshed Abdulrahman
PhD in Electrical and Computer Engineering

Free Electronic Edition
Licensed under Creative Commons Attribution (CC BY).

This book is freely available for educational, personal, and scholarly use. You are welcome to share, reproduce, and adapt its content, including for commercial purposes, provided that appropriate credit is given to the author and the original source is properly cited.

This book and its companion app, *ANN Studio*, are provided free of charge for anyone who wishes to learn. Voluntary donations in support of their continued development are warmly welcomed and appreciated.

A note on AI assistance: AI tools were used during the preparation of this book under the author's direction and review. AI assistance may have contributed to the drafting, refinement, explanation, organization, or development of certain examples and other content. Every effort has been made to carefully check and verify the facts, figures, explanations, mathematical derivations, examples, solutions, and other content before publication. Nevertheless, despite these efforts, errors or inaccuracies may remain. Readers are kindly encouraged to inform the author of any mistakes, inaccuracies, or corrections they identify. Such feedback is greatly appreciated and may be considered for correction in future editions (updates).

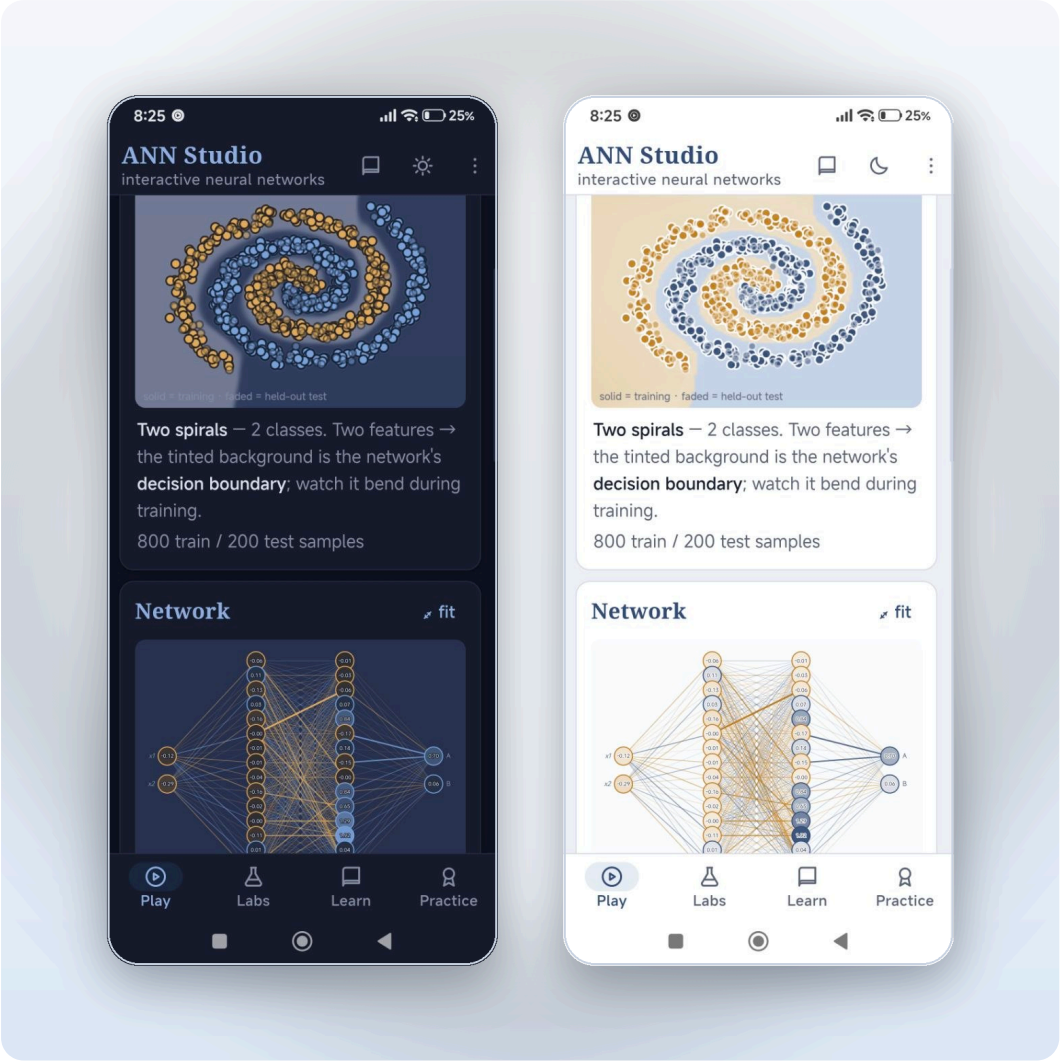
A Note on the Numbers: The datasets, prices, measurements and organizations in this book's worked examples, activities and case studies are illustrative teaching examples, chosen to make a method clear and checkable by hand. They are not verified statistics about any real person, company or institution. Where a real historical fact is stated — who first published a method, and in what year — it is named in the text and listed in the References at the end of the book. Always confirm current figures from a primary source before using them in a real decision.

Last updated: 31 August 2026 · AI model families and dated examples checked as of this date.

COMES WITH A FREE COMPANION APP

Meet ANN Studio

This book has a hands-on twin: everything you read here comes alive in **ANN Studio**, the free companion app.



Train a network on the two-spirals dataset above and watch its **decision boundary** bend as it learns. All 43 chapters live in the app, with interactive **labs**, runnable **code**, worked examples and **quizzes** — and it works fully offline, in light or dark theme. The book gives you the complete written course; the app lets you play with every idea in it.

Play · Labs · Learn · Practice

Preface

This book teaches artificial neural networks the way they are best learned — by reading an idea, seeing it drawn, and then running it.

Neural networks are, at heart, a small number of simple ideas composed at scale. A neuron weighs its inputs; layers of neurons compose simple decisions into rich ones; a loss function says how wrong the network is; gradient descent and backpropagation adjust millions of numbers to make it less wrong. Everything else in modern deep learning — convolutional networks that see, recurrent and attention-based networks that read, autoencoders and diffusion models that imagine — is a variation on those foundations. This book builds them one at a time, in plain language, with every equation's symbols spelled out, every worked example carried to an actual number, and honest notes on where each idea breaks down.

What makes it unusual is that it is the companion to a working application. The same pages you read live inside **ANN Studio**, an offline app for phone, desktop and the web. There you can train the very networks described here and watch their decision boundaries bend, step through backpropagation one number at a time, and run the book's code — real NumPy, executed on the spot — then change a line and run it again. Theory, simulation and code are not three separate resources; they are the same material seen three ways.

How to use this book with the app

Each chapter pairs a concept with a place to *do* it. When you see a shaded code block marked “runnable in the app,” open the matching chapter in ANN Studio and press **Run** — the first run loads Python (once), then results appear instantly. PyTorch listings are meant to be copied into Google Colab or a local environment. The app's **Play** tab is the live laboratory for the foundational and training chapters; the **Labs** animate individual ideas; **Practice** holds worked examples and quizzes. Exercises appear at the end of every chapter — try them first, before checking your work. Full worked solutions are not printed in this student edition; they are collected in a separate **Teacher's & Help Book** (a

solutions manual), available to teachers and instructors on request — please contact the author, Dr. Ismael Khorshed Abdulrahman (ismael.abdulrahman@epu.edu.iq). The same worked solutions are also built into the ANN Studio app.

The mathematics assumes only first-year calculus and linear algebra, and appendices A–D refresh exactly what is needed. Appendices E and F are self-contained primers on Python/NumPy and PyTorch, and Appendix G builds a working autograd engine from scratch, so a motivated reader can start from very little. The five case studies near the end assemble the pieces into complete, real-world pipelines.

My hope is that by the last page you will not just know what a neural network is, but have trained many, broken a few, and understood why they behaved as they did.

Who this is for — and a note on access

These materials are for students, researchers, and anyone who wants to learn Artificial Intelligence (AI), Artificial Neural Networks (ANNs), and Deep Learning in a simple, accessible, and academically rigorous way. The aim is a free, well-structured resource that carries you from the single neuron to modern architectures without assuming more than first-year mathematics.

The book and the ANN Studio app are completely free, so that anyone who wants to learn can use them. If you find them useful and would like to support their continued development, any voluntary donation is warmly welcomed and deeply appreciated — but it is never required.

— Dr. Ismael Khorshed Abdulrahman, PhD in Electrical and Computer Engineering, Tennessee Technological University, USA

Notation & conventions

The book uses one consistent notation, fixed in Chapter 1 and reused everywhere. Vectors and matrices are bold; scalars are not. Batches are stored as rows.

x, x, X	a scalar input, an input vector (one example), and a data matrix (one example per row, m×n)
w, w, W	a weight, a weight vector, and a weight matrix $W^{(l)}$ for layer ℓ (shape $n_{\ell-1}\times n_{\ell}$)
b, b	a bias (scalar) and a bias vector for a layer
z, a	pre-activation $z = \mathbf{x}W + b$ and activation $a = g(z)$; superscript $^{(l)}$ denotes the layer
g, σ	an activation function; σ specifically the logistic sigmoid $\sigma(z)=1/(1+e^{-z})$
ŷ, y	the network's prediction and the true target/label
L	the loss (a single number the training minimizes)
η	the learning rate (step size of gradient descent)
∇, ∂	the gradient operator and partial derivative; $\nabla_{\theta}L$ is the gradient of L w.r.t. parameters θ
$\delta^{(l)}$	the error signal $\partial L/\partial z^{(l)}$ backpropagated to layer ℓ
⊙, ᵀ, ·	elementwise (Hadamard) product, transpose, and dot/matrix product
λ, p	the L2 regularization strength, and the dropout probability
𝔼, 𝒩, ‖·‖	expectation, the Gaussian/normal distribution, and a vector norm (L2 unless noted)

Shaded blue boxes are worked examples; amber boxes state assumptions and limitations; green boxes say when and why an idea is used, and give a real-world application. Grey code blocks marked “runnable in the app” execute inside ANN Studio; PyTorch listings are for Colab.

Contents

FOUNDATIONS

1 From biological to artificial neurons 11

2 The perceptron & linear separability 26

3 Multilayer networks & the forward pass 40

4 Activation functions 53

5 Features, embeddings & what a network learns 71

TRAINING NETWORKS

6 Loss functions & maximum likelihood 88

7 Gradient descent & its variants 103

8 Backpropagation 118

9 Momentum, RMSProp & Adam 131

10 Initialization & the vanishing gradient 147

11 Normalization: batch, layer & beyond 163

12 Regularization & generalization 179

ARCHITECTURES

13 Convolutional neural networks 196

14 Modern CNN architectures & computer vision 211

15 Recurrent networks & sequences 229

16 Attention & the transformer 243

17 Language models & the transformer era 260

18 Graph neural networks 278

19 Autoencoders & representation learning 295

20 Self-supervised & contrastive learning 310

21 Generative models: GANs to diffusion 326

BEYOND SUPERVISED LEARNING

22 Transfer learning & fine-tuning 343

23 Deep reinforcement learning 361

24 Uncertainty, calibration & Bayesian nets 380

IN PRACTICE

25 Training in practice: the complete recipe 397

26 Data-centric deep learning 414

27 Efficiency & deployment 431

28 Interpretability & explainability 446

29 Ethics, fairness & safety 463

CASE STUDIES

30 Case study: handwritten-digit recognition (MNIST) 480

31 Case study: tabular classification & churn 493

32 Case study: text sentiment analysis 510

33 Case study: time-series forecasting 527

34 Case study: anomaly detection with autoencoders 540

APPENDICES

35 Appendix A · Linear algebra refresher 555

36 Appendix B · Calculus & the chain rule 569

37 Appendix C · Probability & statistics 582

38 Appendix D · Information theory 597

39 Appendix E · Python & NumPy primer 610

40 Appendix F · PyTorch primer 627

41 Appendix G · Autograd from scratch 641

42 Appendix H · Datasets & further reading 651

43 Appendix I · Formulas & symbols quick reference 663

REFERENCE

§ References 672

§ Glossary — 155 terms 689

PART I

Foundations

The single neuron, the perceptron and its limits, multilayer networks and the forward pass, and the activation functions that give them their power. Everything else in the book rests on these four chapters.

1 From biological to artificial neurons

2 The perceptron & linear separability

3 Multilayer networks & the forward pass

4 Activation functions

5 Features, embeddings & what a network learns

CHAPTER 1 · FOUNDATIONS

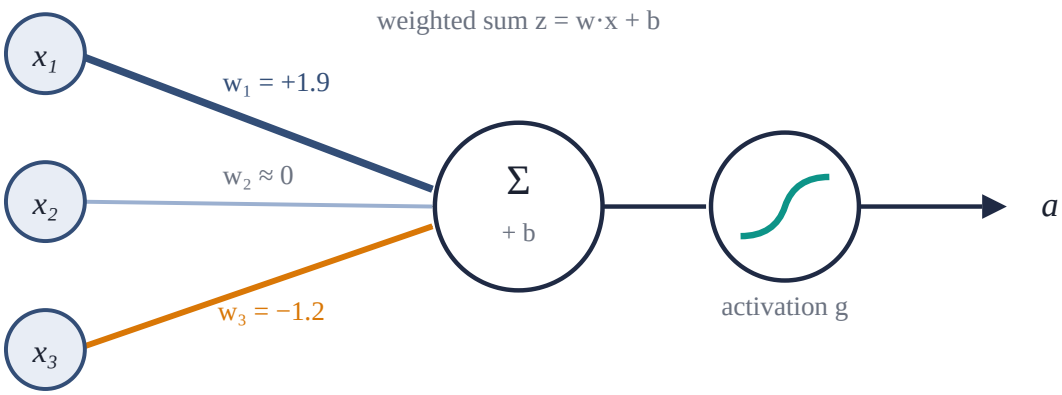
From biological to artificial neurons

> *Every network in this app — and every large language model in a datacenter — is built from one small unit repeated millions of times: a device that takes several numbers, weighs them, adds a small offset, and squashes the result. This chapter builds that unit from its biological inspiration, fixes the notation used everywhere else in the manual, and computes one neuron from start to finish by hand.*

Fundamental concept

A biological neuron is a cell that **listens, votes and shouts**. Through thousands of input fibers (dendrites) it receives chemical signals from other neurons. Each connection — a **synapse** — strengthens or weakens its signal by its own amount. The cell body sums the result. Finally, if the sum crosses a threshold, the neuron fires a spike down its axon to everyone downstream. Learning, in the brain, is largely the slow adjustment of those synaptic strengths.

The artificial neuron of McCulloch & Pitts (1943) and its modern form keep exactly this basic structure and drop the biochemistry. Each input signal becomes a number x_i. Each synapse becomes a **weight** w_i — positive weights excite, negative weights inhibit, near-zero weights mean “ignore this input”. The threshold becomes a **bias** b, a movable baseline that sets how easily the neuron activates. Finally, the firing decision becomes an **activation function** g applied to the sum. That is the entire device. Its power is not in the unit but in the *composition*: chapter 3 shows that layers of these units can approximate essentially any function, and chapter 7 shows how the weights can be learned automatically from examples.



The artificial neuron. Line thickness and color match the app's diagram: blue = excitatory ($w > 0$), amber = inhibitory ($w < 0$), thin gray = ignored ($w \approx 0$).

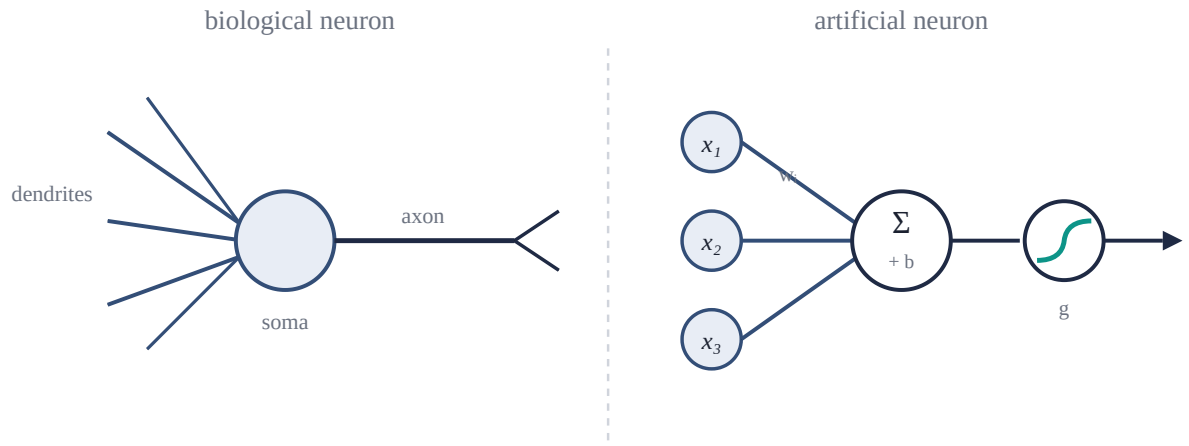
From biology to arithmetic

It is worth setting out the correspondence part by part, because the artificial neuron borrows its whole vocabulary from the cell and the two are used to mean the same thing for the rest of the book. Four parts map across:

1. **Dendrites → inputs.** The branching fibers that collect signals from other cells become the input vector $\mathbf{x} = (x_1, \ldots, x_n)$ — one number per incoming channel.
2. **Synaptic strength → weights.** Each synapse passes on its signal, made stronger or weaker by a strength the brain slowly tunes. That strength becomes the weight w_i. A strong excitatory synapse is a large positive weight, an inhibitory one a negative weight, a cut connection a weight near zero. **Learning is weight change**. This is the one idea the model keeps most faithfully.
3. **Soma summation → weighted sum.** The cell body pools the arriving currents. The model pools them as the linear combination $z = \Sigma_i w_i x_i + b$. The bias b represents the cell's built-in excitability. It sets how much net input the cell needs before it responds at all.
4. **Axon firing → activation.** When the pooled input crosses the threshold, the cell emits a spike down its axon. The model emits $a = g(z)$. A real neuron either fires or does not, but it fires *repeatedly*, and what it communicates is largely the **rate** of firing — a

stronger stimulus means more spikes per second. That firing rate is a nonnegative number that rises with input and then saturates. It is what a continuous activation such as the sigmoid or ReLU really represents: not one spike, but a rate.

This **rate-coding** reading is the honest link from a discrete spike to a smooth number, and it is also where the analogy begins to break down. A cortical neuron is a living dynamical system. It integrates inputs in continuous time with delays, adapts its own threshold, encodes information in the *timing* of spikes and not only their rate, runs on dozens of neurotransmitters and neuromodulators, and sits in a tangled recurrent network with no clear idea of "layers". The artificial neuron discards all of it — time, chemistry, stochastic spikes, dendritic computation — and keeps a single static equation. The claim of this chapter is not that brains are weighted sums. It is that a stripped-down caricature of one, stacked and tuned, turns out to be astonishingly capable. Treat "neural" as inspiration, not description.



The same four parts, twice. Dendrites become the inputs x_i, synaptic strengths become the weights w_i, the soma's pooling becomes the weighted sum $z = \Sigma\, w_i x_i + b$, and the axon's firing rate becomes the activation $a = g(z)$. Everything else about the cell is thrown away.

The McCulloch–Pitts neuron and logic gates

The first artificial neuron came before the computer. In 1943 the neurophysiologist Warren McCulloch and the logician Walter Pitts asked what a network of idealized neurons could compute. They reduced the cell to its simplest logical parts: inputs that are either 0 or 1, weights

of fixed strength, and an all-or-nothing threshold. Their unit fires when the weighted input reaches a threshold θ:

$$a = 1 \text{ if } \Sigma_i w_i x_i \geq \theta, \text{ else } 0$$

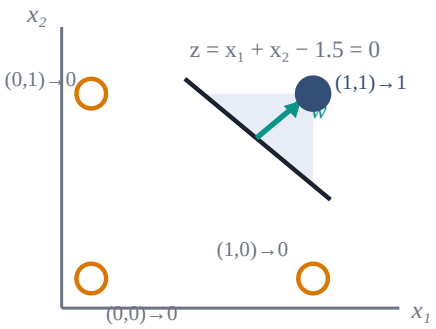
The single step that links their unit to everything that follows is to move the threshold into the sum as a **bias**. Subtract θ from both sides: the neuron fires when $\Sigma\, w_i x_i - \theta \geq 0$, so writing $\mathbf{b = -\theta}$ turns the rule into "fire when $z = \mathbf{w}\cdot\mathbf{x} + b \geq 0$". The threshold has become an ordinary, learnable parameter. It is one more number trained by the same process as the weights. The neuron then reduces to the compact **vectorized form** used everywhere in this book,

$$z = \mathbf{w}\cdot\mathbf{x} + b, \quad a = g(z)$$

with g the hard step for McCulloch and Pitts (smooth replacements arrive two sections below). A high threshold (a very negative bias) makes a cautious neuron that needs very strong input. A low threshold (a positive bias) makes an eager one that fires on almost anything.

What can a single such unit do? With hand-set weights it computes the basic **logic gates**, which is exactly what McCulloch and Pitts set out to show. Take two Boolean inputs. Weights (1, 1) with bias −1.5 give **AND**: $z = x_1 + x_2 - 1.5$ is positive only when both inputs are 1 ($2 - 1.5 = 0.5 > 0$) and negative otherwise. The same weights with bias −0.5 give **OR**: any single 1 already lifts $z = 1 - 0.5 = 0.5$ above zero. A single input with weight −1 and bias +0.5 gives **NOT**. Because AND, OR and NOT together can express any Boolean function, a *network* of these threshold units is in principle a universal logic circuit. This is the 1943 result that first suggested brains might be understood as computers, and computers as brains.

One gate, though, is impossible for a *single* unit: **XOR**, true exactly when the inputs differ. No straight line separates the two "true" corners (0,1) and (1,0) from the two "false" corners (0,0) and (1,1), so no choice of two weights and a bias can compute it. That one observation — a neuron draws only a single straight boundary — is the problem that held back the field for a decade. It is also the reason the next chapter is about stacking neurons. We prove and resolve it in the perceptron chapter (chapter 2).



AND: w = (1, 1), b = −1.5

OR: w = (1, 1), b = −0.5

NOT: w = (−1), b = +0.5

*A single McCulloch–Pitts unit computing **AND**. The line $z = x_1 + x_2 - 1.5 = 0$ separates the corner (1,1) — the only input with both features on — from the other three, and the weight vector **w** = (1, 1) points into the shaded "true" half-plane. Lowering the bias to −0.5 slides the very same line down to compute **OR**. No single line separates XOR's diagonal pairs, which is why one neuron cannot compute it (chapter 2).*

Worked example — a soft AND gate

Give one sigmoid neuron the weights **w** = (4, 4) and bias b = −6, and feed it the four Boolean inputs. The large weights sharpen the sigmoid into a near-hard gate while keeping it differentiable.

1. (0,0): z = 4·0 + 4·0 − 6 = −6, so a = σ(−6) = $1/(1 + e^{6})$ = **0.0025**.
2. (1,0): z = 4 − 6 = −2, a = σ(−2) = **0.119**; and (0,1) gives the same z = −2, a = **0.119** by symmetry.
3. (1,1): z = 8 − 6 = +2, a = σ(2) = **0.881**.
4. Threshold the output at 0.5 and only (1,1) passes it. The neuron has computed AND: it outputs 1 exactly when both inputs are 1.
5. The bias sets the gate's character. Raise it to b = −2 and $z = 4x_1 + 4x_2 - 2$ rises above zero whenever *either* input is 1 (a single 1 gives z = +2 → a = 0.881), turning the identical weights into OR. The threshold b = −θ is the only value that changed.

What the numbers mean: a neuron does not store a truth table, it stores a *geometry* — a line placed so the wanted corner falls on the positive side. Sharpen it with large weights and it behaves like Boolean logic. Soften it with small ones and it reports graded confidence instead. The single function that no straight line can produce is XOR, and that gap is the whole motivation for chapter 2.

Governing equations

One neuron with n inputs computes, in two stages:

$$z = \mathbf{w}\cdot\mathbf{x} + b = \Sigma_{i=1..n}\, w_i x_i + b$$

$$a = g(z)$$

- $\mathbf{x_i}$ input i — a feature of the data, or the output of an upstream neuron (dimensionless number)
- $\mathbf{w_i}$ weight on input i — the learned strength and sign of that connection
- **b** bias — a learned offset that shifts how much total evidence is needed before the neuron responds
- **z** pre-activation — the raw evidence total before squashing (“logit” at an output layer)
- **g** activation function — a fixed nonlinearity such as sigmoid, tanh or ReLU (chapter 4)
- **a** activation — the neuron's output, passed on downstream

Geometrically, the equation z = 0 defines a **hyperplane** — a line in 2-D, a plane in 3-D — with the weight vector **w** perpendicular to it and b setting its distance from the origin. A single neuron therefore answers one linear question: *which side of my hyperplane is the input on, and by how far?* The activation g only reshapes “by how far” into a convenient range.

The geometry of a neuron

It helps to picture the neuron in the space of its inputs, because that picture explains every architecture that follows. The boundary z = 0 is a flat sheet — a **hyperplane** — and the weight vector **w** is its **normal**: it points straight across the sheet, from the “off” side toward the “on”

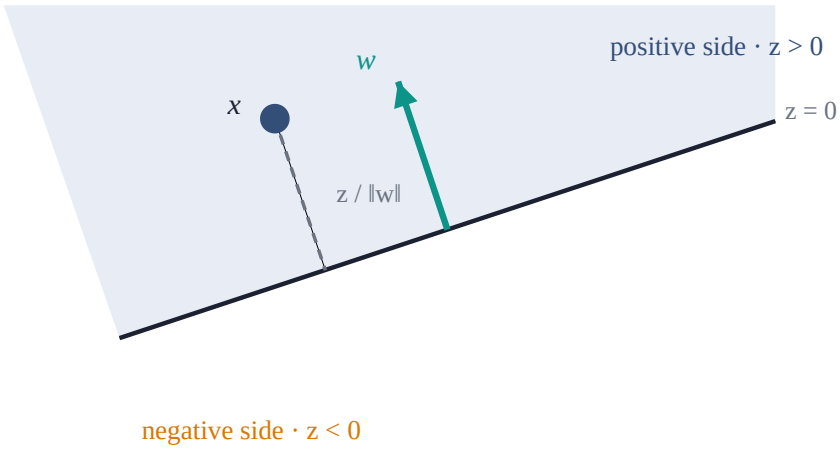
side. Sliding b moves the sheet along that normal without tilting it. Changing **w** rotates it. The two controls therefore have clear, separate jobs: *the weights choose the orientation of the boundary, the bias chooses its position.*

The pre-activation z carries more than a yes/no. Divide it by the length of the weight vector and it becomes the **signed distance** from the input to the boundary:

$$\text{distance}(\mathbf{x}) = \frac{\mathbf{w}\cdot\mathbf{x} + b}{\|\mathbf{w}\|} = \frac{z}{\|\mathbf{w}\|}$$

positive on the side **w** points to, negative on the other, zero exactly on the boundary. A point deep in “spam” territory has a large positive z. One right on the boundary has $z \approx 0$. This distance is the raw material of the **margin** that the perceptron (chapter 2) and support-vector machines try to widen. It also explains a subtle invariance: doubling every weight *and* the bias leaves the boundary fixed but doubles every z. So a sigmoid neuron's *confidence* can be raised without changing a single *decision* — a fact that returns as a calibration hazard in chapter 12.

The *magnitude* $\|\mathbf{w}\|$ carries its own meaning: it sets how steeply the neuron swings from "off" to "on" as the input crosses the boundary. Since $a = \sigma(\mathbf{w}\cdot\mathbf{x} + b) = \sigma(\|\mathbf{w}\| \cdot \text{distance})$, the weight length acts as a **gain** on the signed distance — a slope control on the sigmoid. A small $\|\mathbf{w}\|$ gives a gentle ramp with a wide band of uncertain, near-½ outputs around the boundary. A large $\|\mathbf{w}\|$ gives a steep, almost-step transition that commits to 0 or 1 very close to the line. So the two aspects of **w** separate cleanly again: its *direction* fixes where the boundary sits and which way is "on", while its *length* fixes how confidently the neuron declares it. The soft-AND gate above is exactly this — the plain unit-AND geometry with the weights scaled up until the ramp becomes sharp.



*A neuron's boundary is the hyperplane z = 0. The weight vector **w** is its normal, pointing to the positive side. Any point's pre-activation equals ‖**w**‖ times its signed distance to the boundary — z encodes both which side and how far.*

The same geometry in code — score three emails and read which side of the boundary each lands on, and by how much:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
# One neuron's boundary is the hyperplane  w.x + b = 0.
# The signed distance of a point to it is (w.x + b)/||w||.
w = np.array([0.8, 1.1, -3.0]); b = -2.5
pts = np.array([[4, 2, 1],      # borderline spam
                [4, 2, 0],      # unknown sender
                [0, 0, 1]])     # clean note from a contact
z = pts @ w + b                 # pre-activations (signed evidence)
dist = z / np.linalg.norm(w)    # signed distance to the boundary
for p, zi, di in zip(pts, z, dist):
    side = "spam" if zi >= 0 else "ham "
    print(f"x={p}  z={zi:+.2f}  dist={di:+.2f}  -> {side}")
```

From threshold to smoothness

The original McCulloch–Pitts unit used a hard step: a = 1 if z ≥ 0, else 0. A step is faithful to “fire or don't”, but it has a serious flaw for learning: its derivative is zero everywhere (and undefined at the jump), so it gives no signal about how a *small* change in a weight would

change the output. And every learning algorithm in this manual works by exactly such small-change reasoning. Replacing the step with a smooth ramp such as the sigmoid,

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \sigma'(z) = \sigma(z)(1 - \sigma(z))$$

keeps the same S-shaped decision (near 0 for very negative z, near 1 for very positive) but makes the neuron **differentiable** — every weight now has a well-defined direction to change in. This one modeling choice is what makes gradient descent (chapter 7) and backpropagation (chapter 8) possible.

From threshold to probability: logistic regression

The sigmoid does more than smooth the step. It turns the neuron into a calibrated **probability** device. This is worth stating precisely because it reappears at the output of nearly every classifier in the book. Read a as the model's estimated probability that the label is 1. Inverting σ reveals what the neuron is really computing about that probability:

$$a = \sigma(z) \;\Leftrightarrow\; z = \ln \frac{a}{1 - a}$$

The ratio a/(1 − a) is the **odds** of the positive class, and z = **w**·**x** + b is exactly its logarithm — the **log-odds**, or **logit**. So a neuron is linear not in the probability but in the log-odds: each weight w_i states how much evidence (in log-odds) one unit of feature i contributes. Equivalently, raising x_i by one *multiplies* the odds by e^{w_i}. In the spam example $w_2 = 1.1$, so every extra link multiplies the odds of spam by $e^{1.1} \approx 3.0$ — a clean, checkable statement of what the model learned. A single sigmoid neuron trained to maximize this probability (chapter 6) *is* **logistic regression**, the standard tool of applied statistics. The whole of deep learning is the art of feeding that final classifier *learned* features instead of raw ones.

The neuron in code

Here is the whole neuron in NumPy — the same arithmetic the engine runs, on the very spam example above. Press **Run** (Python loads once, then it is instant), then change a weight and run again:

Python · NumPy ▶ runnable in the app

```python
import numpy as np

def neuron(x, w, b, activation=lambda z: 1/(1+np.exp(-z))):
    z = np.dot(w, x) + b          # weighted sum + bias
    return z, activation(z)       # pre-activation, activation

# the spam detector from the worked example
x = np.array([4, 2, 1])           # CAPS words, links, known-sender?
w = np.array([0.8, 1.1, -3.0])    # learned weights
b = -2.5                          # learned bias

z, a = neuron(x, w, b)
print(f"z = {z:.3f}   (evidence)")
print(f"a = {a:.3f}   -> {a*100:.1f}% spam")

# counterfactual: an unknown sender flips the verdict
_, a2 = neuron(np.array([4, 2, 0]), w, b)
print(f"unknown sender -> {a2*100:.1f}% spam")
```

The same computation, written for the modern deep-learning stack, appears throughout the book as PyTorch (copy it into Colab to run):

PyTorch copy → Colab

```python
import torch, torch.nn as nn

neuron = nn.Linear(3, 1)                  # 3 inputs -> 1 output (a weight row + bias)
x = torch.tensor([4., 2., 1.])
a = torch.sigmoid(neuron(x))              # z = Wx + b, then sigmoid
print(a)
```

In this app's engine

The Playground never computes neurons one at a time. A whole layer of them shares the input, so their weight vectors stack into a matrix W (one column per neuron) and the layer computes **z** = **x**W + **b** in a single matrix multiplication. Chapter 3 develops this notation. When you tap a neuron in the network diagram, the inspector shows precisely this chapter's quantities: its incoming weights, its bias, its z and its a for the sample being examined.

A short history, from 1943 to today

The single neuron carries eighty years of history, and the story is worth knowing because the field's rhythm — hope, disappointment, revival — keeps repeating. **1943:** McCulloch and Pitts publish the threshold unit above and prove that networks of them can compute any logical proposition. This was the founding suggestion that thought might be computation. Their weights, though, are set by hand. Nothing learns. **1949:** Donald Hebb provides a learning principle — "neurons that fire together wire together" — the first account of how synaptic strengths might change with experience. **1958:** Frank Rosenblatt's **perceptron** (chapter 2) makes it mechanical: a neuron with an automatic rule for adjusting its own weights from labeled examples, built in hardware that learned to classify images. The press announced an electronic brain, and the first wave of neural enthusiasm reached its peak.

1969: Marvin Minsky and Seymour Papert's book *Perceptrons* proved with strict rigor what our XOR picture shows — a single-layer perceptron cannot represent functions that are not linearly separable — and observed that nobody knew how to train the multilayer networks that could. Funding drained away. The first "AI winter" began. The missing piece arrived in **1986**, when Rumelhart, Hinton and Williams popularized **backpropagation** (chapter 8), a way to compute gradients through many layers and so train the deep stacks that Minsky and Papert had called too hard to train. Theory soon caught up: a multilayer network is a **universal approximator** (chapter 3). But data and compute had not, and neural networks spent another decade less popular than support-vector machines.

The modern era begins in **2012**, when a deep convolutional network (chapter 13) cut the error rate on the ImageNet contest so sharply that the field changed direction almost overnight. GPUs supplied the compute and the internet the data that 1986 had lacked. Everything since — residual networks, the attention-based Transformer (chapter 16), and the large language models that likely brought you to this book — is assembled from the same unit McCulloch and Pitts sketched in 1943: several numbers, weighed, summed, shifted and squashed. What changed was never the neuron but how many we could afford, how we connected them, and how well we could train them.

Worked example — a spam detector in one neuron

Features of an incoming email: x_1 = number of ALL-CAPS words (say 4), x_2 = number of links (2), x_3 = 1 if the sender is in your contacts (1). Learned parameters: w = [0.8, 1.1, −3.0], b = −2.5, sigmoid activation.

1. Evidence: $z = 0.8\cdot4 + 1.1\cdot2 + (-3.0)\cdot1 + (-2.5) = 3.2 + 2.2 - 3.0 - 2.5 = \mathbf{-0.10}$.
2. Decision: $a = \sigma(-0.10) = 1/(1+e^{0.10}) = \mathbf{0.475}$ → 47.5% spam probability — just under the threshold: the known sender (−3.0) almost exactly cancels the shouting and the links.
3. Counterfactual: from an unknown sender ($x_3 = 0$), z = +2.9 and $a = \sigma(2.9) = \mathbf{0.948}$ — the same email is now 95% spam. One weight changed the decision. That is what “the weights encode knowledge” means.
4. Sensitivity: $\sigma'(-0.10) = 0.475\cdot0.525 = \mathbf{0.249}$ — near its maximum of 0.25. This borderline email is exactly the kind of training example that would move the weights most (chapter 7).
5. Odds view: with a = 0.475 the odds of spam are $a/(1-a) = 0.475/0.525 = \mathbf{0.905} = e^{-0.10} = e^{z}$, confirming z is the log-odds. One more link (x_2: 2 → 3) adds $w_2 = 1.1$ to z, multiplying the odds by $e^{1.1} \approx \mathbf{3.0}$. Then z rises to +1.0 and $a = \sigma(1.0) = \mathbf{0.731}$.

Assumptions & limitations

The artificial neuron is a caricature: real neurons communicate in spike *timing*, adapt their thresholds, run on neurotransmitter chemistry, and are not organized in tidy layers — none of which the model keeps. Treat “neural” as inspiration, not description. Mathematically, a single neuron can only draw one straight boundary: it cannot represent XOR or any pattern needing a curve (chapter 2 proves this), which is why everything beyond this chapter is about *networks* of them. And the numbers fed in must be reasonably scaled. A feature ranging in the thousands next to one in hundredths will silently dominate z, which is why inputs are normalized in practice (chapter 11).

Why & when this matters

Every diagram in this app is read through this chapter's vocabulary: edge color = weight sign, edge thickness = |w|, node shade = activation a. When a network goes wrong, diagnosis starts here — are the z values saturating the activation? Is one input's scale overwhelming the others? Is a bias stuck? A single neuron with sigmoid is also, exactly, **logistic regression** — still the first model to try on simple tabular problems before moving to deep networks.

In the real world

The single sigmoid neuron is not outdated: as logistic regression it is among the most widely used models in the world. Bank credit scorecards, epidemiological risk calculators such as the Framingham heart-disease score, and classic spam filters are each essentially one neuron whose weights humans can read and check — a transparency that deeper models lose. At web scale, the click-through-rate systems that rank advertisements at Google and Meta put a logistic output on top of billions of features.

Google's *Wide & Deep* model (Cheng et al., 2016) literally adds a wide logistic-regression term beside a deep network. Understand this one unit and you understand the last layer of almost every classifier in production.

Exercises

1. For $w = [2, -1, 0.5]$, $x = [1, 2, 4]$ and $b = -1$, compute z and then $a = \sigma(z)$. MATH

2. A neuron has $w = [3, 4]$, $b = -10$. How far is the point $x = [2, 2]$ from its decision boundary, and on which side? MATH

3. Explain why a hard step activation makes gradient-based learning impossible, while the sigmoid does not. CONCEPT

4. Evaluate $\sigma'(z)$ on a grid and confirm numerically that it peaks at $z = 0$. CODE

5. A neuron outputs $a = 0.88$ on some input. What was its pre-activation z (the log-odds)? What are the odds of the positive class? MATH

6. Verify in code that raising an input by one unit multiplies the odds of the positive class by e^{w_i}. CODE

7. Your colleague says "a neuron is just logistic regression, so why bother with networks?" Give the sharpest one-sentence reply, and one situation where they are right. CONCEPT

8. Using a hard-threshold (step) neuron with three binary inputs, design weights and a bias that fire exactly when a **majority** — at least two — of the inputs are 1. Relate your bias to the threshold θ, and say what the same weights compute if the bias is changed to -1 or to -3. MATH

9. In NumPy, run the soft gate $\mathbf{w} = (4, 4)$ over all four Boolean inputs for bias -6 and again for bias -2, and confirm that thresholding the outputs at ½ gives AND and OR respectively.

CODE

10. In what precise sense does a continuous activation such as the sigmoid correspond to what a biological neuron does, and name two features of real neurons the artificial unit throws away. Is discarding them a flaw? CONCEPT

CHAPTER 2 · FOUNDATIONS

The perceptron & linear separability

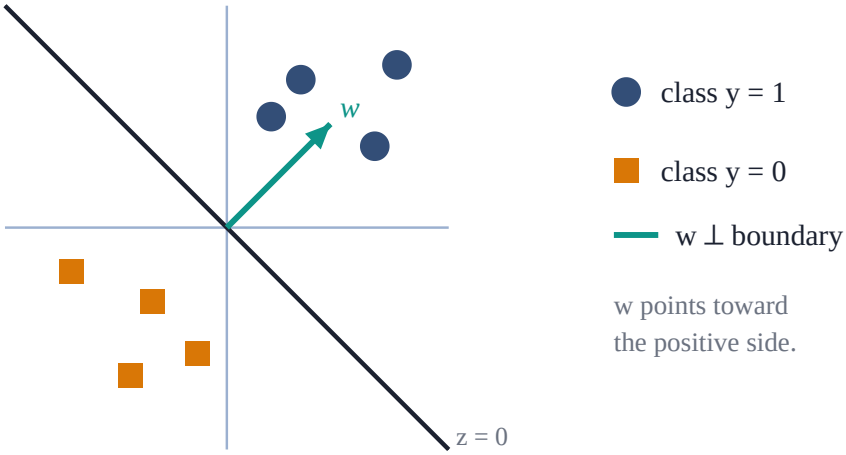
> *The perceptron is the first machine that could learn from its own mistakes. Frank Rosenblatt built it in 1958 as a single neuron with a hard on/off output and a very simple rule for correcting errors, one example at a time. This chapter shows how that rule moves a decision line geometrically. It shows why the rule is guaranteed to succeed when the data allow it. And it shows why one famous four-point problem — XOR — stopped it completely and helped slow the whole field for a decade.*

Fundamental concept

Imagine sorting apples into "ripe" and "unripe" with a balance scale. On one pan you place weights for the evidence — a little for color, more for softness, a counterweight for size — and read which way it tips. When it tips the wrong way you change those weights, and over a whole crate the scale calibrates itself. The perceptron is exactly this self-adjusting scale, built out of arithmetic instead of brass.

Formally it is one neuron (chapter 1) with a **step** output: it forms the weighted sum $z = \mathbf{w}\cdot\mathbf{x} + b$ and fires $\hat{y} = 1$ when $z \geq 0$, otherwise 0. What Rosenblatt added to the McCulloch–Pitts neuron was not the unit but the **learning rule**. This is a method that reads the neuron's error on one example and rewrites the weights, so the same mistake is less likely next time. His hardware, the Mark I Perceptron, wired a 20×20 grid of photocells to banks of motor-driven potentiometers. The "program" was that wiring plus this rule.

Geometrically, the neuron draws a single straight **decision boundary** — the line where $z = 0$, a hyperplane in higher dimensions — and labels everything on one side 1 and the other side 0. Learning means *moving that line* until the two classes sit clearly on opposite sides. The entire story of the perceptron is the story of how one line is pushed and rotated by the points it gets wrong.



*Linearly separable data. The weight vector **w** is perpendicular to the boundary z = 0 and points into the region the neuron labels 1. The bias slides the line along that direction.*

Governing equations

The model is a neuron whose activation is a hard threshold:

$$z = \mathbf{w}\cdot\mathbf{x} + b = \Sigma_{i=1..n}\, w_i x_i + b$$

$$\hat{y} = \text{step}(z) = 1 \text{ if } z \geq 0, \text{ else } 0$$

$$\mathbf{w}\cdot\mathbf{x} + b = 0 \quad \text{(the decision boundary)}$$

After the neuron guesses $\hat{y}$ on a labeled example ($\mathbf{x}$, y), the **perceptron learning rule** corrects the weights and bias in proportion to the error:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$$

$$b \leftarrow b + \eta(y - \hat{y})$$

- $\mathbf{x_i}$ — input feature i (dimensionless). **x** is the input vector, length n
- $\mathbf{w_i}$ — weight on input i — the learned strength and sign of that evidence
- **b** — bias — the learned offset. It sets the distance of the boundary from the origin
- **y** — true label of the example, either 0 or 1
- $\hat{\mathbf{y}}$ — the neuron's prediction, step of z, either 0 or 1
- $\boldsymbol{\eta}$ — learning rate — step size > 0. It scales how far each mistake moves the weights

The error term $(y - \hat{y})$ can take only three values, which is why the rule is so simple:

$$y - \hat{y} = 0 \rightarrow \text{no change}; \quad +1 \rightarrow \mathbf{w} \leftarrow \mathbf{w} + \eta\mathbf{x}; \quad -1 \rightarrow \mathbf{w} \leftarrow \mathbf{w} - \eta\mathbf{x}$$

Correctly classified points are left untouched. The neuron only changes when it is wrong, and only in response to the point it got wrong.

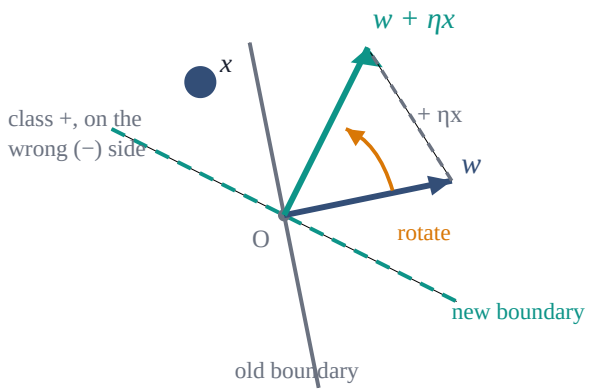
How it works — rotating the line toward its mistakes

Why does adding $\eta\mathbf{x}$ to $\mathbf{w}$ help? Suppose the true label is 1 but the neuron said 0 (a false negative): the point $\mathbf{x}$ is on the wrong side, so $z = \mathbf{w}\cdot\mathbf{x} + b$ was too small. The update $\mathbf{w} \leftarrow \mathbf{w} + \eta\mathbf{x}$ tilts the weight vector *toward* $\mathbf{x}$. This increases the next $\mathbf{w}\cdot\mathbf{x}$ and swings the perpendicular boundary, so that $\mathbf{x}$ is pulled onto the positive side. A false positive does the opposite: subtracting $\eta\mathbf{x}$ tilts the boundary away. Each correction is a small rotation-plus-shift aimed straight at the point that caused the error. In concrete terms, re-scoring that same false-negative point right after its update gives

$$z_{\text{new}} = (\mathbf{w} + \eta\mathbf{x})\cdot\mathbf{x} + b = z + \eta\|\mathbf{x}\|^2 \geq z$$

so the update always pushes that point toward the correct side.

Two motions are combined into a single update, and pulling them apart makes the geometry clearer. The weight vector $\mathbf{w}$ fixes the boundary's *orientation* — it is perpendicular to the line and points into the positive region — so adding $\eta\mathbf{x}$ *rotates* the boundary. The bias b fixes the boundary's *offset* from the origin, so adding $\eta(y - \hat{y})$ *slides* the whole line along $\mathbf{w}$ without turning it. A single false negative therefore does two things at once that work together: it swings the line toward the stray positive point and pushes the line outward past it.



One update on a misclassified positive point. The rule $\mathbf{w} \leftarrow \mathbf{w} + \eta\mathbf{x}$ *tilts the weight vector toward* $\mathbf{x}$ *(blue → teal), rotating the decision boundary (gray → dashed teal) around the origin until* $\mathbf{x}$ *lands on the positive side. The bias update, not drawn, slides the whole line along* $\mathbf{w}$*.*

Seen this way, the perceptron is the original **online learner**: it looks at one example at a time, reacts only to its own mistakes, and never stores the dataset or solves a global system of equations. That error-driven, one-point-at-a-time character is exactly the shape of stochastic gradient descent (chapter 7), which the rest of the book is built on. The perceptron is simply the special case in which the "gradient" is the rough signal $(y - \hat{y})\mathbf{x}$ returned by a hard threshold.

1. Pick a misclassified example $(\mathbf{x}, y)$.
2. Update: $\mathbf{w} \leftarrow \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$, and $b \leftarrow b + \eta(y - \hat{y})$.
3. Sweep the data again. Repeat until a full pass produces zero mistakes.

Remarkably, this rough loop comes with a guarantee. The data are **linearly separable with margin** γ if some ideal weights $(\mathbf{w}^*, b^*)$ classify every point correctly with a clear gap:

$$(2y_i - 1)(\mathbf{w}^* \cdot \mathbf{x}_i + b^*) \geq \gamma > 0\,, \quad \|\mathbf{x}_i\| \leq R \quad \text{for all } i$$

The **perceptron convergence theorem** (Novikoff, 1962) then says the perceptron stops after at most

$$\text{updates} \leq \frac{R^2}{\gamma^2}$$

corrections, regardless of the order of examples or the learning rate. The number is finite and depends only on the geometry: wide margins are learned fast, narrow ones slowly. The weak point is the assumption it needs. If no straight line separates the data, the loop never ends. It circles forever and never settles.

It helps to picture the two quantities in that bound. The **margin** γ is the width of the empty corridor that the ideal boundary cuts between the classes: with $\|\mathbf{w}^*\| = 1$ the quantity $\mathbf{w}^* \cdot \mathbf{x} + b^*$ is a signed distance, so γ is literally how far the nearest point is from the dividing line. The **radius** R is how far the data reach: the smallest ball centered at the origin that contains every input. Their ratio R/γ is a pure, unitless number. Rescale every input by a constant c and both R and γ scale by c, so $(R/\gamma)^2$ stays the same. Change the learning rate η and only the *length* of $\mathbf{w}$ changes, which cancels in the sign test. The bound therefore measures a real geometric difficulty. A wide corridor cutting through a tight cloud is easy. A very thin corridor through a spread-out cloud is hard. The bound does not depend on the unimportant details of the algorithm.

Why it converges: a proof in two inequalities

The bound is not magic. It follows from two short observations. Combine the bias into the weights by adding a constant 1 to every input (so the boundary passes through the origin), write the signed label $t = 2y - 1 \in \{-1, +1\}$, and start from $\mathbf{w} = \mathbf{0}$. Suppose a unit separator $\mathbf{w}^*$ ($\|\mathbf{w}^*\| = 1$) reaches margin γ, meaning $t_i(\mathbf{w}^* \cdot \mathbf{x}_i) \geq \gamma$ for every example, with $\|\mathbf{x}_i\| \leq R$. Let $\mathbf{w}_k$ be the weight vector right after the k-th mistake, made on a point ($\mathbf{x}$, t), whose update is $\mathbf{w}_k = \mathbf{w}_{k-1} + t\mathbf{x}$.

1. **Alignment grows at least linearly.** Dot the update with $\mathbf{w}^*$: $\mathbf{w}_k \cdot \mathbf{w}^* = \mathbf{w}_{k-1} \cdot \mathbf{w}^* + t(\mathbf{x} \cdot \mathbf{w}^*) \geq \mathbf{w}_{k-1} \cdot \mathbf{w}^* + \gamma$. After k mistakes, $\mathbf{w}_k \cdot \mathbf{w}^* \geq k\gamma$.
2. **Length grows at most as $\sqrt{k}$.** $\|\mathbf{w}_k\|^2 = \|\mathbf{w}_{k-1}\|^2 + 2t(\mathbf{x} \cdot \mathbf{w}_{k-1}) + \|\mathbf{x}\|^2$. The middle term is ≤ 0 because the point was *misclassified* (it was on the wrong side, so $t(\mathbf{x} \cdot \mathbf{w}_{k-1}) \leq 0$), and $\|\mathbf{x}\|^2 \leq R^2$, so $\|\mathbf{w}_k\|^2 \leq \|\mathbf{w}_{k-1}\|^2 + R^2 \leq kR^2$.

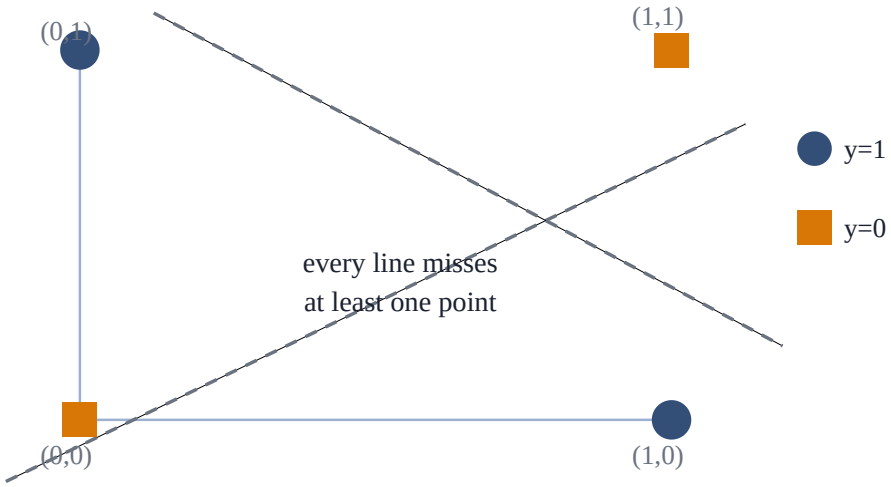
3. **Squeeze the two together.** Cauchy–Schwarz with $\|\mathbf{w}^*\| = 1$ gives $\mathbf{w}_k \cdot \mathbf{w}^* \le \|\mathbf{w}_k\|$, so $k\gamma \le \|\mathbf{w}_k\| \le \sqrt{(kR^2)}$. Squaring, $k^2\gamma^2 \le kR^2$, so $k \le R^2/\gamma^2$.

The mistake count is limited by geometry alone — not by the number of examples, their order, or the learning rate (which only rescales **w** and cancels out). Two forces compete: the margin drives alignment up linearly in k, while the bounded inputs let the length grow only as √k. And a cosine cannot climb past 1 forever. This is one of the oldest convergence proofs in machine learning. Its shape — a lower bound on progress set against an upper bound on drift — appears again and again across optimization.

It is worth looking closely at what the two inequalities are really doing. Every mistake is *forced* to add at least γ to the alignment $\mathbf{w}_k \cdot \mathbf{w}^*$ — a guaranteed step forward — while the length $\|\mathbf{w}_k\|$ is *permitted* to grow only like √k, because a point the neuron got wrong could not already have been pulling **w** the right way. So progress accumulates linearly in the number of mistakes, while drift accumulates only sublinearly. And since alignment can never exceed length (a direction cosine tops out at 1), the linear term must run out of room after a finite number of steps. Notice what the argument never uses: the number of examples, the order they arrive in, and the learning rate. The cap counts *mistakes*, not passes, so a large dataset seen many times still corrects itself at most $(R/\gamma)^2$ times in total.

The wall: XOR and linear separability

That assumption turned out to be a major obstacle. The exclusive-OR function returns 1 when exactly one of two binary inputs is 1: $(0,0) \to 0$, $(1,1) \to 0$, $(0,1) \to 1$, $(1,0) \to 1$. Plot the four points, and the two "1" cases sit on one diagonal while the two "0" cases sit on the other. No single straight line can put both 1s on one side and both 0s on the other. You would need to separate opposite corners at the same time.



XOR is not linearly separable: the two classes occupy opposite corners, so any straight boundary (dashed) misclassifies at least one point. A curved or piecewise boundary is required.

In 1969 Marvin Minsky and Seymour Papert made this precise in their book *Perceptrons*, proving that a single-layer perceptron cannot represent XOR, parity, or connectedness. Together with unmet expectations, the result drained optimism and funding from neural-network research through the 1970s. This was the first "AI winter." The way out was known in theory even then, and it is the subject of chapter 3: stack a **hidden layer** of perceptron-like units. One hidden neuron can cut out the half-plane "at least one input on," another "not both on," and an output neuron can AND them. The result is a curved decision region built from straight pieces. Depth, not cleverness in the rule, is what defeats XOR.

There are, in fact, two ways out, and this book follows both. The first is to *learn* new features. A hidden layer finds, on its own, coordinates in which the classes separate along a straight line (chapter 3). The general study of such learned coordinates — why they work and what makes a good one — is the subject of representations (chapter 5). The second is to *hand-build* features. Add the single product $x_1 \cdot x_2$ to XOR's two raw inputs, and the four points become linearly separable in the resulting three-dimensional space. This is the classic exercise worked below. The two approaches share one idea: bend the space until a straight line is enough. This is also the seed of the kernel trick behind support vector machines. What a single-layer perceptron cannot do is find that bending by itself. It is stuck with the coordinates it is given.

The perceptron in code

The rule fits in a dozen lines. Here it is on a small linearly separable set. Press **Run** and watch it stop the moment a full sweep makes no mistake:

Python · NumPy ▶ runnable in the app

```
import numpy as np
# Perceptron learning rule on a linearly separable set.
X = np.array([[2., 2.], [1., 3.], [3., 1.],       # class 1
              [-1., -2.], [-2., -1.], [-2., -3.]]) # class 0
y = np.array([1, 1, 1, 0, 0, 0])
w = np.zeros(2); b = 0.0; eta = 1.0
step = lambda z: 1 if z >= 0 else 0
for epoch in range(1, 21):
    errors = 0
    for xi, yi in zip(X, y):
        e = yi - step(np.dot(w, xi) + b)
        if e != 0:
            w += eta * e * xi          # rotate boundary toward the mistake
            b += eta * e
            errors += 1
    if errors == 0:
        break
preds = np.array([step(np.dot(w, xi) + b) for xi in X])
print("converged after", epoch, "epochs")
print("w =", w, " b =", b)
print("accuracy =", (preds == y).mean())
```

Point the same loop at XOR and it never settles. The mistake count refuses to reach zero, sweep after sweep, exactly as the theory predicts when no separating line exists:

Python · NumPy ▶ runnable in the app

```
import numpy as np
# The same rule on XOR: it can never reach zero mistakes.
X = np.array([[0., 0.], [0., 1.], [1., 0.], [1., 1.]])
y = np.array([0, 1, 1, 0])                  # XOR
w = np.zeros(2); b = 0.0; eta = 1.0
step = lambda z: 1 if z >= 0 else 0
mistakes = []
for epoch in range(12):
    errs = 0
```

```
    for xi, yi in zip(X, y):
        e = yi - step(np.dot(w, xi) + b)
        if e != 0:
            w += eta * e * xi; b += eta * e; errs += 1
    mistakes.append(errs)
print("mistakes per epoch:", mistakes)
print("never 0 -> XOR is not linearly separable; the loop cycles forever")
```

Modern frameworks express the same linear unit as a single dense layer. The classic 0/1 update is one hand-written line. In practice, though, one trains this layer with a smooth loss (chapter 6) and gradient descent (chapter 7) rather than the step rule, precisely so that it stays differentiable:

PyTorch copy → Colab

```
import torch, torch.nn as nn

layer = nn.Linear(2, 1)                    # one linear unit: z = w·x + b
X = torch.tensor([[2., 2.], [1., 3.], [-1., -2.], [-2., -1.]])
y = torch.tensor([1., 1., 0., 0.])
eta = 1.0
for _ in range(10):                        # sweeps over the data
    for xi, yi in zip(X, y):
        z = layer(xi)
        yhat = (z >= 0).float()
        err = yi - yhat                    # in {-1, 0, +1}
        with torch.no_grad():              # the classic rule is not
autograd
            layer.weight += eta * err * xi
            layer.bias   += eta * err
print("w =", layer.weight.data, " b =", layer.bias.data)
```

Beyond separable data: the pocket algorithm

The XOR cell exposes the perceptron's main weakness: on non-separable data the weights swing back and forth forever, and whichever vector you happen to stop on may be very bad, because the rule keeps over-reacting to the most recent mistake. The **pocket algorithm** (Gallant, 1990) adds one line of memory: run the ordinary perceptron, but keep in your "pocket" the best weights seen so far. These are the weights that lasted through the longest run of consecutive correct classifications. Return *those* at the end. It turns the perceptron into a usable,

though rough, classifier for messy data. It also points ahead to the modern habit of keeping the **best model seen so far** during a noisy training run — the early-stopping idea (chapter 12). The deeper fix, of course, is to give up the hard step. Put in a smooth activation and a differentiable loss, and the perceptron becomes a neuron that gradient descent can train — the path taken from chapter 4 onward.

The pocket fix also points to a deeper weakness. Because the perceptron stops at the *first* separator it finds, the line it returns can lie very close to the training points. This leaves no room for unseen data. The convergence proof already hinted at the cure: the quantity that controlled everything was the margin γ, the width of the corridor between the classes. Why not choose, among all the separating lines, the one that makes γ as *large* as possible — the widest corridor that still fits between the two classes? That single change of objective turns the perceptron into the **support vector machine** (Cortes & Vapnik, 1995): the same linear score $\mathbf{w}\cdot\mathbf{x} + b$, but fitted to *maximize* the margin instead of just stopping at zero mistakes. The maximum-margin boundary is unique and generalizes better. It removes the order-dependence that lets two perceptron runs on identical data finish with two different answers. The margin that only *bounded* the perceptron's mistakes becomes, in the SVM, the very thing to optimize.

In this app's engine

The **Perceptron lab** animates exactly this rule. Drop points of two colors, press step, and watch the boundary line jump after each misclassified point while the weight vector arrow rotates. A mistake counter ticks down toward the convergence bound. Switch the dataset to XOR and the line never settles. It swings back and forth forever, a live demonstration of the theorem's assumption failing. The lab uses the same {0,1} step and $(y - \hat{y})$ rule derived here, not the smooth sigmoid of later chapters.

Worked example — three updates by hand

Two positives $P_1 = (2, 2)$, $P_2 = (-2, -2)$ with $y = 1$. Two negatives $N_1 = (1, -1)$, $N_2 = (2, -2)$ with $y = 0$. Start at $\mathbf{w} = (0, 0)$, $b = 0$, learning rate $\eta = 0.5$. We sweep the four points and update only on mistakes (note step(0) = 1, so the all-zero neuron predicts 1

everywhere at first).

1. **Sweep 1.** P_1, P_2 give $z = 0 \to \hat{y} = 1 = y$: correct, skip. $N_1 = (1,-1)$: $z = 0 \to \hat{y} = 1$, but $y = 0$, so $e = -1$. Update: $\mathbf{w} \leftarrow (0,0) - 0.5(1,-1) = \mathbf{(-0.5, 0.5)}$, $b \leftarrow 0 - 0.5 = \mathbf{-0.5}$. Now $N_2 = (2,-2)$: $z = -0.5 \cdot 2 + 0.5 \cdot (-2) - 0.5 = -2.5 < 0 \to \hat{y} = 0 = y$: correct.
2. **Sweep 2, point $\mathbf{P_1}$** $= (2,2)$: $z = -0.5 \cdot 2 + 0.5 \cdot 2 - 0.5 = -0.5 < 0 \to \hat{y} = 0$, but $y = 1$, $e = +1$. Update: $\mathbf{w} \leftarrow (-0.5,0.5) + 0.5(2,2) = \mathbf{(0.5, 1.5)}$, $b \leftarrow -0.5 + 0.5 = \mathbf{0}$.
3. **Sweep 2, point $\mathbf{P_2}$** $= (-2,-2)$: $z = 0.5 \cdot (-2) + 1.5 \cdot (-2) + 0 = -4 < 0 \to \hat{y} = 0$, $y = 1$, $e = +1$. Update: $\mathbf{w} \leftarrow (0.5,1.5) + 0.5(-2,-2) = \mathbf{(-0.5, 0.5)}$, $b \leftarrow 0 + 0.5 = \mathbf{0.5}$.
4. **Check.** With $\mathbf{w} = (-0.5, 0.5)$, $b = 0.5$ the boundary is $x_2 = x_1 - 1$. All four points now satisfy $z \geq 0$ for the positives and $z < 0$ for the negatives (P_1: $z = 0.5$; N_2: $z = -1.5$), so the next full sweep makes no change: **converged in 3 updates**.

The boundary swung from $x_2 = x_1 + 1$ (after update 1) through a steep tilt to its final $x_2 = x_1 - 1$. Notice that update 1 *over-rotated* and briefly misclassified the positives, which updates 2 and 3 fixed. The errors themselves guided the line to its final place.

Worked example — the mistake bound in numbers

Take four points: positives A = (3, 3), B = (1, 1) with $y = 1$. Negatives C = (−1, −1), D = (−2, −1) with $y = 0$. The line $x_1 + x_2 = 0$ separates them, so the ideal *unit* weight vector is $\mathbf{w}^* = (1, 1)/\sqrt{2}$ (perpendicular to that line, $\|\mathbf{w}^*\| = 1$, $b^* = 0$). From it we read the two geometric quantities the theorem needs.

1. **Margin γ.** The signed distance of each point to the boundary is $t \cdot (\mathbf{w}^* \cdot \mathbf{x})$ with $t = 2y - 1$. For A: $1 \cdot (3+3)/\sqrt{2} = 4.24$; B: $1 \cdot (1+1)/\sqrt{2} = \mathbf{1.41}$; C: $-1 \cdot (-1-1)/\sqrt{2} = \mathbf{1.41}$; D: $-1 \cdot (-2-1)/\sqrt{2} = 2.12$. The smallest — the point closest to the corridor's edge — is $\gamma = \sqrt{2} \approx \mathbf{1.41}$.

2. **Radius R.** The farthest input from the origin is A, with $\|A\| = \sqrt{(3^2+3^2)} = 3\sqrt{2} \approx$ **4.24**, so $R = 3\sqrt{2}$.
3. **Bound.** updates $\le (R/\gamma)^2 = (3\sqrt{2}/\sqrt{2})^2 = 3^2 =$ **9**. The perceptron will make at most nine corrections on this set, in any example order.
4. **What actually happens.** Running the rule from $\mathbf{w} = \mathbf{0}$ (no bias needed, since the boundary passes through the origin), the all-zero neuron fires 1 everywhere. A and B are already right, C triggers the one correction $\mathbf{w} \leftarrow (1, 1)$, and the next sweep is clean. It converges after a **single mistake**, far below the guarantee of nine.

The bound is a *worst case*: it guarantees the process ends and limits the wait, but real runs on well-separated data usually finish much sooner. Slide B and C toward the boundary so the margin halves, and the cap jumps from 9 toward 36. This is the quadratic price of a thinner corridor, exactly the dependence the theorem predicts.

Assumptions & limitations

The convergence guarantee holds only for linearly separable data. On anything else (XOR, overlapping clouds, noise) the rule never stops, and there is no notion of a "best" line it approaches. The step activation has zero gradient, so the perceptron rule is a special hand-made trick, not gradient descent. It cannot be placed into the backpropagation machinery of chapter 8, which needs a smooth output. The neuron also returns a plain 0/1 with no confidence: unlike logistic regression (chapter 1), it does not tell you *how* sure it is. And the final boundary depends on example order and initialization. The perceptron finds *a* separator, not the widest-margin one that a support vector machine would choose.

Why & when this matters

The perceptron is the conceptual seed of everything that follows: a linear score, a threshold, and an error-driven weight update are still the basic structure of a modern network's training step. Its failure on XOR is the clearest single argument for *depth*. It is the reason we build multilayer networks at all (chapter 3). And its margin bound hints at why well-separated, well-scaled data train faster. You will rarely use a raw perceptron today. But every time you watch a boundary move in the Playground, you are watching Rosenblatt's rule, smoothed and generalized.

In the real world

The perceptron rule still lives inside large-scale linear classifiers, where simplicity and speed matter more than the final bit of accuracy. Its direct descendants — the **averaged perceptron** (Freund & Schapire, 1999) and Collins' **structured perceptron** (2002) — were for years the standard choice for part-of-speech tagging and named-entity recognition in natural-language processing, training in minutes on millions of sentences. The same error-driven, mistake-only update is the conceptual core of online systems that must learn from a stream of clicks in real time. And Rosenblatt's 1958 Mark I, wired to a 20×20 grid of photocells, is the direct ancestor of every image classifier that followed.

Exercises

1. Current weights $w = (1, -1)$, $b = 0$, learning rate $\eta = 1$. The point $x = (1, 2)$ has label $y = 1$. Do one perceptron update by hand, then re-score the point. MATH

2. A separable dataset has all inputs within $\|x\| \le 5$ and can be split with margin $\gamma = 0.5$. What does the convergence theorem limit the number of updates to? MATH

3. Why does the perceptron leave the weights unchanged on a correctly classified point, and how does that differ from gradient descent? CONCEPT

4. Confirm in code that the perceptron converges to 100% accuracy on a randomly generated linearly separable set. CODE

5. XOR is not linearly separable in its two raw inputs. Add the product feature $x_1 \cdot x_2$ and show in code that the perceptron now converges. Why does this work? CODE

6. The bound is $(R/\gamma)^2$. For fixed $R = 4$, compare the limits at $\gamma = 1$ and $\gamma = 0.25$ and state the lesson. MATH

7. Two students each run the perceptron on the same separable data but shuffle the examples differently, and they end with different boundaries, though both reach 100% training accuracy. Who is "right", and what model would remove the ambiguity? CONCEPT

8. For the dataset with positives $(2, 0)$, $(0, 2)$ at $y = 1$ and negatives $(-2, 0)$, $(0, -2)$ at $y = 0$, give a unit separating vector $\mathbf{w}^*$, then compute the margin γ, the radius R, and the perceptron mistake bound. MATH

9. Show that, starting from $\mathbf{w} = \mathbf{0}$ and $b = 0$, doubling the learning rate η changes neither the perceptron's final decision boundary nor its sequence of mistakes. Why, then, does η disappear from the bound? CONCEPT

10. On non-separable data the raw perceptron can stop on poor weights. Implement the **pocket algorithm** — carry along the weights that produced the longest run of consecutive correct classifications — and compare its accuracy with the final weights. CODE

CHAPTER 3 · FOUNDATIONS

Multilayer networks & the forward pass

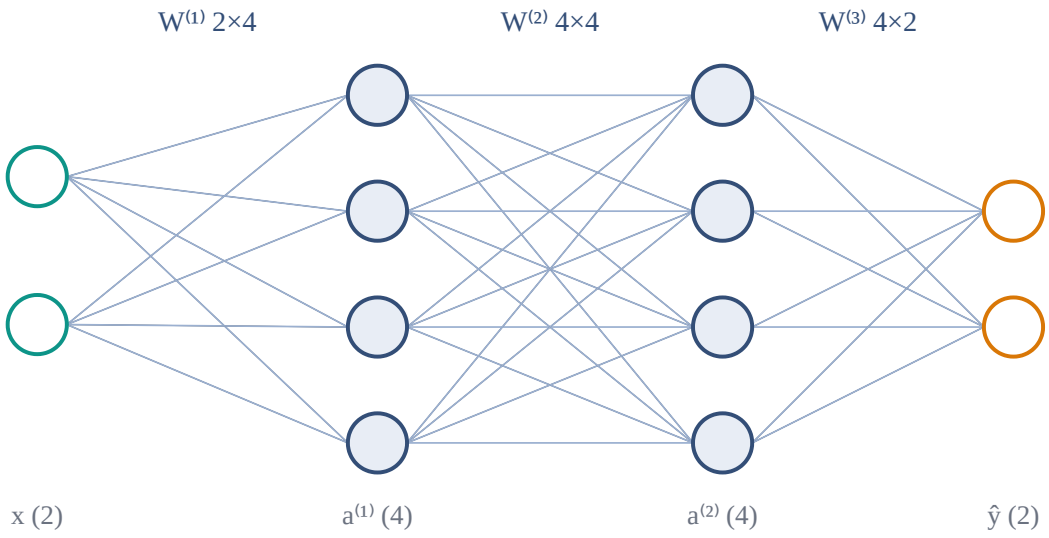
> *One neuron answers one linear question (chapter 1). One layer of neurons asks many questions at once, and stacking layers lets later questions be built from earlier answers. This chapter turns a group of neurons into clean matrix algebra, so a whole layer becomes a single matrix multiply. It explains why a nonlinearity between layers is not optional, and it states the theorem that says such networks can, in principle, approximate almost anything.*

Fundamental concept

Think of a panel of judges scoring a diving competition. Each judge (a neuron) rates the same routine by their own standards and reports one number. A layer is the whole panel: same input, many independent opinions, produced at the same time. Because every judge multiplies the same inputs by their own set of weights, we can stack all their weight lists side by side into a grid — a **matrix** — and compute the entire panel's scores in one round of arithmetic.

That is the first big idea of this chapter: *a layer is a matrix multiply.* If a layer has n inputs and k neurons, its weights form an n×k matrix W. Multiplying the input row by W produces all k weighted sums at once, and adding a length-k bias row and applying an activation finishes the layer. The second idea is **depth**: feed one layer's outputs as the next layer's inputs. Early layers detect simple features. Later layers combine them into complex ones. A network is just this composition repeated a few times.

The **multilayer perceptron** (MLP) is the simplest such network: an input layer, one or more **hidden** layers that you never see directly, and an output layer. Its forward pass, which turns an input into a prediction, is just a chain of "matrix multiply, add bias, squash." Everything in the training chapters (loss, chapter 6; gradient descent, chapter 7; backpropagation, chapter 8; optimizers, chapter 9) exists to choose the numbers inside those matrices.



A small 2–4–4–2 network (to build it in the app, set two hidden layers of 4 in the Network sheet). Each gap is one weight matrix. A forward pass is three multiply-add-squash steps, from left to right.

Governing equations

Number the layers $\ell = 1 \ldots L$. Writing the input as $a^{(0)} = x$, each layer transforms the previous layer's activations:

$$z^{(\ell)} = a^{(\ell-1)} W^{(\ell)} + b^{(\ell)}$$

$$a^{(\ell)} = g(z^{(\ell)})$$

$\mathbf{a}^{(\ell)}$	activations of layer ℓ — a row vector of length n_ℓ (the layer's width); $a^{(0)} = x$, $a^{(L)} = \hat{y}$
$\mathbf{W}^{(\ell)}$	weight matrix of layer ℓ, shape $n_{\ell-1} \times n_\ell$ — one column per neuron
$\mathbf{b}^{(\ell)}$	bias row vector of layer ℓ, length n_ℓ
$\mathbf{z}^{(\ell)}$	pre-activations of layer ℓ (row of length n_ℓ)
$\mathbf{g}$	activation function, applied element-wise (chapter 4)
$\mathbf{L}$	number of layers with weights (depth); n_ℓ is the width of layer ℓ

Each column j of $W^{(\ell)}$ is exactly one neuron's weight vector, so the matrix product $a^{(\ell-1)} W^{(\ell)}$ stacks all n_ℓ weighted sums into one row. For a 2-input, 3-neuron layer this reads:

$$[a_1\ a_2] \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} + [b_1\ b_2\ b_3] = [z_1\ z_2\ z_3]$$

Chaining all L layers writes the whole forward pass as one nested expression. For the app's three-weight-layer default, it reads

$$\hat{y} = g(g(g(x\,W^{(1)} + b^{(1)})\,W^{(2)} + b^{(2)})\,W^{(3)} + b^{(3)})$$

To push a whole **batch** of m examples through at once, stack them as rows of a matrix $A^{(0)} = X$ of shape $m \times n_0$. The equations keep the same form:

$$Z^{(\ell)} = A^{(\ell-1)}W^{(\ell)} + b^{(\ell)}, \quad A^{(\ell)} = g(Z^{(\ell)})$$

only now every quantity has an extra row dimension m, and the bias is broadcast across all m rows. Modern hardware runs this batched matrix multiply far faster than m separate ones.

Keeping track of the shapes is worth the effort, because a mismatched dimension is the most common programming error. For a batch of m examples the pieces of layer ℓ are:

$\mathbf{X = A^{(0)}}$	input batch, shape $m \times n_0$ — m examples down the rows, n_0 features across the columns
$\mathbf{W^{(\ell)}}$	weight matrix, shape $n_{\ell-1} \times n_\ell$ — its rows match the incoming width, its columns are the layer's n_ℓ neurons
$\mathbf{b^{(\ell)}}$	bias, shape $1 \times n_\ell$, added to (broadcast down) every one of the m rows
$\mathbf{Z^{(\ell)}}$, $\mathbf{A^{(\ell)}}$	pre-activations and activations of layer ℓ, both shape $m \times n_\ell$

The product $A^{(\ell-1)}W^{(\ell)}$ then reads $(m \times n_{\ell-1}) \cdot (n_{\ell-1} \times n_\ell) = m \times n_\ell$: the inner $n_{\ell-1}$ must match and cancels, and the surviving $m \times n_\ell$ is exactly "m examples, each now described by n_ℓ numbers." The batch dimension m never enters a weight matrix. It passes through untouched from input to output, which is why the same weights process a batch of 1 or a batch of 10,000 with no change to the code.

Why the nonlinearity is not optional

Suppose we drop g and let each layer be purely linear. Then two stacked layers compute

$$(xW^{(1)} + b^{(1)})W^{(2)} + b^{(2)} = x(W^{(1)}W^{(2)}) + (b^{(1)}W^{(2)} + b^{(2)}) = xW' + b'$$

$$W' = W^{(1)}W^{(2)}, \quad b' = b^{(1)}W^{(2)} + b^{(2)}$$

which is *one* linear layer. Any number of linear layers collapses to a single matrix: depth adds nothing, and the network can still only draw straight boundaries, so it is back to a perceptron. The activation g between layers is exactly what stops this collapse. Because g bends the space before the next matrix acts, stacked layers can trace curved, folded decision regions that no single linear map can. The nonlinearity is what makes depth meaningful.

The smallest problem a line cannot solve: XOR

The collapse argument is abstract. XOR makes it concrete. The exclusive-OR function labels two bits 1 when they differ and 0 when they agree: XOR(0,0) = 0, XOR(0,1) = 1, XOR(1,0) = 1, XOR(1,1) = 0. Plot the four inputs on the unit square. The two positive examples sit on one diagonal and the two negatives sit on the other, forming a checkerboard. No single straight line can put both positives on one side and both negatives on the other, so a single perceptron (chapter 2) cannot solve it. This is the exact example Minsky and Papert used in 1969 to reveal the perceptron's limits, and the discouragement it caused slowed the field for over a decade. One hidden layer with a nonlinearity solves the problem completely.

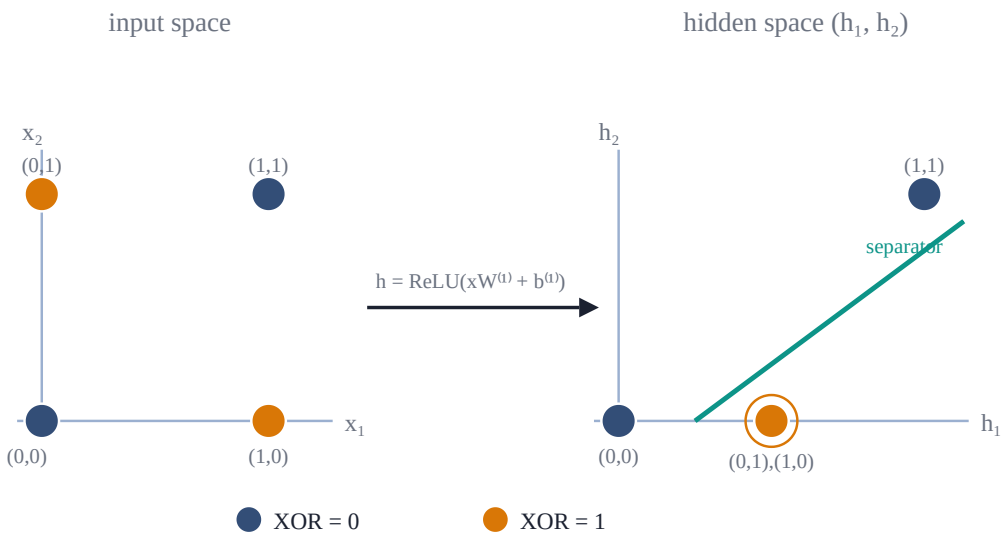
Here is a network that solves it, small enough to check by hand: two inputs, two ReLU hidden units, one linear output (a 2–2–1 MLP):

$$W^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad b^{(1)} = [0, -1], \quad W^{(2)} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad b^{(2)} = 0$$

$$h = \mathrm{ReLU}(x\,W^{(1)} + b^{(1)}), \quad \hat{y} = h\,W^{(2)} + b^{(2)}$$

Look at the two hidden units geometrically. Both share the weight direction (1, 1), so each is a soft step across a diagonal $x_1 + x_2 = c$. They differ only in their threshold. Unit $h_1 = \mathrm{ReLU}(x_1 + x_2)$ switches on for every corner except the origin, and its cut is the line $x_1 + x_2 = 0$. Unit $h_2 = \mathrm{ReLU}(x_1 + x_2 - 1)$ switches on only at the far corner (1, 1), and its cut is the line $x_1 + x_2 = 1$. The two parallel cuts slice the square into three diagonal bands. A point's band is shown by *how*

many hidden units fire: none at (0, 0), exactly one at the two middle corners, both at (1, 1). The output layer then reads XOR directly from that count. It subtracts twice the second unit from the first, which turns the far corner back off while leaving the two middle corners lit.



Left: the four XOR inputs on the unit square, colored by label. The two 1s (amber) sit on one diagonal and the two 0s (blue) on the other, so no straight line separates them, and a single perceptron fails. Right: the ReLU hidden layer re-plots each input as (h_1, h_2). Because both hidden units only measure $x_1 + x_2$, the two positive corners (0,1) and (1,0) fold onto the same point (1,0), and one straight line (teal) now separates the classes. The hidden layer has learned a representation (chapter 5) in which XOR is linearly separable.

Worked example — solving XOR with a 2–2–1 network

Take the weights above: $W^{(1)} = [[1, 1], [1, 1]]$, $b^{(1)} = [0, -1]$, $W^{(2)} = [1, -2]^T$, $b^{(2)} = 0$, ReLU hidden units, linear output. Push all four inputs through by hand. Each input is one row $x = [x_1, x_2]$.

1. $x = [0, 0]$: $z^{(1)} = [0+0, 0+0] + [0, -1] = [0, -1]$; ReLU clips the −1, giving $a^{(1)} = [0, 0]$; $\hat{y} = 0 \cdot 1 + 0 \cdot (-2) + 0 =$ **0** (target 0).
2. $x = [0, 1]$: $z^{(1)} = [0+1, 0+1] + [0, -1] = [1, 0]$; $a^{(1)} = [1, 0]$; $\hat{y} = 1 \cdot 1 + 0 \cdot (-2) =$ **1** (target 1).
3. $x = [1, 0]$: $z^{(1)} = [1+0, 1+0] + [0, -1] = [1, 0]$; $a^{(1)} = [1, 0]$; $\hat{y} = 1 \cdot 1 + 0 \cdot (-2) =$ **1** (target 1).

4. $x = [1, 1]$: $z^{(1)} = [1+1, 1+1] + [0, -1] = [2, 1]$; $a^{(1)} = [2, 1]$; $\hat{y} = 2 \cdot 1 + 1 \cdot (-2) = 2 - 2 = \mathbf{0}$ (target 0).

The four outputs are exactly [0, 1, 1, 0] = XOR. The key event is the fold at steps 2 and 3: the two *different* inputs [0, 1] and [1, 0] produce the *same* hidden activation [1, 0], because each hidden unit sees only $x_1 + x_2$ and both sums equal 1. The ReLU is what makes the fold stick. At $x = [0, 0]$ the second unit's pre-activation is -1, and the clip $\text{ReLU}(-1) = 0$ erases it. Without that clip the layer stays linear and, as the code exercise confirms, can only produce the straight-line map $2 - (x_1 + x_2) = [2, 1, 1, 0]$. This map agrees with XOR at the two middle corners but is wrong at (0, 0) and (1, 1). The nonlinearity (chapter 4) is exactly what turns an unsolvable linear problem into a solved one.

The forward pass in code

The whole forward pass is a short stack of "multiply, add, squash." Here is a $3 \to 4 \to 2$ network run on a batch of five examples. The shapes line up exactly as the worked example below predicts:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)
# a 3 -> 4 -> 2 MLP applied to a batch of 5 examples
X  = rng.randn(5, 3)
W1 = rng.randn(3, 4) * 0.5; b1 = np.zeros(4)
W2 = rng.randn(4, 2) * 0.5; b2 = np.zeros(2)
relu = lambda z: np.maximum(0.0, z)
Z1 = X @ W1 + b1;  A1 = relu(Z1)       # hidden layer
Z2 = A1 @ W2 + b2                       # output logits
print("shapes:  X", X.shape, "-> A1", A1.shape, "-> out", Z2.shape)
print("output row 0:", np.round(Z2[0], 3))
```

And here is the collapse from the section above, made concrete: two *linear* layers act exactly like a single matrix, but a nonlinearity between them changes the output, which makes depth worthwhile:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(1)
x  = rng.randn(1, 3)
W1 = rng.randn(3, 4); W2 = rng.randn(4, 2)
two_linear = (x @ W1) @ W2                # two stacked linear layers
one_linear = x @ (W1 @ W2)                # collapsed to one matrix W1@W2
print("two linear layers vs one matrix, max|diff| = {:.1e}".format(
      float(np.max(np.abs(two_linear - one_linear)))))
nonlin = np.tanh(x @ W1) @ W2             # a nonlinearity between them
print("with tanh inserted, output moves by {:.3f}".format(
      float(np.max(np.abs(nonlin - two_linear)))))
```

In PyTorch the same network is a `Sequential` of layers and activations. The framework stores the matrices and runs the batched matmuls for you (copy into Colab to run):

PyTorch copy → Colab

```python
import torch, torch.nn as nn

mlp = nn.Sequential(
    nn.Linear(3, 4), nn.ReLU(),      # hidden layer
    nn.Linear(4, 2),                 # output layer (logits)
)
x = torch.randn(5, 3)                # a batch of 5 examples
logits = mlp(x)                      # forward pass: (5,3) -> (5,2)
print(logits.shape)
print(sum(p.numel() for p in mlp.parameters()), "parameters")
```

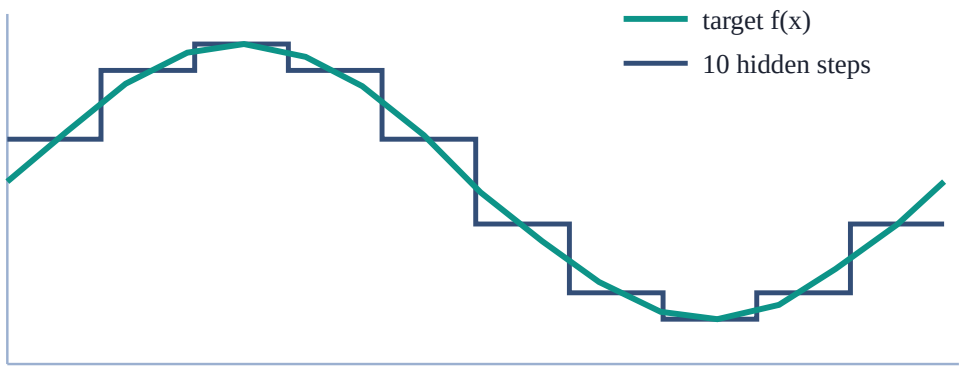
How much can such a network represent?

A great deal, in principle. The **universal approximation theorem** (Cybenko, 1989, for sigmoids; Hornik, 1991, for general activations) states that a feedforward network with a *single* hidden layer of finitely many neurons and a non-polynomial activation can approximate any continuous function on a closed, bounded region to any desired accuracy. But three important

limits keep this claim realistic. First, existence is not the same as learnability: the theorem promises that good weights *exist*, not that gradient descent will find them. Second, the required width can be extremely large. It often grows exponentially with the input dimension, while adding *depth* can reach the same accuracy with far fewer neurons. Third, it says nothing about generalizing to unseen data. That is the job of regularization (chapter 12). The theorem explains why we use neural networks at all, but the practical skill is finding a network that is both expressive and trainable.

A network of bumps: universal approximation you can see

Why should a single hidden layer be enough, even in principle? Here is the picture behind Cybenko's theorem. A sigmoid with a steep slope is almost a step. Place two such steps a little apart, and their difference is a localized **bump**. Give the hidden layer one neuron per step, and the output neuron can raise or lower the result over any small interval. Add enough of these contributions, and you build a staircase that follows any continuous target as closely as you wish. Just use more, finer steps. The hidden layer supplies the steps, and the output neuron does the summing.



Universal approximation, visually: each hidden sigmoid contributes one step, and their sum (blue) traces the smooth target (teal). More, finer steps (more hidden units) drive the gap to zero.

The cell below builds exactly such a staircase for a sine wave out of K sigmoids and reports the worst-case error:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
# One hidden layer of sigmoids builds any smooth 1-D function as a
staircase.
sigmoid = lambda z: 1.0 / (1.0 + np.exp(-z))
xs = np.linspace(0, 1, 300)
f  = np.sin(2 * np.pi * xs)            # target function
K, slope = 60, 80.0                    # K hidden units, steep steps
edges   = np.linspace(0, 1, K + 1)
centers = 0.5 * (edges[:-1] + edges[1:])
deltas  = np.diff(np.sin(2 * np.pi * edges))   # height of each step
approx  = f[0] + sum(d * sigmoid(slope * (xs - c))
                     for d, c in zip(deltas, centers))
print("hidden units:", K, " max|approx - f| =",
      round(float(np.max(np.abs(approx - f))), 3))
```

Raise K and the error shrinks toward zero. This is universal approximation in miniature: enough hidden units of a squashing nonlinearity can trace any shape. The theorem's fine print says that "enough" may mean impractically many, and that *finding* the weights is a separate problem from proving they *exist*. This is exactly what the training chapters, and the case for depth, are about.

In more than one input dimension the same idea takes a different form. A single hidden unit g(x·w + b) is constant along every direction perpendicular to its weight vector w, and it turns on only across the one direction parallel to w. Geometrically it is a **ridge**: a soft, oriented step stretched into an infinite wall. One ridge alone localizes nothing, but combine a few ridges that face different directions and you can wall off a bounded region, then raise a localized **bump** over it. This is the higher-dimensional version of "two steps make a bump." Cover the domain with enough such bumps and you trace any continuous surface. This is why the theorem holds in every dimension. It is also the first hint of its curse: the number of ridges needed to enclose a bump grows with the input dimension, and a single hidden layer must pay for all of them at once. This is the very cost that depth is about to reduce.

Depth versus width

If one hidden layer is already enough, why build deep networks? Because *sufficient* and *efficient* are not the same thing. The staircase argument uses a fresh hidden unit on every wiggle of the target, so a shallow network's width can grow very large with the complexity of the function. Depth trades that width for composition: layer two recombines the bumps of layer one, layer three recombines those, and each added layer can **multiply** the number of linear regions the network expresses rather than just add to it. The parity of n bits is the textbook case. A depth-2 network needs exponentially many neurons to represent it, while a network of depth O(log n) manages with just a few per layer. Real signals are themselves compositional (edges into shapes into objects, letters into words into meaning), so depth matches their structure. This is the practical lesson of the past decade: for a fixed parameter budget, deeper is usually stronger, right up to the training difficulties that the optimization and regularization chapters (optimizers, chapter 9; initialization, chapter 10; normalization, chapter 11; regularization, chapter 12) exist to control.

In this app's engine

The **Playground** is a live MLP. The layer sliders set the widths n_ℓ. The network diagram is the figure above, redrawn for your architecture, with edge color showing weight sign and thickness showing |w| (chapter 1's code). Each training step runs exactly the batched forward pass here, then the backward pass of backpropagation (chapter 8). The Math Log prints the shapes $A^{(\ell-1)}$, $W^{(\ell)}$, $Z^{(\ell)}$ at every layer, so you can watch the dimensions line up on real numbers.

Worked example — parameters and shapes of the 2–4–4–2 network

Count every learnable number, layer by layer. Each layer contributes $n_{\ell-1} \cdot n_\ell$ weights plus n_ℓ biases, so the total is

$$P = \Sigma_{\ell=1..L} \left(n_{\ell-1}\, n_\ell + n_\ell\right)$$

1. Layer 1 ($2 \to 4$): $2 \cdot 4 = 8$ weights + 4 biases = **12**.

2. Layer 2 ($4 \to 4$): $4 \cdot 4 = 16$ weights + 4 biases = **20**.
3. Layer 3 ($4 \to 2$): $4 \cdot 2 = 8$ weights + 2 biases = **10**.
4. Total: 12 + 20 + 10 = **42 parameters**. The entire "knowledge" of this network is 42 numbers.

Now trace the shapes for a batch of $m = 32$ examples (inputs $A^{(0)} = X$ is 32×2):

1. $Z^{(1)} = X \cdot W^{(1)} + b^{(1)}$: $(32 \times 2) \cdot (2 \times 4) = 32 \times 4$, then add the length-4 bias to every row $\to$ $A^{(1)}$ is **32×4**.
2. $Z^{(2)} = A^{(1)} \cdot W^{(2)}$: $(32 \times 4) \cdot (4 \times 4) = 32 \times 4 \to A^{(2)}$ is **32×4**.
3. $Z^{(3)} = A^{(2)} \cdot W^{(3)}$: $(32 \times 4) \cdot (4 \times 2) = 32 \times 2 \to$ the output $\hat{y}$ is **32×2**: two logits for each of the 32 examples.

The inner dimensions always cancel $(\ldots \times 4)(4 \times \ldots)$. This is the shape rule that requires a layer's input width to equal the previous layer's output width. The batch dimension 32 passes along untouched from start to finish.

Assumptions & limitations

The equations assume a plain feedforward stack: no loops, no weight sharing, and every neuron in a layer connected to every neuron in the next. Convolutional networks (chapter 13) and recurrent networks (chapter 14) break these assumptions on purpose to encode structure and save parameters. Universal approximation is an existence result, not a recipe: a network wide enough to fit a function may be impossible to train or may memorize noise. And the "one matmul per layer" picture hides real costs. A width of k makes a layer's compute and memory scale with k^2, which is why very wide layers are expensive. Depth brings its own danger: gradients can vanish or explode through many layers. This is the problem that careful initialization (chapter 10) and normalization (chapter 11) exist to control.

Why & when this matters

Thinking in terms of matrices is the daily language of deep learning. Most bugs are shape mismatches, and reading a model's parameter count tells you at a glance how much it can memorize and how much memory it needs. The MLP is still the default for tabular data and the building block inside every transformer's feed-forward sublayer (attention, chapter 16). Whenever you think about a bigger architecture, you come back to this chapter: it is all just layers of multiply, add, and squash.

In the real world

The plain MLP never left. It is still the first thing to reach for on tabular data (fraud scoring, churn prediction, sensor read-outs), where it competes well with gradient-boosted trees and is easy to deploy. More importantly, it is a *component*. Every transformer block that powers modern language models (attention, chapter 16) ends in a two-layer MLP applied at each position, and these feed-forward sublayers hold roughly two-thirds of a large model's parameters. Recommender systems at YouTube and Netflix push user and item features through a stack of dense layers, and DeepMind's AlphaFold runs MLP-style networks through its protein-geometry pipeline. When you count parameters and trace shapes here, you are practicing the very skill that sizes a billion-parameter model.

Exercises

1. How many learnable parameters (weights plus biases) does a 4–8–8–3 network have? MATH

2. A batch of 64 examples flows through a 10–32–16–1 network. Give the shape of every activation matrix from input to output. MATH

3. Without an activation between layers, a deep network is no more expressive than a single layer. Explain why in one or two sentences. CONCEPT

4. Build a $3 \to 5 \to 4 \to 2$ forward pass in NumPy for a batch of 8, and print the output shape and the parameter count. CODE

5. Take $W^{(1)} = [[1, 2], [0, 1]]$ and $W^{(2)} = [[1, 0], [3, 1]]$ (no activations, no bias). Find the single matrix W' that the two layers collapse to, and verify it on the input $x = [2, -1]$ in code. CODE

6. Both a very wide one-hidden-layer network and a deep network are universal approximators. Give two different reasons practitioners still prefer depth. CONCEPT

7. A dense layer maps 256 inputs to 256 neurons. Its compute scales with the product of the two widths. By what factor does the per-layer cost grow if you double the width to 512? MATH

8. Using the XOR network above ($W^{(1)} = [[1, 1], [1, 1]]$, $b^{(1)} = [0, -1]$, $W^{(2)} = [1, -2]^T$, $b^{(2)} = 0$), carry out the forward pass for $x = [1, 0]$ and report h_1, h_2, and $\hat{y}$. Which of the two cut lines $x_1 + x_2 = 0$ and $x_1 + x_2 = 1$ does this point clear? MATH

9. A classmate insists that stacking two *linear* layers should be enough to learn XOR "because it has more parameters than a perceptron." Explain, using the collapse identity and linear separability, why this is impossible and what exactly fixes it. CONCEPT

10. Implement the 2–2–1 XOR network in NumPy and confirm it outputs [0, 1, 1, 0]. Then swap the ReLU for the identity (a purely linear hidden layer) with the *same* weights and show the output can no longer match XOR. CODE

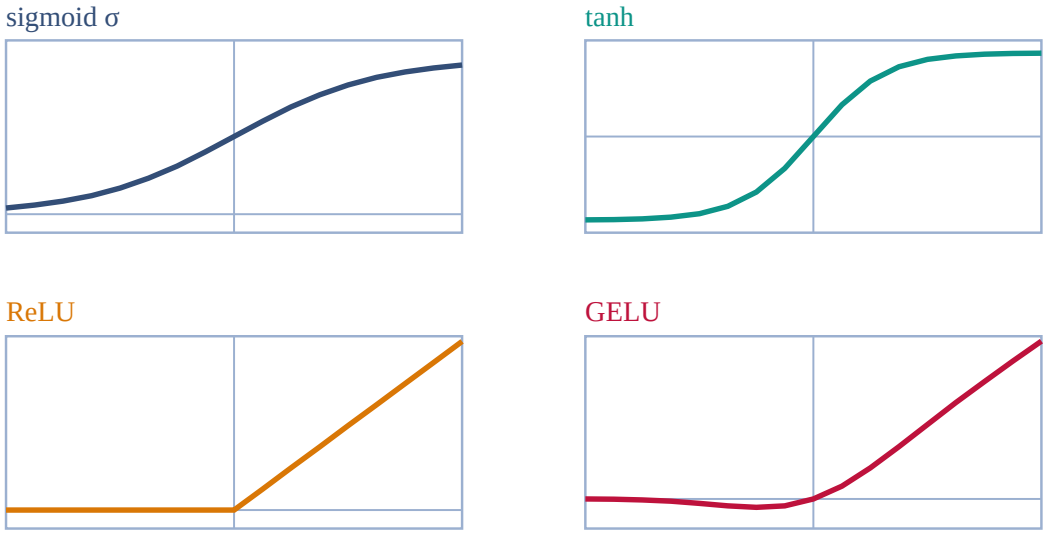
Activation functions

> *The activation function is the small nonlinear valve at the end of every neuron. It decides how a neuron's raw evidence z becomes its output. It also decides, through its derivative, how much learning signal flows back through that neuron during training. Choosing it well is one of the cheapest and most useful decisions in a network. This chapter goes through the eight activations built into the app, explains the saturation trap that once held back deep learning, and shows why a good default matters.*

Fundamental concept

A neuron without an activation is a passive relay. It just reports a weighted sum, and (chapter 3) a stack of such relays collapses into one linear map. The activation is what bends the signal so that layers can build curved, expressive functions. Think of it as a valve on a pipe. Some valves (sigmoid, tanh) are **saturating**. Past a certain pressure they are fully open or fully shut and stop responding to further change. Others (ReLU) are **one-way**: open above a threshold, closed below. The valve's *responsiveness*, its slope, is the key property. During backpropagation (chapter 8) the gradient arriving at a neuron is multiplied by that slope $g'(z)$. Where the valve is flat, the slope is near zero and almost no learning signal passes. The neuron is a closed gate.

A good hidden-layer activation therefore needs three things: it must be **nonlinear** (or layers collapse), **cheap** to compute (it runs on every neuron every step), and it should keep a **healthy gradient** over the range of z the network actually visits, so learning does not stop. The functions below balance these differently.



Four shapes over z ∈ [−3.2, 3.2]. Sigmoid and tanh flatten (saturate) at both ends. ReLU is a hard hinge at 0. GELU is a smooth hinge that dips slightly below zero before rising.

Governing equations: the eight activations

The **sigmoid** squashes any input into (0, 1), which reads as a probability. Its derivative is the simple $\sigma(1 - \sigma)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \sigma'(z) = \sigma(z)(1 - \sigma(z)) \le 0.25$$

Tanh is a rescaled sigmoid centered on zero, with output in (−1, 1). Its zero-centered outputs make the next layer's job easier:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}, \quad \tanh'(z) = 1 - \tanh^{2}(z)$$

$$\tanh(z) = 2\sigma(2z) - 1 \quad \text{(a stretched, re-centred sigmoid)}$$

The **ReLU** (rectified linear unit) is the modern default: a hinge that passes positive z unchanged and zeroes the rest. Its derivative is exactly 1 on the active side, so it does not shrink gradients. It is also almost free to compute:

$$\mathrm{ReLU}(z) = \max(0, z), \quad \mathrm{ReLU}'(z) = 1 \text{ if } z > 0, \text{ else } 0$$

Its cousins fix ReLU's one weakness (a flat, zero-gradient left side). **Leaky ReLU** gives the negative side a small slope α. **ELU** uses a smooth exponential that saturates gently to −α. **softplus** is a smooth everywhere-differentiable ReLU whose derivative is exactly the sigmoid. **GELU** and **Swish** are smooth, slightly non-monotonic hinges that dip a little below zero and now dominate large models:

$$\text{Leaky}(z) = \max(\alpha z, z)$$

$$\text{ELU}(z) = z \text{ if } z > 0, \text{ else } \alpha(e^z - 1)$$

$$\text{softplus}(z) = \ln(1 + e^z)$$

$$\text{GELU}(z) = z{\cdot}\Phi(z) \;\cdot\; \text{Swish}(z) = z{\cdot}\sigma(z)$$

z pre-activation input to the neuron (any real number)

g′(z) the activation's slope, the factor that gates gradient flow in backprop

α small positive constant for the negative branch (e.g. 0.01 for Leaky ReLU, 1 for ELU)

Φ(z) standard-normal cumulative distribution, the probability a N(0,1) draw is below z

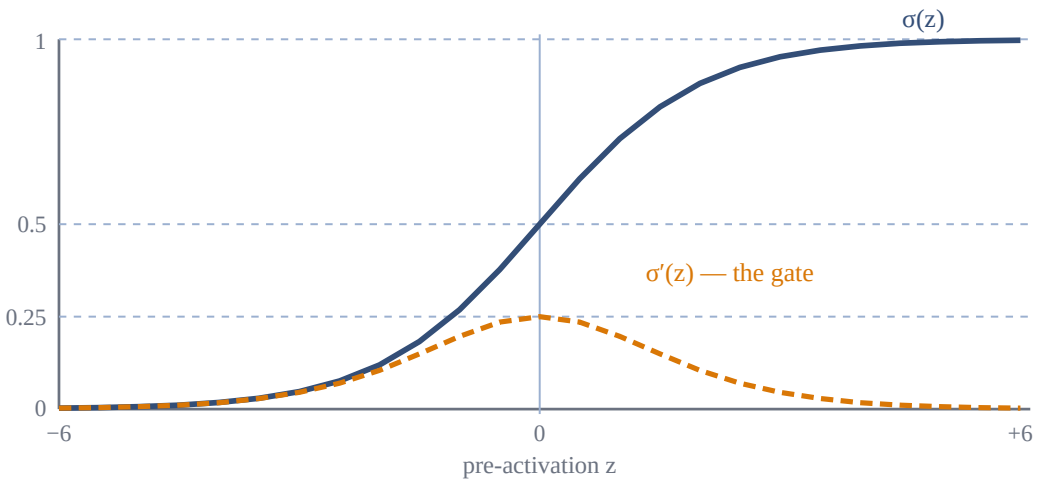
NAME	FORMULA	RANGE	ZERO-CENTERED	SATURATES
Sigmoid	$1/(1+e^{-z})$	(0, 1)	no	both ends
Tanh	$(e^z-e^{-z})/(e^z+e^{-z})$	(−1, 1)	yes	both ends
ReLU	max(0, z)	[0, ∞)	no	z < 0 (flat)
Leaky ReLU	max(αz, z)	(−∞, ∞)	≈ yes	no
ELU	z or $\alpha(e^z-1)$	(−α, ∞)	≈ yes	left → −α
Softplus	$\ln(1+e^z)$	(0, ∞)	no	grad only
GELU	z·Φ(z)	≈(−0.17, ∞)	no	left tail
Swish	z·σ(z)	≈(−0.28, ∞)	no	left tail

Saturation, the gradient gate, and dead ReLUs

Backpropagation (chapter 8) sends each layer's error back through the recursion below, where the activation's slope appears as an element-wise multiplier:

$$\delta^{(\ell)} = (\delta^{(\ell+1)}\, W^{(\ell+1)^{\mathrm{T}}}) \odot g'(z^{(\ell)})$$

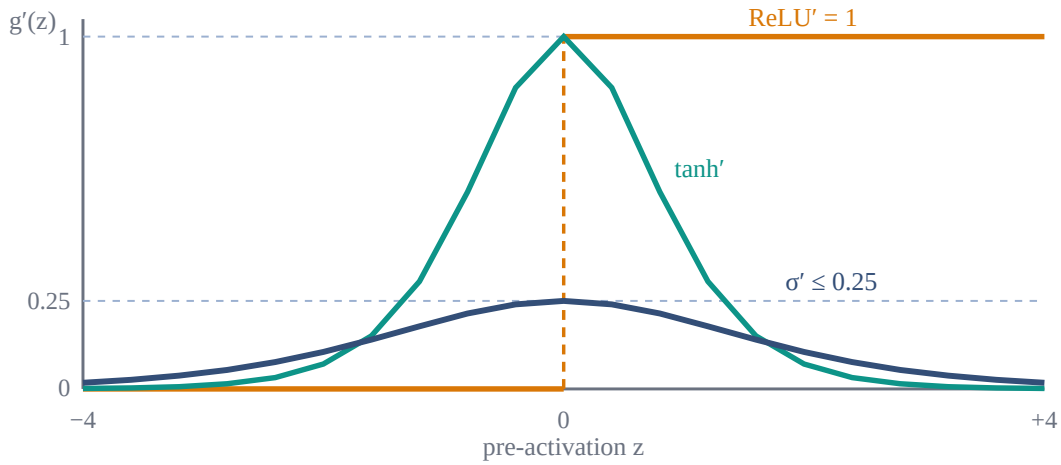
Because that step multiplies the incoming gradient by g′(z), a neuron sitting in a flat region of its activation multiplies by nearly zero and effectively stops learning. For sigmoid and tanh this happens whenever |z| is large, since the tails are flat. A neuron that saturates early can therefore freeze. Worse, sigmoid's slope never exceeds 0.25 *even at its best*, so stacking many sigmoid layers multiplies many factors below a quarter together and the gradient **vanishes** exponentially with depth (chapter 10). This is the historical reason deep sigmoid networks were nearly untrainable.



One activation shown together with its own derivative. The amber gate σ′(z) is literally the slope of the blue curve: it peaks at 0.25 exactly where σ is steepest (z = 0) and drops to zero in both saturated tails, where σ is flat. In backpropagation this bump is the factor g′(z) the incoming gradient is multiplied by. So a sigmoid neuron learns only while it sits in the narrow responsive band around the middle.

ReLU avoids this: on the active side its slope is exactly 1, so gradients pass through without shrinking, and its sparsity (many exact zeros) is often helpful. Its failure mode is the **dead ReLU**: if a large negative bias pushes a unit's z below zero for *every* input, its output and its

gradient are both permanently zero, so it never recovers. The neuron is stuck off forever. Leaky ReLU, ELU, GELU and Swish all fix this by giving the negative side a small nonzero slope, so a sleeping neuron can still feel a gradient and wake up.



The same four activations read as gradient gates: their derivatives. ReLU passes a full slope of 1 while active. Tanh peaks at 1 only near z = 0 and decays fast. Sigmoid's slope is capped at 0.25 and shrinks in both tails. Depth multiplies these factors, which is why slopes below 1 make gradients vanish.

Dead units in practice — diagnosing and reviving them

A dead ReLU is not a rare theoretical curiosity. It is a routine training accident. It usually happens in one of two ways: a **learning rate set too high** takes one enormous step that drives a unit's incoming weights to values where $z < 0$ for the entire dataset, or a **poor initialization / large negative bias** (chapter 10) starts the unit off already switched off. Once every input yields $z < 0$, the output is 0, the local slope is 0, backprop hands it a zero gradient, and the weights never change again. The unit is now permanent dead weight, using up parameters and compute for nothing. On real networks it is common to find 10–40% of ReLU units dead after a careless run.

The symptom is easy to monitor: for a batch, measure the fraction of units whose activation is exactly zero across *all* examples. A rising count of always-zero units is the clear sign. The fixes are equally direct. Lower the learning rate, initialize sensibly (He initialization, chapter 10), or switch to a variant with a nonzero negative slope. **Leaky ReLU** and **ELU** keep a small nonzero slope on the left, so a sleeping unit can still drift back to life. **GELU** and **Swish** go further and

are *self-gated*: they multiply the input z by a smooth gate ($\Phi(z)$ or $\sigma(z)$) that eases from 0 to 1, so a unit fades softly rather than snapping shut. There is always some slope to learn from. This self-gating, more than any range property, is why the smooth hinges dominate very deep and transformer models (chapter 16).

The smooth hinges: GELU and SiLU/Swish, precisely

The two activations that dominate large models earn their place by design, not tuning. Both are **self-gated**: they multiply the input z by a smooth switch that eases from 0 up to 1 as z grows, so a unit fades gently on the left instead of snapping shut like ReLU. The **GELU** (Gaussian Error Linear Unit) uses the standard-normal CDF $\Phi(z)$ as that switch. Differentiating $z\cdot\Phi(z)$ with $\Phi' = \varphi$ (the N(0,1) density) gives a clean two-term slope:

$$\mathrm{GELU}(z) = z\cdot\Phi(z)\,, \quad \mathrm{GELU}'(z) = \Phi(z) + z\cdot\varphi(z)$$

The **probabilistic motivation** is the point. Mask each input with a Bernoulli gate that keeps z with probability $\Phi(z)$, so larger inputs are more likely to survive. GELU is exactly the *expected* output of that stochastic gate, $\mathbb{E}[m\cdot z] = \Phi(z)\cdot z$, a smooth deterministic replacement for data-dependent dropout (chapter 12). This gives transformers three valued properties. It is **smooth** rather than sharply bent at 0, so a deep residual, layer-normalized stack (chapters 11, 16) has a better-conditioned loss surface and cleaner gradients than under ReLU. Its derivative *never touches an exact zero*. It rises slightly above 1 near $z \approx 1$ and dips below 0 for $z < 0$, so no unit freezes and dies the way a ReLU can. These are why the original Transformer, BERT, and the GPT family all place GELU in their feed-forward blocks (chapters 16, 17).

Replacing the Gaussian gate with a logistic one gives the **SiLU** (Sigmoid Linear Unit), which was found independently as **Swish**. It costs a single sigmoid instead of an erf, and follows GELU closely:

$$\mathrm{SiLU}(z) = z\cdot\sigma(z)\,, \quad \mathrm{SiLU}'(z) = \sigma(z) + z\cdot\sigma(z)(1-\sigma(z)) = \sigma(z)\cdot(1 + z(1-\sigma(z)))$$

$\Phi(z)$ standard-normal cumulative distribution, the probability a N(0,1) draw falls below z

$\varphi(z)$ standard-normal density, $\varphi(z) = e^{-z^2/2}/\sqrt{(2\pi)}$, and $\varphi = \Phi'$

The two are very similar. Each passes through the origin. Each is mildly **non-monotonic**, dipping to a shallow minimum at slightly negative z (near −0.75 for GELU, −1.28 for SiLU), and each returns a full gradient for large positive z. They are nearly interchangeable in practice, and frameworks default to a fast *tanh approximation* of GELU, $\mathrm{GELU}(z) \approx \tfrac{1}{2}z(1 + \tanh[\sqrt{(2/\pi)}(z + 0.044715\, z^3)])$, accurate to better than 5×10^{-4} everywhere.

Two ideas generalize the self-gate and are now used in the largest models. **Maxout** (Goodfellow et al., 2013) uses no fixed shape: a unit outputs $\max_k(z_k)$ over k affine pieces $z_k = xW_k + b_k$, so two pieces can *learn* ReLU, Leaky ReLU, or an absolute-value bend (any convex piecewise-linear function) at k× the parameters. **Gated linear units** (GLU) split a layer and let one half gate the other: $\mathrm{GLU}(x) = (xW + b) \odot \sigma(xV + c)$. Replace the sigmoid gate with a GELU or SiLU gate and you get **GEGLU** and **SwiGLU**, the feed-forward block in LLaMA and PaLM (chapter 17), which reliably beats a plain GELU MLP at equal parameter count. Each uses the same idea: let the data decide, smoothly, how far each gate opens.

The table below is the working reference for the hidden-layer choices. It gives the formula, output range, the derivative backprop multiplies by, whether outputs are zero-centered, rough compute cost (a raw compare is far cheaper than an exp, which is cheaper than an erf), and where each is the natural choice:

ACTIVATION	G(Z)	RANGE	G′(Z)	ZERO-CENTERED	COST	TYPICAL USE
Sigmoid	$1/(1+e^{-z})$	(0, 1)	$\sigma(1-\sigma) \le 0.25$	no	exp	binary output, gates
Tanh	$2\sigma(2z)-1$	(−1, 1)	$1 - \tanh^2$	yes	exp	RNN state, zero-centered hidden
ReLU	$\max(0, z)$	[0, ∞)	1 if z>0 else 0	no	compare	default CNN/MLP hidden
Leaky ReLU	$\max(\alpha z, z)$	(−∞, ∞)	1 if z>0 else α	≈ yes	compare	hidden, avoids dead units
ELU	z or $\alpha(e^z-1)$	(−α, ∞)	1 if z>0 else $\alpha\, e^z$	≈ yes	exp on z<0	hidden, robust

ACTIVATION	G(Z)	RANGE	G′(Z)	ZERO-CENTERED	COST	TYPICAL USE
Softplus	$\ln(1+e^z)$	$(0, \infty)$	$\sigma(z)$	no	exp+log	smooth ReLU, positive outputs
GELU	$z\cdot\Phi(z)$	$\approx(-0.17, \infty)$	$\Phi(z) + z\cdot\varphi(z)$	no	erf	transformer feed-forward
SiLU/Swish	$z\cdot\sigma(z)$	$\approx(-0.28, \infty)$	$\sigma(1 + z(1-\sigma))$	no	exp	transformers, vision, gated MLPs

Softmax: the output-layer activation

For multiclass classification the output layer uses **softmax**, which turns a vector of raw scores (logits) into a probability distribution that sums to one:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\Sigma_j e^{z_j}}$$

Unlike the hidden-layer activations, softmax couples all outputs together: raising one logit lowers the others' probabilities. This is exactly what "pick one class" requires. It pairs with the cross-entropy loss (chapter 6). For a single yes/no output, the sigmoid plays the same role. One practical detail: e^z overflows for large z, so implementations first subtract $\max_j z_j$ from every logit. This shifts nothing, since the ratio is unchanged, but it keeps the exponentials in a safe range. That is why the code below always writes `z - z.max()`.

Because every output shares the same denominator, softmax has a genuinely **coupled** derivative: changing one logit z_j moves *all* the probabilities. Collecting the partials gives its Jacobian, a K×K matrix,

$$\partial p_i / \partial z_j = p_i(\delta_{ij} - p_j), \quad \text{equivalently} \quad J = \text{diag}(p) - p\,p^T$$

where p = softmax(z) and δ_{ij} is the Kronecker delta (1 if i = j, else 0). The diagonal terms $p_i(1 - p_i)$ are positive, since raising a logit raises its own probability. The off-diagonal terms $-p_i p_j$ are negative, because the fixed budget of 1 must be taken from the other classes. Every row and column of J therefore sums to zero. This mirrors the rule $\Sigma_i\, p_i = 1$.

On its own this dense Jacobian looks costly to backpropagate through, and hard to compute accurately. The solution appears when softmax meets its natural partner, the **cross-entropy loss** (chapter 6): for a one-hot target y and $L = -\Sigma_k\, y_k \ln p_k$, the Jacobian simplifies against the loss gradient into one of the most useful identities in deep learning:

$$\partial L \,/\, \partial z_i = p_i - y_i$$

So the softmax-plus-cross-entropy layer hands backprop (chapter 8) the raw residual "prediction minus target". This is the same clean form the MSE layer produces in the Backpropagation chapter. Frameworks **fuse** the two (PyTorch's `CrossEntropyLoss` takes logits, not probabilities) both for this simple gradient and to avoid ever forming ln(p) or dividing by a vanishing p. The cell below builds the Jacobian explicitly and checks that routing the cross-entropy gradient through it reproduces p − y exactly:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
def softmax(z):
    z = z - z.max(); e = np.exp(z); return e/e.sum()
z = np.array([2.0, 1.0, 0.1])
p = softmax(z)
J = np.diag(p) - np.outer(p, p)      # Jacobian dp/dz = diag(p) - p p^T
y = np.array([0.0, 1.0, 0.0])        # one-hot target: true class = index 1
dL_dp = -y/p                         # gradient of cross-entropy wrt
probabilities
dL_dz = J.T @ dL_dp                  # chain rule through the softmax
print("p          =", np.round(p, 4))
print("row sums(J) =", np.round(J.sum(1), 6))    # each 0: probabilities sum
to 1
print("dL/dz via J =", np.round(dL_dz, 4))
print("p - y      =", np.round(p - y, 4))       # the famous fused result
matches
```

Temperature — sharpening and softening softmax

Divide the logits by a positive **temperature** T before the softmax and you get a control for the sharpness of the distribution:

$$\text{softmax}(z \,/\, T)_i = \frac{e^{z_i/T}}{\Sigma_j \, e^{z_j/T}}$$

At T = 1 this is ordinary softmax. As $\mathbf{T \to 0}$ the largest logit dominates completely and the output approaches a one-hot argmax, a hard and confident decision. As $\mathbf{T \to \infty}$ the divided logits all shrink toward zero and the distribution flattens toward uniform (1/K each). Temperature therefore trades decisiveness against uncertainty without touching the network's weights. It appears widely in practice. Language models sample the next token at a chosen temperature to tune "creativity" versus determinism. *Knowledge distillation* (Hinton et al., 2015) trains a small student on a large teacher's high-temperature "soft targets", whose spread across the non-winning classes carries the teacher's dark knowledge. Temperature scaling is the standard after-training fix for over-confident classifiers. The runnable cell shows this control in action:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
def softmax(z, T=1.0):
    z = z/T
    z = z - z.max()                 # subtract max for numerical stability
    e = np.exp(z)
    return e/e.sum()

logits = np.array([2.0, 1.0, 0.1])  # same scores, different temperatures
print("temperature   softmax probabilities        peak")
for T in [0.5, 1.0, 2.0, 5.0]:
    p = softmax(logits, T)
    print(f"   T = {T:<4}  {np.round(p,3)}   {p.max():.3f}")
```

The eight activations in code

Here is the whole family in a few lines of NumPy: each function and, just as importantly, each derivative. Run the first cell to reproduce the forward table at z = −2, 0, +2 (its first three columns match the worked example below):

Python · NumPy ▶ runnable in the app

```
import numpy as np
from math import erf
erf_v = np.vectorize(erf)                    # standard-normal helper for
GELU

def sigmoid(z):  return 1/(1+np.exp(-z))
def tanh(z):     return np.tanh(z)
def relu(z):     return np.maximum(0, z)
def leaky(z, a=0.01): return np.where(z>0, z, a*z)
def elu(z, a=1.0):    return np.where(z>0, z, a*(np.exp(z)-1))
def softplus(z): return np.log1p(np.exp(z))
def gelu(z):     return z*0.5*(1+erf_v(z/np.sqrt(2)))   # exact GELU =
z*Phi(z)
def swish(z):    return z*sigmoid(z)

z = np.array([-2.0, 0.0, 2.0])
acts = [("sigmoid",sigmoid),("tanh",tanh),("relu",relu),("leaky",leaky),
        ("elu",elu),("softplus",softplus),("gelu",gelu),("swish",swish)]
print("            z = -2      0     +2")
for name, f in acts:
    v = np.round(f(z), 3)
    print(f"{name:>9}: {v[0]:7.3f} {v[1]:6.3f} {v[2]:6.3f}")
```

The derivatives are what backprop actually multiplies by. The second cell prints the slope of each activation at the same three points. Read column z = 0 as "best-case learning signal" and the z = ±2 columns as "already fading" for the saturating functions:

Python · NumPy ▶ runnable in the app

```
import numpy as np
from math import erf
erf_v = np.vectorize(erf)

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

```
z = np.array([-2.0, 0.0, 2.0])
s = sigmoid(z)
phi = np.exp(-z**2/2)/np.sqrt(2*np.pi)         # standard-normal pdf
Phi = 0.5*(1+erf_v(z/np.sqrt(2)))              # standard-normal cdf

d = [("sigmoid'", s*(1-s)),          ("tanh'", 1-np.tanh(z)**2),
     ("relu'",    (z>0).astype(float)), ("leaky'", np.where(z>0, 1.0,
0.01)),
     ("elu'",     np.where(z>0, 1.0, np.exp(z))), ("softplus'", s),
     ("gelu'",    Phi + z*phi),       ("swish'", s + z*s*(1-s))]
print("              z = -2      0     +2")
for name, g in d:
    g = np.round(g, 3)
    print(f"{name:>10}: {g[0]:7.3f} {g[1]:6.3f} {g[2]:6.3f}")
```

The same functions are included in every deep-learning framework. In PyTorch they are found in `torch.nn.functional` (imported as `F`), so you almost never write them by hand. This listing is for reading, not running in-app:

PyTorch copy → Colab

```
import torch
import torch.nn.functional as F

z = torch.tensor([-2.0, 0.0, 2.0])
print(torch.sigmoid(z))          # or z.sigmoid()
print(torch.tanh(z))
print(F.relu(z))
print(F.leaky_relu(z, 0.01))
print(F.elu(z, alpha=1.0))
print(F.softplus(z))
print(F.gelu(z))                 # exact by default; approximate='tanh' also
available
print(F.silu(z))                 # SiLU is the same function as Swish: z *
sigmoid(z)

logits = torch.tensor([2.0, 1.0, 0.1])
print(F.softmax(logits, dim=0))          # temperature: F.softmax(logits /
T, dim=0)
```

In this app's engine

The **Activation explorer** lab plots any of these eight functions alongside its derivative, and lets you drag z to read both values live. You can watch g′ drop to zero out in a sigmoid's tail. In the Playground, the hidden-layer activation is a dropdown. Switch a saturating tanh to ReLU on the two-spirals dataset and watch training speed up as the vanishing gradient goes away.

Worked example — four activations and their slopes at z = 1

Take one pre-activation z = 1 and evaluate the two classic saturating units, the ReLU, and the modern GELU, each with the derivative backprop would multiply by (rounded to 4 decimals):

ACTIVATION	G(1)	G′(1)	READING
Sigmoid	0.7311	0.1966	already well below its 0.25 peak
Tanh	0.7616	0.4200	steeper, but fading
ReLU	1.0000	1.0000	full gradient, active side
GELU	0.8413	1.0833	slope > 1, mildly amplifying

1. Sigmoid: $\sigma(1) = 1/(1 + e^{-1}) = 1/1.3679 =$ **0.7311**; $\sigma' = \sigma(1 - \sigma) = 0.7311 \cdot 0.2689 =$ **0.1966**.
2. Tanh: $\tanh(1) =$ **0.7616**; $\tanh' = 1 - 0.7616^2 = 1 - 0.5800 =$ **0.4200**.
3. ReLU: since $1 > 0$, $\mathrm{ReLU}(1) =$ **1** and $\mathrm{ReLU}'(1) =$ **1**, with no shrinkage at all.
4. GELU: with $\Phi(1) = 0.8413$ and $\varphi(1) = e^{-½}/\sqrt{(2\pi)} = 0.2420$, $\mathrm{GELU}(1) = 1 \cdot \Phi(1) =$ **0.8413** and $\mathrm{GELU}'(1) = \Phi(1) + 1 \cdot \varphi(1) = 0.8413 + 0.2420 =$ **1.0833**.

What the numbers mean: at the same input the two saturating units already pass less than half the gradient ($\sigma' = 0.20$, $\tanh' = 0.42$), while ReLU passes all of it and GELU passes slightly *more* than all of it, since its derivative peaks just above 1 near $z \approx 1$. Stacked over dozens of layers, the gap between multiplying by ~0.2 and by ~1 is the whole vanishing-gradient story (chapter 10) compressed into a single column.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
from math import erf
z = 1.0
sig = 1/(1+np.exp(-z))
Phi = 0.5*(1+erf(z/np.sqrt(2)))                # standard-normal CDF
Phi(1)=0.8413
phi = np.exp(-z**2/2)/np.sqrt(2*np.pi)         # standard-normal pdf
phi(1)=0.2420
rows = [("sigmoid", sig,         sig*(1-sig)),
        ("tanh",    np.tanh(z),  1-np.tanh(z)**2),
        ("relu",    max(0.0, z), 1.0),
        ("gelu",    z*Phi,       Phi + z*phi)]
print("activation   g(1)     g'(1)")
for name, g, dg in rows:
    print(f"{name:>9}  {g:6.4f}  {dg:6.4f}")
# expected: sigmoid 0.7311/0.1966  tanh 0.7616/0.4200  relu 1.0000/1.0000
gelu 0.8413/1.0833
```

Worked example — values, slopes, and a vanishing product

Evaluate the three classic activations and their derivatives at z = −2, 0, +2 (rounded to 3 decimals):

Z	Σ	Σ′	TANH	TANH′	RELU	RELU′
−2	0.119	0.105	−0.964	0.071	0	0
0	0.500	0.250	0.000	1.000	0	0
+2	0.881	0.105	0.964	0.071	2	1

Read the slopes as gradient gates. Sigmoid peaks at σ′ = 0.25 (at z = 0) and has already dropped to 0.105 by z = ±2. Tanh peaks at 1 but decays even faster in its tails. ReLU is either fully open (1) or fully shut (0). Now chain 10 sigmoid layers and ask what fraction of the gradient survives, in the *best* case where every neuron sits at its most responsive point:

1. Best-case local slope per sigmoid layer: $\sigma'_{max} = 0.25$.
2. Backprop multiplies one such factor per layer, so ten layers give a factor 0.25^{10}.
3. $0.25^{10} = (1/4)^{10} = 1 / 1\,048\,576 \approx \mathbf{9.5 \times 10^{-7}}$.
4. **Meaning:** even under ideal conditions the first layer receives a gradient roughly ten-million times weaker than the last, so it barely moves. Away from z = 0 it is far worse. This is the vanishing gradient, and it is precisely why ReLU (slope 1) replaced sigmoid in deep hidden layers (chapter 10).

Worked example — a dead ReLU and its rescue

Suppose a unit's incoming weights and bias have drifted so that its pre-activation sits at z = −3 for *every* training example. What does each activation do with it, and can the unit ever recover?

ACTIVATION	OUTPUT AT Z=−3	SLOPE G′(−3)	VERDICT
ReLU	0	0.000	dead: zero gradient, never recovers
Leaky (α=0.01)	−0.03	0.010	alive: small signal drifts z up
ELU (α=1)	−0.95	0.050	alive: 5× Leaky's wake-up slope
GELU	−0.004	−0.012	alive: small reversed slope

1. With ReLU the backprop multiplier $g'(-3) = 0$, so the error signal reaching this unit's weights is exactly zero. There is no update on this step or any future step. The unit is frozen off.
2. With Leaky ReLU the multiplier is 0.01: tiny, but nonzero. Every step now changes the bias, and once enough changes lift z above 0 the unit snaps back to full slope 1 and rejoins learning.

3. ELU's slope at $z = -3$ is $e^{-3} = 0.050$, five times Leaky's, so it revives dead units faster. GELU keeps a small slope of its own sign. **The negative-side slope is literally the wake-up signal.** Its size sets how quickly a sleeping neuron can return.

Assumptions & limitations

No activation is universally best. The "ReLU by default" rule is an empirical starting point, not a theorem, and the smooth variants (GELU, Swish) often win in very deep or transformer models (chapter 16). Ranges and saturation describe behavior on the z-values the network actually visits, which depends on initialization (chapter 10) and input scaling (chapter 11). A well-scaled network keeps z near the responsive middle. The ELU/Leaky slopes and GELU's exact form carry hyperparameters (α, the tanh-approximation) that this summary skips over. Finally, softmax and sigmoid outputs are calibrated probabilities only after proper training with a matching loss. A confident-looking 0.99 is not automatically trustworthy.

Why & when this matters

Activation choice is one of the first things to check when a network will not train: flat loss with sigmoids often means vanishing gradients, and dead units often mean a too-aggressive ReLU or learning rate. As a working recipe, use ReLU (or Leaky/GELU) in hidden layers, tanh where a zero-centered signal helps, sigmoid for a single probability output, and softmax for multiclass. Every gate, memory cell, and attention weight in later chapters is built from these same curves. Understanding their slopes means understanding where a deep network's learning signal lives or dies.

In the real world

The activation you pick is built into the biggest deployed models. The original Transformer and BERT use GELU in their feed-forward blocks, and Google's knowledge-distillation work compressed BERT into small, fast students by training on high-*temperature* softmax targets (the exact control from the temperature cell above), so phones can run models that were first trained on TPU clusters. Meta's LLaMA family and Google's PaLM replaced GELU with SiLU/Swish-gated units (SwiGLU), a change worth a measurable drop in perplexity at no extra parameters. In computer vision, Google's EfficientNet is built on Swish throughout. Whenever you use autocomplete, a chatbot, or on-device translation, a specific choice from this chapter's eight curves is running billions of times per query.

Exercises

1. Show that $\tanh(z) = 2\sigma(2z) - 1$, so tanh really is a stretched, re-centered sigmoid. MATH

2. Sigmoid's slope never exceeds 0.25. What best-case fraction of the gradient survives 15 stacked sigmoid layers? MATH

3. Confirm numerically that $\text{softplus}'(z) = \sigma(z)$ by comparing a finite difference of softplus at $z = 0.5$ to $\sigma(0.5)$. CODE

4. Write softmax-with-temperature and verify its two limits: a very cold T approaches the argmax (one-hot), a very hot T approaches the uniform distribution. CODE

5. Why does ReLU not saturate on its positive side while sigmoid saturates on both, and what is the consequence for deep networks? CONCEPT

6. During training you notice a hidden layer outputs exactly 0 for every unit on every example in a batch. Which activation is it likely using, and what is your diagnosis and fix? CONCEPT

7. Compute Swish(z) = $z \cdot \sigma(z)$ at $z = -2, 0, +2$, and its derivative at $z = 0$. MATH

8. When might you deliberately prefer tanh over ReLU in a hidden layer? CONCEPT

9. Derive $\mathrm{GELU}'(z) = \Phi(z) + z \cdot \varphi(z)$ from $\mathrm{GELU}(z) = z \cdot \Phi(z)$, evaluate it at $z = 0$, and explain how a bounded-slope sigmoid can be beaten by a derivative that exceeds 1. MATH

10. Starting from the cross-entropy loss $L = -\Sigma_k\, y_k \ln p_k$ with $p = \mathrm{softmax}(z)$ and a one-hot target y, use the softmax Jacobian $\partial p_k / \partial z_i = p_k(\delta_{ki} - p_i)$ to prove the fused gradient $\partial L / \partial z_i = p_i - y_i$. MATH

11. By how much does the fast tanh approximation of GELU differ from the exact erf form? Compute the maximum absolute error over $z \in [-4, 4]$. CODE

CHAPTER 5 · FOUNDATIONS

Features, embeddings & what a network learns

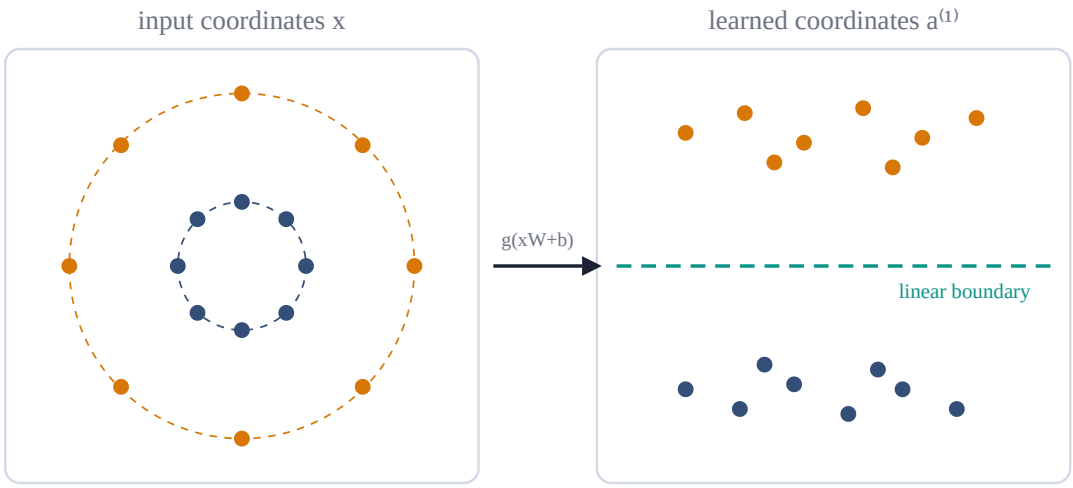
> *A linear model is only as good as the numbers you feed it. Give it the raw pixels of a photograph and it cannot tell a cat from a car. Give it the right few hundred numbers (a whisker here, a wheel there) and the same simple model succeeds. Those numbers are a* ***representation****, and for decades finding good ones was the job of human engineers. The key idea of deep learning is that a network can learn its own representations. Each hidden layer describes the data again in new coordinates that make the next layer's job easier, until the final layer faces a problem so untangled it is nearly linear. This chapter is about what those learned coordinates are, why distributed codes and embeddings are so powerful, and how to read the geometry of the space a network builds.*

Fundamental concept

A **feature** is any number that describes an example: a raw measurement (a pixel's brightness), a hand-designed summary (an image's average redness), or (the subject of this chapter) a quantity the network invents for itself. The classical pipeline combined two skills: a domain expert first designed features by hand (SIFT and HOG for vision, MFCC for speech, TF-IDF for text), and only then did a simple classifier (a perceptron, chapter 2) draw a boundary in that space. The features did the hard work. The classifier was almost an afterthought. Deep learning merges the first skill into the second: *learn the features and the classifier together*, end to end, by gradient descent. This is why the field is called **representation learning**.

Representation matters so much because a problem's difficulty depends entirely on the coordinates you view it in. Take two concentric rings, one class inside the other. In the raw (x, y) plane no straight line separates them. The boundary is a circle, $x^2 + y^2 = r^2$. Switch to polar coordinates (radius r, angle θ) and the same data splits at a single threshold: $r < r_0$ inside, $r > r_0$ outside. Nothing about the points changed, only the language describing them. Yet a hopeless

nonlinear problem became a simple linear one. A hidden layer is a machine for discovering such a change of language from data, without being told that "radius" is the useful quantity. That is the whole idea.



A hidden layer as a learned change of coordinates. On the left, two classes are mixed together as concentric rings. No straight line separates them. The layer's map g(xW + b) stretches and folds the plane so that, in the new coordinates on the right, the same points fall into two clusters that a single hyperplane can split. The network discovers this "untangling" from data. Nobody tells it to compute a radius.

Governing equations

The forward-pass equation is the one from chapter 3, but we now read it geometrically. Layer ℓ takes the previous representation $a^{(l-1)}$ and returns a new one:

$$a^{(l)} = g(a^{(l-1)} W^{(l)} + b^{(l)}) = \varphi_\ell(a^{(l-1)})$$

Each φ_ℓ is a **change of coordinates**: an affine map $W^{(l)}$, $b^{(l)}$ that rotates, scales and shifts the space, followed by the nonlinearity g that bends it (chapter 4). Stacking layers composes these maps, carrying the raw input through a sequence of ever-more-useful descriptions before a final linear readout:

$$x = a^{(0)} \mapsto a^{(1)} \mapsto \cdots \mapsto a^{(L-1)} \ \text{(the learned representation)} \ \mapsto \hat{y}$$

When the input is **categorical** (a word, a product ID, a user), there is no meaningful raw vector to start from, so the first "layer" is a table lookup. Give every one of V categories its own learned row in an **embedding matrix** E of shape V×d. Selecting category i is a matrix multiply by its one-hot row $\mathbf{e}_i$:

$$\text{emb}(i) = \mathbf{e}_i\, E = E_{i,:} \quad \text{(the i-th row of E)}$$

Finally, the standard tool for comparing two representations is the **cosine similarity**. It is the cosine of the angle between them, and it measures alignment of direction while ignoring length:

$$\cos(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\,\|\mathbf{v}\|} = \frac{\Sigma_i\, u_i\, v_i}{\sqrt{(\Sigma_i\, u_i^{\,2})} \cdot \sqrt{(\Sigma_i\, v_i^{\,2})}}$$

$a^{(l)}$ — the representation (feature vector) at layer ℓ — what the network has re-encoded the input into. Here $a^{(0)} = x$, and $a^{(L-1)}$ is the final learned representation feeding the output

φ_ℓ — the coordinate change performed by layer ℓ, $\varphi_\ell(a) = g(aW^{(l)} + b^{(l)})$. The whole network is the composition $\varphi_{L-1} \circ \cdots \circ \varphi_1$

E — embedding matrix, shape V×d — one learned row per category. It is trained by backprop like any other weights

V, d — vocabulary size (number of categories) and embedding dimension, with $d \ll V$ so the code is dense and compressed

$\mathbf{e_i}$ — one-hot row vector for category i — a 1 in position i and zeros elsewhere (length V)

cos(u, v) — cosine similarity: +1 when the vectors point the same way, 0 when orthogonal (unrelated), −1 when opposite

Layers as a learned change of coordinates

Chapter 3 showed that without a nonlinearity every stack of layers collapses to a single linear map. Then chapter 4 named that nonlinearity the valve that bends the signal. From the viewpoint of representation, these become one geometric story. A linear map can only rotate, scale and shear space, never *fold* it, so mixed classes stay mixed no matter how many linear layers you apply. The activation provides the fold: ReLU bends space along a hyperplane, tanh warps it smoothly, and each bend lets the next affine map pull tangled points onto opposite sides of a plane. Over a few layers this **untangling** straightens curved boundaries into flat ones.

Christopher Olah's image fits: the network kneads a sheet of dough, stretching and folding until two linked ribbons come apart and a single straight cut separates them. This is the transformation drawn above.

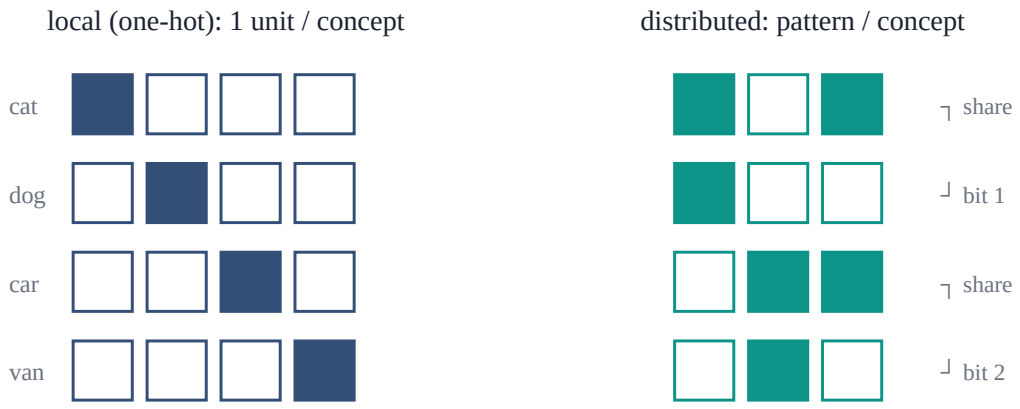
Why should untangling ever be possible? Because real data is not spread uniformly through its space. A 100×100 grayscale image is a point in $\mathbb{R}^{10000}$, yet almost every point there is meaningless static. Natural images lie on an extremely thin, curved sheet within it. This is the **manifold hypothesis**: data of a given kind concentrates near a low-dimensional manifold embedded in the ambient space. Handwritten digits, faces and grammatical sentences each trace out their own surface. Classification is then deciding which class-manifold a point lies nearest, and the network's job is to bend space until those manifolds are flat, separated slabs that a linear layer can slice apart. The concentric rings are a two-dimensional picture of this: each class is a circle (a 1-D manifold), and the useful coordinate (radius) is what the hidden layer learns to expose. The code below runs it for real: a linear classifier stuck at 52% jumps to 100% once that coordinate is supplied.

This also lets us read chapter 3's depth-versus-width argument in terms of representation: each layer recombines the features just below it rather than building from scratch, so a feature learned low in the network is reused by everything above. This is composition, not enumeration. That is why depth expresses far more structure per parameter than width.

Distributed vs local representations

How a representation *encodes* concepts matters as much as which it captures. The naive scheme is a **local** code: one unit per concept, so "cat" is unit 7 firing alone and "dog" is unit 12. This is the one-hot vector, the oversimplified "grandmother cell." Local codes are simple and easy to interpret, but wasteful and blind to meaning: n units name only n concepts. And since every pair of one-hot vectors is orthogonal and equidistant, the code calls cat and dog exactly as dissimilar as cat and car. It cannot say two things are alike, so nothing learned about one concept transfers to a related one.

A **distributed** code fixes both problems at once. A concept is now a *pattern* of activity across many units, and each unit takes part in many concepts: "cat" might be (furry, four-legged, domestic, small) while "dog" shares three of those four bits. There are two benefits. First, capacity is exponential: n binary units support up to 2^n patterns rather than n, so 100 units distinguish not 100 but $2^{100} \approx 10^{30}$ concepts. Second, and more important, *similarity becomes expressible*: concepts that share features get nearby codes, so a model generalizes from cat to dog along what they share. Networks learn such codes by default: each hidden unit responds to many inputs, each input lights many units. This is the mechanism behind everything else in the chapter, and why an embedding can place "king" near "queen."

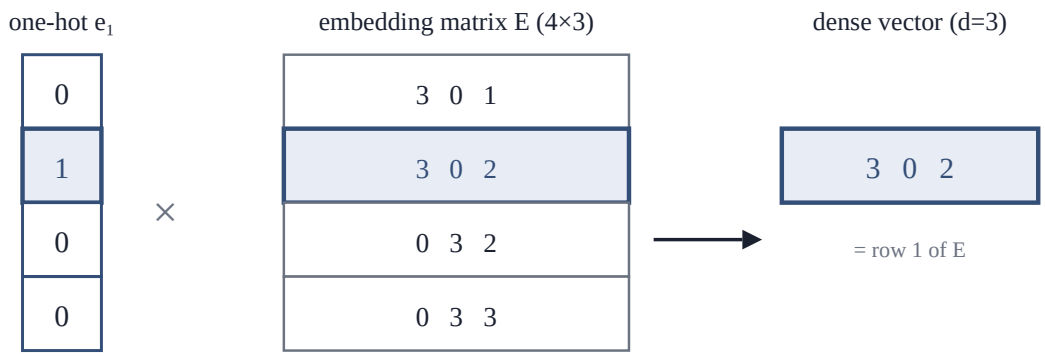


Two ways to code four concepts. The local code (left) uses one unit per concept and can never say that two concepts are alike. Every column is orthogonal. The distributed code (right) represents each concept as a pattern, so k units encode up to 2^k concepts and related items (cat/dog, car/van) share sub-patterns. Distributed codes are both exponentially more compact and able to express similarity. These are the two properties that make learned representations generalize.

Embeddings: from one-hot to dense vectors

Categorical data (words, tokens, product SKUs, user IDs) arrives with no natural geometry. The default encoding is the one-hot vector: a length-V axis-aligned spike, orthogonal to every other category. It is a local code with all the faults of a local code. For a 50,000-word vocabulary each word is a 50,000-dimensional vector of almost all zeros, and no two words are any more alike than any others. An **embedding** replaces it with a learned, dense, low-dimensional vector per category. The matrix E is V×d with d perhaps 64 or 300. Row i is category i's vector, an ordinary block of trainable parameters.

The lookup is where the local and distributed worlds meet. Multiplying the one-hot row $\mathbf{e}_i$ by E selects row i exactly. A one-hot times a matrix *is* a row selection, so an embedding layer is just a linear layer whose input happens to be one-hot. So "lookup" and "matrix multiply" are the same operation. Frameworks implement it as a cheap gather rather than a multiply by a mostly-zero vector, but the two are identical. This is why the layer is differentiable and trains by backprop (chapter 8). The gradient reaches only the rows used in that step, changing each one to serve the task better.



e_1 E = row 1 of E — a one-hot times a matrix is a row selection

An embedding lookup is a matrix multiply. The one-hot vector for category 1 (here "dog") picks out row 1 of the embedding matrix E. Every other row is multiplied by zero. Because it is a linear operation, the dense output vector is differentiable in E, so backpropagation trains the table end to end. Only the rows used on a step receive gradient.

Two forces shape the geometry that results. The bottleneck $d \ll V$ prevents the model from giving every category an independent direction, so it is *forced* to reuse dimensions and place items that behave similarly near one another. And because the embedding is trained end to end against a downstream objective, the meaning of "similar" it discovers is whatever that objective rewards: words that predict the same neighbors, products bought by the same customers, users who click the same items. The embedding is not a fixed encoding but a learned map whose distances mean something task-specific.

The geometry of word embeddings

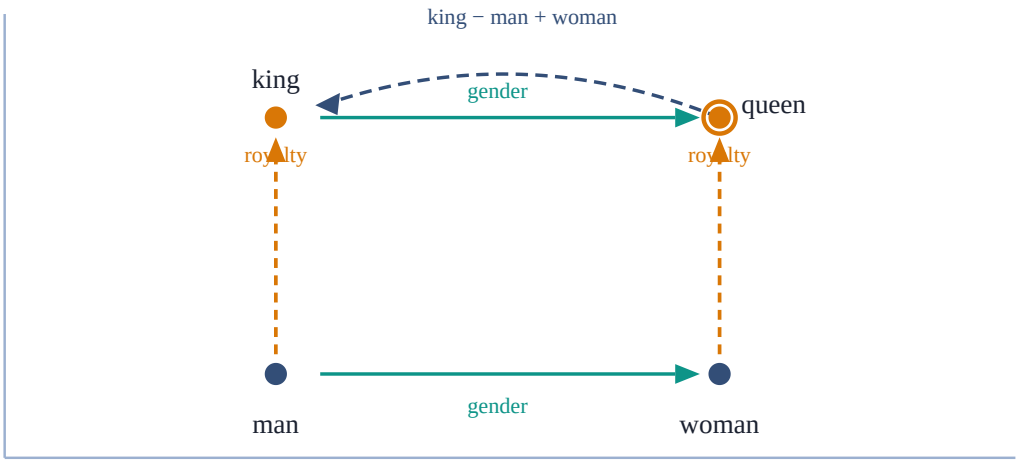
The most striking embeddings are for words. The method that made them popular, **word2vec** (Mikolov et al., 2013), is based on a linguist's slogan: "you shall know a word by the company it keeps" (Firth). Its skip-gram version slides a window over a large corpus and, for each center word, trains the model to predict the words around it. Each word carries a vector. The probability that word c appears in the context of word w is a softmax over their dot product,

$$p(c \mid w) = \frac{\exp(\mathbf{v}_c \cdot \mathbf{v}_w)}{\Sigma_{c'} \exp(\mathbf{v}_{c'} \cdot \mathbf{v}_w)}$$

and gradient descent pushes words that share contexts to point the same way. There is no supervision beyond raw text. The geometry is a side effect of prediction. Two regularities then make embeddings famous. First, **cosine similarity** reflects meaning: a word's nearest neighbors by cosine are its synonyms and related words. We use cosine rather than Euclidean distance because it compares *direction*, ignoring the length differences that come from word frequency. Second, and more surprising, some relations appear as roughly constant **offset vectors**: the step "man" → "king" (the "royalty" direction) is about the same vector as "woman" → "queen," giving the famous analogy arithmetic:

$$\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}$$

It is worth being honest here, because this demonstration is often overstated. The parallelogram is *approximate*: the right-hand side lands near queen, not on it. And the standard evaluation only counts it as a success after removing the three query words from the candidate set. Without that removal, the nearest vector is very often just "king" again. The effect is strongest for frequent words and a few relation types (gender, capital-of-country, verb tense), and weak elsewhere. It depends on the corpus, and it inherits the corpus's biases. So the geometry that returns "queen" will, from "doctor − man + woman," also return "nurse" (Bolukbasi et al., 2016). It is a genuine, useful regularity, not a law of thought.



Analogy as a parallelogram. When a relationship is captured by a roughly constant offset vector, "king − man + woman" walks the royalty edge down, the gender edge across, and lands near "queen" (circled). The picture is idealized: real embeddings satisfy it only approximately, and scoring the analogy requires removing the query words. Still, that consistent relational structure, not any single interpretable axis, is what makes embedding arithmetic possible.

Feature hierarchies and transfer

The untangling story has a spatial counterpart that is even easier to see: the feature hierarchy inside a convolutional network (chapters 13–14). Train a CNN on natural images and look at what each layer responds to, as Zeiler and Fergus did in 2014, and a clear progression appears. The first convolutional layer learns oriented **edges** and color blobs. These are the same Gabor-like filters the visual cortex uses. The next layer combines edges into **textures** and simple contours. The one after combines these into **parts**: an eye, a wheel, a corner of text. The deepest layers combine these into whole **objects**: faces, cars, dogs. Each level is literally built from the level below by the same "recombine the features underneath" mechanism as the coordinate-change view. Here, though, the features are spatial and shared across the image. The hierarchy is the manifold-untangling story spread out over depth: pixels near the raw manifold at the bottom, disentangled object identity near the top.

This hierarchy is why representations **transfer** (chapter 22). Early features (edges, textures) are general, useful for almost any visual task rather than only the one trained on. So a network pretrained on ImageNet can be reused: freeze the early layers, retrain only the task-specific head, and a model that would need millions of labeled examples from scratch adapts to a small

new dataset with a few thousand. The same logic applies to NLP, where a language model's learned representations (chapters 16–17) transfer downstream. It also motivates learning representations *without* labels: autoencoders that reconstruct (chapter 19) and self-supervised methods that pull together two augmented views of one example (chapter 20). In each case, the representation is the product. The original task was only a means to reach it.

Failure modes: collapse, anisotropy, polysemy

Learned representations fail in a few typical ways, and recognizing them is part of using them well. The most dramatic is **representation collapse**: the network maps every input to nearly the same vector, throwing away the very information it was supposed to preserve. Collapse is a constant danger in self-supervised learning. There, an objective that only rewards making two views of an example agree has an easy cheat: map everything to a constant. This satisfies the loss perfectly while learning nothing. Contrastive negatives, stop-gradients, and regularizers exist largely to prevent it. The sign of this is an embedding matrix whose rows have almost no variance, so downstream models cannot tell inputs apart.

A more subtle problem is **anisotropy**. Ideally embeddings spread out to fill the space, so unrelated items are near-orthogonal and cosine can tell them apart. In practice the embeddings of large language models occupy a narrow cone: the average cosine between two random words can be 0.5 or higher instead of near 0 (Gao et al.'s "representation degeneration"; Ethayarajh, 2019). When everything is somewhat similar to everything, cosine loses resolution. This is why practitioners sometimes whiten or re-center embeddings first. Finally, static embeddings suffer from **polysemy**: a word with several senses gets one vector that must average them. "Bank" is stuck between the river cluster and the money cluster, close to neither, with no room for context. This is exactly what contextual embeddings from transformers (chapter 16) fix, producing a different vector for each occurrence from the sentence around it.

Representations in code

The first cell builds the embedding matrix of the worked example, shows that a one-hot times E is a row selection, and computes the full cosine-similarity matrix over four word vectors. Press **Run**: "dog" is closest to "cat," and "van" is closest to "car," just as the numbers below predict.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.random.seed(0)

# 4-word vocabulary, 3-dimensional embedding matrix E (rows are word vectors)
words = ["cat", "dog", "car", "van"]
E = np.array([[3., 0., 1.],      # cat
              [3., 0., 2.],      # dog
              [0., 3., 2.],      # car
              [0., 3., 3.]])     # van

# a one-hot vector times E selects a row -- lookup IS a matrix multiply
onehot_dog = np.array([0., 1., 0., 0.])          # index 1 = "dog"
print("one-hot . E =", onehot_dog @ E, " == row 1 =", E[1])

# cosine-similarity matrix over all four word vectors
unit = E / np.linalg.norm(E, axis=1, keepdims=True)
C = unit @ unit.T                                 # 4x4 cosine similarities
print("cosine similarity matrix:")
print(np.round(C, 3))

# which word is closest to "cat" (ignoring itself)?
i = words.index("cat")
sims = C[i].copy(); sims[i] = -np.inf
print("closest to cat:", words[int(np.argmax(sims))],
      " cos =", round(float(sims.max()), 3))   # expected: dog  cos = 0.965
```

The second cell makes the coordinate-change argument concrete. It draws two concentric rings (a classic not-linearly-separable dataset) and asks a linear classifier to split them, first in the raw (x, y) plane and then after a single fixed nonlinear feature map that appends the radius-squared

feature $r^2 = x^2 + y^2$. A network would *learn* such a coordinate. Here we supply it by hand to isolate the effect. Raw accuracy is at chance level. The lifted representation is perfectly separable.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.random.seed(0)

# concentric rings: inner ring = class -1, outer ring = class +1
def ring(n, r):
    t = np.random.uniform(0, 2*np.pi, n)
    rad = r + 0.12*np.random.randn(n)
    return np.c_[rad*np.cos(t), rad*np.sin(t)]

X = np.vstack([ring(200, 1.0), ring(200, 3.0)])
y = np.r_[-np.ones(200), np.ones(200)]            # labels -1 / +1

def lin_acc(F):                                    # least-squares linear classifier
    Fb = np.c_[F, np.ones(len(F))]                 # append a bias feature
    w, *_ = np.linalg.lstsq(Fb, y, rcond=None)
    return (np.sign(Fb @ w) == y).mean()

raw    = lin_acc(X)                                # raw (x, y): not separable
phi    = np.c_[X, (X**2).sum(1)]                   # add r^2 = x^2 + y^2
lifted = lin_acc(phi)                              # now linearly separable
print("linear accuracy, raw (x,y)   =", round(float(raw), 3))      # ~0.52 (chance)
print("linear accuracy, lifted +r^2 =", round(float(lifted), 3))   # 1.0

# the lift makes the classes occupy disjoint r^2 intervals -> a threshold separates
r2 = (X**2).sum(1)
print("max r^2 inner =", round(float(r2[y < 0].max()), 2),
      "<  min r^2 outer =", round(float(r2[y > 0].min()), 2))
```

In PyTorch the embedding table is a built-in layer, `nn.Embedding`, and the lookup-equals-matmul identity holds exactly. This listing is for reading (copy it into Colab). It sets up the chapter's matrix, does a batched lookup, and shows that indexing and the one-hot multiply

agree:

PyTorch copy → Colab

```
import torch, torch.nn as nn
import torch.nn.functional as F

emb = nn.Embedding(num_embeddings=4, embedding_dim=3)      # a learnable V x
d table
with torch.no_grad():                                      # plant the
chapter's E
    emb.weight.copy_(torch.tensor([[3., 0., 1.],
                                   [3., 0., 2.],
                                   [0., 3., 2.],
                                   [0., 3., 3.]]))

idx = torch.tensor([1, 0, 2])                              # dog, cat, car
gathered = emb(idx)                                        # (3, 3)
differentiable lookup
onehot   = F.one_hot(idx, num_classes=4).float()           # (3, 4)
print(torch.allclose(gathered, onehot @ emb.weight))       # True: gather ==
matmul

cos = F.cosine_similarity(gathered[:, None], gathered[None, :], dim=-1)
print(cos)                    # cat-dog high, car far; only used rows get
gradients
```

Worked example — an embedding lookup and three cosine similarities

Take the 4×3 embedding matrix E with rows cat = [3, 0, 1], dog = [3, 0, 2], car = [0, 3, 2], van = [0, 3, 3]. First look up "dog," then measure which of cat, dog, car point most nearly the same way.

1. **Lookup as a row select.** The one-hot row for index 1 is $\mathbf{e}_1$ = [0, 1, 0, 0]. The product $\mathbf{e}_1$E takes 0·(row 0) + 1·(row 1) + 0·(row 2) + 0·(row 3) = **[3, 0, 2]**, exactly row 1. Only row 1 would receive gradient from a step that used this token.

2. **Norms.** $\|\text{cat}\| = \sqrt{(3^2 + 0^2 + 1^2)} = \sqrt{10} = \mathbf{3.162}$; $\|\text{dog}\| = \sqrt{(9 + 0 + 4)} = \sqrt{13} = \mathbf{3.606}$; $\|\text{car}\| = \sqrt{(0 + 9 + 4)} = \sqrt{13} = \mathbf{3.606}$.
3. **Dot products.** $\text{cat}\cdot\text{dog} = 3\cdot3 + 0\cdot0 + 1\cdot2 = \mathbf{11}$; $\text{cat}\cdot\text{car} = 3\cdot0 + 0\cdot3 + 1\cdot2 = \mathbf{2}$; $\text{dog}\cdot\text{car} = 3\cdot0 + 0\cdot3 + 2\cdot2 = \mathbf{4}$.
4. **Cosines.** $\cos(\text{cat}, \text{dog}) = 11 / (\sqrt{10}\cdot\sqrt{13}) = 11/\sqrt{130} = \mathbf{0.965}$; $\cos(\text{cat}, \text{car}) = 2/\sqrt{130} = \mathbf{0.175}$; $\cos(\text{dog}, \text{car}) = 4/13 = \mathbf{0.308}$.
5. **Closest pair.** cat and dog, at cosine 0.965, are far closer than either is to car (0.175 and 0.308). The two "animal" vectors, both dominated by their large first coordinate, point almost the same way, while "car," whose weight sits on the second coordinate, is nearly orthogonal to cat.

What the numbers mean: cosine measures *direction*, not length, so it shows that cat and dog play the same role even though their magnitudes differ (3.162 vs 3.606). The first coordinate behaves like an "animal" axis and the second like a "vehicle" axis. Similarity is high exactly when two words have their weight on the same axis. This is the same computation a search engine or recommender runs to rank millions of items by relatedness, only with d in the hundreds and V in the millions.

Assumptions & limitations

A learned representation is only ever as good as the objective and data that shaped it. Features optimized for one task can be useless for another. And a representation trained on a biased corpus encodes those biases as geometry: "doctor − man + woman ≈ nurse" comes from the same arithmetic as the king/queen analogy. The manifold hypothesis is an empirical regularity, not a theorem, and some data really is not low-dimensional. Analogy arithmetic is approximate and partly a side effect of how it is scored (removing the query words). Individual embedding dimensions are almost never meaningful on their own. Meaning lives in directions and neighborhoods, not in single coordinates. Cosine similarity assumes direction carries the meaning, an assumption that anisotropy

can quietly weaken. And distributed codes gain their efficiency by giving up the local code's one advantage: there is usually no single neuron you can point to and call the "cat" unit.

Why & when this matters

The moment you feed categorical or high-dimensional data to a model, the representation is your first and often most important design decision. Embeddings are the default encoding for words, tokens, users, items and any high-cardinality category. And reusing a pretrained representation (transfer learning, chapter 22) is frequently the single biggest practical gain available, more valuable than tuning the classifier on top of it. When a model is stuck, its representation is a likely cause: check the embedding space for collapse (near-zero variance) or anisotropy (everything mutually similar) before blaming the optimizer. Thinking in terms of "what coordinates has this network learned, and are the classes untangled in them?" is how practitioners reason about deep models above the level of individual weights.

In the real world

Embeddings run the modern internet. Every large language model begins with a token-embedding matrix (chapter 17), and word2vec and GloVe started the era. Recommender systems at YouTube, Spotify and Amazon embed users and items into a shared space and rank by dot product or cosine, often in a "two-tower" retrieval architecture. Face recognition systems such as FaceNet learn embeddings where two photos of the same person land close together and different people far apart. Vector databases like FAISS index billions of learned embeddings for nearest-neighbor semantic search, powering

retrieval-augmented generation and image search. And transfer of pretrained representations (ImageNet features for vision, BERT features for text) is so routine that training a serious model from scratch is now the exception, not the rule.

Exercises

1. Show algebraically that multiplying a one-hot row vector $\mathbf{e}_i$ by the embedding matrix E returns exactly row i, and explain why, during training, only that row receives a gradient. MATH

2. Using the chapter's matrix, compute the three cosine similarities involving "van" (cos(cat, van), cos(dog, van), cos(car, van)) and state which two words form the tighter cluster with van. MATH

3. A one-hot times a matrix is a row selection. Verify in NumPy that, for a whole batch of token indices, the one-hot matmul and a direct row gather give identical results. CODE

4. Contrast local and distributed codes on capacity and similarity: how many distinct concepts can 8 units represent under each scheme, and which scheme can express that two concepts are alike? CONCEPT

5. Why is cosine similarity, rather than Euclidean distance, the usual way to compare embeddings? Show that scaling a vector by any $c > 0$ leaves its cosine similarity to another vector unchanged. MATH

6. Implement analogy arithmetic on a small hand-built embedding and confirm that "king − man + woman" retrieves "queen." Then note honestly why the query words must be excluded. CODE

7. The static word "bank" is assigned a single embedding vector. Explain the problem this causes (polysemy) and how contextual embeddings resolve it. CONCEPT

8. In terms of coordinate changes and the manifold hypothesis, explain why a linear classifier fails on the concentric-ring data yet succeeds after a single hidden layer, and connect it to the polar-coordinate view from the start of the chapter. CONCEPT

PART II

Training networks

How a network learns: what "wrong" means (loss), how to descend the error surface (gradient descent), how gradients are computed (backpropagation), the optimizers that make it fast, and the initialization and regularization that make it work at all.

6 Loss functions & maximum likelihood

7 Gradient descent & its variants

8 Backpropagation

9 Momentum, RMSProp & Adam

10 Initialization & the vanishing gradient

11 Normalization: batch, layer & beyond

12 Regularization & generalization

Loss functions & maximum likelihood

> *Training a network means turning a vague wish — "get the answers right" — into a single number that a machine can steadily push downward. That number is the **loss**. This chapter shows where the two main losses come from. Squared error comes from assuming Gaussian noise, and cross-entropy comes from counting the probability of the correct labels. Both are not arbitrary formulas. They are the answer to one question: which parameters make the data we actually saw most likely?*

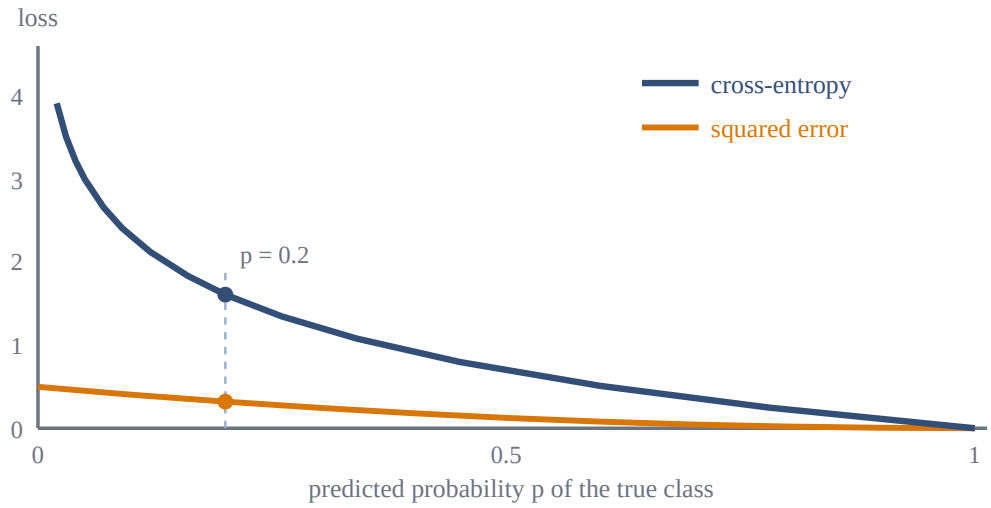
Fundamental concept

Imagine grading an archer. Every shot lands some distance from the bullseye, and you want one score that summarizes a whole set of shots. You could add up the distances, or the squared distances, or count bullseyes. Each choice rewards different behavior. A **loss function** L is exactly this scoring rule for a model. It reads the network's prediction $\hat{y}$ (the output of a forward pass, chapter 3) and the true target y, and returns a non-negative number. This number is zero for a perfect answer and grows as the answer worsens. Training (chapters 7–8) then adjusts the weights to make this score as small as possible, averaged over the data.

The loss is where your *intent* enters the mathematics. Nothing else in the network knows whether you care about being exactly right, roughly right, or confidently right. The loss encodes it. Choose the wrong one and a perfectly good architecture will learn the wrong lesson. So the real question is not "what formula?" but "what do I believe about my data and my mistakes?" Remarkably, two very natural beliefs determine the two losses used almost everywhere.

The guiding principle is **maximum likelihood**. Treat the network as defining a probability of the observed target given the input, $p(y \mid x; \theta)$. The probability background it draws on (densities, independence, expectation) is collected in Appendix C. The best parameters θ are the ones under which the whole dataset was most probable. Because probabilities of independent

samples multiply and are tiny, we take a logarithm and flip the sign. This turns "maximize a product of likelihoods" into "minimize a sum of losses". Every loss below is really a negative log-likelihood.



Cost of being wrong about the true class. Cross-entropy $-\ln p$ diverges as $p \to 0$, so a confident mistake is punished harshly. Squared error $\tfrac{1}{2}(1-p)^2$ flattens out near the corners, so it barely reacts. The two dots mark $p = 0.2$ from the worked example (cross-entropy 1.61, squared error 0.32).

Governing equations

For regression (real-valued targets) the standard loss is the **mean squared error** over m samples:

$$L_{MSE} = \frac{1}{2m} \Sigma_{i=1..m} \|\hat{y}^{(i)} - y^{(i)}\|^2$$

For classification the network outputs a probability vector p = softmax(z) over K classes. Softmax is the output activation of chapter 4. The loss is the **cross-entropy** between the one-hot label y and p:

$$L_{CE} = -\frac{1}{m} \Sigma_{i=1..m} \Sigma_{k=1..K} y_k^{(i)} \ln p_k^{(i)}, \quad p_k = \frac{e^{z_k}}{\Sigma_j e^{z_j}}$$

$\hat{y}^{(i)}, y^{(i)}$	prediction and target for sample i (vectors for multi-output; $\|\cdot\|$ is Euclidean length)
m	number of samples averaged over: a batch, a mini-batch, or the whole set
z	the output layer's pre-activations (logits), one per class: real numbers, any sign
p_k	predicted probability of class k after softmax; $p_k \in (0,1)$ and $\Sigma_k\, p_k = 1$
y_k	one-hot target: 1 for the correct class, 0 otherwise, so $\Sigma_k\, y_k = 1$
σ^2	assumed variance of the output noise (below): sets the loss scale, drops out of the argmin

Derivation — both losses from maximum likelihood

Squared error from Gaussian noise. Suppose targets are a deterministic function plus bell-shaped measurement noise, $y = f(x; \theta) + \varepsilon$ with $\varepsilon \sim N(0, \sigma^2)$. Then each observation has density

$$p(y \mid x; \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - f)^2}{2\sigma^2}\right)$$

1. Log-likelihood of m independent samples: $\ln L = \Sigma_i \left[-\tfrac{1}{2} \ln(2\pi\sigma^2) - (y^{(i)} - f^{(i)})^2 / 2\sigma^2 \right]$.
2. Only the second term depends on θ. Maximizing ln L over θ therefore equals minimizing $\Sigma_i\, (y^{(i)} - f^{(i)})^2$. The constant and the $1/2\sigma^2$ factor cannot change where the minimum sits.
3. Dividing by m gives the mean. Including the surviving ½ in the loss gives L_{MSE}. So squared error *is* Gaussian maximum likelihood.

The ½ is only a convenience, not something deeper: differentiating $\tfrac{1}{2}(\hat{y} - y)^2$ gives exactly $(\hat{y} - y)$, with no stray factor of 2 to carry through every gradient. It also matches the $2\sigma^2$ that MLE gives us.

Cross-entropy from categorical likelihood. For a one-hot label, the model's probability of the observed class is the product $\Pi_k\, p_k^{y_k}$, in which every factor with $y_k = 0$ equals 1, leaving just p of the true class. The negative log-likelihood of one sample is $-\Sigma_k\, y_k \ln p_k$. Averaging over the batch gives L_{CE}. Minimizing it drives the predicted probability of the correct class toward 1.

The name is borrowed from information theory (Appendix D): $-\Sigma_k\, y_k \ln p_k$ is the cross-entropy $H(y, p)$. This is the average surprise (in nats) of labels drawn from y when they are described with a code built for the model's distribution p. It reaches its minimum exactly when $p = y$.

Binary cross-entropy from a Bernoulli likelihood. With just two classes a single probability $p = \sigma(z) \in (0, 1)$ covers the positive class and $1 - p$ the negative, so a binary label $y \in \{0, 1\}$ is a Bernoulli trial. Its likelihood for one observation is the compact $p^y(1 - p)^{1-y}$. This formula gives p when $y = 1$ and $1 - p$ when $y = 0$. The negative log is the **binary cross-entropy** $L = -[y \ln p + (1 - y) \ln(1 - p)]$, precisely the $K = 2$ special case of the categorical loss above (identify $p_1 = p$ and $p_2 = 1 - p$). Minimizing it is Bernoulli maximum likelihood. This is the two-class version of the multiclass case. It is why a sigmoid output pairs with BCE, for exactly the same likelihood argument that pairs softmax with cross-entropy.

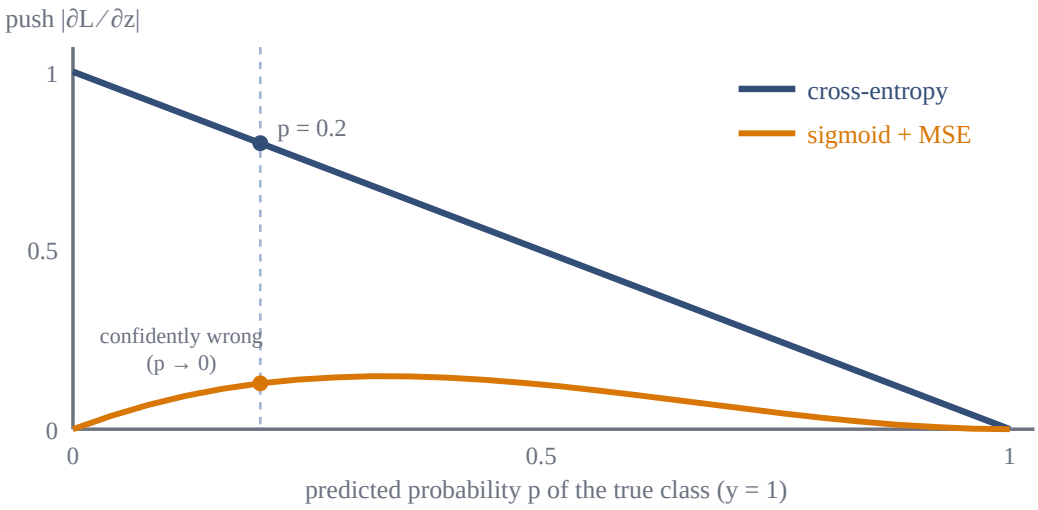
The fused gradient. Softmax and cross-entropy are always paired because their derivative collapses to something surprisingly clean. For one sample:

1. Loss slope on the probabilities: $\partial L / \partial p_k = -\, y_k / p_k$.
2. Softmax Jacobian: $\partial p_k / \partial z_i = p_k(\delta_{ki} - p_i)$, where $\delta_{ki} = 1$ if $k = i$ else 0.
3. Chain them: $\partial L / \partial z_i = \Sigma_k\, (-\, y_k / p_k) \cdot p_k(\delta_{ki} - p_i) = -\Sigma_k\, y_k(\delta_{ki} - p_i) = -y_i + p_i\, \Sigma_k\, y_k$.
4. Since $\Sigma_k\, y_k = 1$, everything cancels to the famous result:

$$\partial L \,/\, \partial \mathbf{z} = \mathbf{p} - \mathbf{y}$$

The error signal given to backpropagation (chapter 8) is simply "predicted minus true". Writing the prediction as $\hat{y} = p$, it is $\hat{y} - y = p - y$. The same form appears for a single sigmoid output with binary cross-entropy, $\partial L/\partial z = p - y$.

This cancellation is why classification avoids squared error. With a sigmoid output and MSE, the chain rule keeps an extra factor: $\partial L/\partial z = (p - y)\cdot\sigma'(z) = (p - y)\cdot p(1 - p)$. When the network is *confidently wrong* (p near 0 while $y = 1$), that $\sigma' = p(1 - p)$ collapses to nearly zero and the gradient nearly vanishes. So the network barely corrects its worst mistakes. Cross-entropy's $p - y$ has no such factor: the more confident the mistake, the harder it pushes. For more robust regression, **MAE** (mean absolute error, $|\hat{y} - y|$) resists outliers with a constant-size gradient but kinks at zero. The **Huber** loss joins a quadratic core to linear tails to get the best of both.



The gradient at the logit, which is just ŷ − y. For a true label y = 1, cross-entropy's push |p − 1| (blue) is largest exactly when the model is confidently wrong (p → 0). But sigmoid-plus-squared-error's push |(p − 1)·p(1 − p)| (amber) is reduced by the σ′ = p(1 − p) factor and collapses toward zero right where a big correction is most needed. The dots mark p = 0.2 from the first worked example (pushes 0.80 vs 0.128, the 6.25× gap).

Robust regression: MAE and the Huber loss

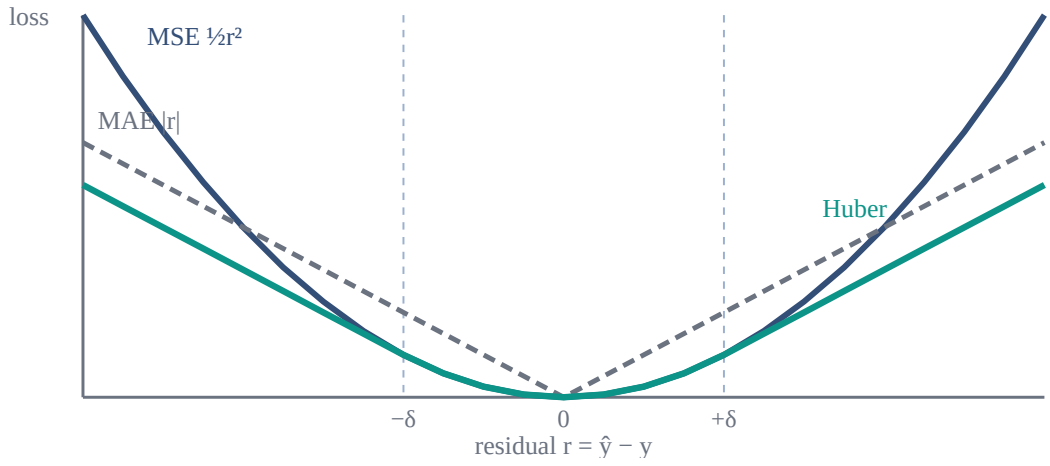
Squared error's main weakness is the square itself. A residual $r = \hat{y} - y$ contributes $\tfrac{1}{2}r^2$ to the loss and r to the gradient. So a single large outlier (a mislabeled point, a sensor glitch) with $r = 10$ contributes a gradient of 10, overwhelming a hundred well-behaved points with $r \approx 0.1$. Under the MLE reading this is exactly right *if* the noise really is Gaussian. But real data has heavier tails, and then the mean-chasing MSE is pulled toward the outliers. Two robust alternatives reduce this problem:

$$L_{\text{MAE}} = |r|\,, \quad \partial L \,/\, \partial r = \operatorname{sign}(r)$$

$$L_{\text{Huber}} = \tfrac{1}{2}r^2 \;\; (|r| \le \delta)\,, \quad \delta(|r| - \tfrac{1}{2}\delta) \;\; (|r| > \delta)$$

MAE penalizes the absolute residual, so its gradient is ±1 regardless of how large the error is. One outlier now counts the same as one ordinary point, and minimizing it fits the *median* rather than the mean. Its weakness is the kink at $r = 0$: the gradient never shrinks as you approach the target, so the final steps of optimization can jitter. The **Huber** loss is a designed compromise. Within a chosen radius δ of zero it behaves like $\tfrac{1}{2}r^2$ (smooth, gradient $\to 0$ at the target, good for fine convergence). Beyond δ it switches to a linear MAE-style penalty (gradient capped at δ,

robust to outliers). It is continuous and once-differentiable at the boundary $r = \pm\delta$, where both the value ($\frac{1}{2}\delta^2$) and the slope (δ) agree. Setting δ small makes Huber more MAE-like and robust. Large δ makes it more MSE-like.



Three regression losses. MSE (blue) grows quadratically, so distant residuals dominate. MAE (gray, dashed) grows linearly and ignores outliers but kinks at 0. Huber (teal) is quadratic inside ±δ and linear outside: smooth at the target, bounded-slope in the tails.

Numerical stability — never take the log of a raw probability

Written literally, cross-entropy asks for $\ln p$, and softmax asks for e^z. Both are traps in finite-precision arithmetic. A large positive logit overflows e^z to infinity. A class the model considers impossible ($p \to 0$) sends $\ln p$ to $-\infty$, producing a `NaN` that ruins the whole training run. Production frameworks never compute the two steps separately. They *fuse* log-softmax into one expression using the log-sum-exp trick,

$$\ln p_k = z_k - \ln \Sigma_j e^{z_j} = (z_k - c) - \ln \Sigma_j e^{(z_j - c)}, \quad c = \max_j z_j$$

Subtracting the maximum logit c makes every exponent ≤ 0, so nothing overflows. The largest term is exactly 1, so nothing underflows to zero either. And c cancels out algebraically, leaving the answer unchanged. This is why PyTorch's `nn.CrossEntropyLoss` takes *raw logits*, not probabilities: it does the stable fused computation internally. The lesson carries into the code below, where `softmax` always begins with `z - z.max()` for the same reason.

Beyond the two losses — smoothing, imbalance, and calibration

Maximum likelihood gives you MSE and cross-entropy. But three practical pressures reshape them in real training runs: labels that are not perfectly certain, classes that are badly imbalanced, and probabilities that must be trustworthy, not just accurate. Each is a small, principled edit to the loss rather than a new one.

Label smoothing. A hard one-hot target is the likelihood ideal only if the label is certain. But softmax can reach $p = 1$ only in the limit where the correct logit goes to $+\infty$. So pure cross-entropy always rewards a larger correct-class logit and trains the model toward *over-confidence*. Label smoothing softens the target from y to $(1 - \alpha)y + \alpha / K$, spreading a small mass α evenly across all K classes. The optimal logit gap then becomes finite, and the network stops chasing infinite certainty. With $\alpha = 0.1$ and $K = 3$ the target $[0, 1, 0]$ becomes $[0.033, 0.933, 0.033]$, and the smallest achievable cross-entropy is no longer 0 but the target's own entropy $H(t) \approx 0.29$ nats (Appendix D). Read as a mild regularizer on the output distribution (chapter 12), smoothing improves calibration and often raises test accuracy slightly. The cost is one extra hyperparameter α and predicted probabilities that are deliberately biased away from 0/1. It is now standard in large image classifiers and transformers.

Class imbalance and focal loss. When negatives outnumber positives a thousand to one (fraud, lesions, rare objects against background), plain cross-entropy is overwhelmed by the many easy negatives. Each one contributes a small but steady gradient, and together they dominate the rare positives. Two remedies are data-centric: re-weighting the loss per class, or resampling the data (chapter 26). The third is loss-centric: **focal loss** multiplies each example's cross-entropy by a modulating factor $(1 - p_t)^\gamma$, where p_t is the model's probability on the true class:

$$L_{\text{focal}} = -(1 - p_t)^\gamma \ln p_t$$

An easy, already-correct example ($p_t \to 1$) has its loss pushed toward zero, so the gradient focuses on the hard, rare, still-misclassified cases. $\gamma = 0$ recovers ordinary cross-entropy, and $\gamma = 2$ is a common default. This down-weighting is what lets a single-stage detector train on a hundred thousand background boxes per image without the easy ones dominating the objective.

Proper scoring rules and calibration. There is a deeper reason MSE and cross-entropy sit at the center of this chapter, and not some arbitrary alternative: both are *strictly proper scoring rules*. A scoring rule is proper when its expected value is optimized only by reporting the true probability. So a model can never lower its expected loss by exaggerating or understating its confidence. Honesty is the unique optimum, and that is what makes the learned probabilities meaningful. Being proper does not, however, guarantee *calibration*: among the inputs a model marks "80% likely", four in five really are positive. Deep networks trained with cross-entropy are known to be over-confident, and a low loss can go together with badly miscalibrated probabilities. Measuring that gap (reliability diagrams, expected calibration error) and closing it after training (temperature scaling) is the topic of the uncertainty chapter (chapter 24). The lesson here is that the number you minimize and the confidence you can act on are related, but not the same thing.

The losses in code

Both losses, and the clean gradients derived above, are only a few lines of NumPy. The first cell computes MSE and checks its analytic gradient $(\hat{y} - y)/m$ against a finite difference:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
y    = np.array([3.0, -1.0, 2.0, 0.5])     # targets
yhat = np.array([2.5, -0.2, 2.0, 1.5])     # predictions
m = len(y)

def mse(p, t): return 0.5*np.mean((p-t)**2)
grad = (yhat - y)/m                          # analytic dL/dyhat

eps = 1e-6                                    # finite-difference check on yhat[0]
yh2 = yhat.copy(); yh2[0] += eps
num0 = (mse(yh2, y) - mse(yhat, y))/eps

print("MSE            =", round(mse(yhat, y), 4))
print("analytic grad  =", np.round(grad, 4))
print("numeric grad[0]=", round(num0, 4), " (matches", round(grad[0], 4), ")")
```

The second cell is the core of the chapter. It computes softmax, then cross-entropy, then checks numerically that the gradient with respect to the logits really is the fused **p − y**. The finite-difference column and the analytic column come out identical:

Python · NumPy ▶ runnable in the app

```
import numpy as np
def softmax(z):
    z = z - z.max()                # numerical stability (subtract max
logit)
    e = np.exp(z)
    return e/e.sum()

z = np.array([1.0, 2.0, 0.5])      # logits for 3 classes
y = np.array([0.0, 1.0, 0.0])      # one-hot: true class is #1
p = softmax(z)
ce = -np.sum(y*np.log(p))
print("p            =", np.round(p, 4))
print("cross-entropy=", round(ce, 4))
print("fused p - y  =", np.round(p - y, 4))

eps = 1e-6; g = np.zeros(3)         # numeric gradient of CE wrt each logit
for i in range(3):
    z2 = z.copy(); z2[i] += eps
    g[i] = (-np.sum(y*np.log(softmax(z2))) - ce)/eps
print("numeric grad =", np.round(g, 4))
```

The third cell compares the three regression losses' gradients on a batch of residuals whose last entry is an extreme outlier. It shows how MAE and Huber cap its influence while MSE lets it dominate:

Python · NumPy ▶ runnable in the app

```
import numpy as np
def mse_grad(r):        return r
def mae_grad(r):        return np.sign(r)
def huber_grad(r, d=1.0):
    return np.where(np.abs(r) <= d, r, d*np.sign(r))

r = np.array([0.2, -0.5, 1.0, 8.0])   # residuals (yhat - y); last one is an
outlier
print("residuals  :", r)
print("MSE   grad :", np.round(mse_grad(r), 2), " <- outlier dominates
```

```
(8.0)")
print("MAE   grad :", np.round(mae_grad(r), 2), " <- all equal magnitude")
print("Huber grad :", np.round(huber_grad(r), 2), " <- outlier capped at
delta=1")
```

In PyTorch you never write these by hand. The framework supplies stable, fused implementations. Note two details: `nn.MSELoss` omits the ½ (so its value is twice the cell above), and `nn.CrossEntropyLoss` takes *raw logits* plus *integer class indices*, not one-hot vectors, and fuses log-softmax with the negative log-likelihood internally:

PyTorch copy → Colab

```
import torch
import torch.nn as nn

# regression: mean squared error (PyTorch omits the 1/2 factor)
yhat = torch.tensor([2.5, -0.2, 2.0, 1.5])
y    = torch.tensor([3.0, -1.0, 2.0, 0.5])
print(nn.MSELoss()(yhat, y))            # 0.4725  == 2 * 0.2362

# classification: cross-entropy on RAW logits + integer labels
logits = torch.tensor([[1.0, 2.0, 0.5]])   # shape (batch=1, classes=3)
target = torch.tensor([1])                  # true class index (not one-hot)
print(nn.CrossEntropyLoss()(logits, target))   # 0.4644, fuses log-softmax +
NLL

# for a single sigmoid output, use BCEWithLogitsLoss (also fused / stable)
z = torch.tensor([0.3]); t = torch.tensor([1.0])
print(nn.BCEWithLogitsLoss()(z, t))
```

In this app's engine

The Playground's loss curve plots exactly this number each step, and its problem picker sets the loss for you: cross-entropy for the classification datasets (XOR, circles, moons, spirals, checkerboard, Gaussians) and MSE for the regression ones (noisy sine, line, bump). Open the Math Log while a classification net trains and you can watch the output layer's raw error vector. It is p − y, the fused gradient derived above, before it flows backward through the rest of the graph (chapter 8).

Worked example — a confident mistake, scored two ways

A binary classifier with a sigmoid output sees a positive example (true label $y = 1$) but reports probability $p = 0.2$. It is wrong, and sure of it. Compare how each loss reacts.

1. Squared error: $L_{MSE} = \tfrac{1}{2}(p - y)^2 = \tfrac{1}{2}(0.2 - 1)^2 = \tfrac{1}{2}(0.64) = \mathbf{0.320}$.
2. Cross-entropy: $L_{CE} = -\ln p = -\ln 0.2 = \mathbf{1.609}$. That is five times the penalty for the same wrong answer.
3. Gradient at the logit, squared error: $\partial L/\partial z = (p - y)\cdot p(1 - p) = (-0.8)(0.2\cdot 0.8) = (-0.8)(0.16) = \mathbf{-0.128}$.
4. Gradient at the logit, cross-entropy: $\partial L/\partial z = p - y = 0.2 - 1 = \mathbf{-0.800}$.
5. Push ratio: $0.800 / 0.128 = \mathbf{6.25\times}$. Both gradients are negative, so both say "raise z to raise p". But cross-entropy pushes 6.25 times harder, precisely because the mistake is confident.

The number means this: squared error lets a confidently wrong neuron sit almost still (the saturated σ' reduces it), while cross-entropy makes the correction proportional to how wrong the probability is. This is the whole reason classifiers train on cross-entropy. See Practice → "Why cross-entropy".

Worked example — the gap widens with confidence

Fix the true label at $y = 1$ and slide the predicted probability p from a mild error toward a confident mistake. Make a table of both logit-gradients, cross-entropy $p - 1$ and squared-error $(p - 1)\cdot p(1 - p)$:

P (WITH Y = 1)	CE GRAD = P − 1	MSE GRAD	|CE| / |MSE|
0.50	−0.500	−0.125	4.0×
0.20	−0.800	−0.128	6.25×

P (WITH Y = 1)	CE GRAD = P − 1	MSE GRAD	|CE| / |MSE|
0.05	−0.950	−0.045	21.1×

As the mistake grows more confident ($p \to 0$), cross-entropy's push strengthens toward its maximum of 1, while squared error's push *weakens* toward 0. The $\sigma' = p(1 - p)$ factor reduces it exactly when a big correction is most needed. By $p = 0.05$ cross-entropy pushes 21 times harder. This is the same gradient reduction that stalls a confidently-wrong sigmoid+MSE classifier, seen as a smooth trend rather than a single point.

Worked example — categorical cross-entropy and its gradient (3 classes)

A 3-class network produces logits $z = [1.0, 2.0, 0.5]$, but the true class is the *first* one, so the one-hot target is $y = [1, 0, 0]$. The model currently favors class #1, a mistake. Compute the loss and the fused gradient $\hat{y} - y = p - y$ it gives to backprop.

1. Stabilize and exponentiate: subtract the max logit 2.0 to get $[-1, 0, -1.5]$, then $e^{-1} = 0.3679$, $e^{0} = 1$, $e^{-1.5} = 0.2231$, whose sum is 1.5910.
2. Softmax: $p = [0.3679, 1, 0.2231] / 1.5910 =$ **[0.2312, 0.6285, 0.1402]** (it sums to 1).
3. Cross-entropy picks out the true class only: $L = -\ln p_0 = -\ln 0.2312 =$ **1.464** nats. This is well above the $\ln 3 = 1.099$ of a uniform guess, as it should be for a confident error.
4. Fused gradient: $p - y = [0.2312 - 1, 0.6285 - 0, 0.1402 - 0] =$ **[−0.769, 0.629, 0.140]**.
5. One gradient-descent step on the logits with $\eta = 0.5$ (chapter 7): $z \leftarrow z - \eta(p - y) = [1.384, 1.686, 0.430]$, which lifts p_0 from 0.231 to **0.365**, straight toward the target of 1.

What the numbers mean: the gradient's three parts sum to exactly zero, because softmax cannot raise one class's probability without lowering the others. The true class carries the largest push, −0.769 (raise its logit), while the wrongly-favored class #1 gets the largest positive entry, 0.629 (lower its logit). The loss shifts probability from the class the model likes to the one it should. Every component is just prediction minus target. No activation-derivative factor survives.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
def softmax(z):
    z = z - z.max()                    # numerical stability
(subtract max logit)
    return np.exp(z)/np.exp(z).sum()
z = np.array([1.0, 2.0, 0.5])          # logits for 3 classes
y = np.array([1.0, 0.0, 0.0])          # one-hot: true class is #0
p = softmax(z)
print("p        =", np.round(p, 4))    # [0.2312 0.6285 0.1402]
print("CE       =", round(float(-np.sum(y*np.log(p))), 4))   # 1.4644
print("grad p-y =", np.round(p - y, 4)) # [-0.7688  0.6285  0.1402],
sums to 0
```

Assumptions & limitations

Squared-error-as-MLE assumes noise that is Gaussian, zero-mean, and the same size everywhere. With heavy-tailed noise or outliers, the squared penalty over-weights these rare points and MAE or Huber does better. Cross-entropy assumes the targets really are class probabilities and that softmax can represent them. Feed it mislabeled or overlapping classes and it will confidently fit the noise. A subtle trap: ln p is undefined at p = 0, so real implementations either fuse softmax with the log for numerical safety or clip p away from the corners. And a single loss value is only meaningful relative to a baseline. Cross-entropy of ln K is exactly the "random guessing" score for K classes.

Why & when this matters

The loss is the first thing to check when training misbehaves. A classifier stuck near loss ln K is predicting uniformly. A regression loss that will not drop below the noise floor has simply reached the irreducible error σ^2. Pairing softmax with cross-entropy (never with MSE) keeps gradients alive on hard examples, and reading a loss as a negative log-likelihood tells you what probabilistic assumption you silently accepted. Every optimizer (chapter 9) and every regularizer (chapter 12) acts on this one scalar. Choosing it well is half of framing a problem correctly.

In the real world

Cross-entropy is the loss behind almost every deployed image classifier, from ResNet on ImageNet to the vision models in medical-imaging triage. To reduce the over-confidence that pure cross-entropy encourages, Google's Inception-v3 introduced *label smoothing*. This softens each one-hot target from [0, 1, 0] to something like [0.05, 0.9, 0.05]. It is now standard in training large classifiers and transformers, and exactly the fix Exercise 8 asks about. The Huber loss (called "smooth L1") is the bounding-box regression loss in object detectors like Faster R-CNN and SSD. It is also the standard TD-error loss in DeepMind's Atari-playing DQN, where it stops rare huge errors from destabilizing the value function. When class imbalance is extreme, RetinaNet's *focal loss* reshapes cross-entropy to down-weight easy examples. This is another reminder that the loss is where you encode what you care about.

Exercises

1. For predictions $\hat{y} = [1, 2]$ and targets $y = [0, 2]$, compute L_{MSE} and its gradient, and confirm the gradient equals $(\hat{y} - y)/m$. MATH

2. A 3-class softmax produces logits $z = [0, 0, \ln 4]$ and the true class is #2 (the last). Compute p, the cross-entropy, and the fused gradient $p - y$. MATH

3. Show that binary cross-entropy $L = -[y \ln p + (1 - y) \ln(1 - p)]$ with $p = \sigma(z)$ has the gradient $\partial L/\partial z = p - y$. MATH

4. Write NumPy that computes cross-entropy for a whole batch (one row of logits per example) in a numerically stable way. CODE

5. Implement the Huber *value* (not just its gradient) and confirm it is continuous at the boundary $r = \delta = 1$. CODE

6. Why is softmax paired with cross-entropy and never with mean squared error? CONCEPT

7. A regression model's loss falls quickly but then refuses to drop below about 0.5, no matter how long you train. What does that most likely indicate? CONCEPT

8. Label smoothing replaces a one-hot target [0, 1, 0] with something like [0.05, 0.9, 0.05]. What problem does it address, and what is the trade-off? CONCEPT

9. With the focal-loss factor $(1 - p_t)^\gamma$ and $\gamma = 2$, compare how much an *easy* example (true-class probability $p_t = 0.9$) and a *hard* one ($p_t = 0.1$) each contribute, relative to plain cross-entropy. MATH

10. Label smoothing with $\alpha = 0.1$ on $K = 3$ classes turns a one-hot target into $t = [0.033, 0.933, 0.033]$. Show that cross-entropy against this soft target can no longer reach 0, and find its minimum value. MATH

11. Implement focal loss on a small batch in NumPy and confirm it down-weights easy examples relative to plain cross-entropy. CODE

Gradient descent & its variants

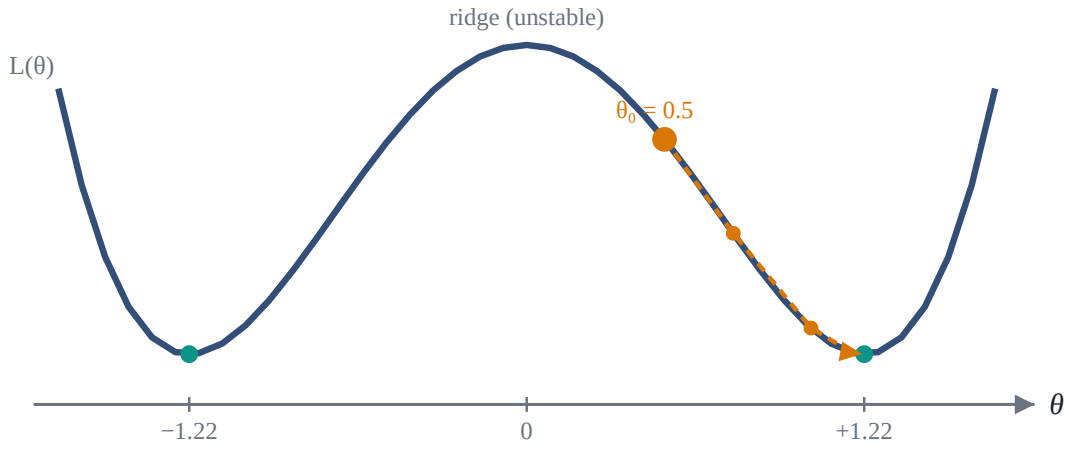
> *A network learns by repeatedly asking one local question: "if I change each weight a tiny bit, does the loss go up or down?" The answer is the gradient, and stepping the opposite way is **gradient descent**. It is the engine under every model in this app and nearly every model in the world. This chapter explains why the negative gradient is the steepest way down, why the learning rate has a hard stability ceiling, and how batching, non-convex terrain, and schedules shape the descent.*

Fundamental concept

Imagine a hiker caught in thick fog on a hillside, wanting to reach the valley. She cannot see the landscape, but she can feel the slope under her boots. So she takes a small step in the steepest downhill direction, feels again, and repeats. With short careful steps she goes down reliably. With reckless jumps she may leap straight over the valley and up the far side. That is gradient descent in one image: the loss surface is the terrain, the weights are her position, the slope she feels is the gradient, and her step length is the **learning rate**.

The method works because the gradient ∇L points in the direction of *steepest ascent* of the loss. Its negative therefore points downhill more steeply than any other direction, so moving against it gives the largest local decrease for each unit of step. Importantly, this is only a *local* fact. The slope right under your feet says nothing about what lies beyond the next hill. This is the source of every difficulty later in the chapter.

Everything else is refinement. How big a step (the learning rate and its schedule)? How accurately do we estimate the slope each time (full batch, one sample, or a mini-batch)? And what happens when the terrain is not a single clean bowl but a rough landscape of valleys, ridges and flats (non-convexity)? Each of these turns the simple rule into a practical algorithm.



Gradient descent on a non-convex loss. From $\theta_0 = 0.5$ the slope is negative, so four steps carry the ball into the $\theta \approx +1.22$ minimum. If it had started left of the central ridge, the same rule would have brought it to the mirror valley. The outcome depends on where you begin.

Governing equations

The update rule is a single line, applied to every parameter at once:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \, \nabla_{\theta} L(\boldsymbol{\theta})$$

We can justify it with the first-order Taylor expansion of the loss around the current point (the calculus refresher in Appendix B works through gradients and Taylor series):

$$L(\boldsymbol{\theta} + \Delta) \approx L(\boldsymbol{\theta}) + \nabla L \cdot \Delta$$

$\boldsymbol{\theta}$	the parameter vector — every weight and bias stacked together (dimensionless)
$\nabla_{\boldsymbol{\theta}} L$	the gradient: the vector of partial derivatives $\partial L / \partial \theta_j$, same shape as θ; points uphill
η	learning rate (step size), a positive scalar, typically 10^{-4} to 10^{-1}
Δ	a small trial step in parameter space; the Taylor rule holds only for small $\lVert\Delta\rVert$
k	local curvature of the loss (second derivative) — large k means a steep, narrow valley

For a fixed step length $\lVert\Delta\rVert = \varepsilon$, the dot product $\nabla L \cdot \Delta$ is made as *negative* as possible by choosing Δ to point in the opposite direction to ∇L. This follows from the Cauchy–Schwarz inequality:

$$\Delta^* = \operatorname{argmin}_{\|\Delta\|=\varepsilon} (\nabla L \cdot \Delta) = -\varepsilon \frac{\nabla L}{\|\nabla L\|}$$

which is why we step along $-\nabla L$ and nothing else.

Convex, non-convex, and what descent guarantees

Before we refine the step, it is worth being clear about what "descent" actually promises. A loss is **convex** if the straight line joining any two points on its graph never dips below the graph. It is a single bowl with no false bottoms. On a convex loss every local minimum is *the* global minimum, and gradient descent with a small enough step is guaranteed to reach it. Linear regression and the softmax/logistic losses (chapter 6) are convex, which is one reason they train so reliably. There is even a clean descent guarantee. Suppose the gradient is **L-smooth**: its value never changes faster than a constant L_s, meaning $\|\nabla L(a) - \nabla L(b)\| \le L_s\|a - b\|$. Then every step with $\eta \le 1/L_s$ strictly lowers the loss, and the hard ceiling $\eta < 2/L_s$ is exactly the 2/k boundary of the next section under a different name.

A neural network's loss is clearly **non-convex**. Stacking linear maps and nonlinear activations folds the surface into many basins, ridges and flats. Here the promise becomes much weaker. Descent still lowers the loss on each small enough step, but all it can guarantee to reach is a **stationary point**, a place where $\nabla L = 0$. This may be a local minimum, and often it is not one at all. What it never guarantees is the *global* optimum. "The loss went down every step" and "I found the best weights" are different claims, and quietly treating them as the same is one of the most common mistakes in practice.

Saddle points dominate in high dimensions. A stationary point is a true minimum only if the curvature is positive along *every* axis at once. In a model with n parameters that is n separate sign conditions. If the curvatures behaved like independent random signs, all n would come up positive only about 2^{-n} of the time. So for any realistic n, the vast majority of critical points are **saddles**: downhill in some directions, uphill in others. This changes the usual worry. In large networks the true minima tend to gather near the global loss value, so being "trapped in a bad minimum" is rarely the real problem. The trouble is instead that a saddle is surrounded by a nearly flat plateau where $\nabla L \approx 0$ and plain descent almost stalls. It escapes only by slowly

moving along the one downhill direction until the surface tips. The gradient noise of mini-batching and the inertia of momentum (chapter 9) are, in part, exactly the tools that slide an iterate off saddles where a noiseless full-batch step would sit.

How it works

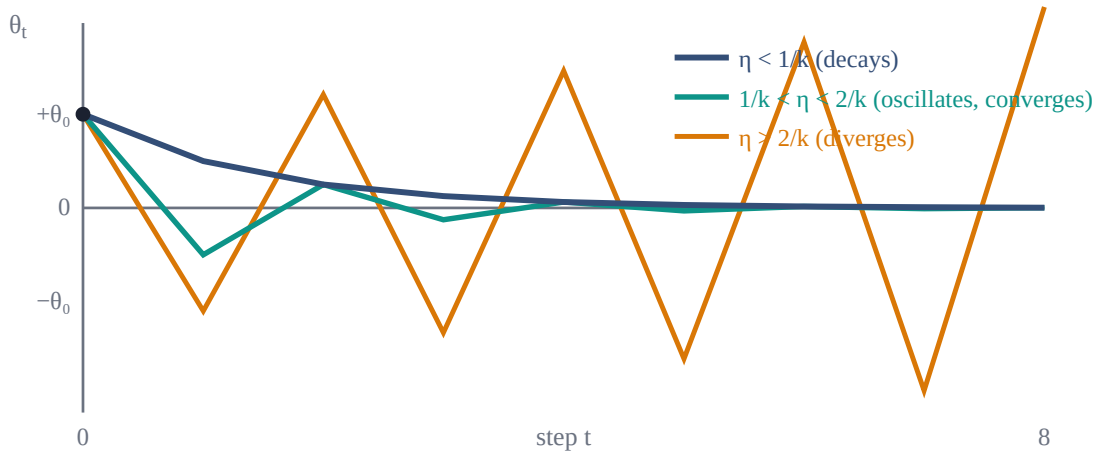
The learning-rate trichotomy. Take the simplest curved loss, $L(\theta) = \tfrac{1}{2}k\theta^2$, whose gradient is $k\theta$. One update becomes

$$\theta_{t+1} = \theta_t - \eta \cdot k\theta_t = (1 - \eta k)\cdot\theta_t \;\Rightarrow\; \theta_t = (1 - \eta k)^t \cdot \theta_0$$

All the behavior is set by the multiplier $(1 - \eta k)$:

1. $0 < \eta < 1/k$: the factor is between 0 and 1, so θ shrinks smoothly to the minimum. It is safe but possibly slow.
2. $1/k < \eta < 2/k$: the factor is between -1 and 0, so θ overshoots but still converges, changing sign each step.
3. $\eta = 2/k$: the factor is exactly -1, and θ bounces between $\pm\theta_0$ forever, never settling.
4. $\eta > 2/k$: the factor is larger than 1 in size, so $\|\theta\|$ grows every step. The loss **diverges** to infinity.

So there is a hard ceiling $\eta < 2/k$ set by the sharpest curvature in the landscape. Too small wastes time. Too large explodes. Real losses have many different curvatures at once, which is exactly the problem that the adaptive optimizers of chapter 9 address.



The trichotomy made visible: the iterate θ_t on $L = \tfrac{1}{2}k\theta^2$ for three learning rates. Blue decays straight to the minimum. Teal overshoots, but the amplitude shrinks each step. Amber's overshoot grows each step and runs away. The boundary between converging and diverging is exactly $\eta = 2/k$.

How much data per step. The gradient is an average over training examples. Each per-example term ∇L_i comes from one backward pass of backpropagation (chapter 8). We estimate it on whatever subset B (a mini-batch) we can afford:

$$\nabla L \approx \frac{1}{|B|} \sum_{i \in B} \nabla L_i(\boldsymbol{\theta})$$

and we can pick B in three ways. **Batch** gradient descent uses all m examples: the direction is exact but each step costs a full pass, and the path is smooth. **Stochastic** gradient descent (SGD) uses a single random example: each step is m times cheaper but the direction is noisy, bouncing around the true gradient. **Mini-batch** descent (typically 32–256 examples) is the practical middle choice. It has cheap steps with acceptable noise, and it maps neatly onto parallel hardware. That noise is not only a drawback: it lets the path shake out of sharp, fragile minima and tends to settle in wide flat ones that generalize better, an *implicit regularizer* (chapter 12). One full pass through the shuffled data is an **epoch**. Reshuffling each epoch keeps successive mini-batches from repeating the same biased order.

Variance versus compute. Why is the middle option the best choice? Treat each example's gradient ∇L_i as a noisy reading of the true gradient with per-coordinate variance σ^2. Averaging $|B|$ independent readings gives an estimate with variance $\sigma^2 / |B|$, so the noise falls only as $1 /$

$\sqrt{|B|}$ while the compute per step rises *linearly* with |B|. Going from |B| = 1 to 16 cuts the noise to a quarter for 16× the arithmetic, which is a bargain. Going from 256 to 4096 cuts it to a quarter again for another 16×, which is rarely worth it once the direction is already good. That diminishing return, together with the fact that a mid-sized batch (roughly 32–512) fills a GPU's parallel lanes without using up its memory, is why nearly everyone works in that band. Push |B| much higher and you must raise η in proportion (the *linear scaling rule*) until you near the stability ceiling and need warmup to survive the opening steps.

Non-convex terrain. A network's loss is not a single bowl. It has many **local minima**, though in high dimensions most are nearly as good as the best. More common are **saddle points**: flat in some directions, uphill or downhill in others. A critical point is a true minimum only if the curvature is positive along *every* one of thousands of axes, which is exponentially unlikely, so saddles vastly outnumber minima. Long **plateaus** of tiny gradient make progress extremely slow. Because the update only feels the local slope, both where you start (chapter 10's initialization) and how the step size changes over time affect the destination. **Learning-rate schedules** use this: *step decay* cuts η at milestones, *warmup* ramps η up gently for the first steps to avoid an early explosion, and *cosine* annealing lowers η smoothly toward zero so the final steps fine-tune rather than jump around.

Curvature, conditioning, and the ravine problem

The 1-D trichotomy generalizes exactly, and it explains the single biggest reason plain gradient descent is slow. Near a minimum any smooth loss looks like a quadratic bowl $L(\theta) = \tfrac{1}{2}\,\theta^{T} H\,\theta$, where the **Hessian** H collects the second derivatives. H has eigenvalues $\lambda_1 \geq \ldots \geq \lambda_n$, the curvatures along its principal axes. Along each of those axes the update *decouples* into an independent 1-D step with its own multiplier $(1 - \eta\lambda_i)$. Two facts follow immediately:

1. **Stability is set by the sharpest axis.** Every axis must satisfy $|1 - \eta\lambda_i| < 1$, so the largest curvature λ_{max} sets the ceiling $\eta < 2/\lambda_{max}$. Go beyond it along even one direction and the whole run diverges.
2. **Speed is set by the gentlest axis.** Along the smallest curvature λ_{min} the multiplier $(1 - \eta\lambda_{min})$ is just under 1, so that direction converges slowly. You cannot simply raise η to

fix it, because fact 1 does not allow it.

The ratio of these two curvatures is the **condition number** $\kappa = \lambda_{max} / \lambda_{min}$. With the best possible fixed learning rate, the error shrinks each step by a factor $(\kappa - 1)/(\kappa + 1)$: when $\kappa = 1$ (a perfectly round bowl) that is 0, meaning one step, but when $\kappa = 100$ it is 0.98, so it takes hundreds of steps to make real progress. In shape, the loss is a long narrow **ravine**: steep across, gentle along. Steepest descent points mostly across the ravine, so the iterate *zig-zags* from wall to wall while slowly moving along the floor. The 2-D example in code below has $\kappa = 10$, and you can watch the fast axis oscillate in sign (multiplier ≈ -0.8) while the slow axis crawls (≈ 0.82 per step, close to the ideal $9/11 \approx 0.818$). This is exactly the problem that momentum, per-parameter adaptive rates (Adam, chapter 9), and input normalization or batch-norm (chapter 11) are built to fix. They either speed up the consistent gentle direction or reshape the landscape toward $\kappa \approx 1$.

Learning-rate schedules

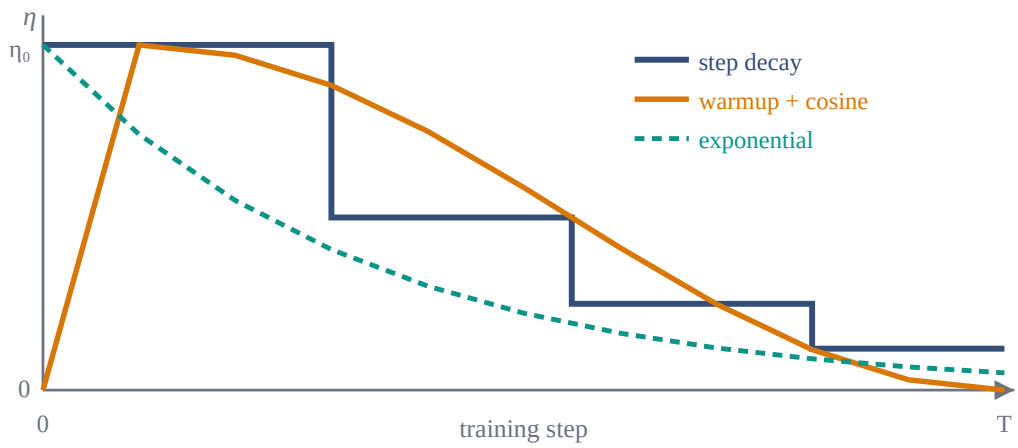
A single fixed η is always a compromise. If it is big enough to make early progress, it is too big to settle cleanly at the end. A **schedule** removes this conflict by letting η change during training: large at first to cover ground, small at the end to fine-tune. The decay comes in a few standard shapes, and choosing one well is often worth more than tuning any other hyperparameter.

Step decay holds η flat, then multiplies it by a factor at fixed milestones: $\eta = \eta_0 \cdot \gamma^{\lfloor t/s \rfloor}$, dropping by γ (classically 0.1) every s epochs. It was the old standard for ImageNet: train at 0.1, cut to 0.01 at epoch 30, then 0.001 at epoch 60. **Exponential decay** smooths those sharp drops into a continuous curve, $\eta = \eta_0 \cdot e^{-\lambda t}$, shrinking by a constant fraction each step. **Cosine annealing** (Loshchilov & Hutter, 2017) moves η along a half-cosine from η_0 down to nearly zero over the run, $\eta_t = \tfrac{1}{2}\,\eta_0\,(1 + \cos(\pi t / T))$. It stays a while at the high and low rates and passes quickly through the middle, works very well in practice, and has become the default for large-scale pretraining.

Warmup runs the opposite way at the very start. Instead of starting at full η, which is risky when the weights are still random and the curvature unknown, it ramps η *up* linearly from near zero over the first few hundred to few thousand steps, then switches to a decay. Warmup is what

makes very large batches and Transformers trainable at all: it keeps the high-variance opening updates below the stability ceiling until the estimates settle. **Cyclical** rates and the **one-cycle** policy (Smith, 2017) go further, deliberately oscillating η, or making a single large rise-then-fall across the whole run, on the idea that the high phases help the iterate hop between basins and slide over saddles while the low phases settle the result.

Where does the starting η_0 come from? The **LR-range test** is the standard answer: begin at a tiny rate and multiply it up geometrically over a few hundred iterations while watching the loss. The loss barely moves at first (η too small), then drops steeply through a useful band, then explodes (past the ceiling). Pick η_0 about an order of magnitude below where the loss turned upward, the steepest part that is still stable. It takes a few minutes and replaces hours of guessing, and it returns as part of the training recipe of chapter 25.



Three learning-rate schedules over one run. Blue step-decay holds η then cuts it by a fixed factor at milestones. Amber ramps η up in a short linear warmup and then follows a cosine down to almost zero. Teal exponential decay shrinks η by a constant fraction each step. All begin high to move fast and end low to fine-tune. The shape in between is the design choice.

Gradient descent in code

The update rule is one line, so a working optimizer is surprisingly short. The first cell runs it on the double-well loss $L(\theta) = \theta^4 - 3\theta^2 + 2$ from the worked example, reproducing the path we computed by hand and converging to the true minimum $\theta = \sqrt{1.5}$:

Python · NumPy ▶ runnable in the app

```
import numpy as np
def L(t):  return t**4 - 3*t**2 + 2      # double-well loss
def dL(t): return 4*t**3 - 6*t           # its gradient

t, eta = 0.5, 0.1
print(f"start: theta={t:.4f}  L={L(t):.5f}")
for step in range(1, 31):
    t = t - eta*dL(t)
    if step in (1, 2, 3, 4, 10, 30):
        print(f"step {step:2d}: theta={t:.4f}  L={L(t):.5f}  grad=
{dL(t):+.4f}")
print(f"true minimum: theta={np.sqrt(1.5):.4f}  L={L(np.sqrt(1.5)):.4f}")
```

The same loop works in any dimension. Only θ becomes a vector. The second cell descends the stretched 2-D quadratic $L(w) = \frac{1}{2}(10w_1^2 + w_2^2)$, whose condition number is 10. Watch the steep first coordinate flip sign each step (the zig-zag) while the gentle second coordinate decays slowly. This is the ravine problem, live:

Python · NumPy ▶ runnable in the app

```
import numpy as np
A = np.array([10.0, 1.0])                # curvature per axis; condition
number = 10
def L(w):  return 0.5*np.sum(A*w**2)
def dL(w): return A*w

w = np.array([1.0, 1.0]); eta = 0.18     # 0.18 < 2/10 = 0.2 stability
ceiling
print(f"kappa = {A.max()/A.min():.0f}   ceiling eta < 2/k_max =
{2/A.max():.2f}")
for step in range(1, 61):
    w = w - eta*dL(w)
    if step in (1, 2, 3, 5, 10, 30, 60):
        print(f"step {step:2d}: w=[{w[0]:+.4f}, {w[1]:+.4f}]  L={L(w):.6f}")
```

Finally, a learning-rate sweep puts numbers on the trichotomy. On $L = \frac{1}{2}\cdot 4\cdot\theta^2$ (curvature $k = 4$, ceiling $\eta = 2/k = 0.5$) it prints the final loss after 50 steps for a series of rates. Note that $\eta = 0.25 = 1/k$ lands the minimum in a single step, $\eta = 0.5$ bounces forever at loss 2, and $\eta = 0.55$ above the ceiling explodes:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
def final_loss(eta, k=4.0, steps=50, t0=1.0):
    t = t0
    for _ in range(steps):
        t = t - eta*(k*t)          # L = 1/2 k t^2, so grad = k t
    return 0.5*k*t*t

print("L = 0.5*4*t^2, start t=1, 50 steps.  Stability ceiling eta = 2/k =
0.5")
print(" eta     final loss")
for eta in [0.05, 0.10, 0.25, 0.45, 0.50, 0.55]:
    print(f"{eta:5.2f}   {final_loss(eta):.3e}")
```

In practice you let a framework's autograd compute ∇L and an optimizer object apply the step. The PyTorch version of the double-well cell (same loss, same η, same answer) reads:

PyTorch copy → Colab

```python
import torch

theta = torch.tensor([0.5], requires_grad=True)   # start point
opt = torch.optim.SGD([theta], lr=0.1)            # plain gradient descent

for step in range(30):
    opt.zero_grad()                                # clear last step's
gradient
    loss = theta**4 - 3*theta**2 + 2               # the double-well loss
    loss.backward()                                # autograd fills
theta.grad
    opt.step()                                      # theta <- theta - lr *
grad

print(theta.item(), loss.item())      # ~1.2247, ~-0.25
# add momentum=0.9 to accelerate down ravines; see chapter 9
```

In this app's engine

The Gradient descent lab is this chapter made interactive: pick a 1-D or 2-D loss surface, set η, and watch the ball step downhill exactly by $\theta \leftarrow \theta - \eta \nabla L$. Raise η past the stability ceiling and you will see the iterate throw itself uphill and diverge, reproducing case 4 above. In the Playground the same rule drives real networks, and the optimizer menu (SGD, Momentum, RMSProp, Adam) and batch-size control let you explore the noise-versus-smoothness trade-off on datasets like two spirals or moons.

Worked example — four steps down a double well

Minimize $L(\theta) = \theta^4 - 3\theta^2 + 2$, whose gradient is $\nabla L = 4\theta^3 - 6\theta$. Setting it to zero gives minima at $\theta^* = \pm\sqrt{1.5} \approx \pm 1.2247$, where $L = -0.25$. Start at $\theta_0 = 0.5$ with $\eta = 0.1$.

1. $\nabla L(0.5) = 4(0.125) - 6(0.5) = 0.5 - 3 = -2.5 \rightarrow \theta_1 = 0.5 - 0.1(-2.5) =$ **0.7500**.
2. $\nabla L(0.75) = 4(0.4219) - 4.5 = 1.6875 - 4.5 = -2.8125 \rightarrow \theta_2 = 0.75 + 0.28125 =$ **1.0313**.
3. $\nabla L(1.0313) = 4(1.0968) - 6.1878 = 4.3870 - 6.1878 = -1.8007 \rightarrow \theta_3 = 1.0313 + 0.18007 =$ **1.2113**.
4. $\nabla L(1.2113) = 4(1.7772) - 7.2678 = 7.1090 - 7.2678 = -0.1585 \rightarrow \theta_4 = 1.2113 + 0.01585 =$ **1.2272**.

In four steps θ has covered almost all the distance to the true minimum 1.2247, and the steps *shrink* (2.5 → 2.81 → 1.80 → 0.16 in gradient size) as the slope flattens near the bottom. Descent naturally slows down into a minimum. The negative gradients all pushed θ to the right because 0.5 sits on the right of the central ridge at $\theta = 0$. Restart at $\theta_0 = -0.5$ and every sign flips: the ball lands in the $\theta \approx -1.2247$ valley instead. Same loss, same rule, opposite answer. Initialization decides which basin you fall into.

Worked example — reading the stability ceiling

Take the pure quadratic $L(\theta) = \tfrac{1}{2}k\theta^2$ with curvature $k = 4$, so $1/k = 0.25$ and the ceiling $2/k = 0.5$. The multiplier is $1 - 4\eta$. Look at each case:

H	FACTOR 1 − 4H	BEHAVIOR
0.10	+0.60	monotone decay to 0 (case 1)
0.25	0.00	exact hit in one step ($\eta = 1/k$)
0.30	−0.20	oscillate but converge (case 2)
0.50	−1.00	bounce $\pm\theta_0$ forever (case 3)
0.60	−1.40	grow, diverge (case 4)

Reading the table: the best choice for a single curvature is $\eta = 1/k$, which removes the error in one step (factor 0). Anything up to $2/k$ still converges. Beyond $1/k$ it does so by overshooting and oscillating. At exactly $2/k$ it is trapped in a permanent bounce. Cross $2/k$ and the amplitude grows without bound. The learning-rate sweep cell prints exactly these outcomes (loss 0 at $\eta = 0.25$, loss 2 at $\eta = 0.50$, and an explosion at $\eta = 0.55$). The problem in a real network is that k is not one number but a whole range of curvatures, so a single η cannot be best for all of them at once. This is the motivation for chapter 9.

Worked example — three steps on a shifted quadratic

Minimize $L(\theta) = \theta^2 - 4\theta + 5$, a parabola with gradient $\nabla L = 2\theta - 4$ and one minimum where $2\theta - 4 = 0$, i.e. $\theta^* = 2$ (there $L = 1$). Its curvature is $k = L'' = 2$, so the ceiling is $2/k = 1$. Start at $\theta_0 = 0$ with $\eta = 0.3$, well inside the stable range.

1. $\nabla L(0) = 2\cdot 0 - 4 = -4 \rightarrow \theta_1 = 0 - 0.3(-4) =$ **1.2000**; $L = 1.2^2 - 4\cdot 1.2 + 5 =$ **1.6400**.
2. $\nabla L(1.2) = 2\cdot 1.2 - 4 = -1.6 \rightarrow \theta_2 = 1.2 - 0.3(-1.6) =$ **1.6800**; $L =$ **1.1024**.

3. $\nabla L(1.68) = 2 \cdot 1.68 - 4 = -0.64 \rightarrow \theta_3 = 1.68 - 0.3(-0.64) =$ **1.8720**; L = **1.0164**.

Now change coordinates to the distance from the minimum, $u = \theta - 2$. In u the loss is the pure bowl $L = u^2 + 1$ and the update reduces to the multiplier of the "How it works" section: $u_{t+1} = (1 - \eta k) \cdot u_t = (1 - 0.3 \cdot 2) \cdot u_t = 0.4 \cdot u_t$. From $u_0 = -2$ the distances run −2, −0.8, −0.32, −0.128, each 0.4× the last. This is a clean geometric decay to zero that matches θ = 1.2, 1.68, 1.872. Because 0.4 lies in (0, 1) this is case 1, monotone convergence, and the loss falls 5 → 1.64 → 1.10 → 1.016 toward its lowest value of 1. The shift is the only new ingredient: any quadratic is a re-centered $\frac{1}{2}k\theta^2$, and the same $(1 - \eta k)$ law controls every one of them.

Assumptions & limitations

The Taylor argument that justifies the step is a *first-order* approximation, trustworthy only for small $\eta\|\nabla L\|$. A step too large leaves the region where the linear approximation holds, and the guarantee disappears. Plain gradient descent has no memory and one global step size, so it crawls through ravines that are steep across and gentle along. This is the motivation for momentum and per-parameter rates (chapter 9). It finds a stationary point, never a certified global optimum. On badly scaled inputs the curvature k varies so much across weights that no single safe η exists (this is why we use input normalization, chapter 11). Finally, the gradient must exist: hard non-differentiable corners need sub-gradients or smoothing.

Why & when this matters

Gradient descent is at the heart of training, so its failure modes are the ones you meet first. A loss that explodes to NaN almost always means η is above the stability ceiling, so halve it. A loss that plateaus early may be stuck on a saddle, or it may be limited by too small an η, so raise it a little or add momentum. The batch-size dial trades wall-clock

speed against gradient noise, and the schedule decides whether late training fine-tunes or shakes around. Understanding the $(1 - \eta k)$ factor turns "pick a learning rate" from guesswork into reasoning about curvature.

In the real world

Every large model you have used was trained by a variant of this one update. Facebook's "Accurate, Large Minibatch SGD" (Goyal et al., 2017) scaled mini-batch SGD to 8192 images across 256 GPUs to train ImageNet in an hour, using two ideas straight from this chapter: the *linear scaling rule* (raise η in proportion to batch size) and a *warmup* that ramps η up gently so the early, high-variance steps do not go past the stability ceiling. Warmup turned out to be essential for training Transformers at all (Vaswani et al., 2017), and modern language-model pretraining lowers η smoothly with a *cosine* schedule over billions of tokens so the final steps fine-tune rather than jump around. Reinforcement-learning agents, recommender systems, and protein-structure predictors all descend the same way. The simple $\theta \leftarrow \theta - \eta \nabla L$, batched and scheduled, is the main tool of the entire field.

Exercises

1. On $L = \tfrac{1}{2}k\theta^2$ with $k = 4$, start at $\theta_0 = 1$ with $\eta = 0.3$. Compute θ_1 and θ_2, and say which trichotomy case this is. MATH

2. For $L(x, y) = \tfrac{1}{2}(10x^2 + y^2)$, find the stability ceiling on η, the condition number, and the long-run per-step convergence factor along the slow axis with $\eta = 0.18$. MATH

3. For the double well $L = \theta^4 - 3\theta^2 + 2$, take one gradient step from $\theta_0 = -0.5$ with $\eta = 0.1$ and confirm it moves toward the left minimum. MATH

4. Modify the double-well descent to start at $\theta_0 = 1.6$ (to the right of the minimum) with $\eta = 0.1$. Does it still converge, and how? CODE

5. For $L = \tfrac{1}{2}k\theta^2$ with $k = 1$, sweep the learning rate and find, by experiment, the largest η that still converges. CODE

6. Why does the noise in mini-batch (or stochastic) gradient descent sometimes *improve* generalization rather than just slow training? CONCEPT

7. Your training loss becomes `NaN` after a few steps. What is the most likely cause and the first fix to try? CONCEPT

8. You are training on a loss whose condition number is about 1000. Why is plain gradient descent very slow here, and what changes would help? CONCEPT

9. You train with initial rate $\eta_0 = 0.1$. (a) Under **cosine annealing** $\eta_t = \tfrac{1}{2}\,\eta_0\,(1 + \cos(\pi t / T))$ over $T = 200$ steps, give η at $t = 0$, 100, and 200. (b) Under **step decay** that multiplies η by 0.1 every 50 steps, give η at $t = 120$. MATH

10. Implement a **linear-warmup-then-cosine** schedule in NumPy: ramp η from 0 up to $\eta_0 = 0.1$ across the first 10 steps, then cosine-anneal to 0 over the rest of a 60-step run. Print η at a few steps. CODE

11. A colleague says "the training loss fell at every step, so the optimizer found the best possible weights." On a neural network, what is wrong with that, and on what kind of loss would it hold? CONCEPT

Backpropagation

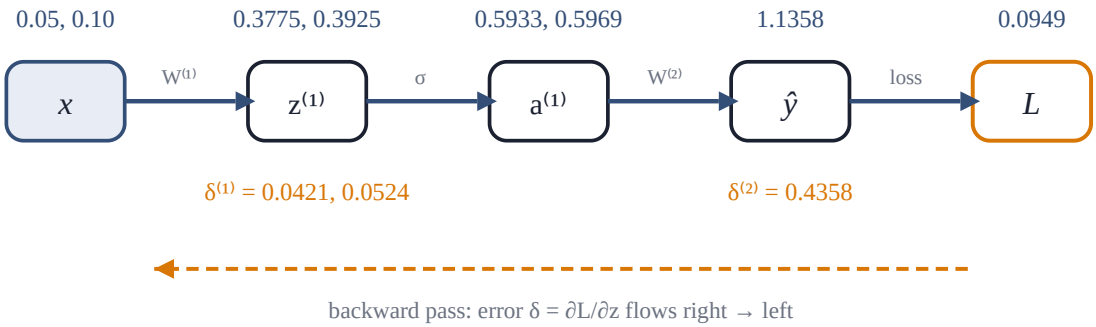
> *A deep network can have millions of weights, and every one of them is partly responsible for a single wrong answer.* ***Backpropagation*** *is the method that gives each weight its exact share of that responsibility — the derivative of the loss with respect to that weight — without computing anything twice. It is simply the chain rule of calculus, applied to a computational graph and arranged so the work is done once, moving backward from the loss to the inputs.*

Fundamental concept

Think of a company that ships a faulty product. The CEO knows only the final loss — an unhappy customer. To improve, the blame must travel back down the organization chart. Each manager learns how much the error would shrink if their team acted a little differently, then passes a correctly scaled message to the teams that feed into them. No one re-derives the whole company's behavior. Each level only combines the message from just above with what it did locally. This is **credit assignment**, and backpropagation is its exact arithmetic.

The local rule is the chain rule. Suppose the loss L depends on a weight w only through an intermediate value z: L depends on z, and z depends on w. Then $\partial L/\partial w = (\partial L/\partial z)\cdot(\partial z/\partial w)$. A network is just a long chain — and branching tree — of such dependencies. Inputs feed pre-activations, which feed activations, which feed the next layer, and finally the loss. Multiplying the chain rule along every path from a weight to the loss gives that weight's gradient. Backpropagation is the key observation that all these products share smaller sub-expressions. So if we compute them *backward* from the loss and store the shared piece — the error signal at each layer — the whole gradient costs about the same as one extra forward pass.

That shared piece has a name. Define $\delta^{(l)} = \partial L/\partial z^{(l)}$, which measures how sensitive the loss is to a layer's pre-activations. Once you know δ at a layer, the gradients for that layer's weights and biases follow immediately, and δ for the layer below follows from one matrix multiply. The whole algorithm is short: find δ at the output, then pass it down, layer by layer.



The app's Backprop lab shown as a computational graph. Blue shows the forward pass (the values above each node); amber shows the backward pass, which carries the error δ from the loss back toward the inputs. Every number is computed by hand in the worked example.

Governing equations

Write $z^{(l)} = a^{(l-1)}W^{(l)} + b^{(l)}$ and $a^{(l)} = g(z^{(l)})$ for each layer l, with the loss L at the top (layer L). The four equations of backpropagation are:

$$\delta^{(L)} = \nabla_a L \odot g'(z^{(L)}) \quad \text{— output error}$$

$$\delta^{(l)} = (\delta^{(l+1)}\, W^{(l+1)T}) \odot g'(z^{(l)}) \quad \text{— recurrence}$$

$$\partial L / \partial W^{(l)} = a^{(l-1)T}\, \delta^{(l)} \qquad \partial L / \partial b^{(l)} = \Sigma_{\text{batch}}\, \delta^{(l)}$$

$\delta^{(l)}$	error signal at layer l — the row vector $\partial L/\partial z^{(l)}$, one entry per unit in that layer
$z^{(l)}$, $a^{(l)}$	pre-activations and activations of layer l ($a^{(0)} = x$, the input); stored on the forward pass
$W^{(l)}$, $b^{(l)}$	weight matrix (shape $n_{l-1} \times n_l$) and bias of layer l; the quantities we want gradients for
$g'(z^{(l)})$	elementwise derivative of the activation at its stored input (e.g. $\sigma' = \sigma(1-\sigma)$)
$\odot$	elementwise (Hadamard) product; ^{T} is transpose; $\nabla_a L$ is the loss slope on the output activations

Derivation

Everything follows from the chain rule applied to $\delta^{(l)} = \partial L/\partial z^{(l)}$. The loss reaches $z^{(l)}$ only through the next layer's pre-activations $z^{(l+1)}$. So for a single unit j:

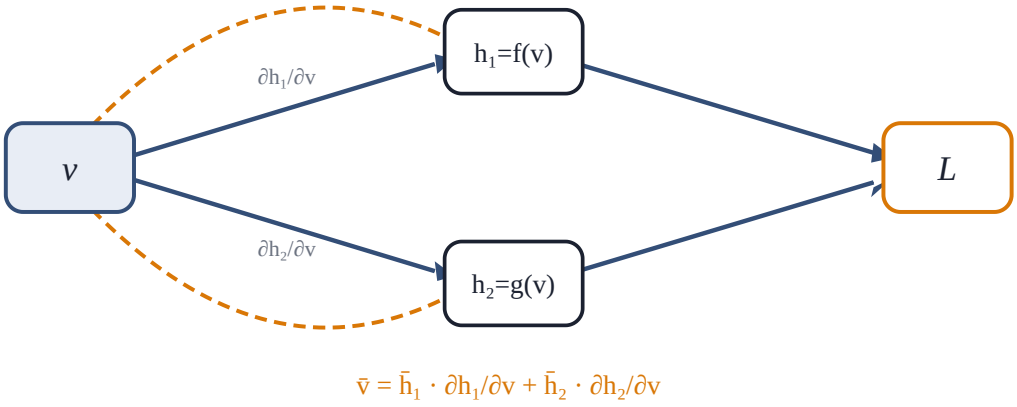
1. $\delta^{(l)}_j = \Sigma_k \dfrac{\partial L}{\partial z^{(l+1)}_k} \cdot \dfrac{\partial z^{(l+1)}_k}{\partial z^{(l)}_j} = \Sigma_k \delta^{(l+1)}_k \cdot \dfrac{\partial z^{(l+1)}_k}{\partial z^{(l)}_j}$.
2. Because $z^{(l+1)}_k = \Sigma_i W^{(l+1)}_{ik}\, g(z^{(l)}_i) + b_k$, the inner derivative is $W^{(l+1)}_{jk}\, g'(z^{(l)}_j)$.
3. So $\delta^{(l)}_j = g'(z^{(l)}_j) \Sigma_k W^{(l+1)}_{jk}\, \delta^{(l+1)}_k$ — in matrix form, the recurrence $\delta^{(l)} = (\delta^{(l+1)} W^{(l+1)\mathrm{T}}) \odot g'(z^{(l)})$.
4. A weight enters its own pre-activation only: $\partial L/\partial W^{(l)}_{ij} = \delta^{(l)}_j \cdot \partial z^{(l)}_j / \partial W_{ij} = \delta^{(l)}_j \cdot a^{(l-1)}_i$, the outer product $a^{(l-1)\mathrm{T}} \delta^{(l)}$.
5. A bias enters with coefficient 1, so $\partial L/\partial b^{(l)}_j = \delta^{(l)}_j$ (summed over a batch).

The **two-pass algorithm** reads straight off these equations: (1) *forward* — compute and store every $z^{(l)}$ and $a^{(l)}$, then the loss; (2) *output error* — evaluate $\delta^{(L)}$; (3) *backward* — apply the recurrence from $l = L-1$ down to 1; (4) *read gradients* — form $\partial L/\partial W^{(l)}$ and $\partial L/\partial b^{(l)}$ for every layer; (5) update with $\theta \leftarrow \theta - \eta \nabla L$ (gradient descent, Chapter 7). What makes it cheap is **dynamic programming**: each $\delta^{(l)}$ is computed once and then reused twice — for that layer's weight gradient and to produce $\delta^{(l-1)}$ — so no path through the graph is ever walked twice. The backward pass costs about twice a forward pass, and its cost is memory: every activation must be kept until its gradient is used.

Backprop as reverse-mode autodiff

The four equations above are one example of a fully general algorithm: **reverse-mode automatic differentiation** (autodiff). Any numerical program — not just a neural network — is a directed acyclic graph of basic operations (add, multiply, matmul, exp, max, …), each with a known local derivative. To differentiate a single scalar output L with respect to every input at once, start the output with $\partial L/\partial L = 1$ and sweep *backward* through the graph in reverse topological order. Each node receives an **adjoint** $\bar{v} = \partial L/\partial v$ from the nodes it feeds, multiplies it by its own local derivative, and passes the result to the nodes that feed it. The $\delta^{(l)}$ of backprop are exactly the adjoints of the pre-activations $z^{(l)}$ — nothing more is happening.

One rule deserves special attention, because it is the only real subtlety: when a value *fans out* to several nodes, its adjoint is the **sum** of the contributions coming back along every branch. This is the multivariable chain rule. It is why a shared weight, a bias reused across a batch, or a residual/skip connection (Chapters 11 and 14) collects gradient from all the paths it affects.



Reverse-mode autodiff at a fan-out node. Blue edges carry the forward values; the two amber dashed arrows carry gradients back from both branches, and the adjoint at v is their sum. Backprop is this rule applied to the whole network, with $\delta^{(l)}$ as the adjoints of the pre-activations.

Two properties make reverse mode the right tool for deep learning. First, its cost is **independent of the number of parameters**: one forward sweep plus one backward sweep computes the gradient for a million weights at the price of about two forward passes. Changing each parameter separately instead (forward mode, or plain finite differences) would cost a million passes. Reverse mode wins exactly when there are many inputs and one output — the shape of every loss. Second, the algorithm is **mechanical**: given the local derivative of each basic operation, the framework builds and walks the graph automatically. This is why you call `loss.backward()` and never derive gradients by hand for a new architecture.

The same equations **vectorize over a mini-batch** at no extra cost. With B examples stacked as rows, $a^{(l-1)}$ is $B \times n_{in}$ and $\delta^{(l)}$ is $B \times n_{out}$. The weight gradient $a^{(l-1)\top}\delta^{(l)}$ is an $n_{in} \times n_{out}$ matrix that has already *summed* the per-example outer products, and the bias gradient sums $\delta^{(l)}$ down its rows. The cost of the whole scheme is memory: every activation must survive from the forward

pass until its adjoint is used. This is why training a deep network needs far more memory than running it — and why **gradient checkpointing**, which recomputes some activations during the backward pass instead of storing them, trades compute for memory in very deep models.

Backprop through common layers

Real networks are built from a small set of layer types, and reverse mode asks only one thing of each: its **vector–Jacobian product** (VJP). Write the adjoint of any quantity v as $\bar{v} = \partial L/\partial v$. Given the adjoint arriving from above, a layer must return two things — the adjoint of its input, to pass downward, and the adjoint of its parameters, to read off as gradients — *without ever forming the full Jacobian matrix*. Four rules cover almost everything you will meet.

Linear / matmul. Forward $z = aW + b$, where a is a $1\times n_{in}$ row and W is an $n_{in}\times n_{out}$ matrix. Given the output adjoint $\bar{z}$ (this is δ), the three VJPs are

$$\bar{a} = \bar{z}\,W^{T} \qquad \bar{W} = a^{T}\,\bar{z} \qquad \bar{b} = \Sigma_{batch}\,\bar{z}$$

These are exactly the recurrence and the weight/bias reads from the four equations above. A matmul routes the incoming adjoint through the *transpose* of the other factor, because the Jacobian of aW with respect to a is W^{T} and with respect to W is a^{T}. Get one of these transposes wrong and the shapes may still match while the gradient points in a useless direction.

Elementwise activation. Forward $a = g(z)$, applied component by component. Its Jacobian is diagonal, so the VJP reduces to a Hadamard (elementwise) product — no matrix at all:

$$\bar{z} = \bar{a} \odot g'(z)$$

The three common activations supply g' as follows: **ReLU** $g'(z) = 1[z > 0]$, a gate that copies the adjoint where $z > 0$ and sets it to zero elsewhere (at $z = 0$ we take the subgradient 0); **sigmoid** $g' = \sigma(z)(1 - \sigma(z)) = a(1 - a)$; and **tanh** $g' = 1 - \tanh^2(z) = 1 - a^2$. Sigmoid and tanh reuse the stored output a, and their slopes are bounded (≤ 0.25 and ≤ 1), so they can only *shrink* the signal. ReLU's gate leaves it unchanged where the unit is active — a first reason ReLU trains deep networks more easily (Chapter 10, initialization & the vanishing gradient).

Softmax + cross-entropy (fused). For a K-class classifier, the last two operations — the softmax $\hat{y} = \text{softmax}(z)$ and the cross-entropy $L = -\Sigma_k\, y_k \log \hat{y}_k$ against a one-hot target y — are differentiated *together*. The softmax's dense Jacobian and the $1 / \hat{y}$ from the log cancel, giving one of the cleanest results in the whole subject:

$$\delta = \partial L \,/\, \partial z = \hat{y} - y$$

the gradient at the logits is simply "prediction minus target." (The same form appears for a sigmoid output paired with binary cross-entropy — see Chapter 6, where these losses are derived from maximum likelihood.) Fusing the pair is also numerically safer than the two separate steps, because it never divides by a possibly tiny $\hat{y}_k$.

Convolution. A convolution is a *linear* operation — every output is the same kernel dotted with a shifted local patch — so it follows the matmul rule, using a structured matrix with shared weights. Its input adjoint is itself a convolution (the incoming adjoint against the flipped kernel, called a "transposed convolution"). Its kernel adjoint sums the patch-times-adjoint products over every spatial position the kernel visited: a shared weight fans out, so its gradients *add* — exactly the rule from the previous section. The details are worked out in Chapter 13 (convolutional neural networks).

Chaining these four VJPs in reverse order *is* the backward pass of a modern network. A framework stores one such rule per basic operation and composes them automatically.

Backpropagation in code

Here is the entire 2–2–1 network from the worked example — forward pass, all four backprop equations, and a finite-difference gradient check — in about a dozen lines of NumPy. Press **Run**; every printed number matches the hand calculation below to the last decimal.

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.set_printoptions(precision=4, suppress=True)
sig = lambda z: 1/(1+np.exp(-z))

x  = np.array([0.05, 0.10])                  # input  (row vector a^(0))
y  = 0.70                                     # target
```

```
W1 = np.array([[0.15, 0.25],                    # 2 inputs -> 2 hidden units
               [0.20, 0.30]])
b1 = np.array([0.35, 0.35])
W2 = np.array([[0.40], [0.50]])                 # 2 hidden -> 1 linear output
b2 = np.array([0.60])

def forward(W1, b1, W2, b2):                     # returns z1, a1, yhat, L
    z1 = x @ W1 + b1;  a1 = sig(z1)
    yhat = a1 @ W2 + b2                          # linear output, g' = 1
    return z1, a1, yhat, 0.5 * (yhat - y) ** 2

z1, a1, yhat, L = forward(W1, b1, W2, b2)
print("z1   =", z1, "  a1 =", a1)
print("yhat =", yhat, "  L =", L)

d2  = yhat - y                                   # delta^(2)  (linear g'=1)
gW2 = np.outer(a1, d2)                           # dL/dW2
d1  = (d2 @ W2.T) * a1 * (1 - a1)                # delta^(1)  (sigmoid g')
gW1 = np.outer(x, d1)                            # dL/dW1
print("dL/dW2 =", gW2.ravel(), "  dL/db2 =", d2)
print("dL/dW1 =", gW1.ravel(), "  dL/db1 =", d1)

# finite-difference check on the output weight W2[0]
e = 1e-5;  Wp = W2.copy(); Wp[0] += e;  Wm = W2.copy(); Wm[0] -= e
num = (forward(W1, b1, Wp, b2)[3] - forward(W1, b1, Wm, b2)[3]) / (2 * e)
print(f"gW2[0]: analytic {gW2[0,0]:.6f}   numeric {num[0]:.6f}")
```

The `analytic ≈ numeric` agreement to six decimals is the everyday test that a hand-written gradient is correct: the central difference $(L(\theta+\varepsilon) - L(\theta-\varepsilon))/2\varepsilon$ approximates the true slope with error proportional to ε^2. A modern framework runs this same backward pass for you the moment you call `loss.backward()`. Here is the same network in PyTorch, with weights set so the gradients reproduce the ones above (note that `nn.Linear` stores the weight transposed, as W^T):

PyTorch copy → Colab

```
import torch, torch.nn as nn

x = torch.tensor([[0.05, 0.10]])                 # batch of one
y = torch.tensor([[0.70]])
net = nn.Sequential(nn.Linear(2, 2), nn.Sigmoid(), nn.Linear(2, 1))
with torch.no_grad():                            # plant the chapter's
weights
```

```
    net[0].weight.copy_(torch.tensor([[0.15, 0.20], [0.25, 0.30]]))  # =
W1^T
    net[0].bias.copy_(torch.tensor([0.35, 0.35]))
    net[2].weight.copy_(torch.tensor([[0.40, 0.50]]))                # =
W2^T
    net[2].bias.copy_(torch.tensor([0.60]))

yhat = net(x)
loss = 0.5 * (yhat - y).pow(2).sum()             # MSE, batch of one
loss.backward()                                  # one call runs all of
backprop
print(yhat.item(), loss.item())                  # 1.1358 , 0.0949
print(net[2].weight.grad)                         # -> [[0.2585, 0.2601]]
print(net[0].weight.grad)                         # hidden-layer gradients
(W1^T layout)
```

In this app's engine

The Backprop step-through lab animates exactly this graph on the 2–2–1 network below: tap "forward" to fill in the blue values, then step the amber δ's back one node at a time and watch each gradient appear. Because a wrong sign or a missing g′ can quietly break training, the app also includes **44 gradient-checking tests** that compare every analytic gradient against a central finite difference, $(L(\theta+\varepsilon) - L(\theta-\varepsilon)) \,/\, 2\varepsilon$, whose error shrinks like ε^2. If backprop and the numerical estimate disagree by more than a tiny tolerance, the test fails — the standard way to trust a hand-written gradient.

Gradient checking in practice

A single wrong sign or a forgotten g′ corrupts training *silently* — there is no error message, just a model that never quite learns. So the one essential habit, when you write a backward pass by hand, is to check it numerically before you trust it. The tool for this is the **central difference**, applied to one scalar parameter θ at a time:

$$\partial L \,/\, \partial\theta \approx \frac{L(\theta + \varepsilon) - L(\theta - \varepsilon)}{2\varepsilon}, \quad \varepsilon \approx 10^{-5}$$

whose error shrinks like $O(\varepsilon^2)$ — far smaller than the one-sided formula $(L(\theta+\varepsilon) - L(\theta))/\varepsilon$ with its $O(\varepsilon)$ error. Compare the estimate to your analytic gradient using a scale-free **relative error**, never an absolute one:

$$\text{rel} = \frac{|\, g_{\text{analytic}} - g_{\text{numeric}} \,|}{|\, g_{\text{analytic}} \,| + |\, g_{\text{numeric}} \,| + 10^{-8}}$$

and read it on a log scale. A relative error at or below about 10^{-7} means the gradient is almost certainly right; about 10^{-4} is borderline, acceptable only if a kink such as a ReLU sits nearby; anything $\geq 10^{-2}$ is a bug. The 10^{-8} in the denominator simply prevents division by zero when both gradients are essentially zero.

The check catches exactly the family of silent mistakes that make backprop easy to get slightly wrong: a **dropped g'** (leaving the Hadamard factor out of the recurrence), a **wrong transpose** (multiplying by W instead of W^T, so the shapes still match but the routing is wrong), a **sign error** ($\delta = y - \hat{y}$ instead of $\hat{y} - y$), an off-by-one in a batch sum, or a swapped axis. Each shows up as a relative error near 1, and the parameter at fault points straight to the layer to inspect.

Three rules of thumb keep it reliable. **Check on a small network** — two or three layers, a few units, a handful of examples — because the procedure costs one pair of forward passes *per parameter* and is meant to test the *equations*, not the scale: once the math is right, it stays right at any width. Use **double precision**, because in float32 the ε^2 truncation error and floating-point round-off work against each other and you cannot reach 10^{-7}. And **avoid non-differentiable kinks** (sharp corners): at a ReLU's $z = 0$ the two-sided difference crosses the corner and will disagree with whatever subgradient you chose, so test at points away from it. Once the check passes, switch it off — it is far too slow for training — and let the analytic pass run. This is exactly the discipline behind this app's 44 gradient-checking tests.

Worked example — one full pass of the Backprop lab (2–2–1)

Inputs $x = [0.05, 0.10]$, target $y = 0.70$. Sigmoid hidden layer, linear output, MSE loss $L = \tfrac{1}{2}(\hat{y} - y)^2$. Parameters $W^{(1)} = \begin{bmatrix} 0.15 & 0.25 \\ 0.20 & 0.30 \end{bmatrix}$, $b^{(1)} = [0.35, 0.35]$, $W^{(2)} = [0.40, 0.50]^T$, $b^{(2)} = 0.60$.

1. Hidden pre-activations: $z^{(1)}_1 = 0.15 \cdot 0.05 + 0.20 \cdot 0.10 + 0.35 = \mathbf{0.3775}$; $z^{(1)}_2 = 0.25 \cdot 0.05 + 0.30 \cdot 0.10 + 0.35 = \mathbf{0.3925}$.
2. Hidden activations: $a^{(1)}_1 = \sigma(0.3775) = 1 / (1 + e^{-0.3775}) = \mathbf{0.5933}$; $a^{(1)}_2 = \sigma(0.3925) = \mathbf{0.5969}$.
3. Output (linear): $\hat{y} = 0.40 \cdot 0.5933 + 0.50 \cdot 0.5969 + 0.60 = 0.23731 + 0.29844 + 0.60 = \mathbf{1.1358}$.
4. Loss: $L = \tfrac{1}{2}(1.1358 - 0.70)^2 = \tfrac{1}{2}(0.43575)^2 = \mathbf{0.0949}$.
5. Output error (linear $g' = 1$): $\delta^{(2)} = \hat{y} - y = \mathbf{0.4358}$.
6. Output-weight gradients: $\partial L / \partial w^{(2)}_1 = a^{(1)}_1 \cdot \delta^{(2)} = 0.5933 \cdot 0.43575 = \mathbf{0.2585}$; $\partial L / \partial w^{(2)}_2 = 0.5969 \cdot 0.43575 = \mathbf{0.2601}$; $\partial L / \partial b^{(2)} = \delta^{(2)} = \mathbf{0.4358}$.
7. Hidden error: $\sigma'(z^{(1)}) = a(1-a) = [0.2413, 0.2406]$; $\delta^{(1)}_1 = \delta^{(2)} \cdot w^{(2)}_1 \cdot 0.2413 = 0.43575 \cdot 0.40 \cdot 0.2413 = \mathbf{0.0421}$; $\delta^{(1)}_2 = 0.43575 \cdot 0.50 \cdot 0.2406 = \mathbf{0.0524}$.
8. Hidden-weight gradients $\partial L / \partial W^{(1)}_{ij} = x_i \cdot \delta^{(1)}_j$: $[[0.05 \cdot 0.0421, 0.05 \cdot 0.0524], [0.10 \cdot 0.0421, 0.10 \cdot 0.0524]] = [[\mathbf{0.00210}, \mathbf{0.00262}], [\mathbf{0.00421}, \mathbf{0.00524}]]$; $\partial L / \partial b^{(1)} = \delta^{(1)} = [0.0421, 0.0524]$.
9. One update of $w^{(2)}_1$ with $\eta = 0.5$: $0.40 - 0.5 \cdot 0.2585 = 0.40 - 0.12926 = \mathbf{0.2707}$.

What the numbers mean: the output weights feed the large raw error $\delta^{(2)} = 0.44$ directly, so they get gradients near 0.26 — large, and they move the most. The hidden weights get gradients near 0.003 — more than sixty times smaller — because the error was reduced

by $\sigma' \approx 0.24$ *and* by the tiny inputs 0.05–0.10. Central-difference gradient checking on $w_1^{(2)}$ returns 0.2585, matching the analytic value to five decimals — so the network is differentiating correctly.

Assumptions & limitations

Backpropagation needs every operation in the graph to be differentiable (or to have a usable sub-gradient, as ReLU does at 0); a hard non-differentiable step blocks it completely. It computes gradients only — it is not the learning rule itself, which is gradient descent (Chapter 7) acting on those numbers. Its memory cost is real: storing all activations for the backward pass is the main reason very deep networks strain GPU memory, which checkpointing helps reduce. And the same recurrence that makes it efficient also lets error signals shrink or explode as they pass through many layers of g' factors — the vanishing/exploding-gradient problem covered in Chapter 10 (initialization). Backprop is exact; whether the resulting gradient is *usable* depends on architecture and initialization.

Why & when this matters

Backpropagation is the reason deep learning is practical: it makes the cost of all gradients grow linearly with network size, instead of exploding combinatorially, so a billion-parameter model can still be trained. You rarely write it yourself — autodiff frameworks build the graph and run this algorithm for you — but understanding it is how you diagnose training. Gradients that vanish point to saturating activations or poor initialization; gradients that explode point to weights that are too large or a missing normalization. When a network will not learn, the flow of δ through this graph is where you look.

In the real world

Reverse-mode autodiff is the engine under every major deep-learning framework — PyTorch's `autograd`, TensorFlow, and JAX's `grad` all build the computational graph and run exactly this backward sweep. It scales to the largest models on Earth: the gradient of a single scalar loss with respect to the hundreds of billions of parameters in a GPT-class language model is computed by the algorithm in this chapter, spread across thousands of GPUs. The same machinery now reaches beyond neural networks — differentiable physics engines, molecular-dynamics simulators, and probabilistic-programming languages all rely on reverse-mode AD to optimize through long numerical pipelines.

Exercises

1. Redo the output layer of the worked example, but assume the output neuron uses a **sigmoid** activation with the same pre-activation $z^{(2)} = 1.1358$, instead of a linear one. Compute $\hat{y}$, the loss L, and the output error $\delta^{(2)} = (\hat{y} - y)\cdot\sigma'(z^{(2)})$. MATH

2. Explain, using the backprop equations, why the hidden-weight gradient $\partial L/\partial W^{(1)}_{11} = 0.00210$ is about 120× smaller than the output-weight gradient $\partial L/\partial w_1^{(2)} = 0.2585$. MATH

3. Modify the runnable cell to process a **batch of two** examples at once and verify that the weight gradient is the batch sum $a^{(0)\mathrm{T}}\delta^{(1)}$, not a per-example value. CODE

4. Add a finite-difference gradient check for the **hidden** weight $W^{(1)}_{11}$ and confirm it matches the analytic 0.00210. CODE

5. Why must the forward-pass activations be *stored* for the backward pass? What is the trade-off made by gradient checkpointing? CONCEPT

6. A student says "backpropagation trains the network." Correct the statement precisely. CONCEPT

7. Consider a **deep linear** network with no activations, $g(z) = z$. Write the δ recurrence, show the input-layer gradient is a product of all the weight matrices, and explain what this implies for very deep networks even though the function they compute is only linear. MATH

8. Derive the fused **softmax + cross-entropy** gradient. With $\hat{y}_k = e^{z_k} / \Sigma_j e^{z_j}$ and cross-entropy $L = -\Sigma_k y_k \log \hat{y}_k$ for a one-hot target y, show that $\partial L / \partial z_i = \hat{y}_i - y_i$, then evaluate it for logits $z = [2, 1, 0]$ whose true class is 0. MATH

9. Hand-backprop through a **ReLU** layer, then verify it in code. A hidden layer has pre-activations $z = [-1.0, 2.0, 0.5, -3.0]$ with ReLU outputs $a = \max(0, z)$; the adjoint arriving from above is $a = [0.4, -0.2, 0.7, 0.1]$. Compute the input adjoint z, and say what happens to the two units with $z < 0$. CODE

Momentum, RMSProp & Adam

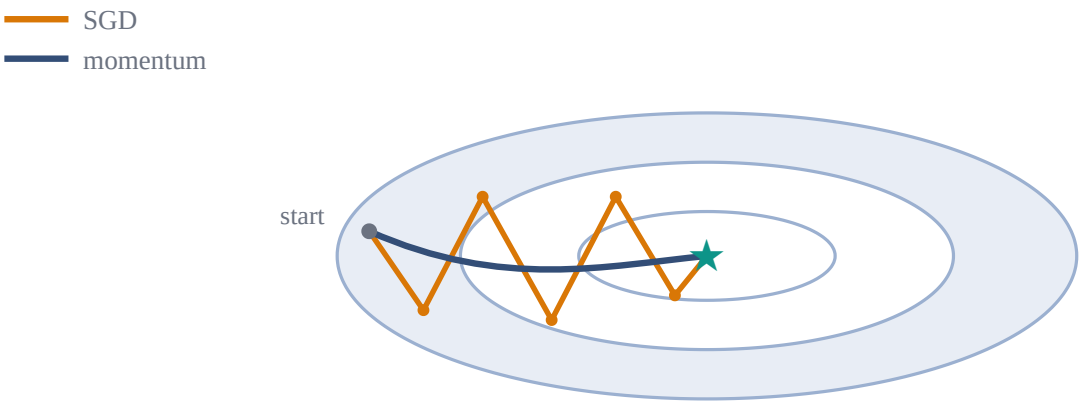
> *Plain gradient descent (chapter 7) walks straight downhill. But on the stretched, curved loss surfaces of real networks, that straight walk becomes a frustrating zig-zag that barely moves forward. This chapter builds the three fixes every practitioner uses — **momentum**, **RMSProp** and **Adam** — one idea at a time, and shows why Adam takes a step of size η no matter how large or small the gradient is.*

Fundamental concept

Imagine a long, narrow valley — a ravine with steep side walls and a floor that slopes only gently toward the exit. A blindfolded hiker who always steps in the steepest downhill direction faces a trap: the steepest direction points almost straight across the valley, into the far wall, not along the floor toward the goal. So the hiker swings from wall to wall, crossing and re-crossing the floor, moving forward only a little with each bounce. That is exactly what plain **stochastic gradient descent** (SGD) does when the loss surface is **ill-conditioned** — much steeper in some directions than others. Progress along the useful, shallow direction is held back by the fear of overshooting the steep one.

Momentum fixes this with a physical idea: give the hiker **inertia**, like a heavy ball rolling down the valley. Side-to-side pushes now alternate in sign and cancel out, while the steady, small pull along the floor adds up step after step. The ball stops bouncing and rolls straight to the exit.

RMSProp solves the same problem in a different way: instead of one shared step size, it gives **every parameter its own**. It shrinks the step on axes whose gradients are large and stretches it on axes whose gradients are tiny. That rescales the ravine into a round bowl where plain descent already works well. **Adam** — by far the most-used optimizer — simply does both at once, plus a small start-up correction. The rest of the chapter makes each of these precise.



An ill-conditioned "ravine": ellipses are loss contours, the teal star is the minimum. SGD (amber) wastes motion bouncing across the steep walls. Momentum (blue) averages those bounces away and rolls along the shallow floor. Reproduce this with the Gradient-descent lab's ravine preset.

Governing equations

Write $g_t = \nabla_\theta L$ for the gradient of the loss (chapter 6) at step t. Plain SGD is the baseline:

$$\theta_{t+1} = \theta_t - \eta \cdot g_t$$

Momentum keeps a running velocity v that blends the new gradient with the old direction:

$$v_t = \beta \cdot v_{t-1} + g_t\,, \quad \theta_{t+1} = \theta_t - \eta \cdot v_t$$

AdaGrad gives every parameter its own step size. It adds up that parameter's squared gradients in r and divides by the root. Large-gradient axes are damped, rarely-seen axes amplified:

$$r_t = r_{t-1} + g_t^{\,2}\,, \quad \theta_{t+1} = \theta_t - \eta \cdot \frac{g_t}{\sqrt{r_t} + \varepsilon}$$

RMSProp tracks a running mean of squared gradients s and divides each step by its root, so every axis is normalized to a comparable scale (and, unlike AdaGrad, the step never fades to zero):

$$s_t = \rho \cdot s_{t-1} + (1-\rho) \cdot g_t^{\,2}, \quad \theta_{t+1} = \theta_t - \eta \cdot \frac{g_t}{\sqrt{s_t} + \varepsilon}$$

Adam keeps both a first moment m (like momentum) and a second moment v (like RMSProp), corrects each for starting at zero, then combines them:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\, g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2)\, g_t^{\,2}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^{\,t}}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^{\,t}}, \quad \theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

θ	the parameters (weights and biases) being optimized
g_t	gradient $\nabla_\theta L$ of the loss at step t — the steepest-ascent direction (backpropagation, chapter 8, supplies it)
η	learning rate (step size); a small positive scalar
v_t	velocity — an exponentially weighted running sum of past gradients (momentum)
β	momentum coefficient, $0 \le \beta < 1$ (typically 0.9); how much of the old velocity is kept
s_t	running mean of squared gradients (RMSProp), one value per parameter
ρ	RMSProp decay rate for s (typically 0.9)
r_t	AdaGrad's running *sum* of squared gradients, one value per parameter
m_t, v_t	Adam's first and second moment estimates (mean and uncentered variance of the gradient)
β_1, β_2	Adam decay rates, default 0.9 and 0.999
$\hat{m}_t$, $\hat{v}_t$	bias-corrected moments, rescaled to remove the zero-start bias
ε	tiny constant ($\approx 10^{-8}$) that prevents division by zero

How it works

Why does momentum accelerate? Suppose the gradient points steadily in one direction with roughly constant size g. The velocity then approaches a fixed point where it stops growing, $v_\infty = \beta v_\infty + g$, which solves to a geometric-series sum:

$$v_\infty = \frac{g}{1-\beta} \;\Rightarrow\; \text{effective step} = \frac{\eta}{1-\beta} \cdot g$$

With $\beta = 0.9$ the effective step is $1/(1-0.9) = \mathbf{10}\times$ the plain SGD step *along directions the gradient agrees on*. Along directions that keep flipping sign — the ravine walls — the terms cancel instead of adding, so those steps stay small. Momentum thus amplifies agreement and damps disagreement, which is exactly the ravine cure. **Nesterov** momentum improves on this by measuring the gradient at the look-ahead point $\theta - \eta\beta v$ rather than at θ, giving the ball a chance to "see" the wall coming and brake earlier.

Why the bias correction in Adam? Both m and v start at zero, so early on they are pulled toward zero and underestimate the true gradient statistics. Unrolling the recursion for a constant-expectation gradient gives $\mathbb{E}[m_t] = (1 - \beta_1^{\,t})\cdot\mathbb{E}[g]$. The factor $(1 - \beta_1^{\,t})$ is the warm-up factor. Dividing by it exactly removes the bias:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^{\,t}} \;\Rightarrow\; \mathbb{E}[\hat{m}_t] = \mathbb{E}[g]$$

At $t = 1$ the raw $m_1 = 0.1\cdot g$ is ten times too small. Dividing by $(1 - 0.9) = 0.1$ restores it. The same correction, with β_2, fixes v. As t grows, $\beta_1^{\,t} \to 0$ and the correction fades to 1. The combined ratio $\hat{m}/(\sqrt{\hat{v}} + \varepsilon)$ has a special property: when the gradient is steady, $\hat{m} \approx g$ and $\sqrt{\hat{v}} \approx |g|$, so the ratio $\approx \pm 1$ and the step $\approx \eta$ *regardless of the gradient's size*. Adam self-scales.

Momentum as a moving average, and Nesterov look-ahead

The velocity recursion $v_t = \beta \cdot v_{t-1} + g_t$ has a second meaning worth making clear. Unroll it and it becomes a weighted sum of *every* past gradient with geometrically fading weights:

$$v_t = g_t + \beta\, g_{t-1} + \beta^2 g_{t-2} + \beta^3 g_{t-3} + \cdots = \Sigma_{k=0}^{\,t-1} \beta^k\, g_{t-k}$$

A gradient from k steps ago still counts, scaled by β^k, so momentum is an **exponential moving average** of the gradient over an effective window of about $1/(1-\beta)$ recent steps — roughly 10 at $\beta = 0.9$. That is the same $1/(1-\beta)$ as the speed-up above: the ball's memory length and its acceleration are one number. Some texts instead write the **normalized** form $v_t = \beta\, v_{t-1} + (1-\beta)$

g_t, whose weights sum to 1 so v is a true running *mean* with the *same* effective step η. The two differ only by the constant (1−β) absorbed into η, and Adam uses the normalized form for both moments.

The name comes from Polyak's **heavy ball**: θ is a ball's position on the loss surface, $-g_t$ the downhill force, β how much momentum it keeps (so 1−β acts like friction), and η the time step. A heavy, low-friction ball (β near 1) moves smoothly through noise but turns slowly and can overshoot a narrow minimum. A light, high-friction ball (small β) reacts at once but keeps SGD's zig-zag. β = 0.9 is the usual compromise.

Nesterov's accelerated gradient improves the heavy ball with one nearly-free change: measure the gradient not where you stand but where your momentum is *about* to put you. In this chapter's convention, form the look-ahead point $\tilde{\theta}_t = \theta_t - \eta\,\beta\,v_{t-1}$, take the gradient there, and only then roll the velocity forward:

$$\tilde{g}_t = \nabla_\theta L(\tilde{\theta}_t)\,, \quad v_t = \beta\,v_{t-1} + \tilde{g}_t\,, \quad \theta_{t+1} = \theta_t - \eta\,v_t$$

Because the gradient is measured a step ahead, a ball racing toward the far wall feels its push *before* arriving and brakes early, so it overshoots less and stays close to the floor. On smooth convex problems this lifts the worst-case rate from O(1/t) to $O(1/t^2)$. In deep learning the gain is small but almost free, which is why `momentum = 0.9, nesterov = True` is a common default.

From AdaGrad to RMSProp: a step size per parameter

Momentum smooths the step's *direction*. The adaptive family rescales its *size*, one parameter at a time. **AdaGrad** (Duchi, Hazan & Singer, 2011) keeps a running *sum* of squared gradients and divides by its root — the rule above, $r_t = r_{t-1} + g_t^2$ then a move of $\eta \cdot g_t / (\sqrt{r_t} + \varepsilon)$. A coordinate that has seen large gradients builds a large r and is slowed down, while a rarely-touched one keeps a small r and moves freely. This is exactly right for sparse features, which made AdaGrad a favorite for text and embeddings.

But it has a hidden flaw that is fatal for deep learning. Because r only ever *adds* non-negative numbers, it grows without limit, so the effective learning rate $\eta / (\sqrt{r_t} + \varepsilon)$ slides steadily toward zero. Hold the gradient steady at size g and after t steps $r_t = t \cdot g^2$, which makes the step

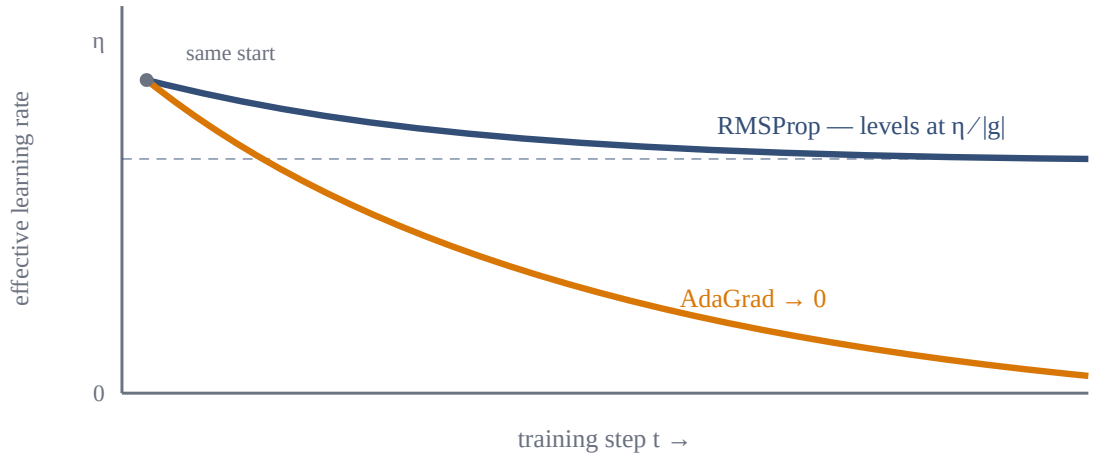
$$\eta \cdot \frac{g}{\sqrt{t \cdot g^2}} = \frac{\eta}{\sqrt{t}} \cdot \operatorname{sign}(g) \longrightarrow 0$$

The step shrinks like $1/\sqrt{t}$ *whether or not the parameter has arrived*. On a convex problem you converge before the denominator grows large, so that decay is a free, automatic annealing. On the long, non-convex trajectory of a neural network it means AdaGrad often stalls with the loss still high, held back by a sum that never forgets.

RMSProp (Tieleman & Hinton, 2012) fixes this with a single-symbol change: replace the unlimited *sum* with an exponential moving *average* of squared gradients — the same averaging trick momentum uses, now applied to g^2:

$$s_t = \rho \cdot s_{t-1} + (1 - \rho) \cdot g_t^{\,2}, \quad \theta_{t+1} = \theta_t - \eta \cdot \frac{g_t}{\sqrt{s_t} + \varepsilon}$$

Because s forgets the distant past at rate ρ, on a steady gradient it settles to $s \to g^2$ instead of growing, so the effective rate levels off near $\eta / |g|$ instead of collapsing, and progress continues. RMSProp is AdaGrad with a finite memory, and that finite memory is the whole fix. (Batch and layer normalization, chapter 11, attack the same per-axis scale mismatch from the architecture side.)

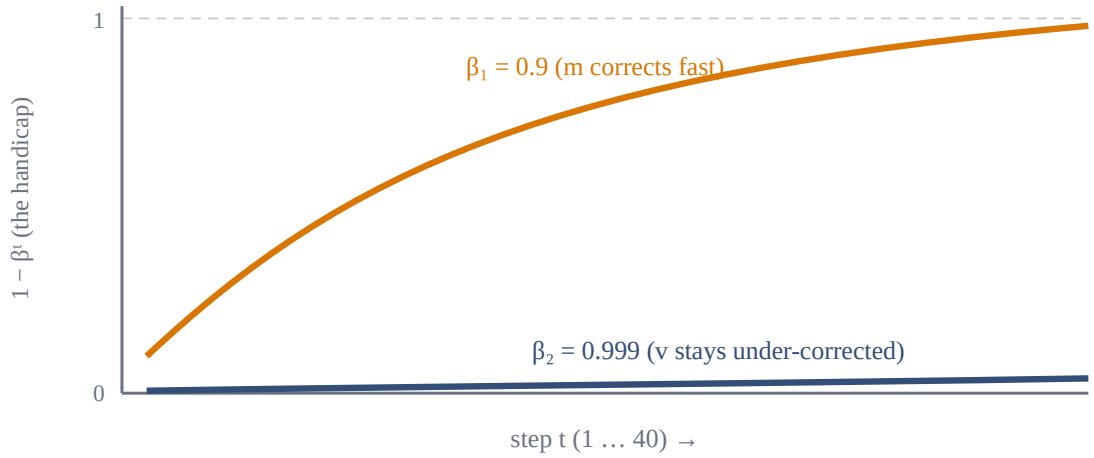


Why AdaGrad stalls. Both methods begin with the same step, but AdaGrad divides by the root of an ever-growing sum of squared gradients, so its effective learning rate decays like $\eta/\sqrt{t}$ toward zero (amber) and training freezes before it finishes. RMSProp divides by a decaying average, so the denominator stops growing and the rate settles at a positive plateau near $\eta/|g|$ (blue). This is the one change that turns AdaGrad into a usable optimizer.

Assembling Adam, and two refinements

With momentum and RMSProp in hand, **Adam** (Kingma & Ba, 2015) is little more than combining them. Momentum's moving average gives the first moment m, the smoothed *direction*. RMSProp's gives the second moment v, the per-parameter *scale*. After bias-correcting both, you divide one by the root of the other. The single update $\theta_{t+1} = \theta_t - \eta \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$ thus gets both momentum's noise-averaging *and* RMSProp's axis-equalizing at once, which is why it trains so many architectures at the single default $\eta = 10^{-3}$.

The bias correction is not a minor detail but the difference between a safe first step and a disastrous one: at t = 1 the raw v has the factor $1 - \beta_2 = 0.001$, so it is a thousandfold too small, its root thirtyfold too small, and an uncorrected step $\eta \cdot m / (\sqrt{v} + \varepsilon)$ would be huge. Dividing by $1 - \beta_2^{\,t}$ restores the true scale from the first iteration. The figure shows why the second moment needs that help far longer than the first.



Why Adam divides by $1 - \beta^t$. Each moving average starts at zero, so a raw moment is only the fraction $1 - \beta^t$ of the truth and must be divided by it to be restored. For the first moment ($\beta_1 = 0.9$, amber) the factor reaches nearly 1 within about twenty steps, so m self-corrects quickly. For the second ($\beta_2 = 0.999$, blue) it is still only ≈ 0.04 at $t = 40$, so an uncorrected v would be twentyfold too small and the first Adam steps would explode.

Two refinements deserve names. **AMSGrad** (Reddi, Kale & Kumar, 2018) closes a gap in Adam's original convergence proof: on some problems v *falls* just after a rare informative gradient, letting the step spike and the iterates diverge. It forces the denominator to be non-decreasing by carrying the running maximum $\hat{v}^{max}_t = \max(\hat{v}^{max}_{t-1}, \hat{v}_t)$ and dividing by $\sqrt{\hat{v}^{max}_t}$. **Nadam** (Dozat, 2016) puts Nesterov's look-ahead into Adam's first moment. Both are small, safe changes. Plain Adam and AdamW stay the defaults. You only reach for these refinements when Adam wobbles or diverges late in a run.

Decoupled weight decay: AdamW

Weight decay gently pulls every parameter toward zero each step — a form of L2 regularization (chapter 12) that improves generalization. With plain SGD its two standard forms are the same: adding a penalty $\lambda/2\|\theta\|^2$ to the loss contributes $\lambda\theta$ to the gradient, and subtracting $\eta \cdot \lambda\theta$ from θ each step is the very same thing. With **Adam they diverge**, and the difference matters. If the $\lambda\theta$ term is added to the gradient g, it then passes through Adam's per-parameter denominator $\sqrt{\hat{v}} + \varepsilon$, so a weight with a large gradient history is decayed *less* and one with a

small history is decayed *more*. The strength of the regularization ends up depending on each weight's gradient scale, which is not what "shrink every weight toward zero" was meant to mean.

AdamW (Loshchilov & Hutter, 2019) fixes this by *decoupling* the decay from the adaptive step: run ordinary Adam on the data gradient, then shrink the weights directly.

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} - \eta \cdot \lambda \cdot \theta_t$$

Now every weight is scaled down by the same fraction $\eta\lambda$ regardless of its gradient, exactly as intended. This one change measurably improves the generalization of adaptive optimizers, costs nothing but a single elementwise multiply, and is the default for training transformers and large vision models.

Warmup, schedules, and the SGD–Adam trade-off

Learning-rate warmup increases η linearly from near zero over the first few hundred to few thousand steps, then continues normally. The reason is visible in Adam's second moment: $\hat{v}$ is an average over only a few gradients early on, so its estimate of the gradient scale is noisy, and a full-size adaptive step taken on that unreliable estimate can throw the weights into a bad region training never recovers from. Warmup lets the moment estimates settle before large steps are trusted. It is essential for transformers, whose deep stacks are especially fragile in the first thousands of updates.

After warmup a **schedule** usually *decays* η over training — step drops, or a smooth **cosine** anneal $\eta_t = \tfrac{1}{2}\eta_0(1 + \cos(\pi t/T))$ — because large steps explore early while small steps settle late. A good schedule routinely helps more than changing the optimizer does. Choosing the optimizer, its learning rate, and its schedule together is one ingredient of the full training recipe (chapter 25).

Finally, the honest trade-off: **Adam trains faster, but a well-tuned SGD+momentum often generalizes better.** Adam's per-parameter rescaling races down the training loss but tends to settle in *sharper* minima that test slightly worse. SGD's uniform step size drifts toward *flatter*

minima that are more robust to the shift between training and test distributions. This is why many state-of-the-art image classifiers are still trained with SGD+momentum and a hand-tuned schedule, while nearly all large language models use AdamW. Their sparse, high-dimensional, wildly-scaled gradients are exactly where Adam's robustness outweighs that last fraction of a percent.

Optimizers in code

The cleanest way to see the difference between these rules is to watch the *step* each one takes when the gradient is held constant at g = 0.4. Press **Run**:

Python · NumPy ▶ runnable in the app

```
import numpy as np
g, eta = 0.4, 0.1                    # constant gradient, learning rate
v = s = m = vv = 0.0                  # every optimizer's state starts at 0
b1, b2, rho, eps = 0.9, 0.999, 0.9, 1e-8
print(" t |   SGD   Moment  RMSProp   Adam")
for t in range(1, 6):
    sgd = eta * g
    v   = 0.9 * v + g;                mom = eta * v
    s   = rho * s + (1 - rho) * g * g; rms = eta * g / (np.sqrt(s) + eps)
    m   = b1 * m + (1 - b1) * g
    vv  = b2 * vv + (1 - b2) * g * g
    mh  = m / (1 - b1 ** t);  vh = vv / (1 - b2 ** t)
    adam = eta * mh / (np.sqrt(vh) + eps)
    print(f"{t:2d} | {sgd:.4f}  {mom:.4f}  {rms:.4f}  {adam:.4f}")
```

Read the columns: SGD takes the same 0.04 forever. Momentum climbs from 0.04 toward its ceiling $\eta \cdot g/(1-\beta) = 0.4$. RMSProp starts *too large* (0.32) because it has no bias correction and its running mean of g^2 is still near zero, then settles toward $\eta = 0.1$. Adam holds a steady $0.1 = \eta$ from the first step, thanks to the corrected ratio $\hat{m}/\sqrt{\hat{v}} \approx 1$. Now the practical result on the ravine of the figure — how many iterations each needs to reach the minimum of $L = \tfrac{1}{2}(20w_0^2 + w_1^2)$:

Python · NumPy ▶ runnable in the app

```
import numpy as np
A = np.array([20.0, 1.0])             # curvatures: steep axis, shallow
axis
```

```
grad = lambda w: A * w                  # L = 0.5*(20*w0^2 + 1*w1^2), a
ravine
tol, cap = 1e-3, 5000
w0 = np.array([1.0, 1.0])

w = w0.copy()                           # --- plain SGD ---
for t in range(1, cap + 1):
    w = w - 0.02 * grad(w)
    if np.linalg.norm(w) < tol: break
print("SGD      :", t, "iters")

w = w0.copy(); v = np.zeros(2)          # --- Momentum ---
for t in range(1, cap + 1):
    v = 0.9 * v + grad(w); w = w - 0.02 * v
    if np.linalg.norm(w) < tol: break
print("Momentum :", t, "iters")

w = w0.copy(); m = np.zeros(2); s = np.zeros(2)   # --- Adam ---
for t in range(1, cap + 1):
    g = grad(w); m = 0.9*m + 0.1*g; s = 0.999*s + 0.001*g*g
    w = w - 0.1 * (m/(1-0.9**t)) / (np.sqrt(s/(1-0.999**t)) + 1e-8)
    if np.linalg.norm(w) < tol: break
print("Adam     :", t, "iters")
```

SGD needs hundreds of iterations, held back by the shallow axis it can only creep along. Momentum and Adam reach the same tolerance about four times faster by adding up that gentle slope. In PyTorch you never write the update by hand. You pick an optimizer and call `.step()`. Switching SGD for Adam or AdamW is a one-line change:

PyTorch copy → Colab

```
import torch

w = torch.tensor([1.0, 1.0], requires_grad=True)
A = torch.tensor([20.0, 1.0])                # ravine curvatures
def loss(w): return 0.5 * (A * w * w).sum()

opt = torch.optim.Adam([w], lr=0.1)          # try SGD([w], lr=0.02,
momentum=0.9)
for step in range(200):
    opt.zero_grad()
    loss(w).backward()                       # autograd fills w.grad
```

```
    opt.step()                                  # one update rule applied
print(w.detach())                               # -> close to [0., 0.]

# AdamW: decoupled weight decay is a drop-in replacement
opt = torch.optim.AdamW([w], lr=0.1, weight_decay=0.01)
```

In this app's engine

The Playground's optimizer selector offers SGD, Momentum, RMSProp and Adam with editable η, β, β_1, β_2 and ε. The Gradient-descent lab draws the trajectory on the same ravine of the figure above so you can watch SGD bounce and momentum glide. The Math Log prints m_t, v_t, their corrected forms and the resulting step for a chosen weight — the same quantities the worked example computes below. Learning rates here behave in the usual way: Adam likes η in the 10^{-3}…3×10^{-2} range, while SGD+momentum often wants a larger η.

Worked example — two steps on one weight, gradient fixed at g = 0.4

To compare the three optimizers clearly, hold the gradient constant at g = 0.4 and use η = 0.1 (Adam: β_1=0.9, β_2=0.999, ε=10^{-8}). All moments start at 0.

1. **SGD.** Each step moves η·g = 0.1·0.4 = **0.04**. After two steps the weight has moved 0.08 — the same amount both times.
2. **Momentum** (β=0.9). Step 1: v = 0.9·0 + 0.4 = 0.4, move = 0.1·0.4 = **0.04**. Step 2: v = 0.9·0.4 + 0.4 = 0.76, move = 0.1·0.76 = **0.076**. The step is already growing. Its ceiling is η·g/(1−β) = 0.1·4 = 0.4, ten times the SGD step.
3. **Adam**, step 1: m = 0.1·0.4 = 0.04, v = 0.001·0.16 = 0.00016. Correct: $\hat{m}$ = 0.04/(1−0.9) = 0.4, $\hat{v}$ = 0.00016/(1−0.999) = 0.16. Step = 0.1·0.4/($\sqrt{0.16}$ +ε) = 0.1·0.4/0.4 = **0.1**.
4. **Adam**, step 2: m = 0.9·0.04 + 0.1·0.4 = 0.076, v = 0.999·0.00016 + 0.001·0.16 = 0.00031984. Correct: $\hat{m}$ = 0.076/(1−0.81) = 0.4, $\hat{v}$ = 0.00031984/(1−0.998001) = 0.16. Step = 0.1·0.4/0.4 = **0.1** again.

5. **Meaning.** Adam moved 0.1 both times — exactly η — because $\hat{m}/\sqrt{\hat{v}} = 0.4/0.4 = 1$. Had the gradient been 0.004 or 40 instead of 0.4, Adam would *still* step ≈0.1, while SGD would crawl at 0.0004 or leap by 4. That scale-invariance is why Adam trains so many different networks with the single default $\eta = 10^{-3}$. Try it in *Practice → One Adam update by hand*.

Worked example — one Adam step on two parameters at once

Adam's per-parameter scaling is easiest to see on a gradient whose two components differ sharply. Take g = [1.0, 0.1] — a steep axis and a shallow one, ten to one — with η = 0.1, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$, and m = v = 0.

1. **First moment** $m = (1 - \beta_1)\cdot g = 0.1\cdot[1.0, 0.1] = [\mathbf{0.100}, \mathbf{0.010}]$.
2. **Second moment** $v = (1 - \beta_2)\cdot g^2 = 0.001\cdot[1.00, 0.01] = [\mathbf{0.00100}, \mathbf{0.0000100}]$.
3. **Bias-correct** at t = 1 (divide m by $1 - \beta_1 = 0.1$ and v by $1 - \beta_2 = 0.001$): $\hat{m} = [1.0, 0.1]$, $\hat{v} = [1.00, 0.0100]$.
4. **Roots:** $\sqrt{\hat{v}} = [1.0, 0.1]$ — equal to |g| componentwise, as it must be at the first step.
5. **Update:** $\Delta\theta = -\eta \cdot \hat{m}/(\sqrt{\hat{v}} + \varepsilon) = -0.1\cdot[1.0/1.0, 0.1/0.1] = [-\mathbf{0.10}, -\mathbf{0.10}]$.
6. **Meaning.** Both axes move by the same 0.10 though their gradients differ tenfold — Adam has *equalized* the ravine. Plain SGD would step η·g = [0.10, 0.010], crawling ten times slower on the shallow axis that actually leads to the minimum. This equal step is the algebra behind the smooth blue glide in the figure at the top of the chapter.

Worked example — one AdamW step vs coupled L2

Reuse the first Adam step above: gradient $g = 0.4$, $\eta = 0.1$, so the adaptive part $\hat{m}/(\sqrt{\hat{v}} + \varepsilon) = 1$ and the data step is $\eta \cdot 1 = 0.1$. Add weight decay $\lambda = 0.01$ to a weight currently at $\theta = 0.5$.

1. **AdamW (decoupled).** $\theta \leftarrow 0.5 - 0.1 \cdot 1 - 0.1 \cdot 0.01 \cdot 0.5 = 0.5 - 0.1 - 0.0005 =$ **0.3995**. The decay contributes a clean -0.0005: shrink θ by $\eta\lambda = 0.1\%$ of its own value, independent of the gradient.
2. **Coupled L2.** Here $\lambda\theta = 0.005$ is added to the gradient *before* the adaptive scaling, so it is divided by $\sqrt{\hat{v}} \approx |g| = 0.4$. Its effective pull becomes $\eta \cdot 0.005/0.4 =$ **0.00125** — 2.5× stronger here, and it would be weaker still for a weight with a larger gradient. The regularization strength has become tied to the gradient magnitude.
3. **Meaning.** Decoupling restores the intended rule "shrink every weight by the same fraction each step", which is why AdamW — not Adam+L2 — trains today's large models. The runnable AdamW cell in the exercises reproduces the -0.0005 decay exactly.

Assumptions & limitations

These update rules are heuristics, not guaranteed descent: on a strongly non-convex surface momentum can overshoot a good minimum, and the original Adam convergence proof was later shown to be flawed (fixed variants such as AMSGrad exist). Adam's per-parameter scaling can hurt **generalization**: a well-tuned SGD+momentum frequently reaches a flatter, better-testing minimum, which is why many vision papers still use it. Standard L2 weight decay interacts badly with Adam's scaling. **AdamW** decouples the decay from the adaptive step and is now the default for training large models (chapter 12). Finally, adaptive methods add memory: Adam stores two extra numbers per parameter.

Why & when this matters

Choose Adam first when you want a network to train with almost no tuning — it is the sensible default for a new problem, an unfamiliar architecture, or sparse/noisy gradients (transformers are trained almost exclusively with AdamW). Prefer SGD+momentum when you can afford to tune the learning-rate schedule and the last fraction of a percent of test accuracy matters. If training zig-zags, diverges, or stalls, the optimizer and its η are the first settings to adjust: too large and the loss explodes, too small and it creeps. Everything here assumes the gradient itself is trustworthy, which depends on good initialization (chapter 10).

In the real world

Adam and AdamW are the main tools of modern large-scale training. GPT, BERT, and essentially every large language model are optimized with AdamW under a warmup-then-decay schedule. The original Transformer paper (Vaswani et al., 2017) specifies a linear warmup over 4000 steps followed by inverse-square-root decay, tuned so precisely that reproducing it is part of reproducing the result. In computer vision the situation is mixed: ResNet and many ImageNet classifiers still report their best accuracy with SGD+momentum and a step or cosine schedule. This is a constant reminder that the "best" optimizer is task-dependent, not universal.

Exercises

1. Derive the effective step size of momentum along a direction of constant gradient g for $\beta = 0.9$ and for $\beta = 0.99$, and compare each to plain SGD. MATH

2. Show that Adam's step is $\approx \eta$ whether the steady gradient is $g = 0.004$ or $g = 40$, and state what SGD would do in each case. MATH

3. Add **RMSProp** to the ravine convergence race and report how many iterations it needs. CODE

4. Implement AdamW on a single weight with a constant gradient and confirm the decoupled decay contributes exactly $\eta\lambda\theta$ per step. CODE

5. Why does RMSProp take a step larger than η on its first few iterations, while Adam does not? CONCEPT

6. You switch a well-trained pipeline from Adam to SGD+momentum. Training gets slower but the final test accuracy improves. Explain. CONCEPT

7. At Adam's step $t = 2$ with $\beta_1 = 0.9$ and a steady gradient $g = 0.4$, show numerically that the raw first moment m undershoots g and that the bias correction restores it. MATH

8. On a constant gradient, show numerically that **AdaGrad's** step decays toward zero while RMSProp's does not, and name the one-line difference responsible. CODE

9. Write the **Nesterov** update in this chapter's momentum convention and say in one sentence why the look-ahead reduces overshoot. CONCEPT

10. Contrast the two momentum conventions $v = \beta v + g$ and $v = \beta v + (1-\beta)g$ on a constant gradient g, giving the effective step size of each. MATH

Initialization & the vanishing gradient

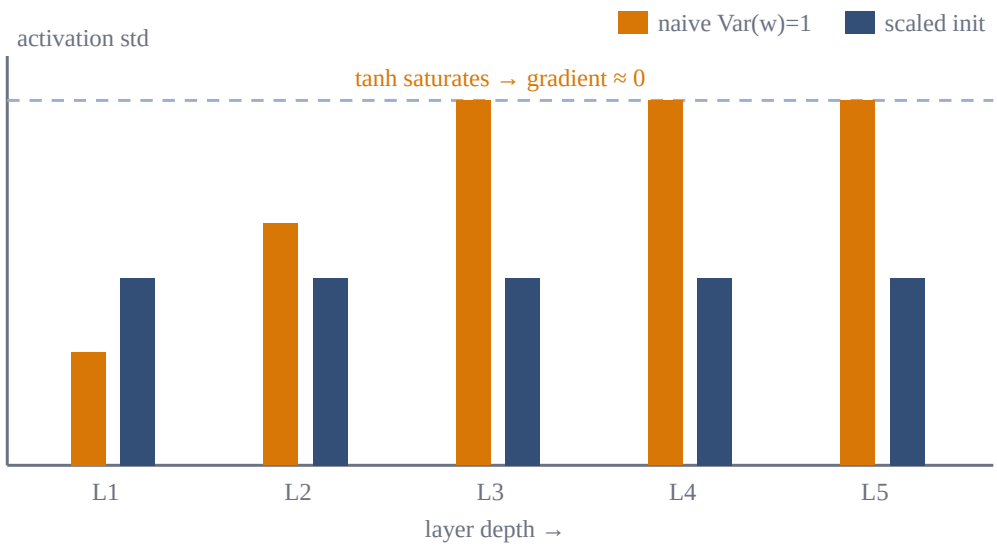
> *Before a network can learn anything, someone must choose its starting weights. This choice may seem small, but it decides whether training works well or fails right away. This chapter shows why weights must be random but carefully scaled. It derives the fan-in variance rule behind LeCun, Xavier and He initialization. It also follows the* ***vanishing/exploding gradient*** *problem that makes deep networks so fragile.*

Fundamental concept

Think of a signal passing through a long line of people. Each person whispers a number to the next after multiplying it by their own factor. If every factor is a little above 1, the number grows very fast. If it is a little below 1, the number fades to nothing by the end of the line. The last person hears something useful only if the factors are set so the typical message keeps roughly the **same loudness** at every stage. A deep network works exactly like this relay. Each layer multiplies its input by a weight matrix, and the same multiplying effect controls both the forward signal and the backward gradient.

Two starting choices fail immediately. First, **all-equal weights** (in particular all zeros). If every neuron in a layer starts identical, they see the same inputs and produce the same output. Because their outgoing connections are identical too, they receive the same gradient. They therefore update identically and stay perfect copies forever. A hundred such neurons have the representational power of one. The fix is randomness, to **break symmetry**. Second, weights that are random but the *wrong size*. Too large and the relayed signal (and gradient) explodes. Too small and it vanishes. Good initialization is the just-right scaling that keeps the variance of the signal constant as it moves through the layers.

The right scale turns out to depend on how many inputs feed each neuron, called its **fan-in**. A neuron summing 100 inputs should use smaller weights than one summing 5, so that the totals are about the same size. Writing this precisely gives one simple formula.



Spread of activations at each layer of a deep tanh net. With naive large weights (amber) the spread grows layer by layer until it hits the saturation limit, where tanh flattens and gradients vanish. Properly scaled weights (blue) hold a steady, healthy spread through all the layers.

Governing equations

Consider one pre-activation z formed from n_{in} inputs, with weights and inputs independent, zero-mean, and mutually independent:

$$z = \Sigma_{i=1..n_{in}} w_i x_i \;\Rightarrow\; \mathrm{Var}(z) = n_{in} \cdot \mathrm{Var}(w) \cdot \mathrm{Var}(x)$$

To keep the signal the same size layer after layer, we need the multiplier $n_{in} \cdot \mathrm{Var}(w)$ to equal 1. This single requirement gives the three standard methods:

$$\text{LeCun: } \mathrm{Var}(w) = \frac{1}{n_{in}} \qquad \text{Xavier/Glorot: } \mathrm{Var}(w) = \frac{2}{n_{in} + n_{out}} \qquad \text{He: } \mathrm{Var}(w) = \frac{2}{n_{in}}$$

w_i	a single weight, drawn i.i.d. from a zero-mean distribution (Gaussian or uniform)
x_i	an input to the neuron (activation of the previous layer)
z	the pre-activation (weighted sum) whose variance we want to control
$\mathrm{Var}(\cdot)$	variance — the statistical "spread" or power of a quantity
n_{in}	fan-in: number of inputs feeding the neuron
n_{out}	fan-out: number of neurons the layer feeds forward
γ, β	batch-norm's learnable scale and shift (below). β here is unrelated to momentum

For a **uniform** distribution U(−r, r) the variance is $r^2/3$, so the app converts any target Var(w) into a symmetric range:

$$U(-r, r): \mathrm{Var}(w) = \frac{r^2}{3} \Rightarrow r = \sqrt{3 \cdot \mathrm{Var}(w)}$$

A fixed uniform range that ignores n_{in} is exactly the naive choice that goes wrong.

Derivation

1. The variance of a sum of independent terms is the sum of their variances. Each term $w_i x_i$ is a product of independent zero-mean variables, whose variance is $\mathrm{Var}(w)\cdot\mathrm{Var}(x)$. With n_{in} identical terms, $\mathrm{Var}(z) = n_{in}\cdot\mathrm{Var}(w)\cdot\mathrm{Var}(x)$.
2. This per-term identity is where the assumptions become important. For independent, zero-mean w_i and x_i, the mean $\mathbb{E}[w_i x_i] = 0$ and $\mathrm{Var}(w_i x_i) = \mathbb{E}[w_i^2]\mathbb{E}[x_i^2] = \mathrm{Var}(w)\cdot\mathrm{Var}(x)$. The cross-terms that would otherwise complicate the sum all average to zero. If we drop independence or zero-mean, the clean multiplier disappears.
3. Requiring $\mathrm{Var}(z) = \mathrm{Var}(x)$, which keeps the signal size unchanged, forces $n_{in}\cdot\mathrm{Var}(w) = 1$, that is $\mathrm{Var}(w) = 1/n_{in}$ (**LeCun**).
4. The *backward* pass multiplies gradients by the same weights but sums over the fan-out, so it needs $\mathrm{Var}(w) = 1/n_{out}$. Xavier takes a middle value between the two by averaging, giving $2/(n_{in}+n_{out})$.
5. A **ReLU** sets roughly half its inputs to zero, which halves the output variance. To undo that loss, the weights must be twice as strong: $\mathrm{Var}(w) = 2/n_{in}$ (**He**). This is the default for ReLU networks.
6. More precisely, for z symmetric about zero, ReLU keeps only its positive half, so $\mathbb{E}[\mathrm{ReLU}(z)^2] = \tfrac{1}{2}\cdot\mathbb{E}[z^2] = \tfrac{1}{2}\cdot\mathrm{Var}(z)$. The forward second moment is exactly halved. Preserving it requires $\tfrac{1}{2}\cdot n_{in}\cdot\mathrm{Var}(w) = 1$, that is $\mathrm{Var}(w) = 2/n_{in}$. The He factor of two is the inverse of that one-half.

All three methods follow one template, Var(w) = gain ⁄ fan. They differ only in the **gain** (1 for LeCun and Xavier, 2 for He) and the **fan** (n_{in}, n_{out}, or their average). The app's initializer menu is exactly this template with a switch for the gain. Everything below is about what happens when that gain is even slightly wrong.

Now consider depth. The gradient of the loss with respect to an early layer is a **product of per-layer Jacobians**, each roughly a weight matrix times the activation's derivative:

$$\nabla_{\theta_1} L \propto \Pi_{l=2..D} W_l^{\top} \cdot \mathrm{diag}(g'(z_l)) \;\Rightarrow\; \text{magnitude} \sim c^D$$

If the typical factor per layer is c, then after D layers the gradient scales like c^D. If $c < 1$ it **vanishes**, and if $c > 1$ it **explodes**. Sigmoid shows this clearly. Its derivative peaks at only 0.25, so ten saturated sigmoid layers shrink the gradient by at most

$$0.25^{10} = \frac{1}{4^{10}} \approx 9.5 \times 10^{-7}$$

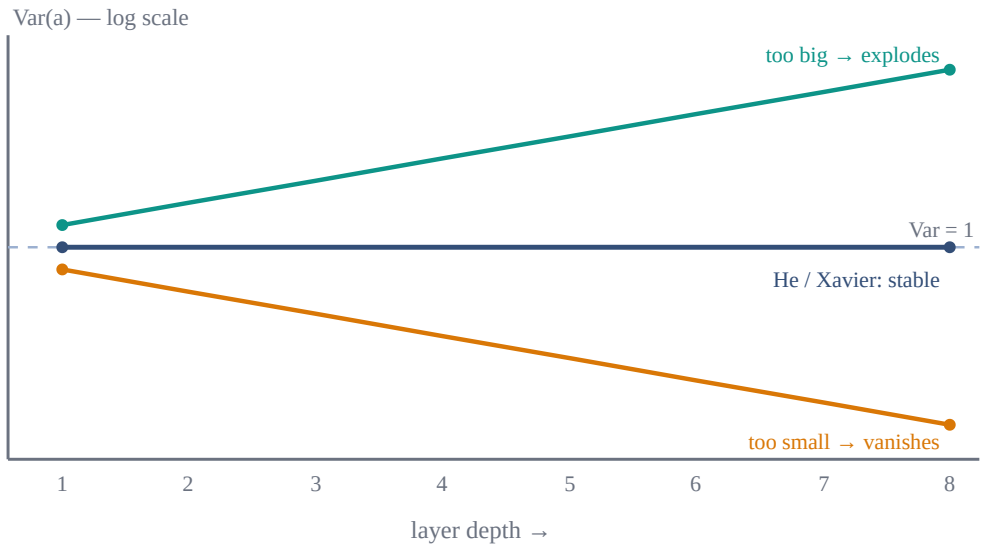
The early layers receive almost no learning signal. This is why progress stalled on deep sigmoid/tanh networks until good initialization and ReLU arrived.

Vanishing and exploding gradients: a product of Jacobians

That c^D estimate is worth a closer look, because the same product controls the signal on the way *up* and the gradient on the way *down*. Follow the forward power layer by layer. Writing $s_l = \mathrm{Var}(a^{(l)})$ for the typical activation variance at layer l, the variance identity leads to a recursion $s_l \approx (n_l \cdot \mathrm{Var}(W_l) \cdot \kappa_l) \cdot s_{l-1}$, where κ_l accounts for the activation ($\kappa = 1$ for a linear or near-origin tanh unit, $\kappa = ½$ for ReLU). Expanding from the input turns this into a **product**, not a sum:

$$\mathrm{Var}(a^{(D)}) = \mathrm{Var}(a^{(0)}) \cdot \Pi_{l=1..D}\, n_l \cdot \mathrm{Var}(W_l) \cdot \kappa_l$$

Variance does not add up across depth. Instead it *multiplies*. If every factor equals a constant c, the depth-D variance is c^D times the input's. This grows exponentially with depth, and stays steady only at the single value $c = 1$. That is exactly why the fan-in rule must be met exactly, not just approximately.



Activation variance across the layers of a deep network, on a logarithmic scale. When the per-layer multiplier n·Var(w) exceeds 1, the variance climbs geometrically and the signal explodes (teal). Below 1 it decays toward zero and the gradient vanishes (amber). Only the variance-preserving scale of He or Xavier holds it flat (blue). Because the effect multiplies, every extra layer widens the gap. Depth turns a small scaling error into a large one.

The backward pass is the mirror image. Backpropagation (Chapter 8) sends the gradient through the *transposes* of the same matrices, so the error reaching layer 1 is again a product of D−1 factors, $\delta^{(1)} = \delta^{(D)} \prod_{l=2..D} \mathrm{diag}(g'(z_l)) \cdot W_l^{T}$. Each factor is a weight matrix scaled by activation slopes. Its size is set by the same per-layer c. So a scale that preserves the forward variance also, to first order, preserves the backward gradient. One choice of Var(w) controls both directions at once. The size of a matrix product is governed by its **singular values**. If each factor's typical singular value is c, the product's is about c^D. It vanishes for $c < 1$ and explodes for $c > 1$. The boundary $c = 1$ is a narrow edge that gets narrower as D grows. A 10% scaling error is invisible across two layers but disastrous across fifty. Sigmoid, whose slope never exceeds 0.25, pushes c far below 1 (the 0.25^D collapse above), while ReLU's pass-through gate and He's factor of two (Chapter 4, activation functions) keep it near 1.

In practice you rarely compute c by hand. Instead you *watch* it. Setting up a fresh network to print the standard deviation of the activations (forward) and of the gradients (backward) at each layer shows the problem at a glance. A healthy network holds both roughly constant with depth. A vanishing one shows a steady geometric decay toward the early layers. An exploding one

shows the same trend reversed. A steady trend in one direction across layers is the clear sign of a badly scaled initialization, and the fix is almost always the matching fan-in rule or a normalization layer, not a smaller learning rate, which only slows the symptom.

Breaking symmetry, and initializing biases

The variance rule fixes the *scale* of the weights but quietly assumes they are also *random*, and that second requirement matters just as much. Suppose every weight in a layer starts at one shared constant, with zero being the well-known case. Then every unit forms the same pre-activation from the same inputs and produces the same activation. Because its outgoing weights are identical too, it receives the same error δ on the backward pass. Gradient descent (Chapter 7) applies the identical update to each unit, and they stay exact copies for all of training. The loss has a **permutation symmetry**: swapping two identical units changes nothing. A gradient can never break a symmetry it respects, so a layer of n such copies has the expressive power of a single unit. Only unequal, random starting weights break the tie and let units specialize. The fan-in scale is then added on top. Randomness and scale are *independent* requirements. Tiny random weights break the symmetry but still let the signal vanish. Huge random weights break it but let the signal explode. A workable start needs both a random *direction* to separate the units and the right *magnitude* to preserve variance. This is why symmetry-breaking and the fan-in rule are two conditions and not one. (All-zero weights are broken in two ways. Beyond the symmetry, they pass no signal at all, since every activation collapses to a constant.)

Biases have none of this problem. Once the weights are random, the units already differ, so biases are almost always initialized to **zero**, which also keeps each pre-activation zero-mean, exactly as the variance derivation assumed. Two deliberate exceptions are worth knowing. A small positive bias (around 0.01) is sometimes given to ReLU units so they start on their active side and fewer of them start out dead. The forget-gate biases of LSTMs and GRUs are set positive (≈ 1) so the gates begin open and gradients can flow through time. Without a specific reason, zero biases with scaled random weights is the standard approach.

Orthogonal initialization and a data-driven fix

The variance rule controls only the *average* gain of a weight matrix. A random Gaussian matrix still has a spread of singular values, so it stretches some input directions and shrinks others, and through depth that spread grows on its own. **Orthogonal initialization** removes it. Draw W to be orthonormal by taking the Q factor of the QR decomposition (or the U of an SVD) of a random matrix, so every singular value equals exactly 1. Such a matrix preserves the norm of *every* input vector, not just on average, and a deep stack of orthogonal layers neither amplifies nor weakens any direction. This condition, sometimes called **dynamical isometry**, lets very deep and recurrent networks train even without normalization. An orthogonal (or identity) recurrent matrix is the starting point for the same idea behind gating and residual connections (Chapter 14, modern convolutional architectures).

When the analytic assumptions are weak (an unusual activation, correlated inputs), you can *measure* the scale instead of computing it. **LSUV** (Layer-Sequential Unit-Variance) initializes each layer orthogonally, then passes one real mini-batch through the network and, layer by layer, divides each layer's weights by the observed standard deviation of its output. This forces the activation variance to exactly 1 before the first gradient step. It is the measured version of the fan-in rule, and it works well exactly where the closed-form derivation does not.

Fixes beyond initialization

Careful scaling and ReLU only set the network up at step 0. Three architectural tools keep it trainable throughout, and each gets a full chapter later. Here we preview how they *complement* good initialization rather than replace it. **Batch normalization** (Chapter 11, normalization) standardizes each layer's pre-activations, then lets the network rescale them:

$$\hat{x} = \frac{x - \mu_{\text{batch}}}{\sqrt{\sigma_{\text{batch}}^{2} + \varepsilon}}, \quad y = \gamma \cdot \hat{x} + \beta$$

Fixing the mean μ and variance σ^2 at every layer keeps activations off the saturating tails automatically, and the learnable γ, β let the layer recover any scale it really needs. **Layer normalization** does the same across features instead of across the batch and is standard in

transformers (Chapter 16, attention). **Residual connections** (Chapter 14, modern convolutional architectures) add a layer's input to its output, $y = x + F(x)$, so the gradient has a direct "shortcut" path of factor 1 back through the network. This avoids the c^D decay and enables hundreds of layers. Finally, **gradient clipping** caps the gradient norm to rescue the exploding case. The Playground offers it directly.

Normalization layers, step by step

Initialization sets activations to a healthy scale at step 0, but training moves the weights and the scale drifts. **Batch normalization** (Ioffe & Szegedy, 2015) re-applies the scale at *every* step. For each feature it standardizes the pre-activations across the current mini-batch, then applies a learnable scale γ and shift β so the layer can recover any distribution it actually needs:

$$\mu = \frac{1}{B}\Sigma x, \quad \sigma^2 = \frac{1}{B}\Sigma(x-\mu)^2, \quad \hat{x} = \frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}}, \quad y = \gamma\hat{x} + \beta$$

Three consequences follow. First, the activations are held off the saturating tails automatically, so the exact initialization matters far less. BatchNorm is a large part of why deep networks became easier to train. Second, γ and β are learned by backprop like any other parameters, so the layer keeps its full expressive range. If passing the raw statistics through is best, it can learn $\gamma = \sqrt{\sigma^2}$, $\beta = \mu$. Third, the batch statistics only exist during training, so at **inference** the layer instead uses a running average of μ and σ^2 collected over training. This is a subtle but critical detail. A model that forgets to switch to evaluation mode computes wrong outputs.

BatchNorm has one weakness: it **couples the examples in a batch**, so it works poorly for very small batches and fits awkwardly into recurrent or sequence models. **Layer normalization** removes the coupling by normalizing across the *features of each single example* instead of across the batch. It has no train/inference difference and is the standard choice in transformers (Chapter 16, attention) and recurrent networks, where it sits inside every block. The original "internal covariate shift" motivation has since been questioned. Later work argues the real benefit is a **smoother, better-conditioned loss surface**, but the measured value of

normalization is not in doubt. Its batch coupling also adds a little noise into every activation, a mild regularizing side effect (Chapter 12, regularization). This is welcome, but no substitute for real regularization.

Residual connections and gradient highways

Even with good initialization and normalization, stacking a hundred layers brings back the c^D problem. Each extra layer is one more factor in the product of Jacobians, and the gradient still tends to fade. **Residual connections** (He et al., 2016) defeat this with a surprisingly simple change: add each block's input to its output:

$$y = x + F(x) \;\Rightarrow\; \frac{\partial y}{\partial x} = I + \frac{\partial F}{\partial x}$$

Because the Jacobian now carries an **identity term**, the backward product is a product of factors each close to I, rather than a product of small numbers. The gradient has a direct shortcut path of factor exactly 1 from the loss to any early layer, so it can no longer vanish just from depth. In the same way, the block learns a *residual*, which is how much to change its input. The easy default "change nothing" (F = 0) is the identity, so adding depth can never make an untrained residual net worse than a shallower one. This single idea unlocked networks of 100–1000 layers (ResNet, the modern convolutional architectures of Chapter 14, built on the CNNs of Chapter 13). It is also a structural part of every transformer block (Chapter 16, attention), where a residual wraps both the attention and the feed-forward sub-layers.

Initialization in code

The variance rule is easiest to believe when you watch it. This first cell sends a batch through six *linear* layers (no activation, so nothing but the weight scale acts on the signal) and prints the activation standard deviation at each layer for three choices of Var(w). Press **Run**:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)
n, depth = 256, 6
```

```
x0 = rng.randn(512, n)                # a batch of 512 inputs, unit
variance
for label, gain in [("too big   Var(w)=4/n  ", np.sqrt(4.0 / n)),
                    ("too small Var(w)=1/4n  ", np.sqrt(0.25 / n)),
                    ("Xavier    Var(w)=1/n   ", np.sqrt(1.0 / n))]:
    x = x0.copy(); stds = []
    for l in range(depth):
        W = rng.randn(n, n) * gain
        x = x @ W                      # LINEAR relay: pure variance flow
        stds.append(round(float(x.std()), 3))
    print(label, "std/layer:", stds)
```

With Var(w) = 4/n the per-layer multiplier is n·Var(w) = 4, so the std doubles every layer and **explodes** (1 → 2 → 4 → … → 64). With 1/4n it halves and **vanishes** (→ 0.016). Only Var(w) = 1/n holds it steady near 1, exactly what Var(z) = n·Var(w)·Var(x) = 1 predicts. A real activation changes the healthy scale but not the lesson, as the second cell shows for tanh and ReLU:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)
n = 256
def relay(gain, act):                 # 6-layer relay -> per-layer std,
final sat%
    x = rng.randn(512, n); stds = []
    for l in range(6):
        x = act(x @ (rng.randn(n, n) * gain))
        stds.append(round(float(x.std()), 2))
    sat = 100 * np.mean(np.abs(x) > 0.99)
    return stds, sat
relu = lambda z: np.maximum(0.0, z)
s, sat = relay(1.0, np.tanh)
print("tanh naive std=1 :", s, f" saturated={sat:.0f}%")   # frozen at the
rails
s, sat = relay(np.sqrt(1.0 / n), np.tanh)
print("tanh Xavier 1/n  :", s, f" saturated={sat:.0f}%")   # healthy mid-
range
print("ReLU naive std=1 :", relay(1.0, relu)[0])            # explodes
print("ReLU He   2/n    :", relay(np.sqrt(2.0 / n), relu)[0])
```

Naive tanh freezes the std near 0.97, but that is *saturation*. About 87% of units are stuck at ±1 where the gradient is dead. Xavier keeps tanh in its responsive middle (0% saturated). Naive ReLU explodes toward a million. He (Var(w) = 2/n, the factor of two that makes up for the half of inputs ReLU zeroes) holds it near 1. In PyTorch each rule is a single call:

PyTorch copy → Colab

```
import torch, torch.nn as nn

layer = nn.Linear(256, 256)
nn.init.kaiming_normal_(layer.weight, nonlinearity='relu')   # He, for ReLU
nn.init.zeros_(layer.bias)

tanh_layer = nn.Linear(256, 256)
nn.init.xavier_normal_(tanh_layer.weight)                    # Xavier, for
tanh

# initialize a whole network by walking its modules
net = nn.Sequential(nn.Linear(256, 256), nn.ReLU(),
                    nn.Linear(256, 64),  nn.ReLU(),
                    nn.Linear(64, 10))
for m in net.modules():
    if isinstance(m, nn.Linear):
        nn.init.kaiming_normal_(m.weight, nonlinearity='relu')
        nn.init.zeros_(m.bias)
```

In this app's engine

The Playground's initializer selector offers He, Xavier, LeCun and plain Uniform, and reports the resulting weight histogram. Run this experiment. On the two-spirals dataset with 5 hidden layers, sigmoid activation and plain Uniform init, the loss barely moves. The deep sigmoids saturate exactly as the 0.25^{10} estimate predicts. Switch the activation to tanh and the initializer to He, and the same network suddenly trains. The Math Log will show the per-layer activation spread staying constant instead of collapsing.

Worked example — He and Xavier std, and variance carried layer to layer

Take a ReLU hidden layer with fan-in $n_{in} = 256$ and fan-out $n_{out} = 64$, given unit-variance activations (Var(x) = 1).

1. **He std.** $\text{Var}(w) = 2/n_{in} = 2/256 = 0.007813$, so the weights are drawn with standard deviation $\sqrt{2/256} = \mathbf{0.0884}$.
2. **Xavier std.** $\text{Var}(w) = 2/(n_{in}+n_{out}) = 2/320 = 0.00625$, std $= \sqrt{2/320} = \mathbf{0.0791}$. (LeCun would use $\sqrt{1/256} = \mathbf{0.0625}$.) The three differ only in their fan and gain.
3. **Forward variance with He.** $\text{Var}(z) = n_{in} \cdot \text{Var}(w) \cdot \text{Var}(x) = 256 \cdot (2/256) \cdot 1 = \mathbf{2}$. ReLU keeps the positive half, so $\text{Var}(a) \approx \tfrac{1}{2} \cdot \text{Var}(z) = \mathbf{1}$. The layer's output variance equals its input variance. Ten such layers hold it at 1, so variance is preserved.
4. **Forward variance with Xavier.** $\text{Var}(z) = 256 \cdot (2/320) \cdot 1 = \mathbf{1.6}$, and after ReLU $\text{Var}(a) \approx \mathbf{0.8}$, a gentle 20% shrink each layer, because Xavier splits its budget between the forward (n_{in}) and backward (n_{out}) passes. Over ten ReLU layers, $0.8^{10} \approx \mathbf{0.11}$, a real fade.
5. **Meaning.** The two standard deviations differ by only 12% (0.0884 vs 0.0791), yet, built up through depth, that tiny difference is the gap between a flat activation profile and a slow collapse. That is why He, tuned to fan-in and to ReLU's one-half, is the default for ReLU networks, while Xavier, balanced across fan-in and fan-out, is the choice for the symmetric tanh.

Worked example — a 10-layer tanh network, $n_{in} = 100$

Assume inputs with Var(x) = 1 and, for the near-origin (linear) part of tanh, treat the activation's derivative as ≈1.

1. **Naive**, Var(w) = 1. Per layer the variance is multiplied by $n_{in} \cdot \text{Var}(w) = 100 \cdot 1 = \mathbf{100}$. So Var(z) at layer 1 is 100, giving a standard deviation of $\sqrt{100} = \mathbf{10}$.
2. A typical pre-activation is now around ±10. But tanh(10) ≈ 1 (0.9999999959), with derivative $1 - \tanh^2(10) \approx \mathbf{8 \times 10^{-9}}$. The unit is stuck at ±1 and passes essentially *no gradient*. After 10 such layers the signal is hopeless (variance $\sim 100^{10} = 10^{20}$ in the linear estimate, though in reality everything saturates by layer 2).
3. **Xavier**, Var(w) = 2/(100+100) = 0.01. Now the per-layer multiplier is $n_{in} \cdot \text{Var}(w) = 100 \cdot 0.01 = \mathbf{1}$, so the variance is *preserved*. Var(z) = 1, so a typical $z \approx \pm 1$.
4. There tanh(1) = **0.7616** with derivative $1 - 0.7616^2 = \mathbf{0.420}$, a healthy slope. Every one of the ten layers passes a strong gradient, and the network learns.
5. **Meaning.** The only difference is a factor of 100 in weight variance, yet it is the difference between a dead network and a working one. Initialization is not a small detail. It sets whether gradient descent (Chapter 7) ever gets a signal to descend.

Worked example — batch normalization on a 3-example batch

One feature takes the values x = [2, 4, 6] across a mini-batch of three. Apply BatchNorm with γ = 1, β = 0 (ε very small).

1. Mean: $\mu = (2 + 4 + 6)/3 = \mathbf{4}$.
2. Variance: $\sigma^2 = [(2-4)^2 + (4-4)^2 + (6-4)^2]/3 = (4 + 0 + 4)/3 = \mathbf{2.667}$, so $\sigma = \sqrt{2.667} = \mathbf{1.633}$.
3. Normalize: $\hat{x} = [(2-4), (4-4), (6-4)] / 1.633 = [\mathbf{-1.225}, \mathbf{0}, \mathbf{+1.225}]$, now zero-mean and unit-variance.

4. Scale and shift: $y = 1 \cdot \hat{x} + 0 = \hat{x}$. If the layer had learned $\gamma = 2$, $\beta = 5$, the outputs would be [2.55, 5, 7.45]. BatchNorm fixes the *shape* of the distribution, then lets the network choose its scale.
5. **Meaning.** Whatever the incoming weights did, the pre-activations leave this layer centered and unit-scaled, off the saturating tails. This is why the rest of the network barely feels a poor initialization. The runnable cell in the exercises confirms mean ≈ 0, std ≈ 1 on a random batch.

Assumptions & limitations

The variance rule assumes inputs and weights are independent, zero-mean, and that the activation behaves linearly near the origin (true for tanh and, in a one-sided way, for ReLU). Those assumptions weaken once units saturate, so the formulas set the *starting* conditions, not a permanent guarantee. During training, weights drift and normalization layers take over the job of holding activations in range. The fan-in rule also says nothing about the mean of the weights or about biases, which are typically initialized to zero (a small positive bias is sometimes used with ReLU to keep units active early).

Why & when this matters

Initialization matters most when a network is **deep** or uses **saturating** activations, and least for one or two ReLU layers. If a fresh network's loss is flat from step one, refuses to move, or produces NaNs, suspect initialization or scaling before anything else. Pick He for ReLU, Xavier for tanh, normalize your inputs, and add batch/layer norm or residual connections for real depth. These choices decide whether the optimizers of Chapter 9 have a usable gradient to work with in the first place. A well-initialized shallow net often needs nothing more complex than SGD.

In the real world

These fixes are the reason modern depth is possible at all. He initialization was introduced in the 2015 work that first trained networks deep enough to surpass human top-5 accuracy on ImageNet. Residual connections (ResNet, 2015) took the same benchmark to 152 layers and won the ILSVRC competition. Batch normalization made training such networks fast enough to be practical. Every transformer you have used, for translation, code completion, or chat, stacks residual connections and layer normalization in exactly the arrangement of this chapter, dozens of times over. Without them, the gradient could not reach the bottom of the stack.

Exercises

1. A deep network uses sigmoid activations, each contributing at most $\sigma' = 0.25$ per layer to the gradient. How many such saturated layers reduce the gradient below 10^{-6}? MATH

2. Compute the target weight standard deviation for (a) He init on a ReLU layer with $n_{in} = 512$, and (b) Xavier init on a tanh layer with $n_{in} = n_{out} = 256$. MATH

3. Modify the relay experiment to use **ReLU with He** initialization and confirm the per-layer std stays roughly constant rather than exploding or vanishing. CODE

4. Implement a BatchNorm forward pass in NumPy on a deliberately off-scale batch and confirm the output has zero mean and unit variance per feature. CODE

5. Why can a network never be initialized with all weights set to the same constant (for example all zeros), no matter how the constant is chosen? CONCEPT

6. Explain, using the Jacobian $y = x + F(x)$, why a residual connection prevents the gradient from vanishing through depth. CONCEPT

7. Using $\mathrm{Var}(z) = n_{\text{in}} \cdot \mathrm{Var}(w) \cdot \mathrm{Var}(x)$, find the weight variance that preserves signal for a linear layer with $n_{\text{in}} = 100$ and unit-variance inputs, then explain why a ReLU layer needs double that. MATH

8. A 20-layer **linear** network draws each weight matrix with i.i.d. entries of variance c/n, so the per-layer variance multiplier is exactly c. For which c is the signal variance preserved, and by what factor does the output variance differ from the input's when $c = 1.1$? When $c = 0.9$? MATH

9. Why is initializing every bias to zero harmless, when initializing every weight to zero is fatal? Give one situation where a small positive bias is used deliberately. CONCEPT

10. Verify numerically that an **orthogonal** matrix preserves a vector's norm exactly, while a Gaussian matrix scaled by $\sqrt{1/n}$ preserves it only on average. CODE

Normalization: batch, layer & beyond

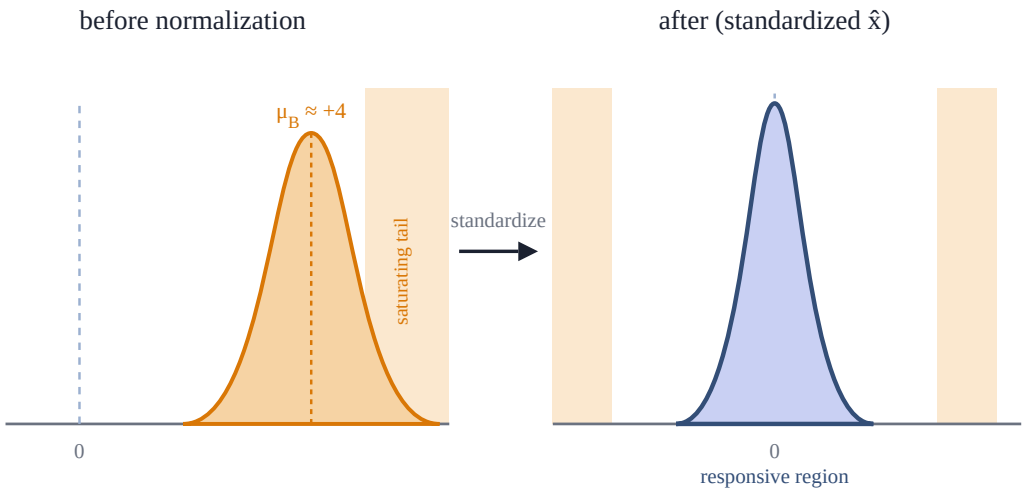
> *Initialization (Chapter 10) sets every layer's activations to a healthy scale at step zero. But the moment training begins, the weights move, and that carefully chosen scale drifts. A pre-activation that started near ±1 can, a few dozen updates later, sit at ±40, buried in the flat tail of its activation where no gradient survives.* ***Normalization*** *prevents this. It re-centers and re-scales the signal at every layer, on every step, so each layer always receives inputs of a predictable size. It then gives the network two learnable knobs to recover any scale it truly needs. This one idea turned networks of a dozen delicate layers into networks of a thousand, and it is now built into almost every model you use.*

Fundamental concept

You already normalize the *input*. Before the first layer ever sees the data, good practice (the neuron of Chapter 1, and the recipe of Chapter 25) rescales every feature to roughly zero mean and unit variance. This way, a feature measured in thousands cannot overwhelm one measured in fractions, and gradient descent (Chapter 7) sees comparable slopes along every weight. Normalization layers make one bold observation. The input to layer 5 is *also* just a set of features (it is the output of layer 4), so why standardize only the first layer's inputs and leave the other forty layers to manage on their own?

Picture a factory line where each station receives parts from the station before it. If an early station starts to misbehave and doubles the size of everything it passes downstream, every later station must rush to recalibrate. By the time the drift reaches the end of the line, the last stations spend all their effort chasing a moving target instead of doing useful work. A normalization layer is a recalibration desk placed between stations. Whatever arrives, it is re-centered and re-scaled to a fixed, agreed size before the next station touches it. The downstream stations can now assume a stable input and simply do their job. That stability is the entire point, and it is why a well-normalized network is so much more forgiving of a mediocre initialization or an aggressive learning rate.

There is one problem that a naive recalibration would miss. Forcing every layer's output to be exactly zero-mean and unit-variance is too restrictive. Sometimes the best thing a layer can do *is* to output a large or shifted signal. So each normalization layer standardizes first and then applies a learnable **scale γ** and **shift β**. Standardization sets a clean default, and γ, β let the network overrule it, even learning to pass the raw signal straight through. Normalization removes a flaw without removing any expressive power.



What a normalization layer does to one feature. Left: raw pre-activations drift off-center (mean ≈ +4) and spread wide, so much of the mass lands in the flat, saturating tail where the gradient dies. Right: after subtracting the mean and dividing by the standard deviation, the same values become a tight zero-mean, unit-variance bump sitting in the activation's responsive middle. The learnable γ and β can then move and stretch this bump back out if the network needs it.

Governing equations

Batch normalization (Ioffe & Szegedy, 2015) applies the recalibration desk one feature at a time, using the statistics of the current mini-batch. Take a single feature and let $x_1, \ldots, x_m$ be its values across the m examples of the batch. BatchNorm computes that feature's batch mean and variance, standardizes, then scales and shifts:

$$\mu_B = \frac{1}{m} \Sigma_{i=1..m} x_i \qquad \sigma^2{}_B = \frac{1}{m} \Sigma_{i=1..m} (x_i - \mu_B)^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma^2_B + \varepsilon}} \qquad y_i = \gamma\,\hat{x}_i + \beta$$

x_i	the value of *one* feature for example i in the batch (BatchNorm treats each feature independently, with its own μ, σ^2, γ, β)
m	the mini-batch size, the number of examples the statistics are averaged over
μ_B, σ^2_B	the batch mean and (population, ÷m) variance of that feature
ε	a tiny constant (e.g. 10^{-5}) added for numerical safety, so a zero-variance feature does not divide by zero
$\hat{x}_i$	the standardized value, which is zero-mean and unit-variance across the batch by construction
γ, β	learnable scale and shift (one pair per feature), trained by backprop like any weight. They let the layer recover any mean and variance it needs
y_i	the layer's output, handed to the next layer (or activation)

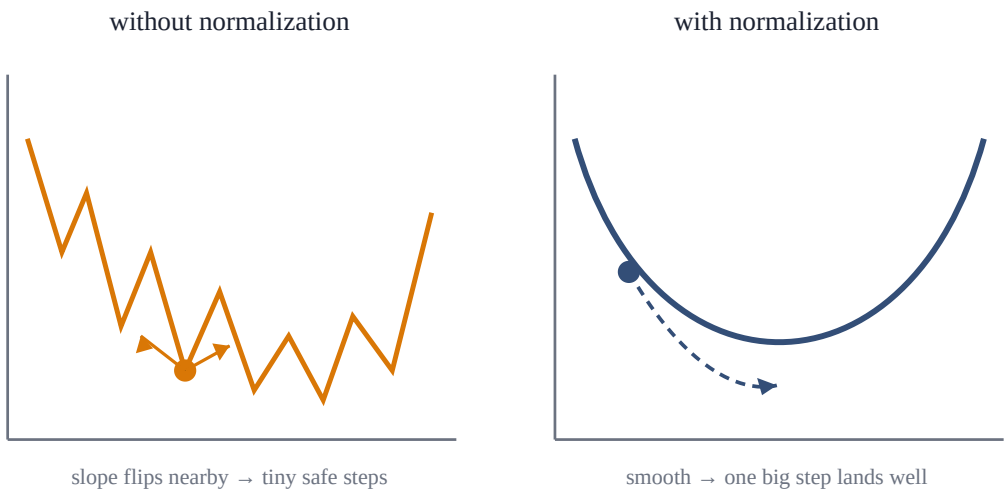
Two facts are worth noting right away. First, the standardization is **per feature, over the batch**: for a batch stored as rows (m examples × D features, the book's convention), μ_B and σ^2_B are computed *down each column*, and there are D independent γ, β pairs. Second, because μ_B and σ^2_B depend on the whole batch, the output for example i now depends on the *other* examples it happened to be batched with. This coupling has deep consequences, both good (a free regularizer) and bad (a fragility at small batch size), which the rest of this chapter explains.

Two stories: covariate shift, and the smoother truth

Why does standardizing hidden activations help so much? The original paper told a simple story called **internal covariate shift**. As the lower layers update, the distribution of the activations they feed upward keeps shifting. So each higher layer is always chasing a target that keeps moving, instead of settling into a solution. By fixing the first two moments of every layer's inputs, BatchNorm holds that distribution roughly still, so the higher layers face a stationary problem and training speeds up. It is an intuitive explanation, and for years it was the standard one.

It is also, at best, incomplete. Santurkar, Tsipras, Ilyas & Madry (2018), in *How Does Batch Normalization Help Optimization?*, put the covariate-shift story to a direct test. They trained a network with BatchNorm and then *deliberately injected noise after each BN layer*. The noise was non-zero and time-varying, and it shifted the distributions on purpose, creating severe internal covariate shift by any reasonable measure. Training was still fast and stable. Whatever BatchNorm was giving the network, it was not primarily a still distribution. Their alternative, supported by both experiment and analysis, is that normalization **smooths the optimization landscape**. It makes the loss and its gradients more Lipschitz (bounded in how fast they can change), so the loss surface has fewer sharp cliffs and sudden flats. On a smoother surface the gradient computed at one point stays a good guide for a longer step, which is exactly why normalized networks tolerate much higher learning rates without diverging. Related work (Bjorck et al., 2018) makes the same point from the viewpoint of gradient magnitudes. BatchNorm keeps them well-conditioned, so a large step helps everywhere instead of exploding in the steepest direction.

Both explanations appear in the literature, so keep both in mind. The mechanism is genuinely debated. Better conditioning, a smoother loss, a decoupling of weight magnitude from direction, and a mild regularizing noise all contribute, in proportions that depend on the architecture. What is *not* in dispute is the empirical payoff: faster convergence, higher usable learning rates, and a striking insensitivity to the exact initialization. Normalization works. The simple 2015 slogan for *why* it works has been substantially revised.



The modern reinterpretation (Santurkar et al., 2018). Normalization does not really freeze the input distribution. Instead, it flattens the ruggedness of the loss. On the rugged surface (left) the gradient at one point is a poor predictor of the loss a short distance away, so only tiny learning rates are safe. On the smoothed surface (right) the gradient stays trustworthy over a long distance, so a single large step glides toward the minimum. Better conditioning, not a still distribution, is the key benefit.

Batch normalization in practice

Three practical facts turn the two clean equations above into a layer you can actually train and deploy.

Training versus inference. The batch statistics μ_B, σ^2_B exist only while a mini-batch flows through the network. At test time you may score a single example, where a batch mean is meaningless. Also, a prediction should never depend on which other examples share its batch. BatchNorm resolves this. During training, it keeps an **exponential moving average** of the statistics, and it uses those fixed running values at inference:

$$\mu_{run} \leftarrow (1 - \alpha)\,\mu_{run} + \alpha\,\mu_B \qquad \sigma^2_{run} \leftarrow (1 - \alpha)\,\sigma^2_{run} + \alpha\,\sigma^2_B$$

with a small momentum α (commonly 0.1). At inference the layer uses these frozen numbers, $\hat{x} = (x - \mu_{run}) \big/ \sqrt{\sigma^2_{run} + \varepsilon}$. It becomes a plain fixed affine map: deterministic, batch-independent, and identical for one example or a thousand. This is the most common BatchNorm bug: a model left in training mode at test time recomputes statistics from the evaluation batch and produces slightly or badly wrong outputs. In PyTorch the switch is `model.eval()`. Forgetting it is a common mistake.

What it gives you. Because the loss is better conditioned, BatchNorm lets you train with *higher learning rates* that would otherwise diverge, and it reaches a given accuracy in far fewer epochs. It also makes the network clearly *less sensitive to initialization* (Chapter 10): the layer re-applies a good scale every step, so a merely decent start is enough. And it provides a *mild regularization* for free. Each example is standardized with the noisy statistics of whatever random batch it landed in, so its normalized value varies from batch to batch, a random perturbation similar to dropout (Chapter 12). Networks with strong BatchNorm often need little or no dropout, and the two work awkwardly together when stacked.

Where to put it: the placement debate. Ioffe & Szegedy inserted BatchNorm on the *pre-activation*, between the linear map and the nonlinearity ($z = Wa$, then BN, then g), arguing that z is nearer to Gaussian and that standardizing it keeps the activation off its saturating tails. Others normalize after the activation, and some architectures prefer it. The identity-mapping ResNet (He et al., 2016) rearranged the block into BN → ReLU → conv for a cleaner gradient path. There is no universal winner, only a default (normalize the pre-activation) and a reminder to treat the ordering as something you can tune. The same debate appears again for transformers as **Pre-LN versus Post-LN** (Chapter 16): the original used Post-LN and needed a learning-rate warm-up to stay stable, while Pre-LN proved far more robust in very deep stacks and is now standard.

Its main weakness: batch size. Every benefit of BatchNorm depends on the quality of μ_B and σ^2_B as estimates, and these estimates get worse as the batch shrinks. Their sampling error grows like $1/\sqrt{m}$. At a batch of 4 or 8 the statistics are so noisy they hurt more than they help. At a batch of 1 the variance is zero and $\hat{x}$ collapses to a constant. This is not a rare case. Memory-heavy tasks (high-resolution detection and segmentation, video, and large models split so each device holds only a few examples) routinely run tiny per-device batches, and BatchNorm fails exactly there. It also fits awkwardly into recurrent networks (Chapter 15), and it has the train/inference discrepancy described above. These failures are exactly what the alternative normalizers below were invented to avoid.

Layer normalization

Layer normalization (Ba, Kiros & Hinton, 2016) removes the batch entirely. Instead of standardizing one feature across many examples, it standardizes one example across its many features. For a single example with feature vector $x = (x_1, \ldots, x_D)$, LayerNorm computes the mean and variance *over those D features*:

$$\mu = \frac{1}{D}\sum_{k=1..D} x_k \quad \sigma^2 = \frac{1}{D}\sum_{k=1..D} (x_k - \mu)^2 \quad \hat{x}_k = \frac{x_k - \mu}{\sqrt{\sigma^2 + \varepsilon}} \quad y_k = \gamma_k \hat{x}_k + \beta_k$$

The change of axis looks small, but it is not. Because the statistics are computed *within* each example, LayerNorm has **no batch dependence at all**: an example's output is a fixed function of that example alone, at any batch size, even one. As a result there is *no train/inference discrepancy*, no running averages, and no `eval()` trap. The layer behaves the same way everywhere. The per-feature γ_k, β_k keep its full expressive range.

These properties are exactly what sequence models need, which is why **transformers and RNNs use LayerNorm, not BatchNorm**. A transformer (Chapter 16) processes sequences of very different lengths, batched together with padding. A batch statistic pooled across positions and sentences would be poorly defined and unstable, while LayerNorm simply standardizes each token's own feature vector, independent of its neighbors. A recurrent network (Chapter 15) applies the same cell at every time step, and batch statistics that change from step to step would break the recurrence. LayerNorm's per-example normalization ignores the time axis. The rule of thumb is simple: normalize across the batch when the batch is large and i.i.d. (classic vision with BatchNorm). Normalize across features when it is not (sequences with LayerNorm).

A zoo of norms, and the axes they average over

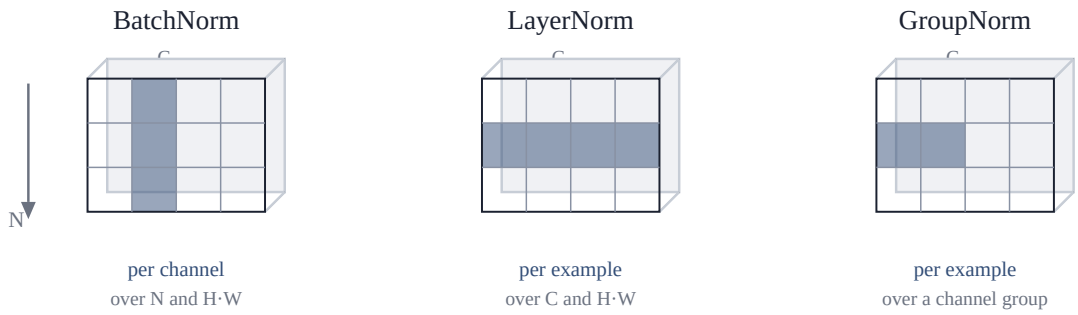
BatchNorm and LayerNorm are the two ends of a spectrum defined by a single question: which axes of the activation tensor do we average over to form μ and σ^2? For images the activation is a four-index tensor (N, C, H, W): N examples, C channels, and H×W spatial positions. For sequences it is (N, T, D): N examples, T time steps, and D features. Choose a different subset of axes and you get a different normalizer, each a good fit for a different situation.

BatchNorm averages over N and the spatial positions H, W, separately for each channel: one μ, σ^2 per channel, pooled across the whole batch. **LayerNorm** averages over the feature axes (C and, in the vision convention, the spatial ones too) separately for each example: one μ, σ^2 per example, no batch involved. Three more complete the picture:

Instance normalization (Ulyanov et al., 2016) averages over only the spatial positions H, W, separately for each example *and* each channel. It removes per-image contrast and color statistics, which is exactly what you want in *style transfer*, where the content should remain but the overall "look" of a particular image should not. **Group normalization** (Wu & He, 2018) splits the C channels into a few groups and averages over each group plus its spatial positions, per example. It is a middle ground between LayerNorm (one group of all channels) and InstanceNorm (one channel per group). Importantly it is **batch-independent**, so it matches BatchNorm's accuracy on detection and segmentation at the tiny batch sizes where BatchNorm fails, and it is the standard alternative when memory forces a small batch. **RMS normalization** (Zhang & Sennrich, 2019) is a simplified LayerNorm that *skips the mean subtraction entirely*, dividing only by the root-mean-square of the features:

$$\mathrm{RMS}(x) = \sqrt{\frac{1}{D}\sum_k x_k^2 + \varepsilon} \qquad y_k = \gamma_k \cdot \frac{x_k}{\mathrm{RMS}(x)}$$

Dropping the centering removes a mean, a subtraction, and (usually) the β shift. This makes RMSNorm cheaper and, in practice, just as effective, which is why it is the preferred normalizer in modern large language models such as LLaMA and T5. Finally, **weight normalization** (Salimans & Kingma, 2016) approaches the problem from the other side: it reparameterizes each weight vector as $w = g \cdot v / \|v\|$, separating direction v from magnitude g so the two learn independently. It is data-independent and adds no batch coupling, but lacks the automatic re-centering that makes BatchNorm so robust.



Shaded cells share one μ and σ². Each cell hides a depth of H·W (or T) positions that join the same average. InstanceNorm = one cell per (N, C); RMSNorm shares LayerNorm's axes but skips the mean.

Which axes each normalizer averages over, drawn on the (N channels-wide, batch-tall) grid with the spatial/time axis hidden as depth inside each cell. BatchNorm pools a channel down the whole batch (a column). LayerNorm pools an example across all channels (a row). GroupNorm pools an example across a group of channels (part of a row). Cells sharing a color share one mean and variance, so BatchNorm's answer depends on the batch, while LayerNorm's and GroupNorm's do not.

Backprop through batch normalization

BatchNorm is a differentiable layer, so it must pass a gradient back to whatever produced its inputs. Here the batch coupling creates a subtle point that is easy to get wrong. The output $y_i = \gamma\,\hat{x}_i + \beta$ does not depend on x_i alone: through μ_B and σ^2_B, *every* input in the batch influences *every* normalized output. Backprop (Chapter 8) must therefore differentiate through the batch statistics themselves, not treat them as constants. The two easy gradients come first. γ and β see the standardized value and the raw upstream signal:

$$\frac{\partial L}{\partial \gamma} = \sum_i \frac{\partial L}{\partial y_i}\,\hat{x}_i \qquad \frac{\partial L}{\partial \beta} = \sum_i \frac{\partial L}{\partial y_i} \qquad \frac{\partial L}{\partial \hat{x}_i} = \gamma\,\frac{\partial L}{\partial y_i}$$

The gradient with respect to the input x_i is where μ_B and σ^2_B matter most. Carefully applying the chain rule through both statistics and collecting terms gives the compact result:

$$\frac{\partial L}{\partial x_i} = \frac{1}{m\sqrt{\sigma^2_B + \varepsilon}}\left(m\,\frac{\partial L}{\partial \hat{x}_i} - \sum_j \frac{\partial L}{\partial \hat{x}_j} - \hat{x}_i \sum_j \frac{\partial L}{\partial \hat{x}_j}\,\hat{x}_j\right)$$

Read the three terms. The first is the "obvious" gradient you would get if μ_B and σ^2_B were frozen constants. The second, $-\Sigma_j\ \partial L/\partial \hat{x}_j$, is the correction that flows back *through the mean*: because subtracting μ_B mixed every example into every output, the gradient must subtract the batch-average upstream signal. The third, $-\hat{x}_i\ \Sigma_j\ (\partial L/\partial \hat{x}_j)\ \hat{x}_j$, is the correction that flows back *through the variance*: dividing by σ_B tied the scale of every output to the spread of the batch, so the gradient must subtract the component that would only re-inflate that spread. Together the two corrections make the returned gradient **centered and decorrelated** across the batch. This is the backward-pass echo of what the forward pass did going up. It also means BatchNorm's gradient at one example really depends on the others, which is the mechanism behind both its regularizing noise and its small-batch fragility. A framework computes all of this automatically the moment you call `loss.backward()`, but the two "extra" terms are exactly what a hand-written BatchNorm must not forget.

Normalization in code

The first cell implements the BatchNorm forward pass twice: once on the four scalars of the worked example below, so you can check the arithmetic to the last decimal, and once on a deliberately off-scale random batch, to confirm that every feature comes out zero-mean and unit-variance. Press **Run**:

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)

# --- the worked example: ONE feature, a batch of 4 examples ---
x = np.array([[2.], [4.], [6.], [8.]])          # column = one feature over
the batch
mu  = x.mean(0)                                  # mu_B  -> [5.]
var = x.var(0)                                   # sigma^2_B (population,
/m) -> [5.]
xhat = (x - mu) / np.sqrt(var + 1e-5)
y = 2.0 * xhat + 1.0                             # gamma = 2, beta = 1
print("mu_B =", mu, " var_B =", var)             # [5.] [5.]
print("xhat =", xhat.ravel())                    # [-1.3416 -0.4472 0.4472
1.3416]
print("mean(xhat) =", round(float(xhat.mean()), 6),
```

```
      " var(xhat) =", round(float(xhat.var()), 6))   # ~0 and ~1
print("y    =", y.ravel())                          # [-1.6833 0.1056 1.8944
3.6833]

# --- BatchNorm forward on an off-scale (N x D) batch: normalize DOWN each
column ---
N, D = 64, 5
X = np.random.randn(N, D) * np.array([3, .5, 10, 2, 7]) + np.array([1, -4,
20, 0, 8])
mu, var = X.mean(0), X.var(0)                       # per-feature stats over
the batch
Xhat = (X - mu) / np.sqrt(var + 1e-5)
print("per-feature mean of xhat:", np.round(Xhat.mean(0), 4))   # all ~0
print("per-feature var  of xhat:", np.round(Xhat.var(0), 4))    # all ~1
```

The four standardized scalars are ±1.3416 and ±0.4472. They are symmetric about zero, with mean 0 and variance 1 exactly, and the random batch confirms that BatchNorm flattens any per-feature scale and shift back to the same clean distribution. The second cell does the LayerNorm computation on a single off-scale example, standardizing *across its features* rather than across a batch:

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(1)
d = 6
x = np.random.randn(d) * 4 + 10                     # ONE example, 6 features,
off-scale
mu  = x.mean()                                      # mean ACROSS the 6
features
var = x.var()                                       # var  ACROSS the 6
features
xhat = (x - mu) / np.sqrt(var + 1e-5)
gamma, beta = np.ones(d), np.zeros(d)
y = gamma * xhat + beta
print("x    =", np.round(x, 3))
print("xhat =", np.round(xhat, 3))
print("mean across features =", round(float(y.mean()), 6))   # ~0
print("var  across features =", round(float(y.var()),  6))   # ~1
```

No batch appeared anywhere: LayerNorm's output for this example is a function of the example alone, which is exactly why it needs no running averages and no train/eval switch. In PyTorch each normalizer is a single module, and the only extra step BatchNorm requires is remembering to switch between training and evaluation mode:

PyTorch copy → Colab

```
import torch, torch.nn as nn

x = torch.randn(32, 64)                          # batch of 32, 64 features

bn = nn.BatchNorm1d(64)                           # per-feature, statistics
over the batch
ln = nn.LayerNorm(64)                             # per-example, statistics
over the 64 features
img = torch.randn(8, 32, 16, 16)                 # (N, C, H, W)
gn = nn.GroupNorm(num_groups=4, num_channels=32)   # per-example, per
channel-group

bn.train(); y = bn(x)                             # TRAIN: use batch stats,
update running mean/var
bn.eval();  y = bn(x)                             # EVAL:  use the frozen
running mean/var (batch-free)
print(ln(x).shape, gn(img).shape)                # torch.Size([32, 64])
torch.Size([8, 32, 16, 16])

class RMSNorm(nn.Module):                         # as used in LLaMA / T5: no
mean subtraction
    def __init__(self, d, eps=1e-6):
        super().__init__()
        self.g = nn.Parameter(torch.ones(d)); self.eps = eps
    def forward(self, x):
        rms = x.pow(2).mean(-1, keepdim=True).add(self.eps).sqrt()
        return self.g * x / rms                   # scale by the root-mean-
square only
print(RMSNorm(64)(x).shape)                       # torch.Size([32, 64])
```

Worked example: batch-normalizing four scalars by hand

One feature takes the values $x = [2, 4, 6, 8]$ across a mini-batch of four. Apply BatchNorm with learnable $\gamma = 2$, $\beta = 1$ (take ε negligible against σ^2_B).

1. Batch mean: $\mu_B = (2 + 4 + 6 + 8)/4 = 20/4 = \mathbf{5}$.
2. Batch variance: $\sigma^2_B = [(2-5)^2 + (4-5)^2 + (6-5)^2 + (8-5)^2]/4 = (9 + 1 + 1 + 9)/4 = 20/4 = \mathbf{5}$, so $\sigma_B = \sqrt{5} = \mathbf{2.2361}$.
3. Standardize each value, $\hat{x}_i = (x_i - 5)/2.2361$: $\hat{x} = [-3, -1, +1, +3]/2.2361 = [\mathbf{-1.3416}, \mathbf{-0.4472}, \mathbf{+0.4472}, \mathbf{+1.3416}]$.
4. Check the standardization: $\text{mean}(\hat{x}) = (-1.3416 - 0.4472 + 0.4472 + 1.3416)/4 = \mathbf{0}$; $\text{var}(\hat{x}) = (1.8 + 0.2 + 0.2 + 1.8)/4 = 4/4 = \mathbf{1}$. Zero-mean, unit-variance, as promised.
5. Scale and shift, $y_i = 2 \cdot \hat{x}_i + 1$: $y = [\mathbf{-1.6833}, \mathbf{0.1056}, \mathbf{1.8944}, \mathbf{3.6833}]$.

What the numbers mean: standardization removed the incoming mean of 5 and the spread of $\sqrt{5}$, mapping the batch onto the clean template $\hat{x}$. The learnable γ, β then *chose* a new scale and center (the outputs now have mean $\beta = 1$ and standard deviation $\gamma = 2$), so BatchNorm fixes the *shape* of the distribution while leaving the network free to pick the size. If the network had instead learned $\gamma = \sqrt{\sigma^2_B + \varepsilon} = 2.2361$ and $\beta = 5$, the outputs would have reproduced the original $[2, 4, 6, 8]$ exactly: normalization can always undo itself, which is why it costs no expressive power.

Assumptions & limitations

BatchNorm assumes the mini-batch is a large, representative, i.i.d. sample of the data. This assumption fails for tiny batches, for non-i.i.d. batches, and inside recurrent models, and it is the reason LayerNorm, GroupNorm, and the others like them exist. It introduces a train/inference gap (batch statistics versus running averages) that quietly corrupts

predictions if you forget to switch to evaluation mode. Its output for one example also depends on the others in the batch, which helps regularization but is a hazard for reproducibility. None of these layers is free: each adds parameters (γ, β), extra memory for the saved statistics, and a synchronization cost across devices for BatchNorm. And normalization does not replace good initialization (Chapter 10) or a sensible learning rate. It gives you more room for error around them, but a badly mis-scaled network or an absurd step size will still fail. Finally, the reason it helps is still debated. Treat "internal covariate shift" as a historical name, not a mechanism.

Why & when this matters

Use normalization whenever a network is deep enough that activation scales drift during training, which, past a few layers, is almost always. Use **BatchNorm** for convolutional vision models trained with large enough batches. Switch to **GroupNorm** when memory forces small batches (detection, segmentation, video). Use **LayerNorm** (or **RMSNorm**) for anything sequential: RNNs, transformers, language models. The practical benefit is that you can train faster, with higher learning rates, and worry far less about the exact initialization. When a deep network trains slowly, diverges at learning rates that "should" be fine, or is very sensitive to its starting weights, a missing or misplaced normalization layer is one of the first things to check.

In the real world

Normalization is essential in almost every model you have used. Every modern convolutional vision backbone (ResNet, EfficientNet) is built from BatchNorm layers, and their small-batch detection and segmentation variants switch to GroupNorm to keep working when only a few high-resolution images fit in memory. Every transformer, from the original translation model to the language model answering you now, wraps

LayerNorm around each attention and feed-forward sub-layer (Chapter 16), and the largest recent LLMs have largely switched to RMSNorm to reduce the cost of that operation across trillions of tokens. Style-transfer and image-generation pipelines rely on InstanceNorm to separate content from appearance. The specific normalizer is one of the defining choices of an architecture. But having *some* normalization present is, for anything deep, no longer optional.

Exercises

1. Batch-normalize the feature $x = [0, 2, 2, 8]$ over its four examples, with $\gamma = 2$, $\beta = 1$ (ε negligible). Compute μ_B, σ^2_B, each $\hat{x}_i$, verify mean($\hat{x}$) = 0 and var($\hat{x}$) = 1, then give the outputs y_i. MATH

2. BatchNorm behaves differently in training and inference. Explain what statistics it uses in each mode, why the difference exists, and the concrete bug that appears if a model is left in training mode at test time. CONCEPT

3. Show that BatchNorm can represent the identity: find the γ and β for which $y_i = x_i$ for all i. What does this prove about the layer's expressive power? MATH

4. Implement LayerNorm on a batch of examples (N×D) and confirm each *row* comes out zero-mean and unit-variance, the opposite axis from BatchNorm. Show that the columns are *not* individually standardized. CODE

5. Why do transformers and RNNs use LayerNorm rather than BatchNorm? Give two distinct reasons based on how those models process data. CONCEPT

6. RMSNorm drops the mean subtraction: $y_k = \gamma_k \, x_k / \mathrm{RMS}(x)$ with $\mathrm{RMS}(x) = \sqrt{(1/D) \sum x_k^2 + \varepsilon}$. Under what condition on the input does RMSNorm give the same standardized direction as LayerNorm, and why is it cheaper? MATH

7. Demonstrate BatchNorm's small-batch fragility: estimate how the sampling noise in the batch mean shrinks with batch size m, and compare against the $1/\sqrt{m}$ law. CODE

8. The backward pass through BatchNorm has two "extra" terms beyond the naive gradient: $\partial L/\partial x_i \propto m\cdot(\partial L/\partial \hat{x}_i) - \Sigma_j\, \partial L/\partial \hat{x}_j - \hat{x}_i\, \Sigma_j\, (\partial L/\partial \hat{x}_j)\, \hat{x}_j$. Explain where each of the two subtracted terms comes from, and why the gradient must depend on the whole batch. MATH

Regularization & generalization

> *A network that scores perfectly on its training data but fails on new data has learned nothing useful. It has **memorized**, not generalized. This chapter defines the generalization gap, explains overfitting through the bias–variance trade-off, and derives the common tools that fight it: L2 weight decay, L1 sparsity, dropout, early stopping and data augmentation. It then looks at the modern puzzle of why today's hugely over-parameterized networks, which could easily memorize their data, so often generalize anyway.*

Fundamental concept

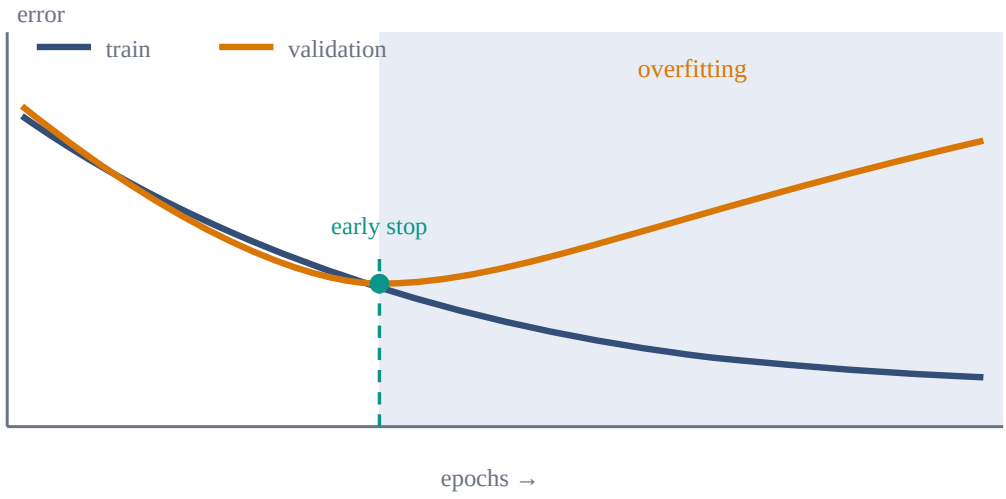
A student who memorizes last year's exam answers word-for-word can get full marks on that exact paper and still fail this year's. The point of studying is to understand the pattern, not the paper. Machine learning has the same split. The **training error** measures performance on the examples the model has already seen. The **test error** (estimated on a held-out validation set) measures performance on examples it has not. The difference between them is the **generalization gap**, and closing it is the whole goal.

When a model has more **capacity** — more neurons, more layers, more freedom — than the data needs, it starts fitting the *noise* in the training set: the random details of these particular samples that will not appear again. Training error keeps falling while test error turns around and rises. That is **overfitting**. The opposite failure, **underfitting**, is a model too simple to capture the real pattern, so both errors stay high. The sweet spot is between them, and this trade-off has a classic name: **bias–variance**. A rigid model has high *bias* (it is systematically wrong) but low variance (it is stable across datasets). A flexible model has low bias but high *variance* (it changes a lot depending on which samples it saw). Formally the expected error at a point splits into three parts:

$$\mathbb{E}[(y - \hat{y})^2] = \text{bias}^2 + \text{variance} + \sigma_{\text{noise}}^2$$

We can change only the first two. The noise term is irreducible. Regularization is any technique that deliberately adds a little bias to get a large reduction in variance.

The most common tool is to keep the weights **small**. Small weights make the function *smoother* — nearby inputs get nearby outputs — so the model cannot create the very bumpy boundaries needed to fit individual noisy points. That single idea is the basis of weight decay, and related methods attack the same problem in different ways.



The clear sign of overfitting. Training error (blue) keeps falling, but validation error (amber) reaches its lowest point and then rises as the model begins to memorize noise. Early stopping stops training at the teal marker — the bottom of the U. This is exactly the plot the Overfitting lab draws.

The bias–variance decomposition

The three-way split stated above is worth deriving, because it is the mathematical basis of the whole chapter. Fix a single input x and imagine drawing many different training sets, each producing a slightly different fitted model. Write $\hat{y}$ for the model's prediction at x — a *random* quantity, since it depends on which training set we drew — and $y = \mathbb{E}[\hat{y}]$ for its average over all those draws. The target is $y = f(x) + \varepsilon$, where f is the true function and ε is observation noise with mean 0 and variance σ^2. The expected squared error then separates into three non-negative parts:

1. Separate out the noise. Since $y - \hat{y} = (f - \hat{y}) + \varepsilon$ with ε independent of $\hat{y}$ and zero-mean, the cross-term averages to zero, leaving $\mathbb{E}[(y - \hat{y})^2] = \mathbb{E}[(f - \hat{y})^2] + \sigma^2$.

2. Add and subtract the mean prediction: $f - \hat{y} = (f - y) + (y - \hat{y})$. Square and take the expectation. The new cross-term disappears because $\mathbb{E}[y - \hat{y}] = 0$, so $\mathbb{E}[(f - \hat{y})^2] = (f - y)^2 + \mathbb{E}[(\hat{y} - y)^2]$.
3. Name the remaining terms: $(f - y)^2$ is the squared **bias** (how far the average model is from the truth), $\mathbb{E}[(\hat{y} - y)^2]$ is the **variance** (how much the model varies between datasets), and σ^2 is the **irreducible noise**.

$$\mathbb{E}[(y - \hat{y})^2] = (f - y)^2 + \mathbb{E}[(\hat{y} - y)^2] + \sigma^2 = \text{bias}^2 + \text{variance} + \text{noise}$$

f	the true, noise-free value at the point x
$\hat{y}$	the model's prediction at x — random, because the training set is random
$y = \mathbb{E}[\hat{y}]$	the average prediction over many independently drawn training sets
σ^2	variance of the observation noise ε — irreducible; no model can beat this floor

A rigid model keeps y nearly fixed across datasets — small variance, large bias. A flexible one follows each dataset — small bias, large variance. Because only these two are under our control, **regularization deliberately accepts a little bias to gain a larger reduction in variance**. The noise floor σ^2 cannot be changed. The worked example below works through a five-dataset split with numbers.

Governing equations

The quantity we really care about but cannot see is the generalization gap:

$$\text{gap} = E_{\text{test}} - E_{\text{train}}$$

L2 regularization (weight decay) adds a penalty on weight size to the loss, pushing the optimizer to keep weights small:

$$J = L + \frac{\lambda}{2m} \Sigma_i w_i^2$$

L1 regularization penalizes absolute value instead, which drives many weights exactly to zero (feature selection):

$$J = L + \frac{\lambda}{m} \Sigma_i |w_i|$$

Inverted dropout multiplies each hidden activation by a random 0/1 mask that keeps a unit with probability (1−p), then rescales the surviving units so the expected output is unchanged:

$$\tilde{a}_i = \frac{\text{mask}_i}{1 - p} \cdot a_i, \quad \mathbb{E}[\tilde{a}_i] = (1 - p) \cdot \frac{a_i}{1 - p} = a_i$$

L	the ordinary data loss (e.g. cross-entropy or MSE, Chapter 6)
J	the regularized objective actually minimized
λ	regularization strength (≥ 0); larger λ means stronger pull toward small/zero weights
m	number of training examples (normalizes the penalty per example)
w_i	an individual weight (biases are usually left un-penalized)
p	dropout rate — probability a hidden unit is silenced on a given step
mask_i	a random draw, 1 with probability (1−p) and 0 with probability p
a_i, $\tilde{a}_i$	a hidden activation and its dropped-and-rescaled version

How it works

Weight decay gets its name from the update rule. Differentiate the L2 objective and substitute it into a gradient-descent step (gradient descent, Chapter 7):

1. The penalty's gradient is $\partial/\partial w$ of $(\lambda/2m)w^2 = (\lambda/m)w$, so the full gradient simply adds this term to the data gradient that backpropagation supplies (Chapter 8): $\partial J/\partial w = \partial L/\partial w + (\lambda/m)w$.
2. A gradient-descent step becomes $w \leftarrow w - \eta(\partial L/\partial w + (\lambda/m)w)$.
3. Group the w terms: $w \leftarrow (1 - \eta\lambda/m) \cdot w - \eta \cdot \partial L/\partial w$.

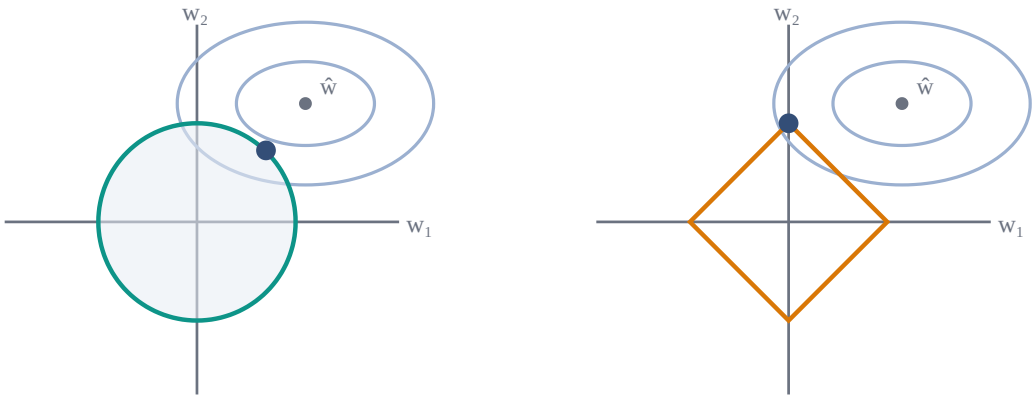
Every step therefore first *shrinks* the weight by the constant factor $(1 - \eta\lambda/m) < 1$ and then applies the usual data-driven update. The weight continuously "decays" toward zero, stopped only where the data actively pushes it:

$$w \leftarrow (1 - \frac{\eta\lambda}{m})\, w - \eta \frac{\partial L}{\partial w}$$

(With adaptive optimizers this decoupling is done explicitly by **AdamW** — Chapter 9, optimizers.) The L1 penalty instead adds a subgradient of constant size:

$$\frac{\partial}{\partial w}\left(\frac{\lambda}{m}|w|\right) = \frac{\lambda}{m} \cdot \operatorname{sign}(w)$$

This is a push toward zero of the *same size* no matter how small w already is, which is why L1 forces many weights to exactly zero and produces **sparse** models. The difference is easiest to see geometrically. Imagine the loss as elliptical contours around the unpenalized optimum $\hat{w}$, and the penalty as a budget that limits the solution to a ball around the origin — a round ball for L2, a pointed diamond for L1. The regularized answer is where the smallest reachable contour first touches that budget region.



Why the norm shapes the answer (schematic). Gray ellipses are loss contours around the unpenalized optimum $\hat{w}$. The colored region is the weight budget. The round L2 ball is first touched at a smooth off-axis point, so both weights stay small but nonzero. The L1 diamond has corners on the axes, and contours almost always touch a corner first — forcing a weight to exactly zero.

The figure's main point — corners make sparsity — deserves a sharper argument. The $\ell 1$ "diamond" $\{w : \|w\|_1 \le t\}$ has its vertices exactly on the axes, where all but one coordinate are zero, and all its curvature is concentrated at those vertices while the flat faces do not bend at all. An expanding elliptical loss contour therefore almost always makes first contact at a pointed

vertex, just as a smooth curve touches a convex body where it bends most. Only a contour arriving perfectly parallel to a face touches mid-face — a coincidence of measure zero. In d dimensions the cross-polytope has faces of every order, and a typical contact lands on one where *most* coordinates are forced to zero, so sparsity grows with dimension. The round ℓ2 ball has identical curvature everywhere and no special points, so contact almost never lands on an axis: L2 shrinks weights but never zeroes them. For a single decoupled coordinate the rules are exact — L1 is the **soft-threshold** $w^* = \text{sign}(\hat{w}) \cdot \max(0, |\hat{w}| - \lambda)$, set to zero whenever $|\hat{w}| \le \lambda$, while L2 is the proportional shrink $w^* = \hat{w} / (1 + \lambda)$, reaching zero only if $\hat{w}$ was already zero.

Why weight decay works: the Bayesian view

Weight decay can look like an arbitrary "keep the weights small" trick. In fact it is the exact result of a principled belief. Ordinary training performs **maximum likelihood** — it chooses the weights that make the observed data most probable, i.e. minimizes the negative log-likelihood $-\log p(D \mid w)$, which is exactly the data loss L. Bayesian reasoning instead adds a **prior** $p(w)$, our belief about the weights *before* seeing any data, and gives the **MAP** (maximum a posteriori) estimate — the peak of the posterior $p(w \mid D) \propto p(D \mid w)\, p(w)$. Taking negative logarithms turns that product into a sum:

$$\hat{w}_{\text{MAP}} = \operatorname{argmin}_w \left[-\log p(D \mid w) - \log p(w) \right] = \operatorname{argmin}_w \left[L - \log p(w) \right]$$

Now place an independent zero-mean Gaussian prior on every weight, $w_j \sim N(0, \tau^2)$: we state in advance that weights are probably small. Its negative log is $-\log p(w) = (1/2\tau^2) \sum_j w_j^2 + \text{const}$, and with Gaussian observation noise of variance σ^2 the data term is $(1 / 2\sigma^2) \sum (y - \hat{y})^2$. Minimizing the sum, then multiplying through by σ^2 (which cannot move the minimizer), gives

$$\hat{w}_{\text{MAP}} = \operatorname{argmin}_w \left[\tfrac{1}{2} \sum (y - \hat{y})^2 + \frac{\lambda}{2} \|w\|^2 \right], \quad \lambda = \frac{\sigma^2}{\tau^2}$$

— exactly L2 regularization, with the strength shown as a *ratio of variances*. A tighter prior (small τ^2, a firm belief that weights are close to zero) makes λ large and shrinks strongly. A flat prior ($\tau^2 \to \infty$) sends $\lambda \to 0$ and gives back plain maximum likelihood. So weight decay is Gaussian-prior MAP estimation: its pull of every weight toward zero is just the prior pulling the

estimate back to its mean of zero. The same reasoning re-derives L1 — use a **Laplace** prior $p(w) \propto \exp(-\Sigma|w_j|/b)$, whose sharp spike at the origin is what forces weights to zero. Carrying the whole posterior forward, not just its peak $\hat{w}_{MAP}$, is the subject of Chapter 24 (uncertainty).

Dropout works by a different mechanism: **ensembling**. Because a random subset of units is silenced on every step, no unit can rely on any particular partner being present, so the network cannot build fragile co-adapted detectors. Each unit must be useful on its own. Training with dropout is like training an exponential family of thinned sub-networks that share weights, then averaging them at test time. The $1/(1-p)$ rescaling (done during training — hence "inverted") makes the expected activation match, so **at test time dropout is simply switched off** and the full network is used unchanged.

There is a precise sense in which this "averaging" is literal. A layer with n droppable units defines 2^n thinned sub-networks that share one pool of weights. Each step samples one and updates it, so the shared weights end up encoding an **ensemble** of exponentially many models. Averaging that ensemble exactly is not feasible, but scaling activations by the keep-probability at test time — which inverted dropout has already built into training — is a fast approximation to the average (exact for one linear layer, close otherwise). Dropout thus gets much of a giant ensemble's variance reduction for the cost of one network, and the pressure it applies forbids **co-adaptation**: since any other unit may disappear on the next step, every unit must be useful on its own rather than as one part of a fragile detector.

More ways to regularize: early stopping and augmentation

Weight decay and dropout are only two members of a large family. Every practical recipe (Chapter 25, the training recipe) combines several. Four more deserve their own name.

Early stopping is the cheapest regularizer of all, and you already do most of the work for it: just watch the validation error each epoch, keep a copy of the weights whenever it reaches a new low, and stop once it has failed to improve for a fixed number of epochs — the *patience*. At the end you restore the best-scoring weights, not the last ones. Why does stopping early regularize? Training starts from small initial weights (Chapter 10, initialization) and grows them. Stopping the run therefore limits how large the weights can become, so limiting *training*

time acts as a soft limit on *effective capacity*. For a quadratic loss one can show the two are exactly equivalent: stopping after a finite number of gradient steps is equivalent to an L2 penalty whose strength shrinks as training continues. Early stopping needs no extra hyperparameter search and costs nothing beyond the validation checks you already run.

Data augmentation attacks variance from the data side rather than the model side. If you know a transformation that changes the input but not the label, you can make new training examples for free. For an image, flip, crop, rotate or color-jitter it. For a spectrogram, add background noise or shift the pitch. For text, swap a word for a synonym. Each augmented copy tells the network "this variation still means the same thing", which is exactly the invariance a plain convolution (Chapter 13) does not get for free. On limited data, augmentation is often the single most effective regularizer — more powerful than any weight penalty, because it adds real prior knowledge about the task, the data-centric view developed in Chapter 26.

Two lighter tools complete the toolkit. **Label smoothing** replaces hard one-hot targets (1 for the true class, 0 elsewhere) with slightly softened ones (say 0.9 and a small share spread over the rest), which stops the network from driving its logits to ±∞ and over-confidently memorizing, and it also improves the calibration of the predicted probabilities (Chapter 24, uncertainty). And **noise injection** — adding small random changes to inputs, weights or activations — is, for the input case, mathematically close to L2 weight decay. Dropout itself is one example of this idea. The common theme is clear: make the model's job slightly harder or its answers slightly less confident, and it generalizes better.

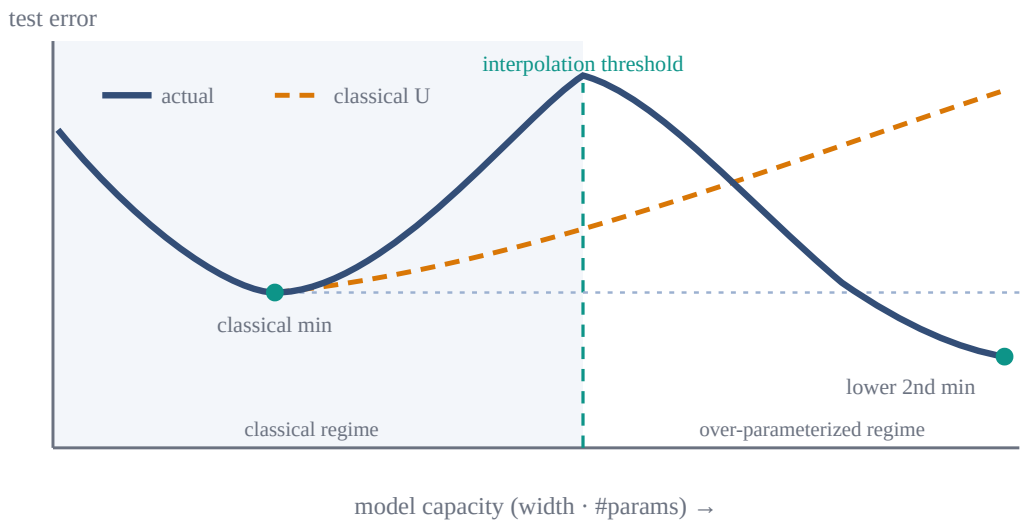
Double descent: why over-parameterized networks still generalize

The bias–variance U tells a simple story: beyond some optimal capacity, variance grows fast and error must rise, so the best model is the smallest that still fits. Modern deep learning clearly breaks this. State-of-the-art networks have far more parameters than training examples — enough to fit even random labels — yet generalize very well. The resolution, **double descent**, does not overturn the decomposition above. Instead it reveals a regime the classical picture never reached.

Plot test error against capacity and grow the model past the point where it fits the training set exactly — the **interpolation threshold**, where training error first reaches zero. The curve falls (underfitting), rises to a peak at the threshold (the classical overfitting spike), then — surprisingly — falls a *second* time, often below the classical sweet spot. Falls, rises, falls again: two descents, which is where the name comes from.

Why the second fall? At the threshold there is basically one way to fit every point exactly, and forced through noisy data it must bend sharply — maximal variance, the peak. Past the threshold there are *infinitely many* zero-error solutions, so the optimizer can prefer a well-behaved one: gradient descent from small weights moves to the smallest-norm interpolant (Chapter 10, initialization), and weight decay and normalization (Chapter 11) pull in the same direction. Among the huge set of solutions these **implicit and explicit biases** reliably choose a smooth, low-norm function that generalizes — extra capacity adds not variance but *room to find a flat solution*.

The honest summary is not "big models cannot overfit" — they can, and will memorize random labels if you remove every safeguard — but "at scale, the interpolating solution regularized SGD actually finds tends to be simple." This is why the modern recipe (Chapter 25) is **make the model big, then regularize** rather than search for the smallest network that fits. Double descent now appears along width, along epochs, and along dataset size — a reminder that the bias–variance trade-off describes a *fixed* model class, not a rule against larger and larger ones.



Double descent. The classical analysis predicts the dashed amber curve — error must rise once the model can memorize. The solid blue curve is what deep networks do: past the interpolation threshold (teal), where training error first reaches zero, test error peaks and then falls a second time, often below the classical sweet spot — because among the many zero-error solutions, regularized gradient descent chooses a smooth one.

Regularization in code

The clearest way to *feel* overfitting is to build it, then remove it. This cell fits a degree-10 polynomial to eleven noisy points twice — once with plain least squares, once with an L2 (ridge) penalty — and reports the error on the training points versus a dense grid of the true curve. Press **Run**:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)
f = lambda t: np.sin(2.5*t)
xtr = np.sort(rng.uniform(-1, 1, 11))          # 11 unevenly-spaced samples
ytr = f(xtr) + rng.normal(0, 0.15, xtr.shape)  # with observation noise
xte = np.linspace(-1, 1, 200)                  # dense grid of the TRUE
curve
V = lambda t: np.vander(t, 11)                 # degree-10 polynomial
features
rmse = lambda a, b: float(np.sqrt(np.mean((a-b)**2)))

w_ols = np.linalg.lstsq(V(xtr), ytr, rcond=None)[0]        # no penalty
A = V(xtr); lam = 0.3                                       # L2 / ridge:
w_l2 = np.linalg.solve(A.T@A + lam*np.eye(11), A.T@ytr)    # (AᵀA + λI)w =
```

```
A^T y

for name, w in [("no L2 ", w_ols), ("with L2", w_l2)]:
    print(f"{name}: train RMSE={rmse(V(xtr)@w, ytr):8.3f}   "
          f"test RMSE={rmse(V(xte)@w, f(xte)):8.3f}   max|w|=
{np.abs(w).max():.1f}")
```

The unpenalized fit interpolates every training point (train error ≈ 0) but its weights grow past a million and it swings wildly between the samples, so its test error is huge. The L2 fit accepts a little training error, keeps every weight near 1, and generalizes — the overfitting gap closes. That is the whole chapter in six printed lines. Now watch the second mechanism, inverted dropout, preserve the expected activation while randomly silencing units:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
a    = np.array([2.0, -1.0, 4.0, 0.5])   # a hidden layer's activations
p    = 0.5; keep = 1.0 - p                # drop rate p -> keep probability
(1-p)
mask = np.array([1., 0., 1., 0.])         # one sampled keep/drop mask

print("scale 1/(1-p) =", 1/keep)
print("kept & scaled =", a * mask / keep)          # survivors amplified ->
[4, 0, 8, 0]

# averaged over many random masks the expected activation is unchanged:
rng = np.random.RandomState(0)
M = rng.binomial(1, keep, size=(200000, a.size)) / keep    # random inverted
masks
print("E[dropped]    ~", np.round((M * a).mean(0), 3), " vs a =", a)
```

The surviving units are doubled to make up for the missing half, so the layer's expected output is exactly the undropped activation — which is why inference can drop the whole mechanism and use the full network. In PyTorch both tools are one line each: weight decay is an optimizer argument, and `nn.Dropout` is a layer that is active only in training mode:

PyTorch copy → Colab

```python
import torch, torch.nn as nn

net = nn.Sequential(
    nn.Linear(20, 128), nn.ReLU(),
```

```
    nn.Dropout(0.5),                          # inverted dropout, active in
train() only
    nn.Linear(128, 1))

# L2 weight decay lives in the optimizer; AdamW decouples it cleanly
(Chapter 9)
opt = torch.optim.AdamW(net.parameters(), lr=1e-3, weight_decay=1e-2)

net.train()                                   # dropout ON, masks resampled each
step
for xb, yb in loader:                         # your DataLoader of (features,
target)
    opt.zero_grad()
    loss = nn.functional.mse_loss(net(xb), yb)
    loss.backward(); opt.step()

net.eval()                                    # dropout OFF at test time — full
network used
```

In this app's engine

The Overfitting lab is this chapter made interactive: pick a small noisy dataset, use a big network on it, and watch the validation curve separate from the training curve into the U-shape of the figure. Sliders for L2 (λ), dropout (p) and gradient clipping let you pull the two curves back together and see the generalization gap shrink in real time. The Playground uses **inverted** dropout, so the scaling happens during training and inference needs no adjustment. Its Math Log shows the pre- and post-dropout activations for a probed unit.

Worked example — weight decay and a dropout mask

Take $\lambda = 0.1$, $m = 100$ training examples, learning rate $\eta = 0.5$, and a weight vector $w = [0.8, -0.6, 0.0, 0.3]$.

1. The L2 penalty added to the loss is $(\lambda / 2m)\,\Sigma\, w^2 = (0.1 / 200)(0.8^2 + 0.6^2 + 0^2 + 0.3^2) = 0.0005 \cdot 1.09 =$ **0.000545** — deliberately small, since it is averaged over all $m = 100$ examples.

2. Its gradient contribution is $(\lambda/m)\, w = 0.001 \cdot [0.8, -0.6, 0, 0.3] = [\mathbf{0.0008}, \mathbf{-0.0006}, \mathbf{0}, \mathbf{0.0003}]$ (in the un-normalized convention $J = L + \tfrac{1}{2}\lambda\|w\|^2$ this is simply $\lambda w = [0.08, -0.06, 0, 0.03]$). The pull on each weight is *proportional to that weight* — the 0.8 weight gets a pull of 0.0008, the zero weight gets nothing — and, times η, it is exactly the shrink applied next.
3. The per-step decay factor is $1 - \eta\lambda/m = 1 - (0.5 \cdot 0.1)/100 = 1 - 0.0005 =$ **0.9995**. Each step multiplies every weight by 0.9995 before the data update.
4. Ignoring the data term, after 1000 steps a weight is scaled by $0.9995^{1000} = \exp(1000 \cdot \ln 0.9995) = \exp(-0.5001) \approx$ **0.606**. With no data pushing it, a weight loses about 40% of its size — a steady pressure toward simplicity.
5. Now dropout with $p = 0.5$, so the keep-scale is $1/(1-0.5) =$ **2**. Take a hidden vector $a = [2, -1, 4, 0.5]$ and a sampled mask $[1, 0, 1, 0]$ (keep units 1 and 3, drop 2 and 4).
6. Apply mask then rescale $\times 2$: $\tilde{a} = [2 \cdot 2, 0, 4 \cdot 2, 0] =$ **[4, 0, 8, 0]**.
7. **Meaning.** The surviving units are made larger to make up for the missing half, so the expected output stays the same: for unit 1, $\mathbb{E}[\tilde{a}_1] = 0.5 \cdot 4 + 0.5 \cdot 0 = 2 = a_1$. The network sees the right average signal on every step but can never depend on any single unit.

Worked example — a bias–variance split by hand

At a fixed test point the true value is $f = 1.00$, and observations have noise with standard deviation $\sigma = 0.20$, so the irreducible term is $\sigma^2 =$ **0.0400**. We retrain the same model on five independent datasets and record its prediction at that point: 1.30, 0.90, 1.50, 0.80, 1.00.

1. Average prediction: $y = (1.30 + 0.90 + 1.50 + 0.80 + 1.00)/5 = 5.50/5 =$ **1.10**.
2. Bias: $y - f = 1.10 - 1.00 = 0.10$, so bias$^2 =$ **0.0100** — the model is a little high on average.

3. Variance: the deviations $\hat{y} - y = [0.20, -0.20, 0.40, -0.30, -0.10]$ have squares $[0.04, 0.04, 0.16, 0.09, 0.01]$ summing to 0.34, so variance = $0.34 / 5$ = **0.0680**.
4. Combine: $\mathbb{E}[(y - \hat{y})^2] \approx \text{bias}^2 + \text{variance} + \sigma^2 = 0.0100 + 0.0680 + 0.0400$ = **0.1180**.

What the numbers mean: variance (0.068) is much larger than bias² (0.010), so nearly all of the *reducible* error is the model changing from dataset to dataset — the high-variance, overfitting regime. Regularization trades bias for variance, and it helps here: changing the model to be more rigid so that bias rises to 0.20 (bias² = 0.04) while variance drops to 0.02 would *lower* the total to 0.04 + 0.02 + 0.04 = 0.10. No amount of tuning, however, can get below the 0.04 noise floor.

Assumptions & limitations

Regularization assumes the problem really is overfitting. Applying it to an *underfitting* model only makes things worse. It also assumes the validation set is a fair, independent sample of the target distribution — if validation data leaks into training, or comes from a different distribution, the U-curve is misleading. There is no single best λ or p: they are hyperparameters tuned on validation data, and too much λ under-fits while too little does nothing. Dropout slows convergence and does not work well with batch normalization (Chapter 11), so modern architectures often prefer weight decay plus heavy data augmentation. And no amount of regularization can replace the best fix of all — more and better data.

Why & when this matters

Use the two curves as a diagnosis. A **large gap** (low train error, high validation error) is overfitting: add regularization (L2, dropout), stop early, augment, or gather more data. **Both errors high** is underfitting: the model lacks capacity or has not trained long

enough — add layers/width, train longer, or reduce λ (the recipe of Chapter 25). Early stopping is the cheapest regularizer you already have — just keep the weights from the validation minimum. Use these tools whenever training data is limited or the network is large compared to it, which in practice is nearly always.

In the real world

Every large model you have used uses a lot of regularization. Vision backbones such as **ResNet** are trained with weight decay around 1e-4 plus heavy augmentation (random crops, flips, and mixing schemes like *mixup* and *Cutout*). The AlexNet paper (2012) credited dropout and augmentation for a large part of its ImageNet win. Modern large language models rely on weight decay via **AdamW** and on label smoothing, and their huge amount of training data is itself the regularizer — with trillions of tokens, memorizing is harder than learning the pattern. Whenever a deployed model is retrained on a smaller in-house dataset (fine-tuning), these settings — dropout rate, weight decay, early-stopping patience — are among the first an engineer tunes to keep the gap closed.

Exercises

1. With learning rate η = 0.1, penalty λ = 0.01 and m = 50 examples, compute the per-step weight-decay factor, then the fraction of its size a weight keeps after 10,000 pure-decay steps. MATH

2. For inverted dropout with keep probability (1−p), show that the rescaled activation ã has mean a but is noisier, and give its variance. MATH

3. Modify the polynomial cell to try λ over several values and print which λ gives the lowest test error — a one-line model-selection loop. CODE

4. Why are bias terms usually left out of the L2/L1 penalty, while weights are penalized?

   CONCEPT

5. Estimate bias² and variance numerically as model complexity grows, and confirm the trade-off. CODE

6. A weight is at $w = 0.005$ with data-gradient $\partial L/\partial w = 0.002$, and $\lambda/m = 0.02$. Compare one L2 step with one L1 step ($\eta = 1$) and explain why L1 can reach exactly zero but L2 cannot.

   MATH

7. You train a classifier and see 99% training accuracy but 71% validation accuracy that rose early and then fell. Diagnose the situation and list three concrete fixes in priority order.

   CONCEPT

8. Show that an independent Gaussian prior $w_j \sim N(0, \tau^2)$ on the weights turns MAP estimation into L2 regularization, and identify λ in terms of the noise and prior variances.

   MATH

9. A colleague objects: "our network has ten times more parameters than training examples, so it is guaranteed to overfit — we must shrink it." Use double descent to argue against this and name two things you would check first. CONCEPT

10. In code, apply one L2 shrink step and one L1 soft-threshold step to the same small weight vector, and confirm that only L1 produces exact zeros. CODE

PART III

Architectures

Beyond the fully-connected network — convolutional networks that see, recurrent networks that remember, attention and transformers that read, graph networks that reason over structure, and autoencoders and generative models that imagine.

13 Convolutional neural networks

14 Modern CNN architectures & computer vision

15 Recurrent networks & sequences

16 Attention & the transformer

17 Language models & the transformer era

18 Graph neural networks

19 Autoencoders & representation learning

20 Self-supervised & contrastive learning

21 Generative models: GANs to diffusion

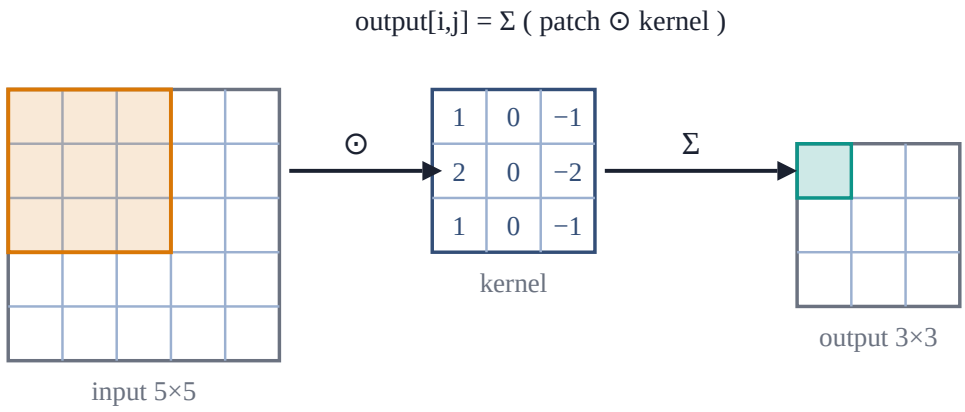
Convolutional neural networks

> *A photograph is a grid of numbers, and neighboring numbers belong together. An eye is an eye whether it sits top-left or bottom-right. The convolutional network is the architecture that builds these two facts, locality and repetition, directly into its wiring, so it learns to see with a thousand times fewer parameters than a plain network would need. This chapter derives the convolution, counts the savings, and slides one edge-detecting filter across an image by hand.*

Fundamental concept

Imagine describing a face to someone over the phone. You would not read out a million pixel brightnesses. You would say "two eyes, a nose between them, a mouth below". Vision is **local** (a feature lives in a small patch), **compositional** (small parts build bigger parts) and **translation-tolerant** (the part means the same thing wherever it appears). A fully connected network (the multilayer network of Chapter 3) ignores all three. Flatten a modest 224×224 color image into 150,528 inputs, connect them to just 1000 hidden neurons, and that layer already holds ≈ 150 *million* weights. It also treats pixels (0,0) and (0,1) as unrelated, so a cat shifted three pixels right looks like a completely new input.

The **convolution** fixes both problems with one idea: instead of a giant weight per pixel, learn one tiny **kernel** (say 3×3) and slide it across the image, computing a weighted sum at each stop. Because the same nine weights are reused everywhere (**weight sharing**), the layer is tiny. And because the detector runs at every location, a feature found in one corner is found in all. The network is **translation-equivariant**. The kernel is a little template. A high response means "this patch looks like my template".



One step of a convolution. The 3×3 window (amber) is multiplied elementwise with the kernel (blue) and summed into a single output cell (teal). Sliding the window over all valid positions produces the whole output map.

Governing equations

For a single-channel input I and a k×k kernel K, the convolution (in practice the closely related *cross-correlation*) writes each output pixel as an inner product of the overlapping patch with the kernel:

$$z[i,j] = \Sigma_{m=0..k-1} \Sigma_{n=0..k-1} I[i+m, j+n] \cdot K[m,n] + b$$

Real images have C input channels (red, green, blue) and a layer learns F separate kernels, one per output **feature map**:

$$z[i,j,f] = \Sigma_{c=1..C} \Sigma_{m,n} I[i+m, j+n, c] \cdot K[m,n,c,f] + b_f$$

The output shrinks unless you pad. With input side n, kernel k, zero-**padding** p on each side and **stride** s (the step between windows), the output side is

$$o = \left\lfloor \frac{n - k + 2p}{s} \right\rfloor + 1$$

Picking p = (k−1)/2 with stride 1 gives o = n. This is a **"same"** convolution that preserves resolution, the usual default inside a block:

$$p = \frac{k-1}{2}, \quad s = 1 \;\Rightarrow\; o = n$$

Each map is passed through a nonlinearity, almost always ReLU, and the number of learned parameters in the layer is small and independent of image size:

$$a = \max(0, z) \qquad \text{params} = k \cdot k \cdot C \cdot F + F$$

Compare this with the dense layer that opened the chapter, whose weight count scales with the entire image. The ratio between the two is the whole argument for convolution:

$$\text{params}_{\text{dense}} = (H \cdot W \cdot C) \cdot M \;\gg\; \text{params}_{\text{conv}} = k \cdot k \cdot C \cdot F + F$$

I	input tensor of shape (height, width, C) — pixels or an earlier layer's maps
K	kernel weights of shape (k, k, C, F); the same K is reused at every (i,j) — weight sharing
k	kernel (receptive) side length, e.g. 3 — the width of the local patch each output sees
C, F	number of input channels and output feature maps (filters)
H, W, M	image height and width, and the unit count of a comparison dense layer
b_f	one scalar bias per output map f
p, s	padding (border of zeros) and stride (window step); p = (k−1)/2 keeps the size unchanged
z, a	pre-activation and post-ReLU activation at each spatial position of each map

How it works

A convolutional stack alternates three operations. **Convolution** detects patterns. **ReLU** discards negative (absent-pattern) responses. And **pooling** summarizes a neighborhood, most often by taking the maximum:

$$y_{\max}[i,j] = \max_{(m,n) \in W} a[i+m, j+n] \qquad y_{\text{avg}}[i,j] = \frac{1}{|W|} \sum_{(m,n) \in W} a[i+m, j+n]$$

Max-pooling over a 2×2 window with stride 2 halves each spatial dimension, cutting computation and giving a little **translation invariance**. If the strongest edge shifts by one pixel, the pooled value is unchanged. Average pooling instead takes the mean. This is smoother, but it

blurs sharp responses, so pooling always trades spatial precision for robustness. After several conv → ReLU → pool blocks the maps are small and deep. They are **flattened** into a vector and fed to an ordinary dense head (Chapter 3, multilayer networks) that reads out the class.

Two structural ideas explain why depth helps. First, the **receptive field** (the region of the original image that influences one deep unit) grows with every layer. Stacking two 3×3 convolutions lets a unit see a 5×5 patch, three sees 7×7, and pooling multiplies this reach. Second, features form a **hierarchy**. Zeiler & Fergus (2014) visualized how early filters learn oriented edges. The next layers combine edges into textures and corners, then object parts, then whole objects. This is exactly the compositional story vision demanded, and a concrete instance of the learned feature hierarchy of Chapter 5 (representations), here made visible by the filter-visualization tools of Chapter 28 (interpretability). A single 1×1 convolution mixes channels at one pixel without looking sideways. It is a cheap way to change depth.

The milestones trace this logic outward. The full history and its modern refinements are the subject of Chapter 14 (modern CNN architectures & computer vision). Here we note only the main thread. **LeNet-5** (LeCun, 1998) read handwritten digits with two conv layers. **AlexNet** (Krizhevsky, 2012) scaled the same recipe with ReLU, dropout and GPUs and roughly halved the ImageNet error rate. This was the moment deep learning became mainstream. **VGG** (2014) showed that deep stacks of plain 3×3 convolutions are enough. Then **ResNet** (He, 2015) let networks pass 100+ layers by adding **skip connections** that learn a residual on top of the identity,

$$y = F(x) + x$$

so gradients flow straight back through the "+ x" path and deep training stops vanishing (Chapter 10, initialization & the vanishing gradient). Together with batch normalization (Chapter 11), residual connections made networks of a hundred-plus layers routine to train. This is the story Chapter 14 tells in full. Modern nets also replace the flatten-and-dense head with **global average pooling**, reducing each final map to its mean. This gives far fewer parameters and less overfitting (a built-in regularizer; Chapter 12).

Cross-correlation, weight sharing and sparse connectivity

Two subtle points hide inside the simple word "convolution." First, the operation the equations above define (sliding the kernel without flipping it) is strictly **cross-correlation**. A *true* convolution reflects the kernel in both axes first. The two differ only by whether an index is added or subtracted:

$$(I * K)[i,j] = \Sigma_{m,n}\, I[i-m, j-n] \cdot K[m,n] \quad \text{(convolution)} \quad \text{vs.} \quad (I \star K)[i,j] = \Sigma_{m,n}\, I[i+m, j+n] \cdot K[m,n] \quad \text{(cross-correlation)}$$

Because the kernel is *learned*, the network can just as well learn the flipped weights. The two forms span the identical hypothesis space, and every deep-learning library computes the cheaper cross-correlation while calling it "convolution." The distinction is not pointless. The flip reappears in the backward pass, and it is why the input-gradient below is a genuine convolution.

Second, two independent savings make the layer small, and it helps to keep them separate. **Sparse connectivity:** a dense layer wires every output to every input, but a convolution wires each output only to the k×k pixels under the kernel. An output "sees" k^2 inputs, not the whole n^2 image. **Weight sharing (parameter tying):** the very same k^2 numbers are reused at every location instead of a fresh weight per position. Measure both on a single-channel n×n → n×n map: a dense layer needs $(n^2)\cdot(n^2) = n^4$ weights. For n = 32 that is $32^4 \approx$ **1.05 million**, while a 3×3 convolution needs just **9**. Sparse connectivity alone (a private 3×3 kernel per output) would still cost $9\cdot n^2$. Sharing reduces that to 9. Together they are the whole reason a CNN's parameter count is independent of image size, and fewer parameters is also why it generalizes from less data (Chapter 12, regularization & generalization).

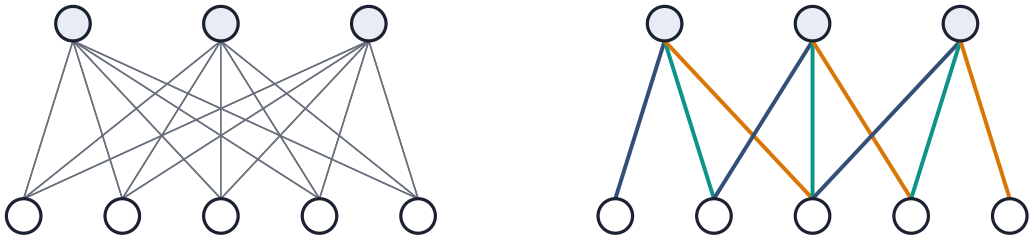
The reward for sharing is **translation equivariance**, which can be stated exactly. Let S_t shift an image by t pixels. Because one kernel runs at every position, convolving a shifted image simply shifts the feature map. The operations commute:

$$\text{conv}(S_t\, x) = S_t\, \text{conv}(x)$$

so a pattern found in one place is found identically everywhere (ignoring the boundary). Equivariance means "move the input, the response moves with it." It is *not* invariance, which means "move the input, the response is unchanged." Pooling gives a little of this invariance: a

2×2 max-pool returns the same value for any shift that keeps the winning activation inside the window, so the stack becomes **locally translation-invariant**. It is robust to a shift of a pixel or two, though never to large shifts, which is why training still relies on data augmentation (Chapter 12).

two economies that shrink a convolution layer



dense: 5×3 = 15 private weights

conv (k=3): 3 shared weights

Two savings, side by side. Dense wiring (left) gives each output its own weight to every input, fifteen here. A convolution (right) is sparse: each output reads only a local three-input window. It also shares weights, so the same three colors are reused at every position. This means it stores three numbers however wide the input grows.

Sizing a convolution: padding, stride, dilation and receptive field

Designing a CNN is largely keeping track of shapes, and one formula does most of it. Where does it come from? Count the valid places a window may start along one axis. After padding, the row is n + 2p wide. A window of width k fits only while its left edge a obeys a + k ≤ n + 2p, so the largest legal start is a = n + 2p − k. The starts step by the stride s:

1. Legal start positions along the axis: a ∈ {0, s, 2s, …} with a ≤ n + 2p − k.
2. The number of such multiples of s is ⌊(n + 2p − k)⁄s⌋ + 1. The "+ 1" counts the start a = 0.
3. Each start produces exactly one output pixel, so o = ⌊(n − k + 2p)⁄s⌋ + 1. The floor discards any final window that would extend past the padded edge.

Adding a **dilation** d, which spreads the kernel taps apart to widen coverage without adding weights, generalizes the output-size rule to

$$o = \left\lfloor \frac{n + 2p - d(k-1) - 1}{s} \right\rfloor + 1$$

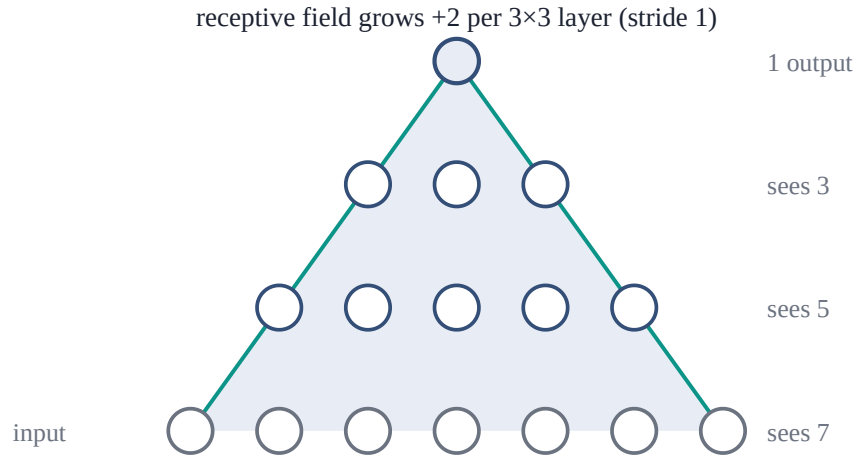
The term $d(k-1)+1$ is the kernel's **effective size**: a 3×3 kernel at dilation 2 covers a 5×5 span but still holds only 9 weights. Setting $d = 1$ recovers the earlier formula. A few cases worth memorizing, all from this one equation:

1. $n = 5, k = 3, p = 0, s = 1$: $o = \lfloor (5 - 2 - 1)/1 \rfloor + 1 = \mathbf{3}$. This is the "valid" shrink of the worked example.
2. $n = 7, k = 3, \mathbf{p = 1}, s = 1$: $o = \lfloor (7 + 2 - 2 - 1)/1 \rfloor + 1 = \mathbf{7}$. Padding one ring gives "same" size.
3. $n = 7, k = 3, p = 1, \mathbf{s = 2}$: $o = \lfloor 6/2 \rfloor + 1 = \mathbf{4}$. A stride-2 conv roughly halves the map, an alternative to pooling.
4. $n = 224, k = 7, p = 3, s = 2$ (the ResNet stem): $o = \lfloor (224 + 6 - 6 - 1)/2 \rfloor + 1 = \mathbf{112}$.

The **receptive field** follows its own recurrence. If layer ℓ has kernel k_ℓ and every earlier layer had stride s_i, then

$$r_\ell = r_{\ell-1} + (k_\ell - 1) \cdot \Pi_{i<\ell}\, s_i$$

With all strides 1 and 3×3 kernels this is simply $r = 2\ell + 1$. Two layers see 5 pixels, three see 7, and so on. This is why deep stacks of small 3×3 filters (the VGG idea) reach as far as one big filter but with fewer weights and more nonlinearity. Every stride-2 or pool multiplies the step size, so receptive field grows *geometrically* with depth once downsampling begins.



The receptive field as a cone. Reading top-down, one output unit depends on three units of the layer below, which depend on five, which depend on seven input pixels. So three stacked 3×3 convolutions give each output a 7-pixel view. Downsampling layers widen the cone faster.

Finally, the simple **1×1 convolution** is where parameter counting pays off. It looks sideways at nothing (k = 1) but mixes all C channels into F new ones, so it is a per-pixel linear layer costing only C·F + F weights. That is how a **bottleneck** block saves compute: instead of a 3×3 convolution on 256 channels (3·3·256·256 + 256 = **590,080** weights), a ResNet bottleneck first squeezes 256→64 with a 1×1 (16,448 weights), does the 3×3 on the cheap 64-channel tensor (36,928), then expands 64→256 with another 1×1 (16,640), for **70,016** weights total. That is the same receptive field for roughly 17× fewer parameters. Counting weights is not a minor detail. It is the main design pressure inside every modern vision backbone.

The backward pass is another convolution

Convolution is a *linear* operation, so backpropagation (Chapter 8) treats it as a shared-weight matrix multiply. And both gradients it must return turn out to be convolutions themselves. Write the layer as $z = I \star K$ and let $z = \partial L / \partial z$ be the gradient arriving from above. The **kernel gradient** is the input correlated with that upstream gradient, summed over every spatial position the kernel visited. A shared weight fans out to many outputs, so its gradients *add* (the fan-out rule of Chapter 8):

$$\partial L / \partial K[m,n] = \Sigma_{i,j}\, I[i+m, j+n] \cdot z[i,j]$$

The **input gradient** (needed to keep propagating downward) is a *full* convolution of the upstream gradient with the **flipped** kernel, the operation usually called a "transposed convolution":

$$\partial L / \partial I = z * \text{flip}(K)$$

Here the kernel flip we set aside in the forward pass comes back for real: cross-correlation forward forces a genuine convolution backward. The practical result is that everything a framework needs to train a conv layer is one more pass of the same primitive it already optimizes heavily. This is why convolutions are about as cheap to differentiate as to evaluate. That same transposed convolution, run in the forward direction, is how segmentation and generative networks *upsample* a small map back to full resolution (Chapter 14).

Convolution in code

The first equation of this chapter is a double loop. This cell implements a "valid" 2-D convolution from scratch and runs it on the exact image from the worked example below (a flat dark field with a bright band on the right) using the vertical Sobel kernel. The printed feature map should match the hand computation exactly:

Python · NumPy ▶ runnable in the app

```
import numpy as np
img = np.full((5, 5), 10.0)          # a flat dark image …
img[:, 3:] = 90.0                    # … with a vertical edge: left dark,
right bright
K = np.array([[1, 0, -1],
              [2, 0, -2],
              [1, 0, -1]], float)    # vertical Sobel kernel

def conv2d_valid(I, K):
    kh, kw = K.shape
    oh, ow = I.shape[0]-kh+1, I.shape[1]-kw+1
    out = np.zeros((oh, ow))
    for i in range(oh):
        for j in range(ow):
            out[i, j] = np.sum(I[i:i+kh, j:j+kw] * K)   # patch ⊙ kernel,
summed
    return out
```

```
fmap = conv2d_valid(img, K)
print("feature map (valid, 3x3):\n", fmap.astype(int))
print("matches hand calc:", np.allclose(fmap,
      [[0,-320,-320],[0,-320,-320],[0,-320,-320]]))
```

Zero in the flat region, a strong band where brightness jumps. One small template found the edge at every position at once. Pooling then shrinks a map while keeping the strongest responses. This cell downsamples a 4×4 map with 2×2 windows and stride 2, by both max and average:

Python · NumPy ▶ runnable in the app

```
import numpy as np
A = np.array([[1, 3, 2, 4],
              [5, 6, 1, 2],
              [7, 2, 9, 1],
              [3, 4, 2, 8]], float)      # a 4x4 feature map

def pool(A, k=2, s=2, mode="max"):
    oh = (A.shape[0]-k)//s + 1
    ow = (A.shape[1]-k)//s + 1
    out = np.zeros((oh, ow))
    for i in range(oh):
        for j in range(ow):
            win = A[i*s:i*s+k, j*s:j*s+k]
            out[i, j] = win.max() if mode == "max" else win.mean()
    return out

print("max-pool 2x2 /2:\n", pool(A).astype(int))          # -> [[6, 4], [7,
9]]
print("avg-pool 2x2 /2:\n", pool(A, mode="avg"))          # each 2x2 block's
mean
```

PyTorch supplies both as optimized layers, and a working image classifier is a short stack of them. Note how each block's output shape is pure arithmetic from the sizing formula. Two 2×2 pools give $28 \to 14 \to 7$:

PyTorch copy → Colab

```
import torch, torch.nn as nn

class SmallCNN(nn.Module):
    def __init__(self, n_classes=10):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, padding=1), nn.ReLU(),   # 28x28
-> 28x28
            nn.MaxPool2d(2),                                          #
-> 14x14
            nn.Conv2d(16, 32, kernel_size=3, padding=1), nn.ReLU(),  #
-> 14x14
            nn.MaxPool2d(2))                                         #
-> 7x7
        self.head = nn.Sequential(nn.Flatten(), nn.Linear(32*7*7,
n_classes))
    def forward(self, x):
        return self.head(self.features(x))

net = SmallCNN()
x = torch.randn(8, 1, 28, 28)          # a batch of 8 grayscale 28x28 images
print(net(x).shape)                     # torch.Size([8, 10]) — one score
per class
```

In this app's engine

Open the **Convolution lab**. You paint a small grayscale image, pick a kernel (edge, blur, sharpen, or your own numbers), and watch the window slide cell by cell while the running weighted sum lights up the output map. That is this chapter's first equation, animated. The Playground trains fully connected nets, so the lab is where you feel weight sharing directly: one 3×3 kernel, reused everywhere, does all the work.

Worked example — sliding a Sobel edge filter

Take a 5×5 image with a vertical edge: the left three columns are dark (intensity 10), the right two are bright (90). The vertical **Sobel** kernel is [[1,0,−1],[2,0,−2],[1,0,−1]]. With no padding and stride 1 the output side is o = (5 − 3 + 0)/1 + 1 = **3**, so we get a 3×3 map. Compute two cells.

1. Top-left cell — window over rows 0–2, columns 0–2, all pixels = 10 (a flat dark patch): z = 1·10 + 0·10 − 1·10 + 2·10 + 0·10 − 2·10 + 1·10 + 0·10 − 1·10 = (10−10) + (20−20) + (10−10) = **0**. No brightness change across the patch → no edge.
2. Its right neighbor — window over columns 1–3, so each row reads (10, 10, 90): z = 1·10 + 0·10 − 1·90 + 2·10 + 0·10 − 2·90 + 1·10 + 0·10 − 1·90 = (10−90) + (20−180) + (10−90) = −80 − 160 − 80 = **−320**. A large-magnitude response: the filter has found the vertical edge, and the sign says brightness rises from left to right.
3. Sliding on gives the full map [[0,−320,−320],[0,−320,−320],[0,−320,−320]]. Zero in the flat region, a strong band exactly where the edge is. One 3×3 template detected the edge everywhere at once.

The parameter story. A conv layer with 3×3 kernels, C = 3 input channels and F = 32 filters holds k·k·C·F + F = 3·3·3·32 + 32 = **896** weights. The dense layer we opened with held about 150,528,000. The convolution does the same job with roughly **168,000× fewer** parameters. That ratio is why images are where CNNs work best.

Worked example — a 3×3 input with a 2×2 kernel, by hand

Take the smallest convolution worth computing in full. The input I and a "diagonal-difference" kernel K are

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Valid, stride 1, no padding, so the output side is $o = \lfloor (3 - 2 + 0)/1 \rfloor + 1 = \mathbf{2}$: a 2×2 map. Each cell multiplies the overlapping 2×2 patch elementwise by K and sums. With this kernel that reduces to *top-left minus bottom-right* of the patch.

1. z[0,0] — patch [[1,2],[4,5]]: 1·1 + 2·0 + 4·0 + 5·(−1) = 1 − 5 = **−4**.
2. z[0,1] — patch [[2,3],[5,6]]: 2 − 6 = **−4**.
3. z[1,0] — patch [[4,5],[7,8]]: 4 − 8 = **−4**.
4. z[1,1] — patch [[5,6],[8,9]]: 5 − 9 = **−4**.

The output is the constant map [[−4, −4], [−4, −4]]. On this smooth ramp the diagonal slope is the same at every location, so a translation-equivariant detector reports one value everywhere. This is the chapter's central property, visible in four numbers. (The runnable exercise below reproduces exactly these numbers.)

Parameter count. Used as a real layer with C = 16 input channels and F = 32 output maps, this 2×2 kernel holds $k \cdot k \cdot C \cdot F + F = 2 \cdot 2 \cdot 16 \cdot 32 + 32 = 2048 + 32 = \mathbf{2{,}080}$ weights. It is the same 2,080 whether the input is 8×8 or 8000×8000, because sparse connectivity and weight sharing make the count independent of the image size.

Assumptions & limitations

Convolution assumes the useful statistics are *local* and *stationary*: the same feature is worth detecting anywhere. That is excellent for natural images and audio spectrograms but wrong where absolute position matters, or where inputs have no grid neighborhood (unordered tabular columns). Pooling gives only small, local translation tolerance. A CNN is not automatically invariant to large shifts, rotation or scale, so training still uses

data augmentation. And equivariance is not free understanding: a convolutional net can be fooled by textures and adversarial noise, because "responds strongly to this template" is not the same as "knows what an object is".

Why & when this matters

Use convolution whenever inputs sit on a grid with meaningful neighbors: images, video frames, audio spectrograms, even 1-D sensor traces. The benefit is parameter efficiency and built-in translation handling on modest data. When position is arbitrary or relationships are global rather than local (words in a sentence, nodes in a graph), the attention mechanism of Chapter 16 (attention & the transformer) is usually the better tool, though convolutions still power most vision front-ends.

In the real world

Convolutional networks are the main tool of applied computer vision. Medical-imaging systems built on CNN backbones read chest X-rays, CT scans and retinal photographs. An important 2016 study (Gulshan et al., *JAMA*) trained a CNN to detect diabetic retinopathy at specialist level. Self-driving perception stacks use convolutions to segment roads and detect pedestrians in every camera frame. Phones run compact CNNs (MobileNet) on-device for face unlock and photo enhancement. Even systems better known for other things rely on them: AlphaGo evaluated board positions with a CNN, and many audio and speech models convolve over spectrograms. When the input is a grid and the budget is real hardware, the convolution's parameter efficiency is exactly what makes deployment possible.

Exercises

1. An input feature map is 32×32. It passes through a 5×5 convolution with padding 2 and stride 1, then a 2×2 max-pool with stride 2. Give the spatial size after each step. MATH

2. How many weights (including biases) does a 5×5 convolution with 64 input channels and 128 output channels have? How many would a 1×1 convolution with the same channel counts have, and what is the ratio? MATH

3. Modify the convolution cell to detect a *horizontal* edge with the horizontal Sobel kernel, and predict the feature map before running. CODE

4. Explain how a 2×2 max-pool provides a small amount of translation invariance, and what it throws away in exchange. CONCEPT

5. Using zero-padding, turn the valid convolution into a "same" convolution and verify numerically that the output size equals the input size. CODE

6. Using only 3×3 convolutions with stride 1 (no pooling), how many layers are needed for a unit's receptive field to reach at least 15 pixels? MATH

7. A colleague proposes replacing every 3×3 convolution in a network with a 1×1 convolution "to save parameters". Why is this usually a bad idea? CONCEPT

8. A convolution applies 128 filters of size 3×3 over a 32×32 input with 64 channels, using padding 1 and stride 1. How many learnable weights does it have, and how many multiply–accumulate operations does one forward pass cost? MATH

9. Deep-learning libraries compute cross-correlation but call it "convolution," dropping the kernel flip of the textbook definition. Explain why this does not change what a CNN can learn, and name one place where the flip genuinely reappears. CONCEPT

10. Verify **translation equivariance** numerically (convolving a shifted image should equal shifting the convolution of the original) and confirm the 3×3-with-2×2-kernel worked example in the same cell. CODE

CHAPTER 14 · ARCHITECTURES

Modern CNN architectures & computer vision

> *Chapter 13 showed us a single filter sliding across an image. This chapter is about what happens when you stack thousands of them and let the pile grow a hundred layers deep. The history of computer vision since 1998 is really the history of that pile: how to make it deeper without breaking it, how to use its parameters wisely, and how to point it at problems harder than "name the one object in this photo". The turning point is a surprisingly small idea: a single "+ x" wired around every few layers. It let networks jump from tens of layers to hundreds almost overnight and still train. We will build that idea — the **residual connection** — starting from the gradient. We will count the parameters of a real ResNet block by hand. Then we will follow convolutions past classification into detection, segmentation, and the attention-based models now challenging them.*

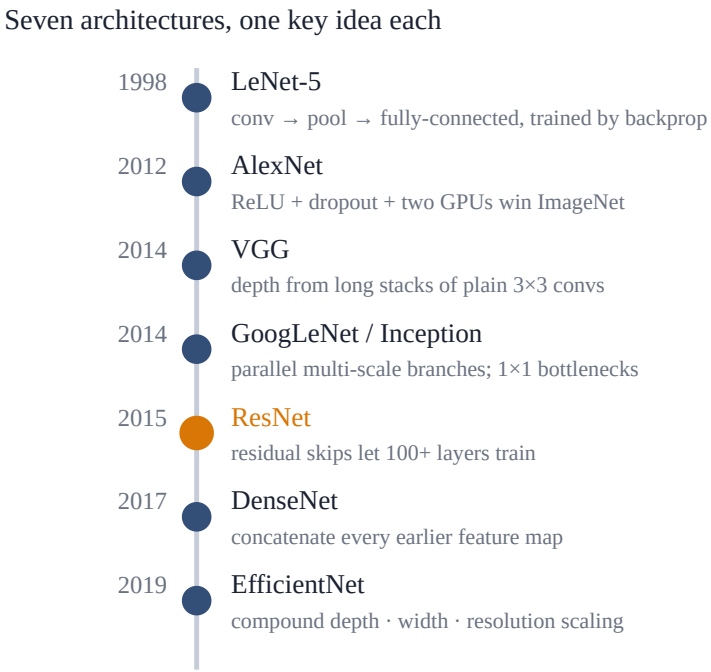
Fundamental concept: depth, and why more of it stopped helping

Chapter 13 gave two reasons depth should help vision. Features build on each other — edges into textures, textures into parts, parts into objects — so more layers give a taller hierarchy. Also, the receptive field grows with depth, so deep units finally see enough of the image to recognize a whole object. By 2014 the recipe seemed to be "keep stacking 3×3 convolutions": **VGG** reached nineteen layers and set records. The obvious next move was to keep going. It failed.

He and colleagues described the failure precisely and named it the **degradation problem**. Take a working 20-layer network, add thirty-six more plain layers, and both the *training* and test error get *worse* — and not by a little. This is not overfitting. Overfitting drives training error *down* while test error rises. Here the training error itself climbs, which can only mean the deeper network is *harder to optimize*. This is strange, because a deeper network contains the shallower one as a special case: if the extra thirty-six layers each computed the identity, the deep net

would exactly reproduce the shallow net's function. We can prove such a solution exists. But stochastic gradient descent, still struggling with the vanishing and exploding gradients of Chapter 10, simply could not find it. Asking a stack of nonlinear layers to learn "copy your input unchanged" turns out to be surprisingly hard.

The residual idea is a way of rethinking the problem that is so small it looks like a trick. Instead of asking a block of layers to compute some desired mapping H(x) directly, ask it to compute only the **residual** F(x) = H(x) − x — the *correction* to the input — and then add the input back at the end: output = F(x) + x. Now the awkward identity solution is easy. If the best thing a block can do is nothing, it just pushes its weights toward zero so that F(x) → 0, and the "+ x" carries the input through unchanged. It is like the difference between editing a good draft and rewriting it from a blank page. Given a working answer x, the block only has to learn how to improve it, and "leave it alone" is the easy default rather than a target it must work hard to hit. With this one change, He's team trained 152-layer networks that beat every shallower competitor and won the 2015 ImageNet challenge. As a demonstration, they also trained a 1001-layer network that still optimized.



The ImageNet-era family tree. Each row adds exactly one lasting idea to the one above it. ResNet (amber) is the turning point, where depth stopped being dangerous. Read the chapter as a walk down this column.

Seven architectures, seven ideas

Every architecture in the figure earns its place by adding one idea that outlived it. Walking down the column is the fastest way to see how the field arrived at the modern backbone.

LeNet-5 (LeCun et al., 1998) is the ancestor: two convolutional layers alternating with subsampling, then two fully-connected layers, trained end-to-end by backpropagation to read handwritten digits on checks. It set the template — convolve, downsample, repeat, then classify — and everything since has built on it. **AlexNet** (Krizhevsky, Sutskever & Hinton, 2012) is that template scaled up for real images, and it was the paper that started the deep-learning era. Its wins came from decisive engineering: the **ReLU** nonlinearity (non-saturating, so far faster to train than tanh), **dropout** in the dense head to fight overfitting, heavy data augmentation, and training split across two GPUs. Together these cut ImageNet top-5 error from 26% to 15% in one year — a jump big enough to end the debate over whether deep networks were practical. **VGG** (Simonyan & Zisserman, 2014) took one clear principle from that mix of tricks: use nothing but 3×3 convolutions and get power from *depth*. As Chapter 13 showed, two stacked 3×3 layers match a 5×5's receptive field with fewer weights and an extra nonlinearity. VGG used that trade-off to reach nineteen uniform layers, and its clean design made it the field's default feature extractor for years.

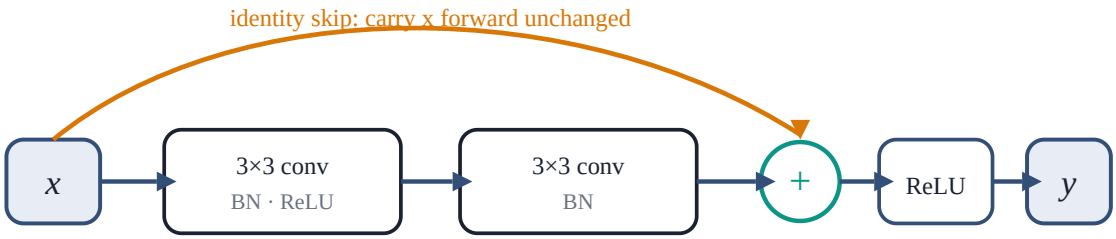
GoogLeNet / **Inception** (Szegedy et al., 2014) asked why choose one kernel size at all. Its **Inception module** runs 1×1, 3×3 and 5×5 convolutions and a pooling branch *in parallel* and concatenates them, letting the network pick the scale each location needs. To keep this affordable, it uses **1×1 convolutions** (Chapter 13) as bottlenecks that shrink the channel count before the costly branches. It was the first architecture to make 1×1 convolutions load-bearing. It also replaced the large dense head with **global average pooling**. Then **ResNet** (He et al., 2015), the main subject of the rest of this chapter, added the residual skip and broke the depth barrier. **DenseNet** (Huang et al., 2017) took skipping to its extreme: instead of *adding* the input to a block's output it *concatenates* the feature maps of *every* preceding layer, so layer ℓ sees layers $0 \dots \ell-1$ stacked along the channel axis. This gives maximal feature reuse and gradient flow, with layers kept thin (a small "growth rate") to hold the parameter count down. Finally **EfficientNet** (Tan & Le, 2019) asked how to spend extra compute: deeper, wider, or higher-resolution? Its answer, **compound scaling**, grows all three together in a fixed ratio (depth $\propto \alpha^{\varphi}$,

width $\propto \beta^{\varphi}$, resolution $\propto \gamma^{\varphi}$, with $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$ so FLOPs roughly double per step of φ), starting from a small baseline found by search. It gave the best accuracy-per-FLOP of its day. Six of these ideas — ReLU, 3×3-stack depth, multi-branch, 1×1 bottlenecks, residual skips, global average pooling — now sit together inside a single modern backbone.

Residual connections

Everything depends on one wiring diagram. A residual (or "skip") connection sends a block's input around its layers and adds it to their output. If the block's stacked layers compute a function F, the block as a whole computes

$$y = F(x) + x \quad \cdot \quad \partial y / \partial x = I + \partial F / \partial x$$



F(x): the residual branch (two conv layers)

The residual block. The amber skip copies the input x straight to the addition node, so the two convolutional layers only have to learn the correction F(x). Because the skip is a plain identity, the gradient reaching x is I plus the branch's gradient. This is a path that can never be reduced to zero.

$$y = F(x, \{W_i\}) + x$$

For the ResNet **basic block** used in ResNet-18 and ResNet-34, the branch F is two 3×3 convolutions, each followed by batch normalization, with a ReLU between them (σ denotes ReLU and ∗ convolution):

$$F(x) = W_2 * \sigma(W_1 * x)$$

Both convolutions use "same" padding and stride 1, so F(x) has exactly the shape of x and the sum F(x) + x is defined element-by-element. A final ReLU is applied *after* the addition. The whole point is in the gradient, so we differentiate the block with respect to its input:

$$\frac{\partial y}{\partial x} = I + \frac{\partial F}{\partial x}$$

x, y	the block's input and output feature tensors, of identical shape C×H×W
F(x)	the residual branch, here two 3×3 convolutions. It learns the *correction* to x, not the whole mapping
$\mathbf{W_1}, \mathbf{W_2}$	the branch's two convolution kernels (its learnable weights)
σ, ∗	the ReLU nonlinearity and the convolution operation. BN denotes batch normalization (folded in above)
I	the identity Jacobian from the skip, the term that guarantees an open gradient path
$\mathbf{x_\ell}, \mathscr{L}$	the input to the ℓ-th block in a stage, and the scalar training loss

Why residual networks train: the gradient highway

The identity term in $\partial y/\partial x = I + \partial F/\partial x$ is the whole explanation, and it is worth seeing what it does across many blocks. Suppose a stage stacks blocks in the clean additive form $x_{\ell+1} = x_\ell + F(x_\ell)$ (this is the "identity mappings" refinement of He et al., 2016, with no ReLU on the main path). Unrolling from block ℓ up to a deeper block L gives a striking telescoping sum:

1. One block: $x_{\ell+1} = x_\ell + F(x_\ell)$.
2. Two blocks: $x_{\ell+2} = x_{\ell+1} + F(x_{\ell+1}) = x_\ell + F(x_\ell) + F(x_{\ell+1})$.
3. In general $x_L = x_\ell + \Sigma_{i=\ell..L-1} F(x_i)$: the deep feature is the shallow feature *plus* a sum of corrections.
4. Differentiating the loss and using the chain rule, $\frac{\partial \mathscr{L}}{\partial x_\ell} = \frac{\partial \mathscr{L}}{\partial x_L} \cdot \left(I + \frac{\partial}{\partial x_\ell} \Sigma_{i=\ell..L-1} F(x_i) \right)$.

Read the last line against Chapter 10. In a *plain* deep network the gradient reaching an early layer is a *product* of many Jacobians. If each has typical singular value c, the product scales like c^L and either vanishes ($c < 1$) or explodes ($c > 1$) exponentially with depth. The residual "I" splits that product into a *sum*: the term $\partial \mathscr{L}/\partial x_L \cdot I$ passes the deep gradient back to x_ℓ **without shrinking it at all**, however small the branch derivatives get. The F terms can still add to the gradient, but nothing can reduce it to zero, because the identity contributes a clean 1, not a product of small numbers. This is the "gradient highway": a direct lane from the loss back to every layer. By construction, it removes the vanishing-gradient obstacle that capped plain nets at a couple of dozen layers. The runnable cell below makes this concrete: the c^L collapse versus the residual's survival.

Identity versus projection shortcuts. The plain "+ x" works only when x and F(x) have the same shape. But real backbones regularly *double* the channel count and *halve* the spatial size (a downsampling "stage transition"). At such a point x might be 64×56×56 while F(x) is 128×28×28, so you cannot add them. ResNet's fix is a **projection shortcut**: put a 1×1 convolution with the matching stride on the skip path to reshape x before adding, $y = F(x) + W_s * x$. The 1×1 kernel changes the channel count and its stride-2 setting halves the resolution, so the two tensors line up again. Projection shortcuts appear only at the handful of stage boundaries. Everywhere else the parameter-free identity skip is used, which is why the residual machinery costs almost nothing.

Spending parameters wisely: bottlenecks, 1×1 convolutions and pooling

Depth is only useful if you can afford it, and the arithmetic of Chapter 13 is what makes a hundred layers affordable. Two levers dominate. The first is the **1×1 convolution**, which mixes channels at a single pixel for C·F + F weights and no spatial extent. ResNet-50 and deeper use it to wrap every expensive 3×3 in a **bottleneck**: a 1×1 that squeezes 256 channels down to 64, the 3×3 acting on the cheap 64-channel tensor, then a 1×1 that expands back to 256. Chapter 13 counted the payoff: roughly 70,000 weights for the bottleneck against 590,000 for a plain 3×3 on 256 channels, the same receptive field for about 17× fewer parameters. Stacking fifty bottleneck blocks is therefore no more costly than a handful of plain ones.

The second lever is where you put the parameters at the *end*. LeNet and AlexNet flattened their final feature maps and fed an enormous fully-connected head. AlexNet's first dense layer alone held ~38 million weights, most of the network. **Global average pooling** (GAP), introduced by Inception and standard ever since, instead collapses each final feature map to its single spatial mean, turning a 512×7×7 tensor into a 512-vector with *zero* parameters, which a small linear classifier then reads. GAP greatly cuts the parameter count and removes the biggest source of overfitting. And because it averages over all spatial positions, it makes the classifier robust to where in the frame the object sits.

Separate two costs that beginners often confuse: **parameters** and **FLOPs**. A convolution's parameter count k·k·C·F is *independent of image size*, because the kernel is reused at every location (weight sharing, Chapter 13). Its compute instead scales with the output positions. A single 3×3 conv, 64→64 on a 56×56 map, does H·W·F·(k·k·C) = 56·56·64·(9·64) ≈ 1.16×10^{8} multiply-accumulates (116 million), while holding only 36,864 weights. So one basic block is ~37k weights but ~232 million MACs at that resolution. This is why early downsampling matters for speed, and why efficiency-focused designs (MobileNet, EfficientNet) use the **depthwise separable convolution**: a depthwise 3×3 (one filter per channel, k·k·C) then a pointwise 1×1 (C·F) replaces the full k·k·C·F, cutting parameters and FLOPs by about k^2 (roughly 9× for 3×3) at little accuracy cost.

Residual blocks in code

The residual block is short enough to write from scratch. This first cell implements a "same"-padded multi-channel convolution with plain loops (kept tiny so it runs instantly) and assembles y = conv2(ReLU(conv1(x))) + x on a 3-channel 6×6 input. It checks the two claims that make the block legal. First, F(x) has the same shape as x, so the addition is defined. Second, if the second convolution is zero-initialized (the real "BN γ = 0" trick), F(x) = 0 exactly and the block *starts life as the identity*.

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)
```

```python
def conv2d_same(x, W, b):
    # x:(C,H,Wd)  W:(F,C,k,k)  b:(F,)  ->  out:(F,H,Wd) with zero-pad 'same'
    C, H, Wd = x.shape
    F, _, k, _ = W.shape
    p = k // 2
    xp = np.pad(x, ((0, 0), (p, p), (p, p)))
    out = np.zeros((F, H, Wd))
    for f in range(F):
        for i in range(H):
            for j in range(Wd):
                out[f, i, j] = np.sum(xp[:, i:i+k, j:j+k] * W[f]) + b[f]
    return out

relu = lambda t: np.maximum(0.0, t)
C, H, Wd, k = 3, 6, 6, 3
x  = np.random.randn(C, H, Wd)
W1 = 0.10 * np.random.randn(C, C, k, k); b1 = np.zeros(C)
W2 = 0.10 * np.random.randn(C, C, k, k); b2 = np.zeros(C)

Fx = conv2d_same(relu(conv2d_same(x, W1, b1)), W2, b2)   # the residual branch F(x)
y  = Fx + x                                              # y = F(x) + x
print("x shape:", x.shape, " F(x) shape:", Fx.shape, " y shape:", y.shape)
print("shapes match, so x + F(x) is valid:", y.shape == x.shape)

# Zero the LAST conv (BN gamma=0 trick): F(x)=0 exactly -> block is the identity
y0 = conv2d_same(relu(conv2d_same(x, W1, b1)), np.zeros_like(W2), b2) + x
print("F=0 at init  ->  y == x exactly:", np.allclose(y0, x))   # expected: True
```

The second cell checks the receptive-field claim by experiment instead of trusting the formula. It drops a single impulse into a 1-D signal and convolves it repeatedly with a 3-tap kernel. After L layers, the number of non-zero entries is exactly how many inputs influence the center unit (the receptive field). It lands on 1 + 2L every time, confirming the r = 2ℓ + 1 rule from Chapter 13 by direct measurement rather than derivation.

Python · NumPy ▶ runnable in the app

```python
import numpy as np

def conv1d_same(x, w):                          # 'same' 1-D convolution
```

```
    k = len(w); p = k // 2
    xp = np.pad(x, (p, p))
    return np.array([xp[i:i+k] @ w for i in range(len(x))])

n = 41
for L in [1, 2, 3, 5, 7]:
    s = np.zeros(n); s[n // 2] = 1.0          # a single impulse at the
centre
    for _ in range(L):
        s = conv1d_same(s, np.ones(3))        # each 3-tap layer widens
support by 2
    measured = int(np.count_nonzero(s))       # inputs that influence the
centre unit
    print(f"{L} conv layers:  measured RF = {measured:2d}   formula 1+2L =
{1 + 2*L}")
```

Finally, the same block as a standard PyTorch module: the ResNet `BasicBlock` almost verbatim from `torchvision`. It uses the identity shortcut when shapes match and a 1×1 projection when a stage transition changes them. It zero-initializes the second batch-norm so the block begins as the identity, exactly as the first cell demonstrated:

PyTorch — copy → Colab

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class BasicBlock(nn.Module):
    """ResNet-18/34 residual block:  out = ReLU( branch(x) + shortcut(x)
)."""
    def __init__(self, in_ch, out_ch, stride=1):
        super().__init__()
        # First conv may downsample (stride) and change the channel count.
        self.conv1 = nn.Conv2d(in_ch, out_ch, 3, stride=stride, padding=1,
bias=False)
        self.bn1   = nn.BatchNorm2d(out_ch)
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3, stride=1, padding=1,
bias=False)
        self.bn2   = nn.BatchNorm2d(out_ch)

        # Identity skip when shapes agree; otherwise a 1x1 projection
shortcut.
```

```
        self.shortcut = nn.Sequential()
        if stride != 1 or in_ch != out_ch:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_ch, out_ch, 1, stride=stride, bias=False),
                nn.BatchNorm2d(out_ch))

        nn.init.zeros_(self.bn2.weight)          # F(x)=0 at init -> starts
as identity

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))    # 3x3 -> BN -> ReLU
        out = self.bn2(self.conv2(out))          # 3x3 -> BN
        out = out + self.shortcut(x)             # residual add (identity or
projection)
        return F.relu(out)                       # ReLU AFTER the addition

# block = BasicBlock(64, 64); x = torch.randn(8, 64, 56, 56)
# block(x).shape  ->  torch.Size([8, 64, 56, 56])   # shape preserved
```

Worked example — a ResNet basic block, shapes and parameters by hand

Take the first residual block of ResNet-18, operating on a 64-channel 56×56 feature map (the resolution after the 7×7 stride-2 stem and the stride-2 max-pool, as Chapter 13's stem example produced). Input shape is C×H×W = **64×56×56**. The branch is two 3×3 convolutions, both 64→64, both "same" (padding 1, stride 1).

1. First conv: input 64×56×56, kernel 3×3, padding 1, stride 1 → output side o = ⌊(56 + 2·1 − 3)/1⌋ + 1 = **56**, so 64×56×56. ReLU keeps the shape.
2. Second conv: same settings → again **64×56×56**. So F(x) is 64×56×56, identical to x, and the addition x + F(x) is valid with a plain identity skip (no projection needed).
3. Parameters, first conv: k·k·C·F = 3·3·64·64 = **36,864** weights (no bias term, because batch-norm supplies the offset).
4. Parameters, second conv: another 3·3·64·64 = **36,864**. Convolutions total 36,864 + 36,864 = **73,728**.

5. Batch-norm adds a scale γ and shift β per channel: 2·64 per BN, two BNs = **256**. Whole block ≈ 73,728 + 256 = **73,984** parameters, and the skip connection itself adds *zero*.
6. Identity at initialization: set the second BN's γ = 0. Then BN_2 outputs 0 for every input, so F(x) = 0 and y = 0 + x = x. The block computes the exact identity before a single gradient step. The deep network starts out behaving like a shallow one and only grows its corrections as training proceeds.

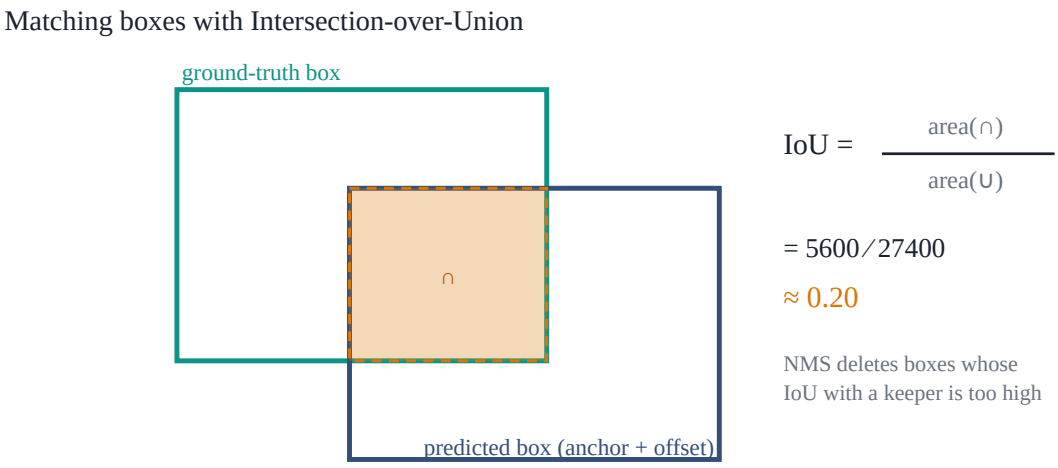
What the numbers mean: 74k parameters is remarkably cheap for two full layers. The whole of ResNet-18 is ~11 million, versus VGG-16's ~138 million, most of which was VGG's dense head. Yet those same two convolutions cost about 232 million multiply-accumulates at 56×56 (116 million each, from H·W·F·k·k·C). This is a reminder that parameters and compute are different budgets. The weights are few because they are shared across all 3,136 spatial positions, but the work is large because it is done at every one of them.

Beyond classification: detection and segmentation

Classification answers "what is in this image?" with a single label. Most real vision systems need more: *where* are the objects, and which *pixels* belong to each. Both tasks are built on a classification backbone (usually a ResNet pretrained on ImageNet) with new machinery added on top. This is why the residual block is the foundation for the whole field.

Object detection outputs a set of bounding boxes, each with a class label and confidence. The **R-CNN** family solved it in two stages, growing steadily faster. The original R-CNN (2014) ran the CNN once per region on ~2,000 classically-proposed regions (accurate but painfully slow). **Fast R-CNN** (2015) ran the backbone once over the whole image and pooled each region from the shared feature map. **Faster R-CNN** (2015) learned the proposals too, with a **Region Proposal Network** sharing the backbone. This gave an end-to-end, near-real-time detector still used as a baseline. Three ideas come up in all detectors. **Anchors** are a fixed grid of reference boxes at several scales and aspect ratios tiled over the feature map. Rather than regress

coordinates from nothing, the network predicts a small *offset* to the nearest anchor plus a class score. This makes detection a dense per-anchor classify-and-refine problem. **Intersection-over-Union** (IoU), overlap area over combined area, is the key measure in the pipeline. It decides which anchors match a ground-truth box in training and whether a prediction is correct at test time. And **Non-Maximum Suppression** (NMS) cleans up the overlapping boxes a dense detector produces: keep the highest-scoring box, delete every other whose IoU with it exceeds a threshold, then repeat. This way each object survives as one detection, not a dozen.

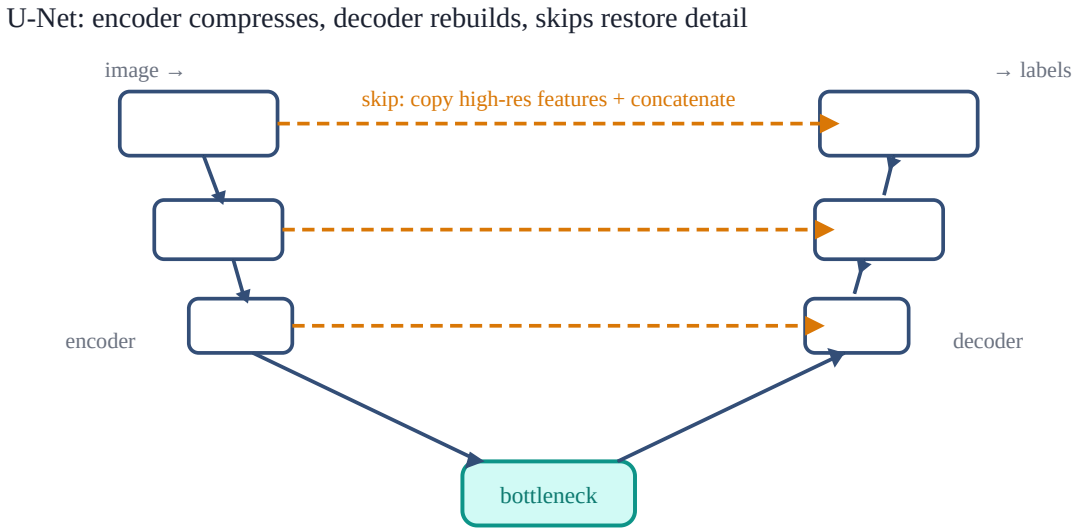


IoU measures box agreement as overlap ÷ union, here 5600/27400 ≈ 0.20. Training uses it to label anchors positive or negative. At test time, Non-Maximum Suppression uses it to collapse a cluster of overlapping predictions down to one box per object.

The two-stage detectors trade speed for accuracy. **One-stage** detectors — **YOLO** ("You Only Look Once", 2016) and **SSD** ("Single-Shot Detector", 2016) — drop the proposal step entirely and predict boxes and classes directly on a grid of anchors in a single forward pass. That makes them fast enough for real-time video, historically at a small cost in accuracy on tiny or crowded objects, a gap that later versions have largely closed. The split between two-stage and one-stage detectors, propose-then-refine versus predict-all-at-once, is still the main axis of the detection literature.

Semantic segmentation goes finer still: label *every pixel* with a class, a dense map rather than boxes. The difficulty is a tension the backbone creates for itself. To recognize *what* a region is, it must pool and downsample to a small, semantically rich representation. But that same downsampling destroys the precise information about *which* pixels are the object. The **encoder-decoder** resolves it. An encoder (an ordinary conv backbone) compresses the image to a coarse,

meaning-dense bottleneck. Then a **decoder** *upsamples* back to full resolution, using learnable **transposed convolutions** (which spread each input value through a kernel to grow the spatial size, the inverse geometry of a strided conv) or a bilinear upsample plus a normal convolution. The decoder rebuilds the "where" while keeping the encoder's "what".



The U-Net. The encoder (left) downsamples to a semantic bottleneck. The decoder (right) upsamples back to full resolution. The amber skip connections copy each encoder level's high-resolution features straight across to the matching decoder level, so fine boundaries lost to pooling are handed back exactly where they are needed.

The key refinement is **U-Net** (Ronneberger et al., 2015), named for its shape. Its innovation is a second family of skip connections, different from ResNet's, that copy each encoder level's high-resolution feature maps *across the U* and concatenate them into the matching decoder level. The bottleneck knows *what* the object is. The skipped features remember precisely *where* its edges were, and the decoder combines the two into crisp boundaries. Compare this with residual skips. A ResNet skip *adds* within a block at one resolution to keep gradients flowing. A U-Net skip *concatenates* across very different depths to restore detail: same instinct, different job. Built for biomedical images, where labels are scarce and boundaries must be exact, U-Net remains the default segmentation architecture. Later it also became the backbone of the denoiser inside diffusion image generators.

Transfer learning: the vision workhorse

Almost nobody trains a vision model from random weights. The main workflow is **transfer learning**, and it is why a small lab can build a strong classifier from a few hundred labeled photos: take a backbone pretrained on ImageNet's 1.2 million images and 1,000 classes and adapt it. The reason is Chapter 13's feature hierarchy. Early and middle layers learn generic primitives (edges, textures, color blobs, object parts) that are useful for essentially *any* natural-image task, whether the goal is spotting tumors, sorting products, or reading road signs. So only the final task-specific layers must change. In practice you either **freeze** the backbone and train just a new head (feature extraction, best when data is scarce) or **fine-tune** it, unfreezing some or all of it at a small learning rate so pretrained features are specialized rather than destroyed. This is why an ImageNet-pretrained ResNet is the starting point for the detectors and segmenters above. The topic is large enough (which layers to unfreeze, how to schedule learning rates, when pretraining hurts) to get its own treatment in Chapter 22. The main point: pretrain once on a giant generic dataset, then reuse it everywhere.

From convolutions to attention: Vision Transformers

For a decade "vision model" meant "convolutional network". That is no longer automatic. The **Vision Transformer** (ViT; Dosovitskiy et al., 2021) brings the Transformer architecture of Chapters 16–17 (built for sequences of words) straight into vision with a single change of view: treat an image as a short *sequence of patches*. Cut the image into a grid of fixed patches (16×16 pixels is typical), flatten each patch and project it linearly to a token embedding, add a positional embedding so the model knows where each patch came from, prepend a special classification token, and feed the whole sequence to a standard Transformer encoder. The stacked self-attention layers let every patch exchange information with every other patch from the very first layer, and the classification token's final embedding is read out for the label. There is not a single convolution in it.

The trade-off comes straight from **inductive bias**. A CNN has locality and translation-equivariance *wired in*. It assumes features are local and position-independent, a prior that is correct for images, so it learns efficiently from modest data. A ViT assumes almost nothing:

self-attention can connect any patch to any other, so it must *learn* that nearby patches usually matter most, and learning a prior from scratch takes enormous data. On ImageNet alone ViT falls behind comparable CNNs. Pretrained on a far larger dataset (the original used 300 million images), it matches or beats them and, crucially, keeps improving as data and compute grow, where CNNs plateau. That scaling is why attention-based and hybrid models (convolutional stems feeding Transformer blocks, or windowed designs like Swin that reintroduce locality) now dominate the largest vision systems. The guidance is simple. With limited data, compute, or latency, a convolutional backbone is still the efficient choice. At very large scale, attention's flexibility wins. The self-attention that makes ViT work is the subject of Chapters 16–17, and this patch-token bridge is where the vision and sequence halves of the book meet.

In this app's engine

The Convolution lab from Chapter 13 is the microscope for this chapter too. Stack a second and third 3×3 convolution and watch the highlighted receptive field widen by two cells per layer. This is the 1 + 2L rule the second runnable cell measured. Add the residual wiring in your mind: the output is that map *plus* the unchanged input. So at initialization, with branch weights near zero, output equals input, the identity the worked example proved by hand. Everything this chapter adds to Chapter 13 is a way of stacking, skipping, or re-scaling that one sliding-window operation.

Assumptions & limitations

Residual connections cure an *optimization* problem, not a modeling one. They let deep networks be trained, but they do not by themselves guarantee that more depth improves accuracy, and past a point extra blocks add cost for very little gain. The gradient-highway argument, in its clean form, assumes the skip is a true identity. The ReLU that ResNet places after the addition, and the projection shortcuts at stage transitions, both weaken that ideal slightly (which is exactly why the "identity mappings" follow-up moved activations off the main path). The broader architectures inherit every limitation of convolution from Chapter 13: locality and translation assumptions that fail when

absolute position matters, and no built-in robustness to large rotation, scale change, or adversarial perturbation. Detection and segmentation add their own weak points. Anchor-based detectors are sensitive to the chosen anchor scales and to the NMS threshold, and segmentation quality degrades on thin structures and object boundaries where a single downsampling step can erase the evidence. And transfer learning assumes the source and target domains share low-level statistics. ImageNet features transfer beautifully to everyday photographs but far less well to radar, medical volumes, or overhead satellite imagery, where pretraining can even hurt.

Why & when this matters

Whenever you build a vision system today, you are almost certainly assembling the pieces of this chapter: a residual backbone, pretrained on ImageNet, with a task-specific head (a linear classifier, a detection neck with anchors and NMS, or a U-Net decoder) fine-tuned on your data. Understanding the residual connection tells you why you *can* reach for a 50- or 101-layer backbone without it failing to train. Understanding parameters-versus-FLOPs tells you which model will actually fit your latency budget. Understanding the encoder-decoder tells you why your segmentation masks need skip connections to have sharp edges. These are the levers you adjust in practice, long before you ever design a new architecture.

In the real world

The architectures of this chapter run much of the visual world. Residual backbones read medical scans, grade manufacturing defects, and power the perception stacks of self-driving cars, where a Faster R-CNN or YOLO-style detector localizes pedestrians and vehicles in every camera frame and a segmentation head paints the drivable road. U-Net segments tumors and cell nuclei in hospitals and outlines fields, floods and deforestation

in satellite imagery. And, unexpectedly, it is the denoiser at the heart of Stable Diffusion and other image generators. Transfer learning from ImageNet is so routine that "load a pretrained ResNet" is the first line of countless production pipelines. And the ViT bridge is already load-bearing. The vision encoders inside multimodal systems that caption images or answer questions about them are, increasingly, patch-token Transformers: the same self-attention you will meet in Chapters 16–17, pointed at pixels.

Exercises

1. A ResNet basic block operates on a 128-channel 28×28 feature map, with two 3×3 "same" convolutions (both $128 \to 128$, no bias). Give the output shape and count the block's convolution parameters. MATH

2. At a stage transition a block must map a 64×56×56 input to a 128×28×28 output (first conv stride 2, $64 \to 128$). Explain why a plain identity skip is impossible here, and count the parameters of the 1×1 projection shortcut that fixes it. MATH

3. Verify numerically that a residual connection keeps gradients alive where a plain deep network kills them. Backpropagate a random gradient through 60 layers with small weights, once through the plain Jacobian W^T and once through the residual Jacobian $I + W^T$, and compare the surviving gradient norms. CODE

4. He et al. report that a 56-layer plain network has *higher training error* than a 20-layer one. Why does this rule out overfitting as the cause, and how does the residual form remove the difficulty? CONCEPT

5. Implement Intersection-over-Union for two boxes and a single pass of Non-Maximum Suppression, then run them on three boxes where two are almost identical. CODE

6. Using only 3×3 stride-1 convolutions (no downsampling), how many layers are needed for a unit's receptive field to exceed 32 pixels? Why do real backbones not build receptive field

this way? MATH

7. U-Net and ResNet both use "skip connections", yet they do different jobs. Compare the two: what is skipped, how the skipped tensor is combined, and what problem each solves. CONCEPT

8. A Vision Transformer with no convolutions can match a CNN on ImageNet, but only after pretraining on a far larger dataset. Explain the role of inductive bias, and say when you would still choose a CNN. CONCEPT

CHAPTER 15 · ARCHITECTURES

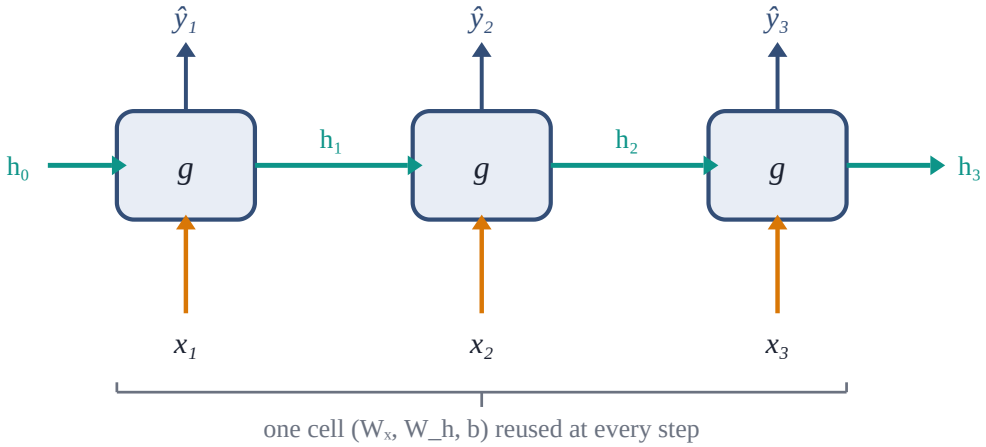
Recurrent networks & sequences

> *Language, sound and sensor streams arrive one piece at a time, and their meaning depends on order: "dog bites man" is not "man bites dog". A plain network needs a fixed-size input and treats every position independently, so it cannot handle a sentence of unknown length or remember what came before. The recurrent network solves this with a single idea taken from note-taking: keep a running summary, and update it as each new item arrives. This chapter builds that loop, shows why its memory both fades and explodes, and runs a tiny counting network by hand.*

Fundamental concept

Suppose you are adding a long column of numbers read aloud to you. Rather than wait for all of them, you hold one **running total** and update it with each new number. That total is a **state**: a compact memory that carries everything relevant from the past into the present. A recurrent neural network (RNN) is exactly this: a small network with a feedback loop that reads one input, blends it with its current state, and passes the new state forward.

Why not a multilayer network (chapter 3)? Because an MLP has a fixed number of input slots and a separate weight for each, so it cannot accept a variable-length sequence, and it has no built-in sense that position 5 follows position 4. The RNN reuses *one* cell (the same weights) at every time step. This is **weight sharing in time**, the mirror of the convolution's weight sharing in space (chapter 11): the same update rule applies at every moment in the sequence. Sharing keeps the parameter count fixed no matter how long the input runs.



The recurrent loop unrolled through time. Amber inputs x_t enter each step, the teal hidden state h_t carries memory forward, and blue outputs $\hat{y}_t$ can be read at each step or only the last. Every box is the same cell with the same weights.

Governing equations

At each time step t the cell mixes the current input with the previous state and squashes the result; an optional readout maps the state to an output:

$$h_t = g(x_t W_x + h_{t-1} W_h + b)$$

$$\hat{y}_t = \varphi(h_t W_y + b_y)$$

$\mathbf{x_t}$	input at step t — shape (1, d_{in}); one token, sample or frame
$\mathbf{h_t}$	hidden state (memory) — shape (1, d_h); the running summary of steps 1…t
$\mathbf{h_0}$	initial state, usually all zeros — the blank notebook before any input
$\mathbf{W_x}$, $\mathbf{W_h}$	input-to-hidden and hidden-to-hidden weight matrices — the *same* at every step
b	bias vector added at each step
g	hidden nonlinearity, classically tanh (Chapter 4, activation functions); φ, W_y are the optional output map
⊙	elementwise (Hadamard) product, used by the gates below

Reading the loop is easier once it is **unrolled**: copy the cell once per step and lay them left to right, as in the figure. That reveals a deep feedforward network whose depth equals the sequence length T and whose layers all share weights.

The same loop supports several **wirings**. They differ only in where the loss reads an output. A **many-to-one** network keeps just the final $\hat{y}_T$: a sentiment label for a whole review, a class for a whole gesture. A **many-to-many** network reads a $\hat{y}_t$ at every step: part-of-speech tags, or a label per audio frame. A **one-to-many** network takes a single input and unrolls a whole sequence out of it: a caption generated from an image. All three are the same recurrence with the loss attached at different steps, which is why one cell implementation serves them all.

How it works

Training uses **backpropagation through time** (BPTT): unroll, run the forward pass, then backpropagate the loss from step T back to step 1. This is exactly the backpropagation of Chapter 8 run over the unrolled graph. One complication comes from weight sharing: because the *same* W_x, W_h and b appear at every step, each one's gradient is the **sum** of the contributions gathered at all T steps. This is the fan-out rule of reverse-mode autodiff. And because the state feeds itself, the gradient of a late state with respect to an early one is a *product* of per-step Jacobians:

$$\partial h_T \,/\, \partial h_1 = \Pi_{t=2..T}\, \partial h_t / \partial h_{t-1} = \Pi_{t=2..T}\, \mathrm{diag}(g'_t)\, W_h^{\top}$$

Multiplying T similar matrices makes the norm behave roughly like a number raised to the power T:

$$\| \partial h_T / \partial h_1 \| \approx (\|W_h\| \cdot |g'|)^T$$

If that base is below 1 the gradient **vanishes**, so distant inputs are forgotten. If it is above 1 the gradient **explodes** into NaNs. More precisely, the growth rate is controlled by the largest singular value (spectral norm) of W_h: once it falls below $1/|g'|$ the product decays to zero, and once it climbs above that the product diverges. This is the same eigenvalue behavior as depth in Chapter 10 (initialization & the vanishing gradient), now stretched across time. This is why the orthogonal and identity-like initializations that stabilize a deep feedforward net also stabilize an RNN's recurrent matrix. The exploding case has a simple fix, **gradient clipping**: rescale any gradient whose norm exceeds a threshold θ,

$$\text{if } \|g\| > \theta : \quad g \leftarrow g \cdot \theta \,/\, \|g\|$$

Unrolling a long sequence is also costly in memory: a ten-thousand-token document would need every one of its ten thousand states cached before a single weight update. **Truncated BPTT** is the practical solution. Split the stream into chunks of a fixed length k (often 20–50 steps). Carry the hidden state *forward* across chunk boundaries so the model never loses its place, but backpropagate the gradient only k steps before *detaching* the state from the graph. Forward memory stays unbounded, but the backward graph and its cost are capped at k. The cost is a limited horizon: any dependency longer than k steps receives no gradient at all, so k directly trades compute and memory against how far back the network can learn to look.

Vanishing needs a fix in the architecture itself. The **LSTM** (Hochreiter & Schmidhuber, 1997) adds a protected **cell state** c that runs along the top like a conveyor belt, edited only by gentle multiplicative **gates**. Each gate is a sigmoid producing values in [0,1] that decide how much to let through:

$$f_t = \sigma(x_t W_{xf} + h_{t-1} W_{hf} + b_f)$$

$$i_t = \sigma(x_t W_{xi} + h_{t-1} W_{hi} + b_i) \quad o_t = \sigma(x_t W_{xo} + h_{t-1} W_{ho} + b_o)$$

$$\tilde{c}_t = \tanh(x_t W_{xc} + h_{t-1} W_{hc} + b_c)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \quad h_t = o_t \odot \tanh(c_t)$$

$\mathbf{f_t, i_t, o_t}$	forget, input and output gates — each a sigmoid in [0,1], one value per cell dimension
$\mathbf{\tilde{c}_t}$	candidate memory (a tanh in [−1,1]) — the new content proposed for writing this step
$\mathbf{c_t}$	cell state — the protected long-term memory carried along the additive highway
$\mathbf{W_{x\bullet}, W_{h\bullet}, b_\bullet}$	a separate input, recurrent and bias set for each of the four units — roughly 4× a vanilla cell's parameters

The **forget** gate f decides what old memory to keep, the **input** gate i how much new candidate c̃ to write, and the **output** gate o how much to expose as the hidden state. Because c is updated by addition rather than repeated matrix multiplication, gradients travel back along it through hundreds of steps without vanishing.

This protected additive path has a name from Hochreiter & Schmidhuber's 1997 paper: the **constant error carousel**. Along the cell state the local gradient is $\partial c_t / \partial c_{t-1} = f_t$, a single elementwise multiply by the forget gate, with no W_h factor and no g' factor anywhere in it. Whenever the learned forget gate sits near 1, the error signal circulates unchanged, as if riding a carousel, for hundreds of steps. When the gate closes toward 0, the model deliberately drops a memory it no longer needs. Because the gates are learned per dimension, one LSTM can hold some memories wide open while letting others decay, all in the same step. This is exactly how it bridges dependencies that a vanilla RNN cannot handle. The **GRU** (Cho et al., 2014) is a lighter two-gate variant that merges cell and hidden state:

$$z_t = \sigma(\, x_t W_{xz} + h_{t-1} W_{hz} + b_z \,) \quad r_t = \sigma(\, x_t W_{xr} + h_{t-1} W_{hr} + b_r \,)$$

$$\tilde{h}_t = \tanh(\, x_t W_{xh} + (r_t \odot h_{t-1}) W_{hh} + b_h \,)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

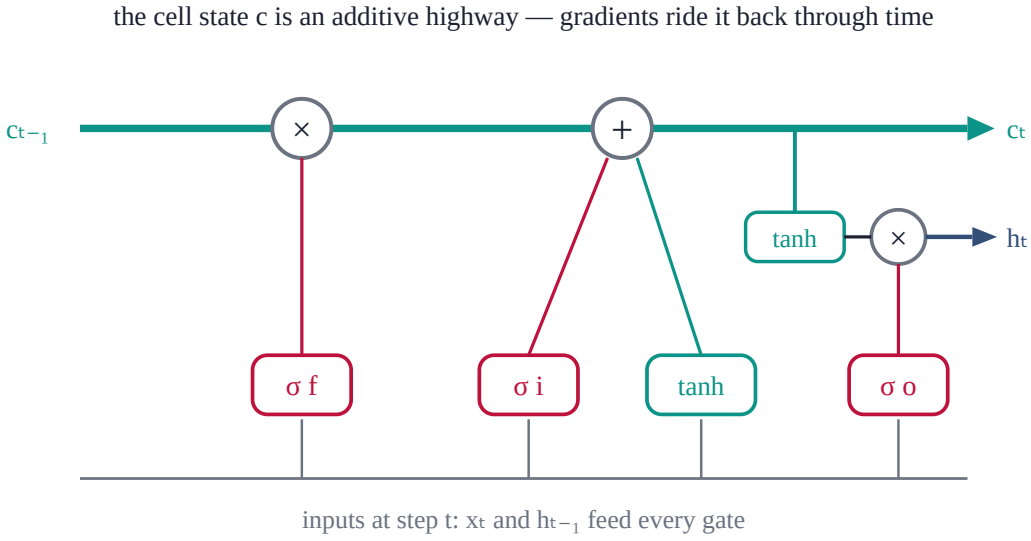
The **update** gate z does the work of the LSTM's forget and input gates together. The blend $(1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$ keeps the old state where z is near 0 and overwrites it where z is near 1. The **reset** gate r controls how much of the previous state is even allowed into the candidate $\tilde{h}$. With two gates rather than three and no separate cell state, a GRU carries roughly three-quarters of an LSTM's parameters and often matches its accuracy, which makes it a common default when the compute budget is limited.

Two more variations, one sentence each: a **bidirectional** RNN runs a second cell backward so each position sees future as well as past, and **stacking** RNN layers lets higher layers model slower, more abstract structure. Today, RNNs have largely been replaced by transformers (attention & the transformer, Chapter 16) and the large language models built on them (Chapter 17) for language, because recurrence forces strictly sequential computation. But they remain a strong choice for streaming, low-latency and small on-device models.

An LSTM step, by the numbers

The gate equations look complicated, but a single step is just a few scalars when the state is one-dimensional. Suppose the cell is carrying an old memory $c_{t-1} = 0.8$, and at this step the network produces forget gate $f = 0.9$, input gate $i = 0.1$, output gate $o = 0.5$ and candidate $\tilde{c} = 0.5$. Then

1. New memory: $c_t = f \cdot c_{t-1} + i \cdot \tilde{c} = 0.9 \cdot 0.8 + 0.1 \cdot 0.5 = 0.72 + 0.05 = \mathbf{0.77}$. The forget gate kept 90% of the old value and the input gate wrote in a little of the new candidate.
2. Exposed state: $h_t = o \cdot \tanh(c_t) = 0.5 \cdot \tanh(0.77) = 0.5 \cdot 0.6469 = \mathbf{0.3235}$. The output gate reveals only half of the (squashed) memory to the rest of the network.
3. **The gradient highway.** Differentiate the memory update along the cell path: $\partial c_t / \partial c_{t-1} = f = 0.9$, a plain multiply by the forget gate, with *no* W_h factor and *no* g' factor. Over 50 steps a vanilla RNN's memory-path gradient decays like $(\|W_h\| \cdot |g'|)^{50}$, but the LSTM's decays like f^{50}. If the network learns to set $f \approx 1$, then $1^{50} = 1$: the gradient is preserved perfectly. That is the whole trick. The additive belt turns a runaway product into a controllable one, and initializing the forget-gate bias positive (so f starts near 1) is a standard adjustment that makes long-range learning work from the first epoch.



The LSTM cell. The teal cell state c runs straight across the top, edited only by a forget multiply (×) and an input add (+). The purple sigmoid gates f, i, o and the teal tanh candidate, all driven by x_t and h_{t-1}, decide what is erased, written and read. The hidden state h_t (blue) is the gated, squashed memory.

Bidirectional, deep, and sequence-to-sequence networks

The single forward cell is a building block, and three standard ways of wiring several of them together cover most of what recurrent networks were built to do.

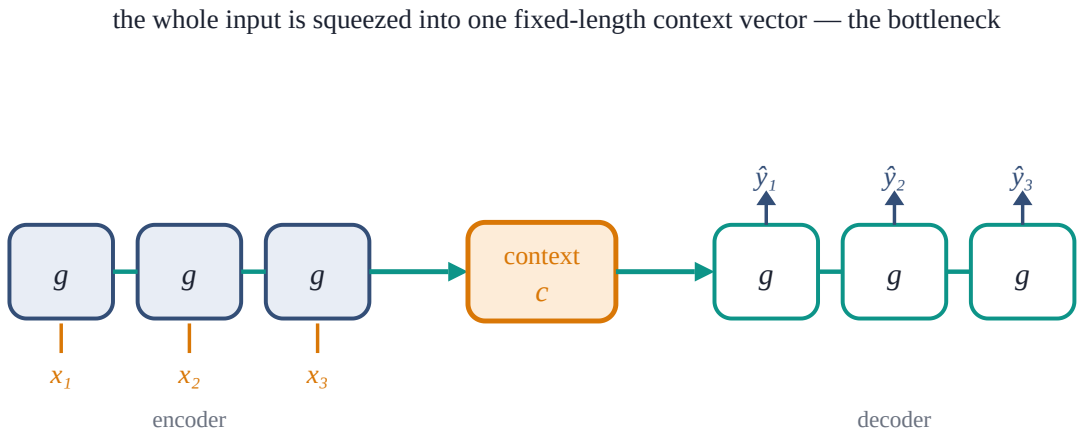
Bidirectional RNNs. When the whole sequence is available at once (a sentence to tag, an audio clip to transcribe offline), the future is as informative as the past. A bidirectional RNN runs one cell left-to-right and an independent second cell right-to-left, then concatenates their hidden states at each position, $h_t = [\, h_t^{\rightarrow} \,;\, h_t^{\leftarrow} \,]$. The representation at position t now summarizes the *entire* sequence rather than just its prefix, which sharply improves tagging and speech recognition. The cost is that nothing can be produced until the sequence ends, so a bidirectional RNN cannot run in a true streaming setting.

Deep (stacked) RNNs. Just as feedforward depth builds a feature hierarchy, stacking recurrent layers feeds the sequence of hidden states from one layer up as the input sequence of the next. Lower layers track fast, local structure (phonemes, characters) and higher layers track slower, more abstract structure (words, phrases). Two to four layers are typical. Residual connections between them (the fan-out rule of Chapter 8 once more) help the gradient survive the extra depth stacked on top of the extra length.

Sequence-to-sequence. Translation, summarization and speech-to-text map an input sequence to an output sequence of a *different* length, so reading one label per step is not enough. The **encoder–decoder** (Sutskever et al., 2014; Cho et al., 2014) uses two RNNs. An **encoder** consumes the whole input and compresses it into its final hidden state. This is a single fixed-length vector often called the **context** or **thought vector**. A **decoder** RNN is initialized from that vector and generates the output one token at a time, feeding each emitted token back in as the next step's input. Google's 2016 translation system was exactly this: a deep LSTM encoder feeding a deep LSTM decoder.

Forcing an entire paragraph through one fixed-length vector is, however, a real **bottleneck**. The longer the input, the more the single context vector must forget, and translation quality drops for long sentences (the figure). The cure, taken up next in Chapter 16 (attention & the transformer), is to let the decoder look back at *all* of the encoder's hidden states and learn, at each output step, which input positions to attend to. This is **attention**. Attention lifts the bottleneck so effectively

that a model built from attention alone, with no recurrence at all, became the transformer that now supports the large language models of Chapter 17. The RNN's fixed-vector bottleneck is the problem that much of the rest of the book is written to solve.



The sequence-to-sequence encoder–decoder. A blue encoder reads the input and packs it into one fixed-length context vector c (amber), the bottleneck, which initializes the teal decoder that produces the output token by token. Because c cannot grow with the input, long sequences overflow it. Attention (Chapter 16) instead lets the decoder read every encoder state, and dropping recurrence altogether yields the transformer.

The RNN in code

The counting network of the worked example is four lines of NumPy. This cell runs the same cell twice. First, with perfect memory (identity, $W_h = 1$), so the state literally counts the 1s. Second, with a contracting tanh cell ($W_h = 0.5$), so the memory fades. It then prints the hidden state after every step:

Python · NumPy ▶ runnable in the app

```
import numpy as np
def run_rnn(x, Wx=1.0, Wh=1.0, b=0.0, g=lambda z: z, h0=0.0):
    h, hs = h0, []
    for xt in x:
        h = g(Wx*xt + Wh*h + b)      # h_t = g(x_t Wx + h_{t-1} Wh + b)
        hs.append(h)
    return np.array(hs)

x = np.array([1., 0., 1., 1.])
```

```
print("perfect memory (Wh=1, identity):", run_rnn(x, Wh=1.0))
print("fading  memory (Wh=0.5, tanh)  :", np.round(run_rnn(x, Wh=0.5,
g=np.tanh), 4))
```

The perfect-memory run returns [1, 1, 2, 3], where the final 3 is the total number of 1s. The fading run stays bounded but forgets, exactly the trade-off the LSTM was built to escape. Next, see *why* BPTT is hard: the gradient across T steps is a product of per-step factors, so it behaves like a base raised to the power T. This cell prints that magnitude for three cases:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
# BPTT gradient magnitude across T steps ~ (||Wh|| * |g'|)^T
T, gp = 50, 0.9                          # gp = a representative tanh slope
|g'|
for Wh in [0.8, 1.0, 1.2]:
    base = Wh * gp
    print(f"||Wh||={Wh}  base={base:.2f}  (base)^{T} = {base**T:.2e}")
```

Base 0.72 collapses to 1e-7 (the signal from 50 steps ago is *gone*). Base 1.08 grows past 40 (an explosion that clipping must control). Only a base right at 1 is stable, a delicate balance no vanilla RNN can hold for long, which is the whole motivation for gated cells. In practice you never write BPTT by hand: PyTorch's `nn.RNN` and `nn.LSTM` unroll and backpropagate for you, and swapping one for the other is a single word:

PyTorch copy → Colab

```python
import torch, torch.nn as nn

rnn = nn.RNN(input_size=1, hidden_size=8, batch_first=True)   # a tanh RNN
cell
x = torch.randn(4, 6, 1)              # batch 4, sequence length 6, 1
feature/step
out, h_n = rnn(x)                     # out: every hidden state; h_n: the
final one
print(out.shape, h_n.shape)          # torch.Size([4, 6, 8])  torch.Size([1,
4, 8])

# swap in an LSTM to defeat vanishing gradients — same call, plus a cell
state:
lstm = nn.LSTM(input_size=1, hidden_size=8, batch_first=True)
```

```
out, (h_n, c_n) = lstm(x)
print(out.shape, c_n.shape)          # torch.Size([4, 6, 8])  torch.Size([1, 4, 8])
```

Finally, one runnable cell pushes real numbers through a single LSTM step in pure NumPy. It is the same scalar cell worked out by hand below. Each gate is one `sig` or `tanh`, and the memory update is the single additive line that defeats the vanishing gradient:

Python · NumPy ▶ runnable in the app

```
import numpy as np
sig = lambda z: 1/(1+np.exp(-z))
x, h, c = 1.0, 0.5, 0.8                       # x_t, h_{t-1}, c_{t-1}
f  = sig(x*1.0 + h*1.0 + 0.5)                 # forget gate   (Wx, Wh, b)
i  = sig(x*0.5 + h*0.5 - 1.0)                 # input  gate
o  = sig(x*1.0 + h*0.0 + 0.0)                 # output gate
ct = np.tanh(x*1.0 + h*1.0 + 0.0)             # candidate  c-tilde
c_new = f*c + i*ct                            # additive cell update:  f*c + i*c~
h_new = o*np.tanh(c_new)
print("f,i,o,cand =", np.round([f, i, o, ct], 4))          # [0.8808 0.4378 0.7311 0.9051]
print("c_t =", round(c_new, 4), " h_t =", round(h_new, 4))  # 1.1009  0.5855
```

In this app's engine

This app has no dedicated RNN tool (its Playground nets are feedforward), but you can see the RNN's main danger there directly: switch on **gradient clipping** in the Playground and watch a diverging run stabilize. That toggle is the exact fix described above, and the Math Log prints gradient norms so you can catch an explosion as it happens. To picture the recurrence, imagine taking one Playground network and feeding its own output back in as an extra input at the next step. Unroll that and you have built an RNN by hand.

Worked example — a one-neuron counting network

Use the smallest possible RNN: one input, one hidden unit, $W_x = 1$, $b = 0$, $h_0 = 0$. Feed the sequence $x = [1, 0, 1, 1]$.

1. **Perfect memory ($W_h = 1$, identity g).** Each step adds the input to the running state: $h_1 = 1\cdot1 + 1\cdot0 = 1$; $h_2 = 1\cdot0 + 1\cdot1 = 1$; $h_3 = 1\cdot1 + 1\cdot1 = 2$; $h_4 = 1\cdot1 + 1\cdot2 = 3$. The state is literally a **counter**: h = [1, 1, 2, 3], and the final 3 is the total number of 1s seen. Memory realized as a number.
2. **Fading memory ($W_h = 0.5$, g = tanh).** Now the previous state is halved before being added, then squashed. Compute each step to 4 decimals:
   $h_1 = \tanh(1\cdot1 + 0.5\cdot0) = \tanh(1.0000) =$ **0.7616**;
   $h_2 = \tanh(1\cdot0 + 0.5\cdot0.7616) = \tanh(0.3808) =$ **0.3634**;
   $h_3 = \tanh(1\cdot1 + 0.5\cdot0.3634) = \tanh(1.1817) =$ **0.8280**;
   $h_4 = \tanh(1\cdot1 + 0.5\cdot0.8280) = \tanh(1.4140) =$ **0.8883**.
3. **Read the numbers.** At step 2 the input is 0, so the only thing keeping h_2 above zero is the memory of step 1. But $W_h = 0.5$ has already halved it (0.7616 → 0.3808) before tanh. Each step the influence of an old input is multiplied by 0.5 again, so it decays geometrically: this is the vanishing memory of a contracting W_h. Part 1 remembers perfectly but has no bound and would explode on a long positive stream. Part 2 is stable but forgetful. Escaping that trade-off is exactly what the LSTM's additive cell state gives you.

Worked example — one LSTM cell step from raw weights

Take a one-dimensional LSTM so every quantity is a scalar. The step's inputs are $x_t = 1.0$, previous hidden $h_{t-1} = 0.5$, previous cell $c_{t-1} = 0.8$. The four small weight sets, written (W_x, W_h, b), are: forget (1.0, 1.0, 0.5), input (0.5, 0.5, −1.0), output (1.0, 0.0, 0.0) and candidate (1.0, 1.0, 0.0).

1. **Gate pre-activations,** each of the form $x_tW_x + h_{t-1}W_h + b$: forget $1\cdot1.0 + 0.5\cdot1.0 + 0.5 = 2.0$; input $1\cdot0.5 + 0.5\cdot0.5 - 1.0 = -0.25$; output $1\cdot1.0 + 0.5\cdot0 + 0 = 1.0$; candidate $1\cdot1.0 + 0.5\cdot1.0 + 0 = 1.5$.

2. **Gates** — σ on the three gates, tanh on the candidate: f = σ(2.0) = **0.8808**; i = σ(−0.25) = **0.4378**; o = σ(1.0) = **0.7311**; $\tilde{c}$ = tanh(1.5) = **0.9051**.
3. **New cell state:** $c_t = f \cdot c_{t-1} + i \cdot \tilde{c} = 0.8808 \cdot 0.8 + 0.4378 \cdot 0.9051 = 0.7046 + 0.3963 =$ **1.1009**. The forget gate (0.88) preserves most of the old 0.8, and the half-open input gate (0.44) writes in about 0.40 of the strongly positive candidate.
4. **New hidden state:** $h_t = o \cdot \tanh(c_t) = 0.7311 \cdot \tanh(1.1009) = 0.7311 \cdot 0.8008 =$ **0.5855**. The output gate exposes roughly 73% of the squashed memory to the rest of the network.

What the numbers mean: the cell state has climbed past 1 even though every gate output is bounded in [0,1]. The additive update lets memory *accumulate*, exactly what an integrator or a counter needs and exactly what a single saturating tanh state cannot do. The runnable LSTM-step cell above prints these same six numbers, and the gradient carried back along the cell this step is simply $\partial c_t / \partial c_{t-1} = f = 0.8808$, near 1, so the error passes almost untouched.

Assumptions & limitations

A vanilla RNN assumes recent context dominates. Its fixed-size state is an information bottleneck, so very long-range dependencies are lost even before the gradient problems begin. LSTMs and GRUs stretch the range but do not make it infinite. Importantly, recurrence is sequential by nature: step t cannot start until step t−1 finishes, so RNNs cannot use parallel hardware across the length of a sequence the way convolutions and transformers can. This is the main reason they lost their leading role in language. They also assume a single time direction unless made bidirectional, which prevents true streaming for the backward pass.

Why & when this matters

Recurrent models work best when inputs truly stream and you must respond before the sequence ends (live speech, keyboard prediction, control signals, sensor anomaly detection), and when the model must be small enough to run on a phone or microcontroller, where a compact GRU beats a large transformer. For offline tasks over long text, where you can see the whole sequence at once and want parallel training, attention (Chapter 16) is now the default. Understanding the RNN first is what makes the transformer's motivation obvious.

In the real world

Long before transformers, LSTMs already ran production language technology. Google Translate switched to a deep LSTM sequence-to-sequence model in 2016 (the GNMT system) and cut translation errors sharply. Apple, Amazon and Google used LSTMs for on-device speech recognition and wake-word detection, where the input really streams and latency matters. They remain the default in many **time-series** settings, such as forecasting energy demand or sensor telemetry, and flagging anomalies in an ECG or a jet engine's vibration trace (the setting of Chapter 33's time-series case study). They are also the default in reinforcement-learning agents that must remember earlier observations (DeepMind's Atari and StarCraft agents used LSTM memory). Whenever the sequence arrives one step at a time and the compute budget is limited, a gated RNN is still a top choice.

Exercises

1. Run the counting RNN by hand on x = [1, 1, 0, 1] with $W_x = 1$, $W_h = 1$, b = 0, identity g. Then give the first two states for the tanh cell with $W_h = 0.5$. MATH

2. Modify the code cell so the recurrence has a bias $b = -0.5$ and input weight $W_x = 2$, using the tanh cell, and print the states. CODE

3. A CNN and a transformer can process all positions of a sequence at once, but a vanilla RNN cannot. Why? CONCEPT

4. With $\|W_h\| = 1.2$ and $|g'| = 0.9$, estimate the gradient magnitude $\|\partial h_T / \partial h_1\|$ after $T = 25$ steps. What value of $\|W_h\| \cdot |g'|$ marks the boundary between vanishing and exploding? MATH

5. An LSTM cell has $c_{t-1} = 0.6$, forget gate $f = 0.8$, input gate $i = 0.3$, candidate $\tilde{c} = -0.5$, output gate $o = 0.6$. Compute c_t and h_t. MATH

6. Implement gradient clipping for a vector and show it rescales a large gradient to norm θ while leaving a small one untouched. CODE

7. You must build a wake-word detector ("Hey, phone") that runs on a battery-powered device and reacts the instant the phrase ends. Would you pick a gated RNN or a transformer, and why? CONCEPT

8. Compute one **GRU** step by hand for a scalar cell with $x_t = 1.0$, $h_{t-1} = 0.5$. The weight sets (W_x, W_h, b) are: update-gate $(0.5, 0.5, -0.5)$, reset-gate $(1.0, 1.0, 0.0)$ and candidate $(1.0, 1.0, 0.0)$, where the candidate uses $r_t \odot h_{t-1}$. Find z_t, r_t, $\tilde{h}_t$ and h_t, then check in code. CODE

9. You must train an RNN language model on documents thousands of tokens long, but a full unroll will not fit in memory. Describe **truncated BPTT** with a window of $k = 40$, and state precisely which dependencies it can and cannot learn. CONCEPT

10. In a plain sequence-to-sequence translator, the encoder compresses the whole source sentence into one fixed-length vector. Explain why translation quality gets worse as sentences grow longer, and how attention (Chapter 16) removes the problem. CONCEPT

CHAPTER 16 · ARCHITECTURES

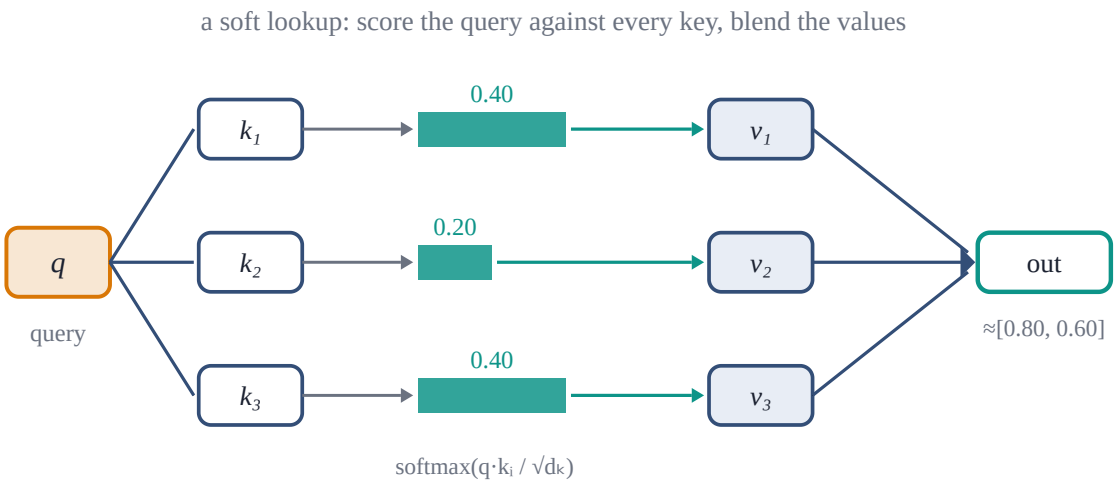
Attention & the transformer

> *A recurrent network reads a sentence word by word and compresses it into a single fixed vector. By the end, the opening words are only a faint echo. Attention removes that bottleneck. It lets every position look directly at every other and pull in exactly the information it needs, in one parallel step. This one mechanism powers every large language model. This chapter derives it from a lookup table, explains the mysterious √d divisor, and computes one attention head by hand.*

Fundamental concept

Think of a dictionary lookup. You hold a **query** — the word you want to define — and you scan a shelf of entries, each tagged with a **key** (the headword) and holding a **value** (the definition). A hard lookup returns the one value whose key matches. Attention is the *soft*, differentiable version. Instead of picking a single match, it scores the query against *every* key, turns those scores into weights that sum to one, and returns a blend of all the values, dominated by the best matches. Nothing is thrown away. Relevance just decides how much each item contributes.

This fixes the recurrent bottleneck (recurrent networks, Chapter 15) in two ways. First, distance disappears. Word 1 and word 50 are compared as directly as word 49 and word 50, so long-range links no longer fade through a chain of updates. Second, every comparison is independent, so the whole operation runs in parallel across the sequence rather than strictly left to right. This is the property that lets transformers train on very large text collections. When the queries, keys and values are all produced from the *same* sequence, the sentence attends to itself. This is **self-attention**, and it is how a model discovers that "it" refers to "the animal" three words earlier.



One attention step. The query is dotted with each key. The scores become softmax weights (teal bars). Each weight scales its value, and the weighted values sum into the output. Here the query matches k_1 and k_3 equally and k_2 least.

Governing equations

For query i, score it against every key j, scale it, normalize with softmax, then take the weighted sum of values:

$$s_{ij} = \frac{q_i \cdot k_j}{\sqrt{d_k}} \qquad \alpha_{ij} = \frac{e^{s_{ij}}}{\Sigma_l \, e^{s_{il}}} \qquad o_i = \Sigma_j \, \alpha_{ij} \, v_j$$

Stacking all queries as rows of Q (and keys, values likewise) gives the compact matrix form used everywhere:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q K^T}{\sqrt{d_k}} \right) V$$

In self-attention the three roles are three learned *views* of the same input sequence X:

$$Q = X W_Q \qquad K = X W_K \qquad V = X W_V$$

$\mathbf{q_i}, \mathbf{k_j}$	query and key vectors, each of dimension d_k; their dot product measures match
$\mathbf{v_j}$	value vector (dimension d_v) — the content returned when key j is selected
$\mathbf{s_{ij}}$	scaled compatibility score of query i with key j
$\boldsymbol{\alpha_{ij}}$	attention weight — how much of value j flows into output i; each row $\Sigma_j \alpha_{ij} = 1$
$\mathbf{d_k}$	key/query dimension; $\sqrt{d_k}$ rescales scores to unit variance (derived below)
$\mathbf{W_Q}, \mathbf{W_K}, \mathbf{W_V}$	learned projection matrices that turn each input token into its query, key and value

Queries, keys, and values

Everything begins with three linear projections of the input. Stack the sequence as a matrix X with one token per row. There are n tokens (the sequence length, written T when we think of it as time) and model dimension d, so X is n×d. Then multiply by three learned matrices to obtain three different *views* of every token:

$$Q = X\,W_Q \;\; (n\times d_k) \qquad K = X\,W_K \;\; (n\times d_k) \qquad V = X\,W_V \;\; (n\times d_v)$$

The query row q_i asks "what am I looking for?". The key row k_j announces "what do I offer?". The value row v_j carries "what I hand over if I am chosen." Because the three matrices are learned independently, one token can look for one thing, be found by another, and deliver a third. No fixed similarity measure could give this flexibility. The whole layer is then a single chain of matrix operations. Score every query against every key, $S = Q\,K^T$ (an n×n matrix whose entry S_{ij} is $q_i \cdot k_j$). Divide by $\sqrt{d_k}$. Apply softmax along each *row* so it becomes a probability distribution over keys. Then read out the blended values A V. That is the entire operation: two matrix multiplies with a softmax between them.

When Q, K and V are all projected from one sequence, the sequence attends to itself. This is **self-attention**, the case that lets a sentence discover its own internal structure. But the three inputs need not share a source. In **cross-attention** the queries come from one sequence and the keys and values from another. A translation decoder generating French forms its queries from the French produced so far, while its keys and values come from the encoded English, so each French position pulls in exactly the English words it needs. Now Q is $n\times d_k$ (target length n) while K and V are $m\times d_k$ and $m\times d_v$ (source length m), the score matrix S is the rectangular n×m,

and the output is $n \times d_v$. Self-attention is simply the special case m = n with a shared input. The same softmax-weighted blend therefore serves two jobs at once. It lets a sequence reason about itself, and it lets one sequence read another.

How it works

Why divide by $\sqrt{d_k}$? A score is a dot product of two d_k-dimensional vectors, $q \cdot k = \Sigma_m \, q_m k_m$. If the components are independent with mean 0 and variance 1, each product term has variance 1, and summing d_k of them gives

$$\mathrm{Var}(q \cdot k) = \Sigma_{m=1..d_k} \mathrm{Var}(q_m k_m) = d_k \;\Rightarrow\; \text{standard deviation} \approx \sqrt{d_k}$$

So with a typical $d_k = 64$ the raw scores swing by about ±8. Feed swings that large into softmax and it saturates. One weight goes to ≈1 and the rest to ≈0. The gradient through softmax nearly vanishes, so learning stalls. Dividing by $\sqrt{d_k}$ restores unit-variance scores and keeps the distribution soft and trainable. It is the same "keep the signal at unit scale" principle as careful initialization (Chapter 10).

One attention layer gives one kind of relevance. **Multi-head attention** runs h attentions in parallel on different learned projections, so one head can track grammar while another tracks reference, then concatenates and mixes them:

$$\mathrm{head}_r = \mathrm{Attention}(X\,W_Q^r, X\,W_K^r, X\,W_V^r) \quad \mathrm{MultiHead} = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h)\,W_O$$

Attention has no built-in sense of order. Permute the input tokens and the outputs just permute with them, because the mechanism only ever compares content. Order is added by giving each token a **positional encoding**. The classic choice is fixed sinusoids of many frequencies (learned position vectors work too):

$$PE(pos, 2i) = \sin(\, pos \,/\, 10000^{2i/d}\,) \quad PE(pos, 2i{+}1) = \cos(\, pos \,/\, 10000^{2i/d}\,)$$

A **transformer block** then wraps attention and a small per-token network in residual connections (Chapter 14) and layer normalization (Chapter 11):

$$z = \mathrm{LayerNorm}(\, x + \mathrm{MultiHead}(x)\,) \quad \mathrm{out} = \mathrm{LayerNorm}(\, z + \mathrm{FFN}(z)\,)$$

$$\mathrm{FFN}(z) = \max(0,\, z\,W_1 + b_1)\,W_2 + b_2$$

Stacking these blocks builds the whole model. An **encoder** lets every token see every other (bidirectional, as in BERT). A **decoder** for text generation adds a **causal mask** so a token can attend only to earlier positions. You set $s_{ij} = -\infty$ for $j > i$ before the softmax. This is how GPT predicts the next word without peeking ahead. Transformers won because their path length between any two positions is O(1), against an RNN's O(T), and because they parallelize over sequence length. The models in this app hold about a hundred parameters. GPT-scale language models (Chapter 17) multiply the very same equations by billions.

Multi-head attention

The head_r and MultiHead equations above hide a detail that matters. The h heads do not each work in the full d dimensions. Instead, the model *splits* its width, giving every head a slice of size $d_k = d_v = d / h$. Head r owns three projections W_Q^r, W_K^r, W_V^r, each of shape $d \times (d / h)$. It runs scaled dot-product attention inside that subspace and produces an $n \times (d / h)$ result. Concatenating the h results rebuilds an $n \times d$ matrix, and a final output projection W_O ($d \times d$) mixes it back into the token's representation. With $d = 512$ and $h = 8$ each head lives in $d_k = 64$ dimensions. This is exactly the case where the $\sqrt{d_k} = 8$ divisor really matters.

Why split one large attention into several small ones? Because a single softmax expresses only *one* pattern of relevance per query. It must fold every relationship a token cares about into one blurred distribution. Eight heads compute eight independent distributions at once, so one head can track subject–verb agreement while another resolves what a pronoun refers to and a third simply looks one token back. The concatenation and W_O then let the next layer combine these specialized signals. The cost is essentially unchanged, because h heads of width d / h do the same arithmetic as one head of width d. Yet the representational richness rises sharply. Probing trained models turns up heads that have specialized into recognizable syntactic and positional roles. Splitting the width is nearly free, and the diversity it gives is much of why attention works.

Multi-head attention. The input is projected into h independent subspaces of width d/h. Each head runs its own scaled dot-product attention (teal), the h results are concatenated back to width d, and W_O mixes them. Because the subspaces tile the full width, h heads cost about the same as one. But each can attend to a different kind of relationship.

The transformer block

Attention mixes information across positions, but on its own it does no per-token processing. A **transformer block** therefore pairs an attention sublayer with a small **position-wise feed-forward network** (FFN) and wraps each in the two devices that make deep stacks trainable. The first is a **residual connection** (Chapter 14) that adds a sublayer's input to its output. The second is **layer normalization** (Chapter 11) that rescales activations to a stable distribution. In the original 2017 arrangement the norm follows the addition (post-norm):

$$z = \mathrm{LayerNorm}(\, x + \mathrm{MultiHead}(x)\,) \qquad \mathrm{out} = \mathrm{LayerNorm}(\, z + \mathrm{FFN}(z)\,)$$

$$\mathrm{FFN}(z) = \mathrm{GELU}(\, z\,W_1 + b_1\,)\,W_2 + b_2 \quad (W_1 : d\times 4d,\ \ W_2 : 4d\times d)$$

The FFN is applied to *each position independently* with shared weights. It widens every token to an inner dimension of typically 4d, passes it through a pointwise nonlinearity, and projects back. The nonlinearity is ReLU in the original and the smoother **GELU** (Chapter 4) in most modern models. If attention is where tokens exchange information, the FFN is where each token does its private thinking. The block alternates mixing and computing all the way up the stack.

The residual highway lets gradients run straight from the loss to the earliest block, the same trick that tamed very deep networks (Chapter 10). Modern implementations move the norm to the *front* of each sublayer (pre-norm). This trains deep stacks more stably and is the form the language-model chapter adopts (Chapter 17).

Two block types build the two halves of the architecture. An **encoder** block uses unmasked self-attention, so every token sees every other in both directions. This is ideal for *understanding* a fixed input, as in BERT. A **decoder** block, used to *generate*, makes two changes. First, its self-attention is **causally masked** so a position may attend only to itself and earlier positions (set $S_{ij} = -\infty$ for $j > i$ before the softmax, making the attention matrix lower-triangular). Second, in the original encoder–decoder transformer for translation, it inserts a middle **cross-attention** sublayer whose queries come from the decoder and whose keys and values come from the encoder. Decoder-only models such as GPT keep the mask, drop the cross-attention, and stack the result into a language model (Chapter 17). The causal mask is the whole trick behind left-to-right generation. It lets the model be trained on every position at once while guaranteeing no position ever sees its own future.

Complexity, parallelism, and the fall of the RNN

Attention's power has a price, and you can see it in the matrix shapes. Forming the score matrix $S = Q\,K^{T}$ multiplies an $n \times d_k$ by a $d_k \times n$ matrix. That is n^2 dot products of length d_k, or $O(n^2 d_k)$ work. The softmax touches all n^2 entries, and the readout $A\,V$ multiplies $n \times n$ by $n \times d_v$ for another $O(n^2 d_v)$. Summed over the heads, with $d_k + d_v \approx d$, one self-attention layer costs

$$\text{time} \approx O(n^2 \cdot d) \quad \text{memory} \approx O(n^2)$$

plus $O(n \cdot d^2)$ for the four projections. The **quadratic** term in n is the one that hurts. Doubling the context quadruples both the compute and the memory for the $n \times n$ weight matrix. This single fact drives a whole research literature. There are sparse, low-rank, and linear-attention variants that trade exactness for a gentler curve. There are also IO-aware kernels such as FlashAttention that keep the $O(n^2)$ arithmetic but never store the full matrix. (At *generation* time the key–value cache avoids recomputing the past. That belongs to Chapter 17.)

Why pay this quadratic cost at all, when a recurrent network (Chapter 15) processes a sequence in linear $O(n \cdot d^2)$ time? Two reasons decided the field. First, **path length**. In an RNN, information from token 1 must survive n sequential update steps to reach token n, and the gradient along that path fades or explodes with distance. In attention any two positions are compared *directly*, a path length of O(1), so a long-range dependency is as easy to learn as a short one. Second, **parallelism**. An RNN's step t cannot begin until step t−1 finishes, a chain of n sequential operations. Attention instead computes all n^2 comparisons at once, a sequential depth of O(1). This is exactly what saturates a GPU and lets a transformer train on trillions of tokens. The transformer spends more arithmetic ($n^2 d$ against nd^2) and more memory to gain short gradient paths and near-total parallelism. On modern hardware that trade has been decisive.

Positional encoding, by the numbers

Because attention only ever compares content, it is **permutation-equivariant**. Shuffle the input tokens and the outputs shuffle with them unchanged. On its own the mechanism cannot tell “dog bites man” from “man bites dog”. Order must be supplied from outside. The classic method adds to each token a fixed vector of sines and cosines whose frequencies span many scales. This gives every position a unique, smoothly varying fingerprint. Take a tiny model dimension d = 4, whose two frequency denominators $10000^{2i/d}$ are {1, 100}. The first four positions then encode as

$$\mathrm{PE}_0 = [0, 1, 0, 1], \quad \mathrm{PE}_1 = [0.841, 0.540, 0.010, 1.000]$$

$$\mathrm{PE}_2 = [0.909, -0.416, 0.020, 1.000], \quad \mathrm{PE}_3 = [0.141, -0.990, 0.030, 1.000]$$

The high-frequency pair (denominator 1) turns quickly from one position to the next, resolving fine “exactly where” detail. The low-frequency pair (denominator 100) barely moves, giving a coarse “roughly where” signal. The two together work like the minute and hour hands of a clock. A useful side effect is that the dot product between two encodings falls off smoothly with their separation. $\mathrm{PE}_0 \cdot \mathrm{PE}_1 = 1.540$ against $\mathrm{PE}_0 \cdot \mathrm{PE}_3 = 0.010$, so relative distance is readable by the very dot products attention already performs. The cell below reproduces these numbers exactly:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.set_printoptions(precision=3, suppress=True)

d = 4                               # model dimension (even)
positions = np.arange(4)            # tokens at positions 0..3
i = np.arange(d // 2)               # 0, 1
denom = 10000.0 ** (2 * i / d)      # one frequency per dimension pair

PE = np.zeros((len(positions), d))
for pos in positions:
    PE[pos, 0::2] = np.sin(pos / denom)   # even dims: sine
    PE[pos, 1::2] = np.cos(pos / denom)   # odd  dims: cosine

print("positional encodings (rows = positions 0..3):")
print(PE)
print("PE[0] . PE[1] =", round(float(PE[0] @ PE[1]), 4))
print("PE[0] . PE[3] =", round(float(PE[0] @ PE[3]), 4))
```

Attention in code

The worked example above is a dozen lines of NumPy. This cell runs scaled dot-product attention on the same one-query, three-key head. Press **Run** and it prints exactly the weights and output we found by hand:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.set_printoptions(precision=4, suppress=True)

def softmax(v):
    v = v - v.max()
    e = np.exp(v)
    return e / e.sum()

# the worked example, in code: one query, three keys, three values
q = np.array([1.0, 0.0])
K = np.array([[1.0, 0.0],
              [0.0, 1.0],
              [1.0, 1.0]])
V = np.array([[1.0, 0.0],
```

```
              [0.0, 1.0],
              [1.0, 1.0]])

dk = q.shape[0]                    # key/query dimension = 2
scores = (K @ q) / np.sqrt(dk)     # scaled dot products
alpha  = softmax(scores)           # attention weights, sum to 1
out    = alpha @ V                 # weighted blend of the values

print("scaled scores:", scores)
print("weights      :", alpha, " sum =", alpha.sum())
print("output       :", out)
```

To generate text, a decoder must never let a position see words that come later. A **causal mask** enforces this. Before the softmax, every score s_{ij} with $j > i$ is set to $-\infty$ (a large negative number in practice), so its weight collapses to zero and each token blends only itself and its past. The resulting attention matrix is lower-triangular, its rows still summing to one:

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.set_printoptions(precision=3, suppress=True)
rng = np.random.RandomState(0)

T, dk = 4, 8                       # 4 tokens, dimension 8
Q = rng.randn(T, dk)
K = rng.randn(T, dk)
V = rng.randn(T, dk)

scores = (Q @ K.T) / np.sqrt(dk)   # (T, T) all pairwise scores
mask = np.triu(np.ones((T, T)), k=1)   # 1s strictly above diagonal = future
scores = np.where(mask == 1, -1e9, scores)   # forbid looking ahead

scores = scores - scores.max(axis=1, keepdims=True)
w = np.exp(scores)
w = w / w.sum(axis=1, keepdims=True)   # row-wise softmax

print("causal attention weights (lower-triangular):")
print(w)
print("row sums:", w.sum(axis=1))
```

The first token attends only to itself (weight 1.0), the second spreads over positions 1–2, and so on down the triangle. This is exactly the information flow that lets a GPT-style model predict the next word without peeking ahead. Written for the modern stack, the same head is a three-line function, while production code uses the fused, batched, multi-head `nn.MultiheadAttention`:

PyTorch copy → Colab

```
import torch, torch.nn as nn
import torch.nn.functional as F

def attention(Q, K, V, mask=None):        # scaled dot-product attention
    d = Q.size(-1)
    scores = Q @ K.transpose(-2, -1) / d ** 0.5
    if mask is not None:
        scores = scores.masked_fill(mask, float('-inf'))
    w = F.softmax(scores, dim=-1)
    return w @ V, w

Q = torch.tensor([[1., 0.]])
K = torch.tensor([[1., 0.], [0., 1.], [1., 1.]])
out, w = attention(Q, K, K)               # here values = keys
print(w)      # tensor([[0.4011, 0.1978, 0.4011]])
print(out)    # tensor([[0.8022, 0.5989]])

# the batched, multi-head path used in real transformers:
mha = nn.MultiheadAttention(embed_dim=64, num_heads=8, batch_first=True)
x = torch.randn(2, 10, 64)                # (batch, sequence, features)
y, attn = mha(x, x, x)                    # self-attention: Q = K = V = x
print(y.shape)                            # torch.Size([2, 10, 64])
```

In this app's engine

This app has no attention lab, but its final ingredient is one you can already watch. The **softmax** (Chapter 4) that turns attention scores into weights is exactly the softmax behind the Playground's softmax output option, which turns output logits into class probabilities on multi-

class datasets (three Gaussians, checkerboard). Open the Math Log on such a run and you are reading the same normalization, $e^{S_i}/\Sigma e$, that sits at the heart of every attention head. Here it just weights values instead of classes.

Worked example — one attention head by hand

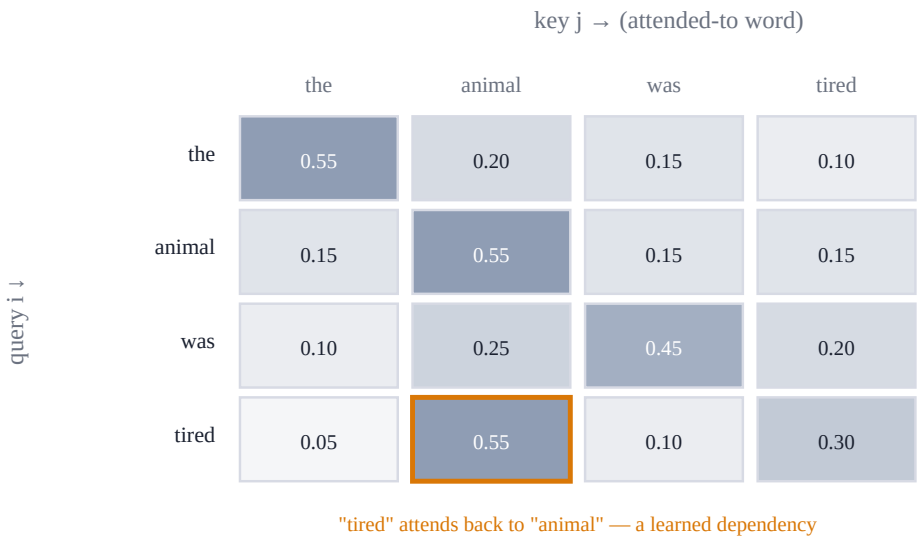
Let $d_k = 2$. Query $q = [1, 0]$. Keys $k_1 = [1,0]$, $k_2 = [0,1]$, $k_3 = [1,1]$. Values $v_1 = [1,0]$, $v_2 = [0,1]$, $v_3 = [1,1]$.

1. **Scores** (dot products): $q \cdot k_1 = 1 \cdot 1 + 0 \cdot 0 = 1$; $q \cdot k_2 = 1 \cdot 0 + 0 \cdot 1 = 0$; $q \cdot k_3 = 1 \cdot 1 + 0 \cdot 1 = 1$. So raw scores = [1, 0, 1].
2. **Scale** by $\sqrt{d_k} = \sqrt{2} = 1.4142$: [1, 0, 1] / 1.4142 = [0.7071, 0, 0.7071].
3. **Softmax**: exponentiate — $e^{0.7071} = 2.0281$, $e^{0} = 1.0000$, $e^{0.7071} = 2.0281$; sum = 5.0562. Weights α = [2.0281/5.0562, 1.0000/5.0562, 2.0281/5.0562] = [0.4011, 0.1978, 0.4011]. Check: 0.4011 + 0.1978 + 0.4011 = 1.0000. ✓
4. **Weighted sum of values**: $o = 0.4011 \cdot [1,0] + 0.1978 \cdot [0,1] + 0.4011 \cdot [1,1]$ = [0.4011 + 0 + 0.4011, 0 + 0.1978 + 0.4011] = **[0.8022, 0.5989]**.
5. **Read it.** The query aligned equally with k_1 and k_3 (both scored 1) and least with k_2, so the output leans on v_1 and v_3. It is a soft blend, not a hard pick. That softness is what makes attention differentiable and trainable. With $d_k = 2$ the $\sqrt{2}$ scaling barely matters, but at $d_k = 64$ the unscaled scores would be roughly eight times larger and the softmax would collapse onto a single value. This is exactly the saturation the divisor prevents.

Self-attention over a whole sequence

The single-query walkthrough above is one row of a larger object. Run self-attention on all n tokens at once and the weights form an n×n **attention matrix** A. Row i is the probability distribution token i places over every token, itself included, and each row sums to one. Reading

a trained model's A is the closest thing attention offers to an explanation. It shows, for every word, which other words it drew on. The diagram below sketches the kind of pattern that emerges, with the adjective “tired” focusing its attention on the noun “animal” it modifies.



A self-attention matrix. Each row is a query word's probability distribution over the key words (rows sum to 1). Darker cells carry more weight. The amber-outlined cell shows the adjective “tired” attending to the noun “animal”. This is the sort of dependency self-attention discovers on its own. These weights are illustrative. The worked example below carries an exact 3×3 case to real numbers.

The worked example below carries a complete three-token attention matrix to numbers, so every entry of A and every output row is one you can check.

Worked example — self-attention over a 3-token sequence

Take three tokens whose (already projected) rows are $x_1 = [1, 0]$, $x_2 = [0, 1]$, $x_3 = [1, 1]$, with $d_k = 2$ and, for clarity, identity projections so $Q = K = V = X$ (a real layer learns W_Q, W_K, W_V instead). We build the full 3×3 attention matrix A and the three output rows.

1. **Scores** $S = Q K^T$, entry $S_{ij} = x_i \cdot x_j$. Row 1: [1, 0, 1]; row 2: [0, 1, 1]; row 3: [1, 1, 2]. The diagonal entry is largest for token 3 because its vector is the longest.
2. **Scale** by $\sqrt{d_k} = \sqrt{2} = 1.4142$. Row 1: [0.7071, 0, 0.7071]; row 2: [0, 0.7071, 0.7071]; row 3: [0.7071, 0.7071, 1.4142].

3. **Softmax each row.** With $e^{0.7071} = 2.0281$, $e^{0} = 1$, $e^{1.4142} = 4.1133$: row 1 sums to 5.0562 → [0.4011, 0.1978, 0.4011]; row 2 → [0.1978, 0.4011, 0.4011]; row 3 sums to 8.1695 → [0.2483, 0.2483, 0.5035]. Every row sums to 1. ✓
4. **Blend the values** O = A V (here V = X). Row 1: 0.4011·[1,0] + 0.1978·[0,1] + 0.4011·[1,1] = **[0.8022, 0.5989]**. Row 2, by symmetry: **[0.5989, 0.8022]**. Row 3: 0.2483·[1,0] + 0.2483·[0,1] + 0.5035·[1,1] = **[0.7517, 0.7517]**.

What the numbers mean: token 1's output reuses the very computation of the one-head example. Self-attention just applies it to every token in parallel. Token 3, whose vector points between the other two, spreads its attention most evenly but still leans hardest on itself (weight 0.5035) and lands at the symmetric blend [0.7517, 0.7517]. The whole layer turned three input vectors into three context-aware outputs with one matrix multiply, a row-wise softmax, and a second matrix multiply. That is the entire forward pass of attention, at a scale you can hold in your head. Exercise 10 reproduces every number in five lines of NumPy.

Assumptions & limitations

Full self-attention compares every position with every other, so its compute and memory grow as $O(T^2)$ in sequence length T. This is why long documents strain transformers and why a whole industry of sparse and linear approximations exists. Attention has no built-in bias toward nearby words or toward order. This is a strength for long-range reasoning, but it means position must be supplied and locality must be learned from data. It is also data-hungry. With the inductive priors of convolution or recurrence removed, transformers need large datasets (or heavy augmentation) to reach their advantage. And a high attention weight signals correlation of representations, not human-style understanding.

Why & when this matters

Attention is the default when relationships are global and order-sensitive but not strictly local. This covers language, code, protein sequences, and increasingly vision, where an image becomes a grid of patch tokens (the Vision Transformer, Chapter 14), multimodal models, and even graphs, where a node attends over its neighbors (graph attention networks, Chapter 18). Its parallelism and O(1) path length make it the backbone of large pretrained models you then fine-tune. Prefer a recurrent model (Chapter 15) for truly streaming, low-latency, tiny-memory settings, and a convolution (Chapter 13) when the signal lives on a grid with strong local structure. Most state-of-the-art systems combine them, but attention is the piece that unlocked scale.

In the real world

Every modern large language model — OpenAI's GPT-4, Anthropic's Claude, Google's Gemini, Meta's open-weight Llama — is a deep stack of the decoder blocks in this chapter, scaled to tens or hundreds of billions of parameters and trained on trillions of tokens. The 2017 “Attention Is All You Need” architecture is essentially unchanged. What grew was scale, data, and engineering: FlashAttention's IO-aware kernels, rotary position embeddings, and grouped-query attention for cheaper inference. The same mechanism, applied to fixed-size image patches, gives the Vision Transformer that now rivals CNNs (Chapter 14). Applied to amino-acid sequences, it underlies AlphaFold's protein-structure breakthrough. When you use any AI chat assistant today, each word of the reply is produced by exactly the causal, masked self-attention you just computed by hand.

Exercises

1. Recompute the worked example with the query **q = [0, 1]** instead of [1, 0], keeping the same three keys and values. Find the attention weights and the output vector. MATH

2. The $\sqrt{d_k}$ divisor exists to stop softmax saturation. With $d_k = 64$ a raw score has standard deviation ≈ 8. Compare softmax([8, 0]) with softmax([1, 0]) (the scaled version) and explain what the divisor gives you. MATH

3. Add a fourth key $k_4 = [0, 0]$ with value $v_4 = [2, 2]$ to the code cell and rerun. What weight does the new key receive, and why does it still influence the output? CODE

4. Self-attention is permutation-equivariant, yet a language model clearly cares about word order. Explain how both can be true. CONCEPT

5. A decoder token at position 2 (0-indexed) produces raw scores [2, 1, 3, 0] against positions 0–3. Apply the causal mask and softmax to get its attention weights. MATH

6. Multi-head attention uses h heads each of dimension d/h rather than one head of dimension d, keeping total compute about the same. Why is the split worth it? CONCEPT

7. Empirically demonstrate that the $\sqrt{d_k}$ scaling keeps softmax soft. Draw 10 raw d=64 scores, and compare the entropy of their softmax with and without the divisor (higher entropy = softer, more spread-out weights). CODE

8. Counting multi-head parameters. A block has model dimension $d = 512$ and $h = 8$ heads. Give the per-head dimension d_k, the shape of one head's three projection matrices, and the total parameter count of all the query, key, value, and output projections. Show it does not depend on h. MATH

9. Cross-attention shapes. A translation decoder of length $n = 4$ attends to an encoded source of length $m = 6$, with $d_k = d_v = 32$. Give the shapes of Q, K, V, the score matrix S, the weight matrix A, and the output. Explain why S is not square and which dimension each row of A sums over. CONCEPT

10. Verify the three-token worked example in code: build Q = K = V from the token rows, form the scaled scores, softmax each row, and read out the outputs. Confirm the attention-matrix rows sum to 1 and that the outputs are [0.8022, 0.5989], [0.5989, 0.8022], [0.7517, 0.7517].

CODE

Language models & the transformer era

> *Give a transformer one job: predict the next token. Then train it on a large enough part of human writing, and something surprising appears. You get a single model that translates, summarizes, writes code, and answers questions it was never directly taught. A **language model** is not a new architecture. It is the causal, masked decoder from the previous chapter. We stack it deep, wrap it around a vocabulary, and train it by maximum likelihood on text. This chapter starts from that decoder and builds the language model around it: the objective, the tokens it reads, the way it is turned into new text, the scaling behavior that made this work worth billions of dollars, and the real limits that come with it.*

The language-modeling objective

A language model assigns a probability to a piece of text. That sounds abstract until you see that it is the most useful self-supervised task in machine learning. The label for each position is simply the word that actually came next, so a huge amount of unlabeled text becomes a huge amount of training examples for free. No annotation is needed and no human is involved. The corpus supervises itself. To make the probability of a whole sequence possible to compute, we factor it with the **chain rule of probability**, from left to right:

$$P(x_1, x_2, \ldots, x_T) = \Pi_{t=1..T}\, P(x_t \mid x_{<t})$$

This is the **autoregressive** factorization. It says: the probability of the sentence is the probability of the first token, times the probability of the second given the first, times the third given the first two, and so on. Every factor is a distribution over the vocabulary, conditioned on everything to its left. This is a **next-token prediction**. The model never has to represent the extremely large joint distribution directly. It only ever predicts one token at a time, and the product joins those predictions into one coherent whole.

The model produces each factor exactly as the attention chapter's classifier produced class scores. The decoder turns the left context $x_{<t}$ into a hidden vector $h_t \in \mathbb{R}^d$. An **unembedding** matrix W_u (shape $d \times V$) maps that vector to one logit per vocabulary item. Then a softmax normalizes the logits into a distribution:

$$P(x_t = w \mid x_{<t}) = \mathrm{softmax}(h_t W_u)_w = \frac{e^{(h_t W_u)_w}}{\Sigma_{w'} e^{(h_t W_u)_{w'}}}$$

Training maximizes the likelihood of the corpus. If we take the negative logarithm and average, this is exactly the **cross-entropy loss** of the loss-functions chapter (Chapter 6), now summed over positions instead of over a labeled batch:

$$\mathscr{L}(\theta) = -\frac{1}{T} \Sigma_{t=1..T} \log P_\theta(x_t \mid x_{<t})$$

x_t the token at position t (an integer index into the vocabulary of size V)

$x_{<t}$ the prefix $x_1 \ldots x_{t-1}$: every token strictly before position t

h_t the decoder's hidden vector at position t: a summary of the left context, with dimension d

W_u unembedding / output projection ($d \times V$), often *tied* to the input embedding matrix

V, T vocabulary size and sequence length. θ collects all model parameters

Two consequences of this loss are worth stating clearly. First, minimizing cross-entropy is **density estimation**: the model is rewarded for placing probability exactly where the data actually is and nowhere else. Second, the loss has a natural unit called the **perplexity**. It is the exponential of the mean cross-entropy, which is the effective number of equally-likely choices the model faces at each step:

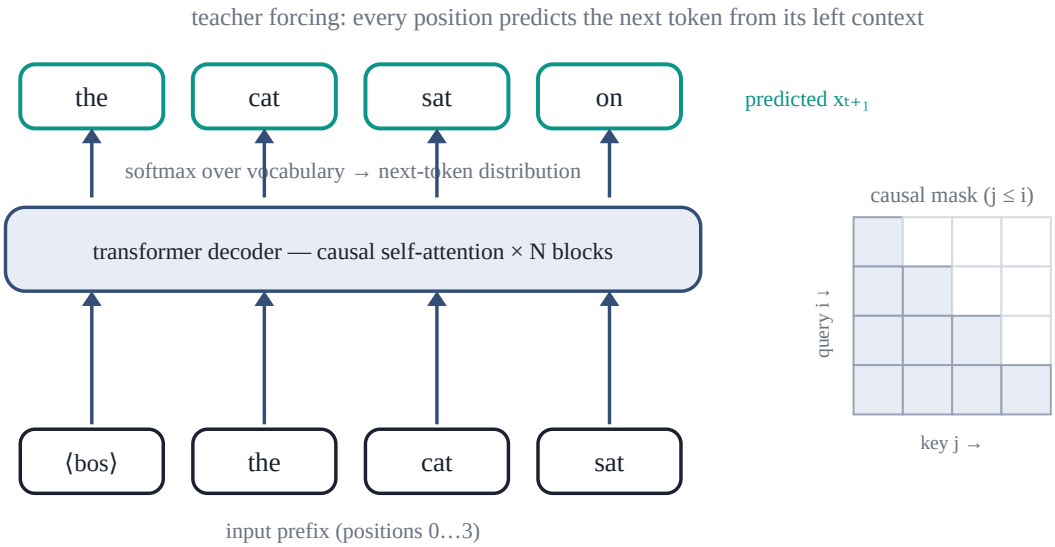
$$\mathrm{PPL} = \exp(\mathscr{L}) = \left(\Pi_{t=1..T} P_\theta(x_t \mid x_{<t}) \right)^{-1/T}$$

A model that is perfectly confused among all V tokens would score PPL = V. A model that always put probability 1 on the correct token would score PPL = 1. Real language models fall between these two. Dropping the perplexity of English text from a few hundred (early n-gram models) down toward the low tens is, honestly measured, most of the progress of the last decade.

Teacher forcing and the training/inference gap

How do we compute all T next-token losses efficiently? During training we already *have* the target sequence, so at every position t we condition on the **true** prefix $x_{<t}$, never on the model's own guesses. This is **teacher forcing**. Its benefit is parallelism. Because the causal mask (Chapter 16) already stops each position from seeing its future, a single masked forward pass over the whole sequence produces all T conditional distributions at once, and the T cross-entropy terms are averaged in one step. There is no left-to-right loop at training time. The loop only appears at generation.

This difference is also a weakness. At inference there is no ground truth to rely on, so the model must use its *own* outputs as the prefix for the next step. If it drifts even slightly away from the distribution it saw during training, the error builds up. This mismatch is known as **exposure bias**. In practice, large-scale pretraining makes the model robust enough that this rarely becomes a big problem. But it is the reason a model can begin a paragraph very well and then drift into repetition or nonsense several hundred tokens later.



One masked forward pass computes every next-token prediction at once. The input prefix ⟨bos⟩, the, cat, sat runs up through the causal decoder. Each position produces a distribution over the vocabulary, whose target is the next actual token (the, cat, sat, on). The lower-triangular mask on the right is the reason: query i may attend to key j only when j ≤ i, so no position sees its own future.

Tokenization: from characters to subwords

Before any of this can run, text must become a sequence of integers. The unit we choose, the **token**, balances two costs. Split into **characters** and the vocabulary is tiny (about a hundred symbols) and never sees an unknown word. But sequences become very long and each token carries almost no meaning, so attention's $O(T^2)$ cost grows very large and the model must relearn spelling from the beginning. Split into **words** and each token is meaningful and sequences are short. But the vocabulary is unlimited: every proper noun, typo, and hashtag is a new word, and any word not in a fixed vocabulary becomes a useless "unknown" symbol.

Subword tokenization is a compromise between these two extremes and is what every modern model uses. The most common method, **Byte-Pair Encoding** (BPE), starts from the raw bytes (or characters) and greedily merges the most frequent adjacent pair into a new symbol, repeating until the vocabulary reaches a target size (typically 30k–100k). Common words like " the" survive as single tokens. A rare word like "tokenization" is broken into familiar fragments such as "token" + "ization". A string never seen before still breaks into bytes it has definitely seen. The result is an **open vocabulary** at a fixed table size: nothing is ever truly out-of-vocabulary, frequent text stays short, and rare or new text breaks down smoothly into pieces rather than disappearing. Byte-level BPE, which merges over the 256 possible bytes, guarantees that every possible input, in any language or script, can be represented. Tokenization is not exciting but it is important: it quietly sets the model's arithmetic ability, its handling of code and non-English text, and even how many words fit in a fixed context window.

Embeddings and position

Each token id selects a row of a learned **embedding matrix** E (shape $V \times d$), turning the integer into a dense vector. This is the same lookup-table-as-linear-layer trick introduced with representations (Chapter 5) and reused for the value vectors in attention. Because self-attention is permutation-equivariant (Chapter 16 showed it only ever compares content), position must be provided separately. The chapter before this used fixed **sinusoidal** encodings, PE(pos, 2i) =

sin(pos / $10000^{2i/d}$) and its cosine partner, added to the token embedding so each position carries a unique multi-frequency pattern. Two modern alternatives are now most common in large models, and both are worth understanding at a basic level.

Learned absolute positions replace the sinusoids with a second trainable table P (shape $T_{max} \times d$), one vector per position, added to the token embedding. This is simple and effective, but it cannot represent any position beyond the T_{max} rows it was trained on. **Rotary position embedding** (RoPE) takes a more subtle approach: rather than *adding* anything, it *rotates* the query and key vectors by an angle proportional to their absolute position before the dot product. Because rotating a query at position m and a key at position n and then taking their inner product gives a result that depends only on the difference $m - n$, the attention score becomes a function of **relative** position at no extra cost. This is a property that generalizes better across different lengths. **ALiBi** is even simpler and cheaper: it adds a fixed, head-specific linear penalty $-s \cdot (i - j)$ to each score before the softmax, gently biasing every query toward nearby keys. Because that penalty is defined for any distance, an ALiBi model can be run on sequences far longer than it was trained on and still behave reasonably. This is a clear example of how a small, well-chosen inductive bias leads to length extrapolation.

The decoder-only transformer block

The attention chapter derived every part we need: scaled dot-product attention, the causal mask, multi-head attention, residual connections, and the position-wise feed-forward network. Here we put them together, using the two improvements that define the modern block. The first is **pre-normalization**: layer normalization (Chapter 11) is applied to the *input* of each sublayer rather than to its output. The second is the **GELU** activation (Chapter 4) in place of ReLU inside the feed-forward network. A single decoder block maps a hidden sequence h (shape $B \times T \times d$) to a refined sequence of the same shape:

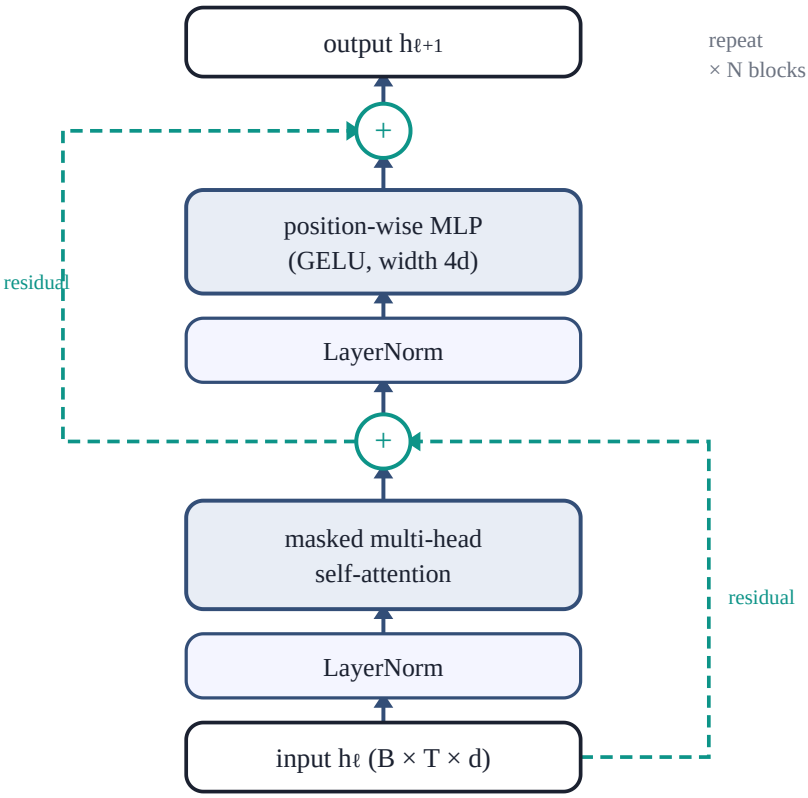
$$\tilde{h} = h + \mathrm{MaskedMHA}(\,\mathrm{LayerNorm}(h)\,)$$

$$\mathrm{out} = \tilde{h} + \mathrm{MLP}(\,\mathrm{LayerNorm}(\tilde{h})\,) \qquad \mathrm{MLP}(z) = \mathrm{GELU}(z\,W_1 + b_1)\,W_2 + b_2$$

Reading these carefully is worth the effort. Each sublayer sits on a **residual** highway: its output is *added* to its input, so the identity path runs freely from the embeddings to the final layer, and gradients flow straight down it. This is why blocks can be stacked dozens deep without the vanishing-gradient failure of Chapter 10. The LayerNorm now sits *inside* the residual branch (pre-norm), which keeps the highway itself un-normalized and makes very deep stacks train stably. The original 2017 transformer put the norm after the addition (post-norm), which needs careful learning-rate warmup to match. The MLP is applied to each position independently, widening to an inner dimension of typically 4d and back. It is where much of the model's factual capacity is stored, while attention does the mixing across positions. GELU(x) = x·Φ(x), with Φ the standard-normal CDF, is a smooth gate that, unlike ReLU, passes a small negative signal near the origin, which in practice trains slightly better in these models.

$$\mathrm{GELU}(x) = x \cdot \Phi(x) = x \cdot P(Z \le x), \quad Z \sim \mathcal{N}(0,1)$$

Stacking N such blocks between the embedding layer and the unembedding head *is* the model. GPT, Llama, and their relatives differ mostly in N, d, the number of heads, the position scheme, and small activation changes (many now use gated variants such as SwiGLU). The basic structure is exactly the one drawn below.



The pre-norm decoder block. Each sublayer normalizes its input, transforms it, and adds the result back onto the residual highway (teal). Attention mixes information across positions under the causal mask. The MLP transforms each position independently. Stacking N of these blocks, between a token+position embedding and a softmax head, is the entire model.

Counting the parameters

A rough count tells you where a model's capacity is and why. Consider the parameters in one block, ignoring the very small LayerNorm and bias terms. The attention sublayer has four d × d projections (query, key, value, and the output mix W_O), for $4d^2$. The MLP has W_1 of shape d × 4d and W_2 of shape 4d × d, for $8d^2$. So one block holds

$$N_{\text{block}} \approx 4d^2 + 8d^2 = 12\,d^2$$

and a stack of L blocks, plus the token embedding (V × d, tied with the unembedding) and any learned position table, gives the model total

$$N \approx 12\,L\,d^2 + V\,d$$

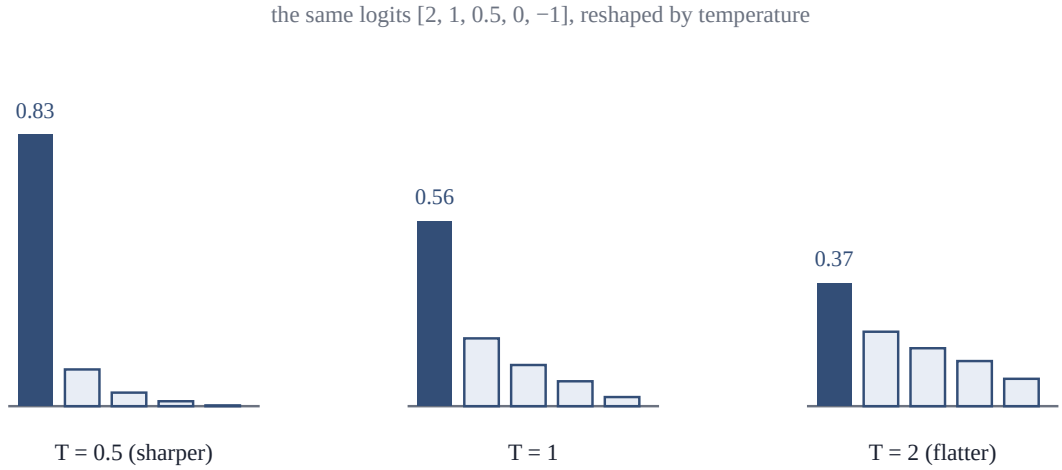
Put in the values for GPT-2 small ($d = 768$, $L = 12$, $V = 50\,257$, $T_{max} = 1024$), and the arithmetic gives a familiar number. Each block is $12 \cdot 768^2 \approx 7.08$M parameters. Twelve blocks are ≈ 84.9M. The embedding is $50\,257 \cdot 768 \approx 38.6$M and the position table ≈ 0.79M, for a total ≈ **124M**. That is exactly the published size of GPT-2 small. The formula scales the same way up to the hundred-billion-parameter scale: the $L\,d^2$ term dominates once the model is deep and wide. This is why the embedding, which is a large fraction of a small model, becomes a rounding error in a large one.

Decoding: turning probabilities into text

A trained model gives a distribution over the next token. Turning that into text is a separate design choice called **decoding**, and it matters as much as the model. The simplest rule, **greedy** decoding, always takes the argmax. It is deterministic and often fluent, but it tends to be dull and to fall into repetition loops, because the single most-probable continuation is rarely the most interesting one, and choices that look best at each step are not always best overall. **Temperature** reshapes the distribution before sampling by dividing the logits by $T > 0$:

$$P_T(w) = \mathrm{softmax}(z / T)_w = \frac{e^{z_w/T}}{\Sigma_{w'} e^{z_{w'}/T}}$$

As $T \to 0$ the distribution sharpens onto the argmax (recovering greedy). At $T = 1$ it is the model's own distribution. As $T \to \infty$ it flattens toward uniform. Sampling from raw temperature still occasionally draws from the long, unreliable tail of the distribution, so two **truncation** methods cut it off first. **Top-k** keeps only the k highest-probability tokens and renormalizes over them. **Top-p** (nucleus) sampling is more adaptive: it keeps the smallest set $\mathscr{V}_p$ of tokens whose cumulative probability first reaches p, then renormalizes. So a confident, peaked distribution keeps just a few candidates, while a flat, uncertain one keeps many. Together, a moderate temperature (≈0.7–1.0) with top-p ≈ 0.9 is the standard setting for open-ended generation, giving up a little fluency for the diversity that keeps text from collapsing into loops.



Temperature is a single knob on confidence. Low T concentrates almost all mass on the top token (near-greedy). High T spreads it toward uniform, which adds diversity and risk. These are the exact probabilities computed in the worked example below.

The KV cache and the cost of context

Generation is a loop: predict a token, append it, predict the next. If done naively, each new step runs attention again over the entire prefix, recomputing the keys and values of every past token. This is $O(T^2)$ work at step T, and $O(T^3)$ to produce a length-T sequence. The fix is the **KV cache**. Because a past token's key and value never change once computed, we store them. Each new token then computes only its own query, key, and value, attends its single query against the T cached keys, and appends its own key/value for next time. Per generated token the cost drops to $O(T)$, and the whole sequence to $O(T^2)$. This is the difference between usable and hopeless at long context.

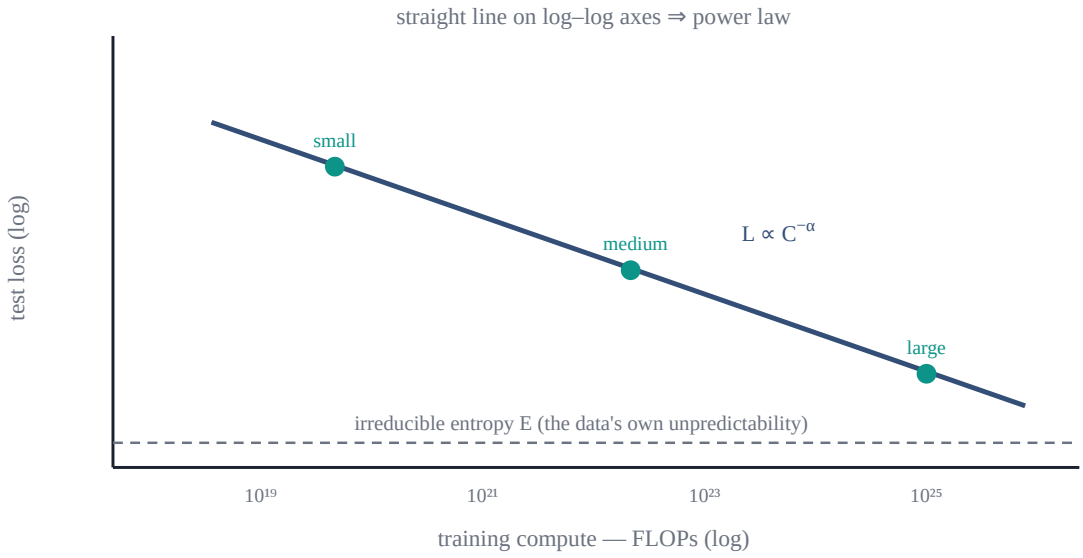
But there is a cost. The cache itself must hold two vectors (a key and a value) for every layer and every past position: roughly $2 \cdot L \cdot T \cdot d$ numbers per sequence. For a 32-layer, $d = 4096$ model at an 8192-token context that is ≈ 2.1 billion numbers, about 4 GB in half precision *per sequence in the batch*. This memory, not compute, is often what limits a deployed model's context length and batch size. It is why cheaper-inference methods such as grouped-query attention (sharing keys and values across heads) and quantized caches are active areas of engineering (Chapter 27). Context length is further limited by the $O(T^2)$ attention cost and by the position scheme's ability to generalize beyond its training lengths, which is the problem RoPE-interpolation and ALiBi were built to reduce.

Scaling laws

The experimental finding that turned language modeling into an industry is that its test loss is **predictable**. Kaplan et al. (2020) trained many models across many orders of magnitude and found the cross-entropy falls as a smooth **power law** in each of three quantities (the number of non-embedding parameters N, the dataset size D, and the training compute C) over a wide range. When the others are not limiting,

$$L(N) \approx (N_c / N)^{\alpha_N}, \quad L(D) \approx (D_c / D)^{\alpha_D}, \quad L(C) \approx (C_c / C)^{\alpha_C}$$

with small exponents ($\alpha_N \approx 0.076$ in the original fit). A power law is a straight line on log–log axes, so a few small training runs let you *extrapolate* the loss of a model too expensive to train. This is an unusually reliable planning tool. The exponents are small, which is the cautionary part of the story: driving the loss down takes exponentially more scale, and each factor of ten in compute gives a small, shrinking drop.



The scaling law. Across many orders of magnitude, test loss falls as a power law in compute, drawing a straight line on log–log axes. A few cheap runs predict the loss of a model too costly to build. The exponent is small (each 10× of compute gives a small, shrinking drop), and the line finally bends toward the irreducible entropy of language itself.

Kaplan's analysis suggested that, given more compute, you should spend most of it making the model *bigger*. Hoffmann et al. (2022), the **Chinchilla** paper, looked at the question again with a more careful fit and reached a different, now-standard conclusion. Modeling the loss as an irreducible term plus two power laws,

$$L(N, D) = E + A / N^{\alpha} + B / D^{\beta}$$

and minimizing it under a fixed compute budget $C \approx 6ND$, they found that parameters and data should grow in **equal proportion**: roughly $N \propto C^{0.5}$ and $D \propto C^{0.5}$, a compute-optimal ratio of about **20 training tokens per parameter**. The key result was concrete and expensive: a 70-billion-parameter model trained on 1.4 trillion tokens (Chinchilla) outperformed a 280-billion-parameter model trained on far fewer tokens (Gopher) at the *same* compute. Most large models before 2022 were badly over-sized and under-trained. The field's common guideline shifted from "make it bigger" to "feed it more." The deeper concern, now visible at the largest scale, is that data is finite. You cannot keep finding 20× more high-quality tokens. So inference cost, data quality, and repetition increasingly control the optimum rather than the clean two-variable law.

Emergent abilities, carefully

Along with the smooth loss curves came a louder claim: that certain abilities (multi-step arithmetic, instruction following, chain-of-thought reasoning) appear **suddenly** past a certain scale, absent in smaller models and present in larger ones (Wei et al., 2022). The effect is real in the sense that a task's measured score can jump from near-zero to useful over a narrow range of scale, and it makes capabilities genuinely hard to predict from loss alone. But the interpretation needs care. Schaeffer et al. (2023) argued that many reported "emergences" are partly artifacts of the **metric**: a task scored by exact-match (all-or-nothing over several tokens) will look like a sharp phase transition even when the underlying per-token probability is improving perfectly smoothly, simply because a strict threshold turns gradual gains into a sudden pass/fail. Under continuous metrics, many such curves become smooth. So the honest summary has two sides. Scale reliably lowers loss along a predictable power law, and some downstream capabilities do become useful over a small scale range. But "emergence" is not magic. It is sensitive to how you measure, and it is not a reason to assume that even more scale will produce any particular ability on demand.

From a pretrained model to a helpful assistant

Pretraining on next-token prediction produces a model that is fluent and knowledgeable but not, on its own, obedient: prompt a raw base model with a question and it may continue with more questions, because that is what its training distribution often did. Turning it into an assistant is a short, separate pipeline added on top of the expensive pretraining. First, **supervised fine-tuning** (SFT), also called **instruction tuning**: continue training on a carefully chosen set of (instruction, good response) demonstrations, so the model learns the *format* of being helpful and generalizes to instructions it never saw. This is a form of transfer learning (Chapter 22). Second, **alignment** from human preferences, because we can more easily judge which of two responses is better than we can write the ideal one. **RLHF** (reinforcement learning from human feedback) trains a reward model on human preference comparisons and then optimizes the language model as a policy against that reward with a reinforcement-learning algorithm (Chapter 23). **DPO** (Direct Preference Optimization) reaches a similar result more simply, skipping the separate reward model and RL loop to optimize a classification-style loss directly on the preference pairs.

Two further ideas need no weight updates at all. **In-context learning** is the striking property, largely emergent from scale, that a frozen model can perform a new task purely from examples placed in its prompt. This is the "few-shot" prompting that made GPT-3 famous. It works because the pattern in the context guides the same next-token machinery. **Prompting** is the practice built on this: phrasing, ordering, and worked examples measurably change behavior, which is powerful but also, as we will see, a source of weakness. Finally, **retrieval-augmented generation** conditions the model on documents fetched at query time from an external store, adding fresh or private facts the frozen weights never memorized and grounding answers in citable sources.

Language models in code

The two runnable cells below make the decoding and evaluation concrete on the exact five-token distribution used throughout this chapter. Each imports only NumPy and prints numbers you can check by hand. The first performs temperature scaling followed by top-k truncation,

then draws one token.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.random.seed(0)

logits = np.array([2.0, 1.0, 0.5, 0.0, -1.0])      # scores over a 5-token vocab
vocab  = ["the", "cat", "sat", "on", "mat"]

def softmax(z):
    z = z - z.max()                                  # subtract max for stability
    e = np.exp(z)
    return e / e.sum()

T = 0.7                                              # temperature below 1 sharpens
k = 3                                                # keep only the top-3 logits
scaled = logits / T
top    = np.argsort(scaled)[-k:]                     # indices of the k largest
filt   = np.full_like(scaled, -np.inf)               # everything else -> -inf
filt[top] = scaled[top]
probs  = softmax(filt)                               # renormalised over the top-k

print("top-k probs:", np.round(probs, 4))            # [0.7369 0.1766 0.0865 0. 0.]
tok = np.random.choice(len(vocab), p=probs)          # sample one token id
print("sampled id :", tok, "->", vocab[tok])         # 0 -> the
```

The second evaluates a model by **perplexity**: exponentiate the mean per-token cross-entropy of a short target sequence. With the model assigning probabilities 0.5, 0.2, 0.1 to the three true next tokens, the perplexity is $(0.5 \cdot 0.2 \cdot 0.1)^{-1/3} = 10^{2/3} \approx 4.64$. The model is about as uncertain as a fair choice among ~4.6 options at each step.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.random.seed(0)
```

```
# the model's probability for each ACTUAL next token in a 3-token target
p_true  = np.array([0.5, 0.2, 0.1])
ce      = -np.log(p_true)                          # per-token cross-entropy
(nats)
mean_ce = ce.mean()
ppl     = np.exp(mean_ce)                          # perplexity = exp(mean
CE)

print("per-token CE :", np.round(ce, 4))           # [0.6931 1.6094 2.3026]
print("mean CE      :", round(float(mean_ce), 4)) # 1.5351
print("perplexity   :", round(float(ppl), 4))     # 4.6416  = 10**(2/3)
```

Finally, a PyTorch listing (shown, not run here) gives a minimal causal self-attention module (the fused-projection, multi-head, triangular-mask form used in real code) and a generation loop that supports greedy and temperature sampling. It is the concrete version of the decoder block and the decoding loop described above.

PyTorch copy → Colab

```
import torch, torch.nn as nn
import torch.nn.functional as F

class CausalSelfAttention(nn.Module):
    def __init__(self, d_model, n_head):
        super().__init__()
        self.n_head = n_head
        self.qkv  = nn.Linear(d_model, 3 * d_model)   # fused Q, K, V
projection
        self.proj = nn.Linear(d_model, d_model)       # output mix  W_O

    def forward(self, x):                             # x: (B, T, d_model)
        B, T, C = x.shape
        q, k, v = self.qkv(x).split(C, dim=2)
        hd = C // self.n_head                          # per-head dimension
        q = q.view(B, T, self.n_head, hd).transpose(1, 2)   # (B, h, T, hd)
        k = k.view(B, T, self.n_head, hd).transpose(1, 2)
        v = v.view(B, T, self.n_head, hd).transpose(1, 2)
        att  = (q @ k.transpose(-2, -1)) / hd ** 0.5   # (B, h, T, T) scores
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        att  = att.masked_fill(mask, float('-inf'))    # forbid attending
ahead
        att  = F.softmax(att, dim=-1)
```

```
        y = att @ v                                          # (B, h, T, hd)
        y = y.transpose(1, 2).reshape(B, T, C)               # concat heads
        return self.proj(y)

@torch.no_grad()
def generate(model, idx, max_new, temperature=1.0, greedy=False):
    # model(idx) returns logits of shape (B, T, vocab); idx is (B, T) token
ids
    for _ in range(max_new):
        logits = model(idx)[:, -1, :] / temperature    # next-token logits
(B, V)
        if greedy:
            nxt = logits.argmax(dim=-1, keepdim=True)   # deterministic
argmax
        else:
            probs = F.softmax(logits, dim=-1)
            nxt   = torch.multinomial(probs, num_samples=1)   # temperature
sample
        idx = torch.cat([idx, nxt], dim=1)              # append and feed
back in
    return idx                                          # add top-k/top-p to
taste
```

Worked example: softmax, temperature, and perplexity by hand

Take the logits z = [2.0, 1.0, 0.5, 0.0, −1.0] over a 5-token vocabulary.

1. **Softmax at T = 1.** Exponentiate: $e^{2} = 7.389$, $e^{1} = 2.718$, $e^{0.5} = 1.649$, $e^{0} = 1.000$, $e^{-1} = 0.368$. Sum = 13.124. Divide: P = [**0.5630**, 0.2071, 0.1256, 0.0762, 0.0280]. The top token holds 56% of the probability.
2. **Sharpen with T = 0.5.** Divide logits by 0.5 → [4, 2, 1, 0, −2] and softmax: P = [**0.8292**, 0.1122, 0.0413, 0.0152, 0.0021]. Halving the temperature squared the odds ratios. The peak rose from 0.56 to 0.83.
3. **Flatten with T = 2.** Divide by 2 → [1, 0.5, 0.25, 0, −0.5] and softmax: P = [**0.3745**, 0.2272, 0.1769, 0.1378, 0.0836]. The peak fell to 0.37 and the tail rose, giving more diversity and more risk.

4. **Cross-entropy of a 3-token target.** Suppose the model assigns the true next tokens probabilities 0.5, 0.2, 0.1. Per-token cross-entropies (in nats) are $-\ln 0.5 = 0.693$, $-\ln 0.2 = 1.609$, $-\ln 0.1 = 2.303$. Their mean is $\mathscr{L} = (0.693 + 1.609 + 2.303)/3 = \mathbf{1.5351}$ nats.
5. **Perplexity.** PPL $= e^{1.5351} = \mathbf{4.6416}$. Equivalently $(0.5 \cdot 0.2 \cdot 0.1)^{-1/3} = (0.01)^{-1/3} = 10^{2/3} = 4.6416$. The model is as uncertain as a uniform choice among ≈ 4.6 tokens per step.

What the numbers mean: temperature is a single confidence dial that leaves the ranking untouched but stretches or compresses the gaps. Cross-entropy scores how much probability the model put on what actually happened. And perplexity re-expresses that score as an intuitive "effective number of choices." Lowering perplexity from hundreds to tens is exactly what pretraining gives you.

Assumptions & limitations

A language model is trained to be *plausible*, not *truthful*: cross-entropy rewards fluent continuations of its training distribution, so it will confidently produce well-formed falsehoods (called **hallucinations**) with no internal signal that it has done so, and its probabilities are not calibrated estimates of factual correctness. It is **brittle to the prompt**: rephrasing a question, reordering few-shot examples, or an extra instruction can change the output, because in-context learning conditions on surface form as well as meaning. It has a hard **context limit** set by $O(T^2)$ attention and linear KV-cache memory, so information outside the window simply does not exist to the model. And it is **expensive**: pretraining at the largest scale costs millions of GPU-hours and megawatts of power, and inference at scale is an ongoing cost. This is the reason quantization, distillation, and efficient attention (Chapter 27) are not optional. Finally, the model inherits the biases, gaps, and out-of-date content of its training corpus, and no amount of fluent phrasing removes them.

Why & when this matters

Use a pretrained language model when the task is expressible as text-to-text and you value broad coverage and fast iteration over guarantees: drafting and rewriting, summarization, code assistance, classification and extraction via prompting, and as a flexible reasoning component inside a larger system. Prefer a smaller, task-specific model when latency, cost, privacy, or auditability are the main concern, and pair the model with **retrieval** and explicit tools whenever factual precision or fresh knowledge is required. The generator supplies fluency and structure, while the retriever supplies grounded facts. Treat every output as a draft to check, not a final answer to trust.

In the real world

Every widely used AI assistant (OpenAI's GPT-4-class models, Anthropic's Claude, Google's Gemini, Meta's open-weight Llama, Mistral, and DeepSeek) is exactly the decoder stack of this chapter, scaled to tens or hundreds of billions of parameters, pretrained on trillions of tokens, then instruction-tuned and preference-aligned. The 2017 architecture is almost unchanged. What grew was scale, data curation, and systems engineering: FlashAttention's IO-aware kernels, rotary embeddings, grouped-query attention and paged KV caches for cheap serving, and the SFT-plus-RLHF/DPO method that turns a next-token predictor into something you can talk to. When a chat assistant streams a reply one token at a time, you are watching temperature-and-top-p sampling from the masked, causal softmax you computed by hand.

Exercises

1. Show the two limits of temperature on the logits $z = [2, 1, 0]$. Compute the softmax at $T = 1$, then describe (with the numbers) what happens as $T \to 0$ and $T \to \infty$. MATH

2. A model spreads its probability uniformly, $1/V$, over a vocabulary of $V = 50\,000$ at every step. What is its perplexity, and what does that tell you about a real model whose perplexity is 12? MATH

3. Explain teacher forcing and why the same choice creates "exposure bias" at generation time. CONCEPT

4. Count the parameters of a decoder with model dimension $d = 1024$, $L = 24$ blocks, and vocabulary $V = 50\,000$ (tied embeddings). How large is the embedding relative to the whole? MATH

5. Implement **top-p (nucleus)** sampling in NumPy on the chapter's logits with $p = 0.9$: keep the smallest set of highest-probability tokens whose cumulative probability first reaches 0.9, then renormalize. Which tokens survive? CODE

6. Why do modern models tokenize into subwords (BPE) rather than into whole words or into single characters? Give the concrete failure of each extreme. CONCEPT

7. Explain why autoregressive generation is $O(T^2)$ total *with* a KV cache and $O(T^3)$ *without* one, and estimate the cache memory for $L = 32$ layers, $d = 4096$, and a $T = 8192$ context. MATH

8. A blog post claims a model "suddenly learned three-digit addition" at a certain size, calling it an emergent ability. Give the balanced reading a careful reader should apply. CONCEPT

Graph neural networks

> *Most data that matters is not a neat grid of pixels or a straight line of words. It is a web of relationships. Atoms bonded into a molecule, users following users, roads meeting at junctions, papers citing papers: each is a **graph**, a set of entities joined by connections that carry the real signal. A **graph neural network** learns on that structure directly, by letting every node repeatedly gather information from its neighbors and mix it with its own. This chapter builds that idea from scratch: the adjacency matrix, the message-passing loop, and the Graph Convolutional Network's normalized aggregation. It then works through one graph convolution by hand, to the last decimal.*

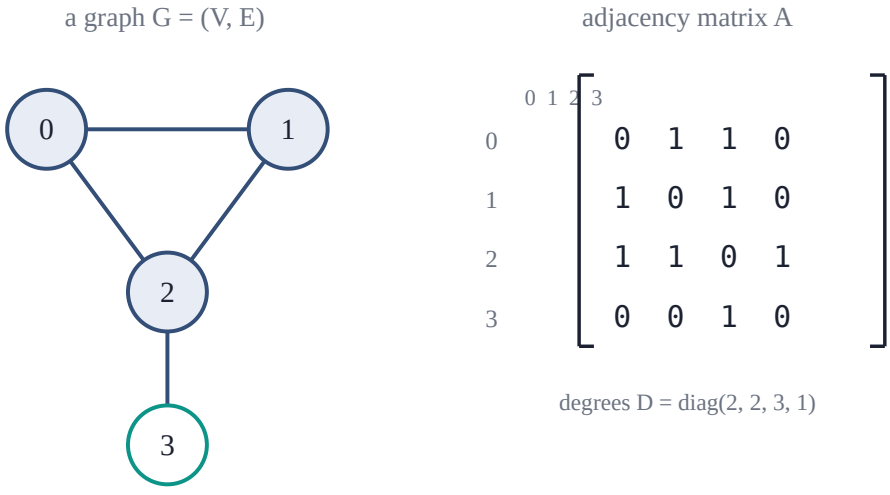
Fundamental concept

Picture a rumor spreading through a circle of friends. You form your view by blending what you already think with what the people you talk to are saying. They do the same with theirs. After one round of gossip you have heard from your immediate friends. After two rounds, news from friends-of-friends arrives, slightly weakened. After a few rounds, information from far parts of the network has reached you. A graph neural network is exactly this process made differentiable: a node's representation is refined, round after round, by aggregating its neighbors'.

Let us make the objects precise. A graph $G = (V, E)$ is a set of **nodes** (vertices) V and a set of **edges** E joining pairs of them. Every node i carries a feature vector x_i: for an atom, its element and charge; for a user, their profile and activity. We stack these as the rows of a feature matrix X of shape $n \times d$ (n nodes, d features each). This is exactly the row-per-example convention used since the multilayer perceptron (chapter 3). Edges may carry features too (a bond's type, a road's length). The connectivity is recorded in the **adjacency matrix** A, an $n \times n$ matrix with $A_{ij} = 1$ when an edge joins i and j and 0 otherwise. For an undirected graph, A is symmetric. Summing each row gives the **degree** of a node, meaning how many neighbors it has. We collect these degrees on the diagonal of the **degree matrix** D, with $D_{ii} = \Sigma_j A_{ij}$ and zeros off the diagonal.

Why does this generalize the networks you already know? Because grids and sequences are just *special* graphs. An image is a graph whose nodes are pixels connected to their four or eight grid neighbors, and a convolution (chapter 13) is aggregation over that grid with a shared filter. A sentence is a chain graph, with each token linked to the next, which a recurrent network (chapter 15) walks through in order. Both build in a *fixed, ordered* neighborhood: always four pixels up-down-left-right, always one token before and after. A general graph has neither of these advantages: a node may have two neighbors or two thousand, with no standard "first" neighbor. Removing those assumptions, namely fixed size and fixed order, is exactly what a graph network must handle. This is why it takes some care to build.

Three kinds of task come up again and again, and naming them clarifies what the output means. **Node classification** labels individual nodes (is this account fraudulent, does this protein bind) using the graph as context. **Link prediction** scores whether an edge *should* exist between two nodes (will this user like this movie), and powers recommendation. **Graph classification** labels a whole graph at once (is this molecule toxic), which needs a way to collapse many node vectors into one. The same message-passing core serves all three. Only the final read-out differs.

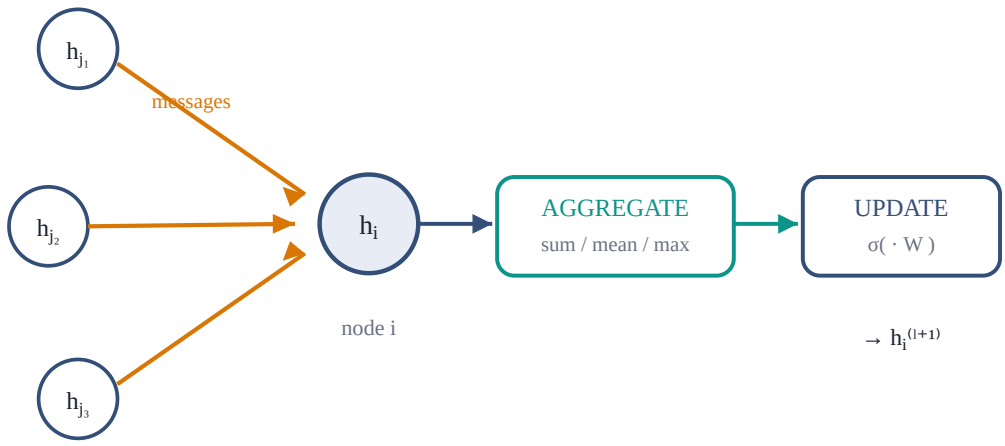


A graph and its adjacency matrix are two views of the same object. Node 2 is a hub of degree 3. Node 3 is a leaf with a single neighbor. Row i of A lists node i's neighbors as 1s, so the row sum is its degree. This exact graph is worked through by hand in a full graph convolution later in the chapter.

Message passing: the core principle

Every graph network is built from one repeated operation, worth stating in words before symbols: *each node computes a new state by aggregating messages from its neighbors and combining that summary with its own, repeated for several rounds.* One round is one **layer**. After K layers a node's state reflects everything within K edges, called its **K-hop neighborhood**, because information relayed each round travels one more hop out. This is the graph generalization of a convolution: where a CNN filter pools a fixed grid patch, a message-passing layer pools a node's variable-sized neighborhood.

The subtle part is the word *aggregate*. A node's neighbors arrive as an unordered set, with no first and no second, so the combining function must return the same answer whatever order we list them in. It must be **permutation-invariant**: a sum, a mean, or an elementwise max, never something order-dependent like concatenation or a plain recurrent read. This single requirement is the most important choice in the whole field, and the next sections show it is exactly the right inductive bias for data with no natural node ordering.



One message-passing layer at node i. Its neighbors send messages (amber). A permutation-invariant AGGREGATE collapses that unordered set into one vector. UPDATE mixes the summary with the node's own state and applies a learned transform and nonlinearity. Stacking layers pushes each node's view one more hop out.

Governing equations

In its most general form, a message-passing layer computes, for every node i, an aggregated message from its neighborhood N(i) and then an update:

$$m_i^{(l)} = \mathrm{AGG}_{j \in N(i)} \left(h_j^{(l)} \right) \qquad h_i^{(l+1)} = \mathrm{UPDATE} \left(h_i^{(l)}, m_i^{(l)} \right)$$

Different choices of AGG and UPDATE give different graph networks. The **Graph Convolutional Network** (GCN) of Kipf & Welling (2017) gives an elegant example: aggregate with a *degree-weighted sum that includes the node itself*, transform by a shared weight matrix, and squash. Written per node, with $\tilde{N}(i) = N(i) \cup \{i\}$ the neighborhood *plus* the node,

$$h_i^{(l+1)} = \sigma \left(\Sigma_{j \in \tilde{N}(i)} \frac{1}{\sqrt{\tilde{d}_i \, \tilde{d}_j}} \, h_j^{(l)} W^{(l)} \right)$$

Stacking all nodes' vectors as the rows of $H^{(l)}$ turns the layer into one matrix expression: a sparse matrix multiply, a dense one, and a nonlinearity:

$$H^{(l+1)} = \sigma \left(\tilde{D}^{-1/2} \, \tilde{A} \, \tilde{D}^{-1/2} \, H^{(l)} W^{(l)} \right), \quad \tilde{A} = A + I, \quad \tilde{D}_{ii} = \Sigma_j \, \tilde{A}_{ij}$$

Abbreviating the fixed, precomputable propagation matrix as $\hat{A} = \tilde{D}^{-1/2} \, \tilde{A} \, \tilde{D}^{-1/2}$, the layer is simply $H^{(l+1)} = \sigma(\hat{A} \, H^{(l)} \, W^{(l)})$: aggregate along the graph ($\hat{A}$), transform the features (W), squash (σ).

A, $\tilde{A}$ — adjacency matrix (n×n). $\tilde{A} = A + I$ adds a self-loop to every node so a node stays in its own neighborhood

$\tilde{D}$ — diagonal degree matrix of $\tilde{A}$. $\tilde{d}_i = \tilde{D}_{ii}$ is node i's degree after self-loops, so a leaf counts as degree 2, not 1

$\hat{A}$ — the symmetric-normalized adjacency $\tilde{D}^{-1/2} \, \tilde{A} \, \tilde{D}^{-1/2}$. Entry $\hat{A}_{ij} = 1 / \sqrt{\tilde{d}_i \, \tilde{d}_j}$ weights each edge by both endpoints' degrees

$H^{(l)}$ — node representations at layer ℓ, one row per node ($H^{(0)} = X$, the raw features). Shape $n \times d_\ell$

$W^{(l)}$ — learned weight matrix of layer ℓ, shape $d_\ell \times d_{\ell+1}$, shared across *all* nodes. This is the graph analog of a convolution's shared filter

σ — elementwise nonlinearity, typically ReLU on hidden layers (chapter 4). The last layer's activation matches the task (softmax for classification)

$N(i)$, $\tilde{N}(i)$ — the neighbors of node i, and the neighbors together with i itself

Why symmetric normalization, and why self-loops

The GCN's aggregator looks more complicated than a plain neighbor sum, but each extra piece is there for a good reason. Start with the most basic choice, AH: multiplying the raw adjacency by the features sums each node's neighbors' features. It has two problems. First, it *leaves the node out*: the sum runs over neighbors only, so after one layer a node's representation forgets what the node itself was. A self-loop fixes this: $\tilde{A} = A + I$ puts a 1 on the diagonal, folding each node into its own aggregation so the update blends "what I am" with "what surrounds me." Self-loops also rescue isolated nodes, which would otherwise have degree zero and break the normalization.

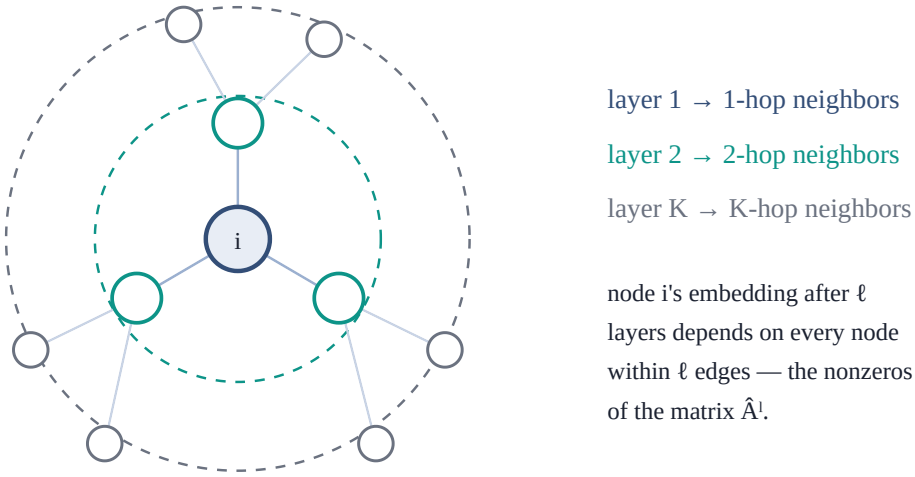
The second problem is scale. A hub with a thousand neighbors gives a summed message a thousand times larger than a leaf's. So raw sums make activations depend strongly on degree, and stacking such layers makes the effect worse until the signal explodes or collapses. This is the instability that careful initialization (chapter 10) and normalization layers (chapter 11) exist to prevent. The fix is to normalize by degree. The simplest choice, $\tilde{D}^{-1}\tilde{A}$, divides each row by its own degree, replacing the sum with an **average**. This is a random walk's transition matrix, whose rows sum to 1. The GCN instead uses the **symmetric** form $\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$, scaling the edge between i and j by $1 \diagup \sqrt{\tilde{d}_i\,\tilde{d}_j}$. This uses *both* endpoints' degrees, not just the receiver's.

The symmetric form is useful in two ways. In practice, a message *from* a high-degree hub is given less weight. A node that connects to everyone is a weak signal about any one neighbor, and the $\sqrt{\tilde{d}_j}$ in the denominator lowers it. A link from a selective, low-degree node counts for more. Mathematically, $\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$ is symmetric (unlike the random walk), so its eigenvalues are real and, with self-loops, bounded by 1 in magnitude. Repeatedly multiplying by such an operator keeps the signal from exploding as layers stack. The cost is that $\hat{A}$'s rows no longer sum to 1. It is a degree-aware diffusion, not a true average. For the leaf edge (2, 3), where $\tilde{d}_2 = 4$ and $\tilde{d}_3 = 2$, the weight is $1 \diagup \sqrt{4 \cdot 2} = 1 \diagup \sqrt{8} = 0.3536$. This number reappears when we compute the example by hand.

Permutation symmetry: the right inductive bias

A graph's node numbering is arbitrary. Relabel the nodes (swap "0" and "3") and you have the same graph, drawn with different name tags. The adjacency matrix is only defined *up to* a simultaneous shuffle of its rows and columns, $A \mapsto PAP^T$ for a permutation matrix P. Any correct graph model must therefore not depend on the labeling, and message passing satisfies this automatically because its aggregator is a symmetric function of an unordered set. The precise statements are worth telling apart. A node-level GNN is **permutation-equivariant**: relabel the input and the per-node outputs relabel identically, $\text{GNN}(PAP^T, PX) = P \cdot \text{GNN}(A, X)$. A graph-level GNN, which pools all nodes into one vector, is **permutation-invariant**: relabeling leaves the graph's prediction unchanged. Equivariance is the correct bias for tasks defined on nodes (classify each node), and invariance for tasks defined on the whole graph (classify the molecule).

This is the same symmetry idea we saw with attention (chapter 16), with one important difference. Self-attention is also permutation-equivariant, and a transformer must *break* that symmetry with positional encodings because word order carries meaning. A graph network keeps the symmetry and adds structure through the adjacency matrix instead: where the transformer says "position 5 follows position 4," the GNN says "node 5 is adjacent to node 4," letting the graph, not an index, define locality. Building the symmetry in rather than learning it is a large part of why GNNs are so sample-efficient on relational problems. They never waste capacity discovering that node names are meaningless.



Depth is reach. One layer mixes a node with its direct neighbors (inner ring). A second layer mixes those already-mixed vectors, so two-hop information arrives (outer ring). After K layers node i's receptive field is its entire K-hop neighborhood, which is exactly where $\hat{A}^K$ is nonzero.

A family of aggregators: GraphSAGE and graph attention

The GCN fixes its aggregation weights by degree, before any learning. Two influential variants relax that in opposite directions: one for scale, one for flexibility.

GraphSAGE (Hamilton, Ying & Leskovec, 2017), from "sample and aggregate," is designed for graphs too large to process all at once. Two ideas make it work. First, instead of aggregating *all* of a node's neighbors (a hub might have millions), it *samples* a fixed-size subset each layer, limiting the work per node regardless of degree. Second, it keeps the node's own vector separate from the neighbor summary, concatenating the two before the transform:

$$h_i^{(l+1)} = \sigma\left(W^{(l)} \cdot \left[h_i^{(l)} \,\|\, \mathrm{AGG}_{j \in S(i)} h_j^{(l)} \right] \right)$$

where S(i) is the sampled neighborhood and ‖ is concatenation. AGG can be a **mean**, a **pool** (pass each neighbor through a small network, then take an elementwise max), or an **LSTM** over a randomly shuffled neighbor list. GraphSAGE's key property is that it is **inductive**: it learns the *functions* W and AGG, not a fixed embedding per node. So a trained model can embed nodes, or whole graphs, never seen in training, which is exactly what a live system needs. A plain GCN, trained *transductively* on one fixed graph, must be re-run when the graph changes.

Graph Attention Networks (GAT; Veličković et al., 2018) replace the GCN's fixed coefficient $1/\sqrt{\tilde{d}_i\,\tilde{d}_j}$ with a *learned* attention weight α_{ij} over each neighbor. This is the same mechanism from chapter 16, restricted to the graph's edges. Each node scores its neighbors, normalizes the scores with a softmax over its neighborhood, and takes the weighted combination:

$$\alpha_{ij} = \mathrm{softmax}_j\left(\mathrm{LeakyReLU}\left(a^{\top}\left[\,W h_i \,\|\, W h_j\,\right]\right)\right), \quad h_i^{(l+1)} = \sigma\left(\sum_{j\in\tilde{N}(i)} \alpha_{ij}\, W h_j\right)$$

Now the network *decides* which neighbors matter instead of trusting degree. A helpful edge gets a large weight even from a high-degree node, and an irrelevant one is suppressed. As in the transformer, GAT runs several attention **heads** in parallel and concatenates them, so different heads focus on different kinds of neighbor. Attention over a graph is just attention whose candidate set is "the adjacent nodes" rather than "all positions". The graph supplies the mask for free.

From nodes to whole graphs: readout

Node classification reads the final layer's per-node vectors directly. A whole-graph task (is this molecule a drug) needs one vector for the entire graph. We get this from a **readout** (or graph pooling) that collapses all node embeddings with, again, a permutation-invariant operator:

$$h_G = \mathrm{READOUT}_{i\in V}\left(h_i^{(L)}\right) \in \left\{\sum_i h_i,\ \ \mathrm{mean}_i\, h_i,\ \ \max_i h_i\right\}$$

The choice matters. A **sum** keeps size information, since a larger graph gives a larger vector, and is the most expressive option (it can, in principle, count substructures). A **mean** is scale-free but discards how many nodes there are. A **max** reports the single most important feature, useful when one key node decides the label. The pooled h_G then feeds an ordinary MLP classifier. More advanced methods pool *hierarchically*, making the graph coarser in stages, just as a CNN's pooling shrinks an image (chapter 13). But clustering an irregular graph is far harder than halving a grid, and simple global readout remains a strong default.

Oversmoothing, scale, and the limits of expressiveness

Message passing is powerful but limited, in three ways worth understanding before you try a hundred layers.

Oversmoothing. Each GCN layer replaces a node's features with a normalized average of its neighborhood, and averaging smooths. Remove the weights and nonlinearity, and K layers apply $\hat{A}^K$. Because $\hat{A}$ is a symmetric diffusion whose largest eigenvalue is 1 and the rest are smaller in magnitude, $\hat{A}^K$ converges to a projection onto that top eigenvector. This drives every node toward the *same* value (up to a degree factor). Once neighboring nodes look the same, no classifier can separate them and accuracy falls. In practice plain GCNs peak at two to four layers. This is a sharp contrast with the hundred-layer CNNs of chapter 13. The fixes come from other areas: residual and skip connections that preserve a node's own signal, "jumping knowledge" that reads out from several depths, and graph-specific normalization.

Scalability. A full-graph layer touches every edge, and the receptive field grows exponentially with depth. A node's two-hop neighborhood can already be most of a dense graph, so simple training on a billion-edge social network is impossible. Neighbor sampling (GraphSAGE), graph clustering, and subgraph mini-batching are what make web-scale GNNs practical.

Expressiveness and the Weisfeiler–Lehman test. There is an exact limit on what message passing can tell apart. A standard message-passing GNN is at most as powerful as the **1-dimensional Weisfeiler–Lehman** (1-WL) graph-isomorphism heuristic: if 1-WL cannot tell two graphs apart, then *no* such GNN can produce different embeddings for them, no matter how deep or wide (Xu et al., 2019; Morris et al., 2019). The classic failure is a 6-cycle versus two disjoint triangles: every node has degree 2 and an identical local view in both, so aggregation gives identical features and identical graph readouts, even though the graphs are not isomorphic. The **Graph Isomorphism Network** (GIN) reaches that 1-WL limit exactly with a *sum* aggregator and an injective MLP update, while mean and max fall strictly short because they lose the neighbor *counts* that a sum preserves. Knowing this limit tells you when to add positional or structural features to the input, rather than expecting depth alone to help.

Graph networks in code

The GCN layer is a dozen lines of NumPy. The cell below builds the running graph, forms $\tilde{A}$, $\tilde{D}^{-1/2}$ and the normalized $\hat{A}$, then computes one propagation $\hat{A}\,H\,W$ and applies ReLU. Every printed number matches the worked example that follows, and node 3's first feature is clipped from −0.5 to 0 by the ReLU.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.set_printoptions(precision=4, suppress=True)

# 4-node graph: triangle 0-1-2 with a leaf node 3 hanging off node 2
A = np.array([[0, 1, 1, 0],
              [1, 0, 1, 0],
              [1, 1, 0, 1],
              [0, 0, 1, 0]], float)

At   = A + np.eye(4)            # add self-loops:  A~ = A + I
d    = At.sum(1)                # degrees of A~  ->  [3, 3, 4, 2]
Dinv = np.diag(d ** -0.5)       # D~^(-1/2), diagonal
Ahat = Dinv @ At @ Dinv         # symmetric-normalized  A^ = D~^-1/2 A~ D~^-1/2

H = np.array([[ 1.,  0.],       # node features: 4 nodes, 2 features each
              [ 0.,  1.],
              [ 1., -1.],
              [-1.,  0.]])
W = np.array([[ 1.,  1.],       # learnable weights: col 0 = sum, col 1 = difference
              [ 1., -1.]])

Z    = Ahat @ H @ W             # aggregate along graph, then transform
Hnew = np.maximum(0.0, Z)       # ReLU

print("degrees d~       :", d)
print("A^ row 3         :", Ahat[3])          # [0, 0, 0.3536, 0.5]
print("aggregation A^H  :\n", Ahat @ H)        # normalized neighbor sums
print("updated features :\n", Hnew)            # node 3 feature 0 clipped to 0
```

The heart of the layer is the single line `Ahat @ H @ W`: the sparse Â mixes each node with its neighbors, and the dense W transforms the features. First aggregate, then transform. Because Â is fixed by the graph, it is computed once and reused every layer. The next cell stacks two such layers to show the defining property of depth: information travels one hop per layer. Nodes 0 and 3 are two edges apart, so $\hat{A}[3,0] = 0$ but $(\hat{A}^2)[3,0] > 0$. We change *only* node 0's features and watch node 3: after one layer it does not move. After two layers it does.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.random.seed(0)
np.set_printoptions(precision=4, suppress=True)

A  = np.array([[0,1,1,0],[1,0,1,0],[1,1,0,1],[0,0,1,0]], float)
At = A + np.eye(4)
d  = At.sum(1)
Ahat = np.diag(d ** -0.5) @ At @ np.diag(d ** -0.5)

H  = np.array([[1.,0.],[0.,1.],[1.,-1.],[-1.,0.]])
W1 = np.random.randn(2, 2)
W2 = np.random.randn(2, 2)
relu = lambda x: np.maximum(0.0, x)

def gcn2(X):                          # two stacked GCN layers
    H1 = relu(Ahat @ X  @ W1)         # after 1 layer: 1-hop information
    H2 = relu(Ahat @ H1 @ W2)         # after 2 layers: 2-hop information
    return H1, H2

H1, H2   = gcn2(H)
Hp       = H.copy(); Hp[0] += [1.0, 1.0]     # perturb ONLY node 0's
features
H1p, H2p = gcn2(Hp)

print("A^[3,0]      =", round(Ahat[3, 0], 4), "  (nodes 0 and 3: NOT 1-
hop)")
print("(A^@A^)[3,0] =", round((Ahat @ Ahat)[3, 0], 4), "  (they ARE 2-hop)")
print("node 3 change after 1 layer :", np.abs(H1p[3] -
H1[3]).sum().round(4))
print("node 3 change after 2 layers:", np.abs(H2p[3] -
H2[3]).sum().round(4))
```

The first layer leaves node 3 untouched by node 0's change (0.0000). The second lets it through (0.7789), because two layers reach two hops. This is the receptive-field figure, now measured on real numbers. Written for a modern framework, one GCN layer is a tiny `nn.Module` wrapping a linear map and a matrix multiply by the precomputed Â:

PyTorch copy → Colab

```
import torch
import torch.nn as nn
import torch.nn.functional as F

def normalized_adjacency(A):                    # A: dense (n, n) of 0/1
    A_tilde    = A + torch.eye(A.size(0))       # add self-loops:  A~ = A + I
    d          = A_tilde.sum(1)                 # degrees
    d_inv_sqrt = torch.diag(d.pow(-0.5))        # D~^(-1/2)
    return d_inv_sqrt @ A_tilde @ d_inv_sqrt    # A^ = D~^-1/2 A~ D~^-1/2

class GCNLayer(nn.Module):
    """One Kipf-Welling graph-convolution layer (dense adjacency)."""
    def __init__(self, in_dim, out_dim):
        super().__init__()
        self.lin = nn.Linear(in_dim, out_dim, bias=False)   # the shared
W^(l)

    def forward(self, H, A_hat):                # H: (n, in);  A_hat: (n, n)
        return A_hat @ self.lin(H)              # A^ (H W)

class GCN(nn.Module):                           # a 2-layer node classifier
    def __init__(self, in_dim, hidden, n_classes):
        super().__init__()
        self.gc1 = GCNLayer(in_dim, hidden)
        self.gc2 = GCNLayer(hidden, n_classes)

    def forward(self, X, A_hat):
        H = F.relu(self.gc1(X, A_hat))          # hidden node embeddings
        return self.gc2(H, A_hat)               # per-node class logits

n, f = 4, 2
A     = torch.tensor([[0,1,1,0],[1,0,1,0],[1,1,0,1],[0,0,1,0]],
dtype=torch.float)
A_hat = normalized_adjacency(A)                 # precompute once, reuse
every layer
model = GCN(f, 16, 3)
logits = model(torch.randn(n, f), A_hat)        # -> (4, 3): 3-class scores
per node
print(logits.shape)                             # torch.Size([4, 3])
```

In this app's engine

This app does not include a graph lab, yet both halves of a GCN layer are operations you have already run. The aggregation $\hat{A}\,H^{(l)}$ is just a matrix multiply. It is the batched multiply the Playground performs every forward pass (chapter 3), with the graph's $\hat{A}$ in place of a dense weight. A graph-attention layer's neighbor weighting is the same softmax $e^{s} / \Sigma\, e^{s}$ that attention (chapter 16) applies to scores, here normalized over a node's neighbors instead of over classes. What is new is only *where* the multiplications reach.

Worked example — one graph convolution on the 4-node graph

Nodes 0–3 with edges {0–1, 0–2, 1–2, 2–3}: a triangle 0-1-2 plus a leaf 3 on node 2. Features (one row per node) $H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \\ -1 & 0 \end{bmatrix}$, weights $W = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ (column 0 sums the two features, column 1 takes their difference).

1. **Self-loops.** $\tilde{A} = A + I = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$. Row sums give degrees $\tilde{d} = [3, 3, 4, 2]$.
   The leaf node 3, with one real neighbor plus itself, has degree 2.
2. **Inverse square-root degrees.** $\tilde{D}^{-1/2} = \mathrm{diag}(1/\sqrt{3}, 1/\sqrt{3}, 1/\sqrt{4}, 1/\sqrt{2}) = \mathrm{diag}(0.5774, 0.5774, 0.5000, 0.7071)$.
3. **Symmetric-normalized adjacency** $\hat{A}_{ij} = \tilde{A}_{ij} / \sqrt{\tilde{d}_i\,\tilde{d}_j}$. For example $\hat{A}_{00} = 1/3 = 0.3333$, $\hat{A}_{02} = 1/\sqrt{12} = 0.2887$, $\hat{A}_{23} = 1/\sqrt{8} = 0.3536$, $\hat{A}_{33} = 1/2 = 0.5000$.

The full matrix is $\hat{A} = \begin{bmatrix} .333 & .333 & .289 & 0 \\ .333 & .333 & .289 & 0 \\ .289 & .289 & .250 & .354 \\ 0 & 0 & .354 & .500 \end{bmatrix}$.

4. **Aggregate: M = Â H.** Take node 0, whose neighborhood is {0, 1, 2}. Its new aggregated features are the Â-weighted sum of those nodes' features: M_0 = 0.3333·[1, 0] + 0.3333·[0, 1] + 0.2887·[1, −1] = [0.3333 + 0.2887, 0.3333 − 0.2887] = **[0.6220, 0.0447]**. This is the whole point in one line: node 0's update is the normalized blend of its own and its neighbors' features.
5. The leaf, node 3, sees only {2, 3}: M_3 = 0.3536·[1, −1] + 0.5000·[−1, 0] = [0.3536 − 0.5, −0.3536] = **[−0.1464, −0.3536]**.
6. **Transform: Z = M W.** For node 0, $Z_0 = [M_{0,0} + M_{0,1}, M_{0,0} - M_{0,1}]$ = [0.6220 + 0.0447, 0.6220 − 0.0447] = **[0.6667, 0.5774]**. For node 3, Z_3 = [−0.1464 + (−0.3536), −0.1464 − (−0.3536)] = **[−0.5000, 0.2071]**.
7. **Nonlinearity.** ReLU leaves node 0 untouched, [0.6667, 0.5774], but clips node 3's negative first coordinate to zero: $h_3^{(1)}$ = **[0, 0.2071]**.

What the numbers mean: node 0, inside the dense triangle, pools three sources into a moderate positive vector. The leaf node 3, with few neighbors, produces a small aggregate that the transform pushes negative and the ReLU partly silences. Degree normalization is visible in the coefficients: the hub-to-leaf edge (2, 3) carries weight 0.354, lower than the triangle's internal 0.333–0.289 links because node 2 is a busy hub. One layer has already combined every node with its immediate surroundings. A second would reach the whole graph.

Assumptions & limitations

A GNN assumes you actually have the graph, with edges that are known and meaningful. A noisy or wrong adjacency degrades everything downstream. The GCN in particular assumes **homophily**, that connected nodes are similar, since it averages over neighbors. On *heterophilic* graphs, where neighbors are typically *dissimilar* (as in some fraud rings), plain smoothing hurts and specialized architectures are needed. Depth is limited by oversmoothing, so GNNs are shallow by CNN standards and can struggle to pass information across long paths. A bottleneck edge can "over-squash" everything routed through it. Transductive GCNs must be retrained when the graph grows, and expressiveness is limited by the 1-WL test: some truly different structures are, provably, invisible to standard message passing.

Why & when this matters

Use a graph network when the relationships between examples carry as much signal as the examples themselves, and those relationships are irregular: molecules, social and citation networks, transaction ledgers, road systems, knowledge bases. If your "graph" is really a grid or a sequence, the specialized architectures are better: use a CNN for images (chapter 13), an RNN or transformer for text (chapters 15–16). And if the edges add little, a plain MLP on node features may match a GNN at a fraction of the cost. The skill is honestly asking whether structure helps. When it does, message passing is often the difference between an average and a state-of-the-art model.

In the real world

Graph networks are in production wherever data is relational. In **drug discovery**, molecules are graphs of atoms and bonds, and message-passing networks predict properties like toxicity and binding. A GNN screen at MIT identified the new antibiotic

halicin (Stokes et al., 2020). **Recommender systems** treat users and items as a bipartite graph: Pinterest's PinSage runs GraphSAGE over billions of pins to power "more like this," and similar models drive recommendations at Alibaba and UberEats. **Google Maps** improved arrival-time estimates with a DeepMind GNN over road-segment graphs, cutting ETA errors in cities worldwide. **Knowledge graphs** use link prediction to complete missing facts and resolve duplicate entities. And in **fraud and anomaly detection**, banks and marketplaces model transactions as graphs so a GNN can catch rings of colluding accounts that look innocent one node at a time but reveal themselves in their connections.

Exercises

1. For the 3-node path graph 0—1—2 (edges {0–1, 1–2}), build $\tilde{A} = A + I$, list the self-loop degrees $\tilde{d}$, and compute the entries $\hat{A}_{01}$ and $\hat{A}_{11}$ of the symmetric-normalized adjacency. MATH

2. Show that $\hat{A} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$ is symmetric, and explain why its rows do *not* sum to 1, unlike the random-walk matrix $\tilde{D}^{-1}\tilde{A}$. Use the running graph's row sums. MATH

3. Why add self-loops? Describe concretely what would happen to leaf node 3's representation if we propagated with the raw A (no self-loop) instead of $\tilde{A}$. CONCEPT

4. Swap the GCN's symmetric normalization for the random-walk normalization $\tilde{D}^{-1}\tilde{A}$ and compare node 0's aggregated output on the running graph. CODE

5. Demonstrate permutation equivariance empirically: relabel the nodes with a permutation P (swap 0 and 3), rebuild $\hat{A}$ from the permuted adjacency, and check that one GCN layer satisfies $\mathrm{GCN}(PAP^{T}, PX) = P \cdot \mathrm{GCN}(A, X)$. CODE

6. A GAT node i has three neighbors with attention logits e = [2, 1, 0.5] and neighbor vectors $h_1 = [1, 0]$, $h_2 = [0, 1]$, $h_3 = [1, 1]$. Compute the normalized attention weights and the

aggregated output, and relate the computation to chapter 16. MATH

7. Explain why stacking many GCN layers drives node representations together (oversmoothing), using the fact that $\hat{A}$ acts like an averaging operator. CONCEPT

8. State, correctly and briefly, the Weisfeiler–Lehman limit on message-passing GNNs, and give a pair of non-isomorphic graphs a standard GNN cannot tell apart. CONCEPT

Autoencoders & representation learning

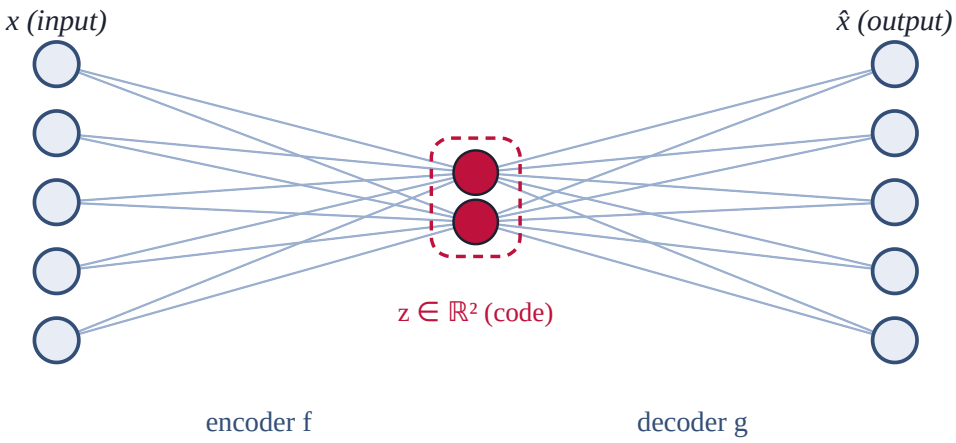
> *An autoencoder learns by copying its input to its output through an obstacle placed on purpose: a narrow middle layer that cannot hold everything. Forced to choose what to keep, the network discovers the hidden structure of the data on its own, with no labels at all. This chapter builds the plain autoencoder, ties it to PCA, works through its regularized versions (denoising, sparse, and contractive), and extends all of them to the variational autoencoder that can generate truly new data. This is the bridge from representation learning (Chapter 5) to the generative models of Chapter 21.*

Fundamental concept

Imagine describing a photograph to a friend over the phone in only ten words. You cannot send every pixel, so you send the essentials: “a beach at sunset, one red umbrella, no people.” Your friend paints the scene back from that summary. If the painting looks like the original, your ten-word **code** kept what mattered. An autoencoder plays exactly this game against itself: one half compresses, the other reconstructs, and the gap between them is the only teacher.

More specifically, an **encoder** f maps the input x to a compact code z, and a **decoder** g maps that code back to an approximation $\hat{x}$, trained so that $\hat{x} \approx x$. Because the target *is* the input, no labels are ever needed. This is **unsupervised** learning, or in the modern framing **self-supervised** learning (Chapter 20). The trick that makes it useful is the **bottleneck**: z is forced to be much smaller than x. A code as wide as the input could copy every number straight through, giving the useless identity map. So the bottleneck forbids this, and the encoder must spend its few numbers on the data's *predictable* regularities rather than noise. These regularities are exactly the structure we wanted.

The code z is therefore a learned, compressed **representation** (the central object of Chapter 5): similar inputs end up at nearby codes, and its axes often align with meaningful factors of variation. This is the structure behind compression, denoising, anomaly detection, pretraining, and generation.



The hourglass: five inputs are squeezed into a two-unit code z (violet), then expanded back to five outputs. The waist is the bottleneck. Only what survives it can be reconstructed.

Governing equations

An autoencoder is two networks trained as one. Written with a single hidden code they are:

$$z = f(x) = s(W_e\, x + b_e)$$

$$\hat{x} = g(z) = s(W_d\, z + b_d)$$

Training minimizes the **reconstruction loss**. This is the squared distance between input and rebuild, averaged over the data:

$$L(x) = \|x - \hat{x}\|^2 = \Sigma_i\, (x_i - \hat{x}_i)^2$$

so the encoder and decoder parameters θ, φ jointly solve

$$\min_{\theta,\varphi} \frac{1}{N} \Sigma_x \|x - g_\varphi(f_\theta(x))\|^2$$

$\mathbf{x}$, $\hat{\mathbf{x}}$	input vector and its reconstruction. Same shape, e.g. $\mathbb{R}^n$ (an image flattened, a row of features)
$\mathbf{z}$	the code or latent vector. This is the bottleneck, dimension $d \ll n$
$\mathbf{f}$, $\mathbf{g}$	encoder and decoder networks (any depth). Here shown with one weight layer each
$\mathbf{W_e}$, $\mathbf{W_d}$	encoder and decoder weight matrices. b_e, b_d are their biases (learned)
$\mathbf{s}$	an activation function such as ReLU or sigmoid. A linear s gives the PCA case below
$\mathbf{L}$	reconstruction error in the units of x^2. Smaller means a more accurate rebuild

The single design constraint that makes this interesting is **undercompleteness**: dim(z) < dim(x). Without it, the pair can choose the identity and learn nothing.

How it works

The bottleneck as a teacher. Real data often lives near a low-dimensional surface. Faces vary in pose, lighting and identity far more than their millions of pixels suggest. A narrow code can only represent points on such a surface, so gradient descent places it where the data actually is. It discards everything perpendicular (sensor noise, irrelevant detail) that would not lower the error.

Undercomplete versus overcomplete. The width of the code defines two cases. An **undercomplete** autoencoder has dim(z) < dim(x): the bottleneck is a hard constraint, and compression alone forces the encoder to keep only structure. An **overcomplete** autoencoder has a code at least as wide as the input, dim(z) ≥ dim(x). With that much room, the pair can copy each coordinate straight through (the identity map again) and learn nothing. So an overcomplete code is useful only when something *else* forbids the shortcut. That something is a **regularizer** (Chapter 12): a loss term that makes trivial solutions too costly. The three variants below (denoising, sparse, and contractive) are exactly such regularizers. They let a wide, expressive code stay informative.

Linear autoencoder = PCA. If f and g are linear (s is the identity), the whole network reduces to a single product,

$$\hat{x} = W_d W_e x$$

and with MSE loss the optimal encoder projects x onto the top principal-component subspace. This recovers what **principal component analysis** finds, with no eigenvectors computed by hand. The benefit of the neural approach appears only when f and g are *nonlinear*: then the code can follow a curved manifold that no flat PCA subspace could. More precisely, the optimal linear encoder spans the same subspace as the top d principal components. These are the d eigenvectors of the data covariance with the largest eigenvalues. The leftover reconstruction error equals the sum of the *discarded* eigenvalues, the variance thrown away in the directions the code has no room to represent (the exercises work this out with numbers).

Denoising autoencoders. A stronger way to forbid the identity map is to break the input on purpose: corrupt each example (with Gaussian noise, or zeroed-out entries) and ask the decoder to restore the *clean* original:

$$\tilde{x} = x + \eta, \quad \text{minimize } \|x - g(f(\tilde{x}))\|^2$$

To undo corruption, the network must learn where clean data lives and project noisy points back onto it. This gives robust features even when the code is not tiny. The corruption can be Gaussian noise added to every input coordinate, or a random mask that deletes a fraction of the entries completely (a close relative of dropout, Chapter 12). Either way, the decoder is scored against the clean original, so the identity shortcut is closed even when the code is wide. A **sparse autoencoder** reaches the same goal without a hard bottleneck: a penalty keeps most code units near zero, so only a few activate per input.

Sparse autoencoders, more precisely. Let $\hat{\rho}_j$ be the average activation of unit j across the training set, and ρ a small target rate such as 0.05. The loss gains a term pulling every $\hat{\rho}_j$ toward ρ. This is commonly the Bernoulli KL divergence $\Sigma_j \, \mathrm{KL}(\rho \,\|\, \hat{\rho}_j)$ (Appendix D, information theory) or an L1 term $\Sigma_j \, |z_j|$. Since only a few units fire per input, each one specializes in a distinct, reusable feature. This old idea is now central to modern **interpretability** (Chapter 28): a large, heavily sparse autoencoder trained on the internal activations of a language model decomposes them into thousands of human-readable features, turning a hard-to-read hidden vector into a short list of the concepts the model represents.

Contractive autoencoders. A third regularizer targets robustness directly: it adds the squared Frobenius norm of the encoder's Jacobian, $\|\partial z / \partial x\|_F^2 = \Sigma_{ij} (\partial z_j / \partial x_i)^2$, penalizing any code coordinate that changes quickly when the input is slightly changed. The encoder is therefore pushed to stay flat, except along the few directions that move *along* the data manifold. Denoising and contractive autoencoders are closely related. Averaging a reconstruction over small input corruptions is, to first order, the same as penalizing this Jacobian. Both give robustness that a plain bottleneck cannot guarantee.

From reconstructing to generating: the VAE. A plain autoencoder is a poor generator: its latent space has holes, so a random z usually decodes to garbage. The **variational autoencoder** (VAE) fixes this with two changes. First, for each input the encoder outputs not a point but a small Gaussian *distribution* over codes. This is a mean μ and a spread σ, so every code now carries its own uncertainty (the probabilistic view developed in Chapter 24). Second, the loss adds a term pulling every one of those Gaussians toward the same standard normal prior N(0, I), packing the codes together with no gaps. Afterwards you can draw $z \sim N(0, I)$, decode, and get a new, plausible sample.

Sampling inside the network creates a problem: you cannot backpropagate through a random draw. The **reparameterization trick** moves the randomness outside. Instead of sampling z directly, we draw a fixed noise ε and shift-and-scale it:

$$(\mu, \sigma) = f(x)$$

$$z = \mu + \sigma \odot \varepsilon, \quad \varepsilon \sim N(0, I)$$

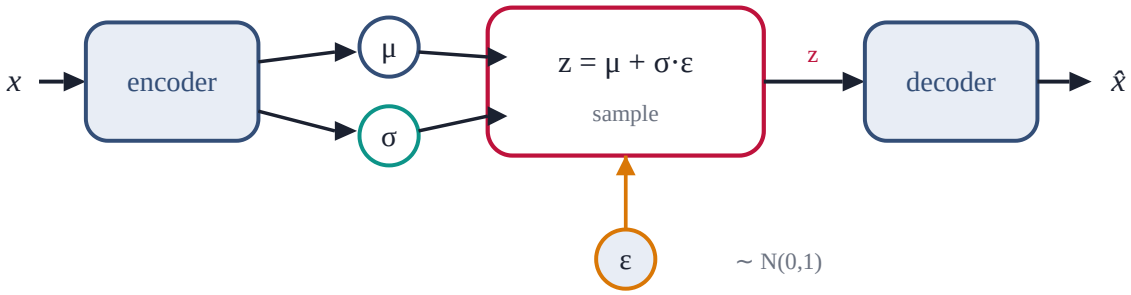
Now μ and σ are deterministic outputs with well-defined gradients. The only random part, ε, has no parameters. The full objective adds a **KL divergence** measuring how far the encoded Gaussian q(z|x) has moved away from the prior:

$$L_{VAE} = \|x - \hat{x}\|^2 + KL(q(z|x) \,\|\, N(0, I))$$

For a diagonal Gaussian the KL is closed-form, summed over the d code units:

$$KL = \frac{1}{2} \Sigma_j (\mu_j^2 + \sigma_j^2 - 1 - \ln \sigma_j^2) = -\frac{1}{2} \Sigma_j (1 + \ln \sigma_j^2 - \mu_j^2 - \sigma_j^2)$$

The KL term is zero exactly when μ = 0 and σ = 1 for every unit. Reconstruction pushes codes apart so they stay distinguishable, while KL pulls them toward the prior. Their balance leaves the latent space smooth enough to sample from (Chapter 21).



The reparameterization trick. Randomness enters only through the external noise ε (amber). μ and σ stay deterministic, so gradients flow back through the encoder during training.

The ELBO — where the loss comes from

The VAE loss is not an arbitrary sum of two terms. It is a lower bound on the one quantity we actually care about: the probability the model assigns to real data, log p(x). We cannot compute log p(x) directly, because it hides an intractable integral over all codes z, so we bound it instead. For any encoder distribution q(z|x) (the probability background is collected in Appendix C), a single application of Jensen's inequality yields the **evidence lower bound** (ELBO):

$$\log p(x) \geq \mathbb{E}_{q(z|x)}[\, \log p(x|z) \,] - \mathrm{KL}(\, q(z|x) \,\|\, p(z) \,)$$

Look at the two pieces. The first term rewards a code from which the decoder can reconstruct x. For a Gaussian decoder, log p(x|z) is exactly $-\|x - \hat{x}\|^2/2\sigma^2$ plus a constant, so *maximizing it is minimizing reconstruction error*. The second term is a regularizer pulling the encoder's Gaussian toward the prior p(z) = N(0, I). Maximizing the ELBO therefore minimizes precisely the loss we wrote before, now with an explicit weight:

$$L_{\mathrm{VAE}} = \|x - \hat{x}\|^2 + \beta \cdot \mathrm{KL}(\, q(z|x) \,\|\, N(0, I) \,)$$

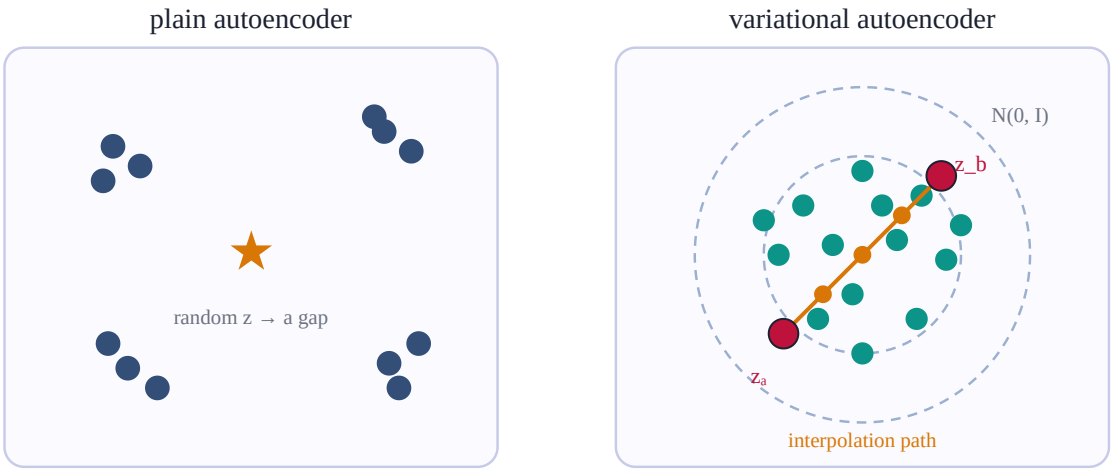
The parameter β balances the two. At $\beta = 0$ the KL becomes zero, and the VAE becomes a plain autoencoder with a scattered latent space you cannot sample from. If you push β too high, the codes are squeezed so hard against the prior that they stop carrying information about x. This failure is called **posterior collapse**, where the decoder learns to ignore z altogether. The **β-VAE** deliberately sets $\beta > 1$ to encourage *disentangled* codes whose individual axes control single interpretable factors (a face's pose, its lighting). The KL itself is closed-form and cheap: one term per latent unit, as the worked example computes below. This is why the whole objective is a single differentiable expression, trainable by ordinary backpropagation.

Sampling and interpolation in latent space

Because the KL term has packed the codes into a smooth ball around the origin, the VAE's latent space supports two operations a plain autoencoder cannot. The first is **sampling**: draw $z \sim N(0, I)$, push it through the decoder, and you get a fresh example the model was never shown. This is the defining action of a generative model (Chapter 21). The second is **interpolation**: take the codes z_a and z_b of two real inputs and move along the straight line between them,

$$z_t = (1 - t)\, z_a + t\, z_b, \quad t \in [0, 1]$$

decoding each z_t along the way. Because nearby codes decode to similar outputs, and the space between the endpoints is populated rather than empty, the reconstructions change *smoothly* from the first input into the second (a face slowly turning, a handwritten 4 bending into a 9) instead of jumping through noise. One warning: a d-dimensional Gaussian keeps almost all its mass in a thin shell at radius $\approx \sqrt{d}$, so for high-dimensional codes we interpolate *along* that sphere (a **spherical** interpolation, or slerp) to keep midpoints at a typical radius. A plain autoencoder fails both operations for one reason: nothing filled the gaps between its codes, so a random or interpolated z lands where the decoder learned nothing. The figure contrasts the two geometries.



Two latent geometries. Left: a plain autoencoder scatters its codes into isolated clusters with empty space between them, so a random draw (amber star) falls where the decoder has learned nothing and rebuilds to garbage. Right: the VAE's KL term packs codes into a smooth cloud filling the standard-normal prior (dashed circles), so every draw decodes to something plausible and the straight interpolation path (amber) between two codes z_a and z_b (violet) passes only through populated regions.

Autoencoders in code

A linear autoencoder is small enough to train from scratch in a few NumPy lines. This one squeezes 4-D data (which secretly lives on a 2-D plane) through a 2-unit code, and watches the reconstruction error fall toward the noise floor:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)

# data that really lives on a 2-D plane inside 4-D space, plus tiny noise
B = np.linalg.qr(rng.randn(4, 2))[0].T   # 2 orthonormal directions
Z = rng.randn(120, 2)
X = Z @ B + 0.05 * rng.randn(120, 4)
X = X - X.mean(axis=0)                    # centre the data

n, d = 4, 2                               # 4 features -> code of size 2
We = rng.randn(n, d) * 0.1                # encoder  x -> z
Wd = rng.randn(d, n) * 0.1                # decoder  z -> x-hat
lr = 0.2
for step in range(401):
```

```
    Zc = X @ We                          # codes  (120, 2)
    Xh = Zc @ Wd                         # reconstruction (120, 4)
    E  = Xh - X                          # residual
    loss = np.mean(np.sum(E**2, axis=1))
    We -= lr * (X.T @ (E @ Wd.T)) * (2 / len(X))
    Wd -= lr * (Zc.T @ E) * (2 / len(X))
    if step % 100 == 0:
        print(f"step {step:3d}   reconstruction error {loss:.4f}")
print("floor set by the injected noise is ~0.005")
```

The error falls from ≈1.95 to ≈0.005 (the residual left by the injected noise) as the two-number code learns the plane the data lives on. The same method does **anomaly detection**: train only on "normal" points, then score any new point by how badly it reconstructs. A point off the learned manifold rebuilds poorly and crosses the threshold:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(1)

# 'normal' beats live near a 1-D line through 3-D feature space
line = np.array([[1.0, 0.8, -0.5]])
Xn = rng.randn(300, 1) @ line + 0.05 * rng.randn(300, 3)

We = rng.randn(3, 1) * 0.1               # bottleneck code of size 1
Wd = rng.randn(1, 3) * 0.1
for _ in range(800):
    Z = Xn @ We; Xh = Z @ Wd; E = Xh - Xn
    We -= 0.1 * (Xn.T @ (E @ Wd.T)) * (2 / len(Xn))
    Wd -= 0.1 * (Z.T @ E) * (2 / len(Xn))

def score(x):                            # reconstruction error = anomaly
score
    return np.sum((x - (x @ We) @ Wd) ** 2, axis=1)

val = rng.randn(100, 1) @ line + 0.05 * rng.randn(100, 3)   # held-out
normal
tau = 5 * score(val).mean()              # threshold = 5x mean normal error
anomaly = np.array([[0.6, 0.5, 0.9]])    # 3rd feature off the line
print(f"mean normal score : {score(val).mean():.4f}")
```

```
print(f"threshold  (5x)   : {tau:.4f}")
print(f"anomaly    score  : {score(anomaly)[0]:.4f}")
print("flagged?          :", bool(score(anomaly)[0] > tau))
```

The anomaly reconstructs hundreds of times worse than a typical normal point, far above the 5× threshold. It is flagged with no labeled anomalies ever seen, exactly the recipe of the heart-monitor example below. In PyTorch, the encoder and decoder become `nn.Module`s. The VAE variant simply has the encoder output a mean and log-variance, and applies the reparameterization trick:

PyTorch copy → Colab

```
import torch, torch.nn as nn
import torch.nn.functional as F

class AE(nn.Module):                          # plain autoencoder
    def __init__(self, n=784, d=32):
        super().__init__()
        self.enc = nn.Sequential(nn.Linear(n, 128), nn.ReLU(),
nn.Linear(128, d))
        self.dec = nn.Sequential(nn.Linear(d, 128), nn.ReLU(),
nn.Linear(128, n))
    def forward(self, x):
        z = self.enc(x)
        return self.dec(z), z

model = AE(); opt = torch.optim.Adam(model.parameters(), 1e-3)
x = torch.rand(64, 784)
xh, z = model(x)
loss = F.mse_loss(xh, x)                      # reconstruction loss
opt.zero_grad(); loss.backward(); opt.step()

class VAE(nn.Module):                          # variational autoencoder
    def __init__(self, n=784, d=32):
        super().__init__()
        self.enc = nn.Linear(n, 2 * d); self.dec = nn.Linear(d, n)
    def forward(self, x):
        mu, logvar = self.enc(x).chunk(2, dim=1)
        z = mu + torch.exp(0.5 * logvar) * torch.randn_like(mu)   #
reparameterize
```

```
        kl = -0.5 * torch.sum(1 + logvar - mu**2 - logvar.exp(),
dim=1).mean()
        return self.dec(z), kl
```

Two changes turn these toy modules into practical ones. The encoder and decoder grow deep, stacking convolutions (for images) and **normalization** layers (Chapter 11) that keep activations well-scaled, as in any deep network. Also, the encoder outputs a log-variance rather than σ directly, so exp(½·logvar) is always positive and the KL stays numerically stable.

In this app's engine

This chapter has no dedicated tool because an autoencoder needs nothing new. It is an ordinary multilayer perceptron whose target is its own input, trained with a squared-error reconstruction loss by backpropagation. You can still see the bottleneck effect in the Playground: load two-spirals, shrink a hidden layer to two units, and watch how much of the spiral survives that squeeze in the hidden activations. This is the kind of two-number code an autoencoder learns to store and rebuild from.

Worked example — anomaly detection on a heart monitor

Train an autoencoder on *normal* heartbeats only. It rebuilds healthy rhythms well. But it has never seen an arrhythmia, so it rebuilds one badly, and that failure is the alarm. Each beat is summarized by four features, and a beat is flagged when its reconstruction error goes above a threshold:

$$\text{anomaly} \iff L(x) > \tau, \quad \tau = k \cdot \text{mean}_{\text{val}}\, L$$

Taking the per-feature errors $|x_i - \hat{x}_i|$:

1. Normal beat, errors [0.1, 0.2, 0.1, 0.15]. Squared: [0.01, 0.04, 0.01, 0.0225]. MSE = (0.01 + 0.04 + 0.01 + 0.0225)/4 = 0.0825/4 = **0.020625**.
2. Arrhythmia, errors [0.9, 1.2, 0.7, 1.0]. Squared: [0.81, 1.44, 0.49, 1.00]. MSE = (0.81 + 1.44 + 0.49 + 1.00)/4 = 3.74/4 = **0.935**.

3. Ratio: 0.935 / 0.020625 = **45×**. The abnormal beat reconstructs 45 times worse than a normal one.
4. Set the alarm threshold at 5× the mean error on held-out normal beats: $\tau = 5 \cdot 0.020625 \approx$ **0.103**. The normal beat ($0.0206 < 0.103$) passes. The arrhythmia ($0.935 \gg 0.103$) is **flagged**.

The number means the beat is far off the manifold of everything the model calls "normal". That is the recipe: fit on clean data, read a threshold off held-out clean data, then flag anything whose error rises above it. No labeled anomalies are required.

Worked example — the KL cost of a small latent

A VAE encoder reports $\mu = 0.5$ and $\sigma = 0.8$ for a single code unit. How far has it moved from the N(0,1) prior? Using $\ln \sigma^2 = 2 \ln \sigma$ and $\ln 0.8 = -0.2231$:

1. $KL = -\tfrac{1}{2}(1 + 2 \ln 0.8 - \mu^2 - \sigma^2) = -\tfrac{1}{2}(1 + 2(-0.2231) - 0.25 - 0.64)$.
2. Inside: $1 - 0.4463 - 0.25 - 0.64 = -0.3363$.
3. $KL = -\tfrac{1}{2} \cdot (-0.3363) =$ **0.168 nats**.

That 0.168 is the penalty this unit pays for sitting off-center ($\mu \neq 0$) and too sharp ($\sigma < 1$). Training pushes it toward $\mu = 0$, $\sigma = 1$, where KL becomes zero, unless reconstruction needs otherwise.

Now take the full latent, a three-unit code with $\mu = [0.5, 0, -1]$ and $\sigma = [0.8, 1, 0.5]$. Because the posterior is diagonal, the total KL is just the per-unit sum $\tfrac{1}{2} \Sigma_j (\mu_j^2 + \sigma_j^2 - 1 - \ln \sigma_j^2)$:

1. Unit 1 ($\mu = 0.5$, $\sigma = 0.8$): the value just computed, **0.1681**.
2. Unit 2 ($\mu = 0$, $\sigma = 1$): $\tfrac{1}{2}(0 + 1 - 1 - \ln 1) = \tfrac{1}{2} \cdot 0 =$ **0**. A unit already exactly at the prior pays nothing.

3. Unit 3 ($\mu = -1$, $\sigma = 0.5$): $\ln \sigma^2 = \ln 0.25 = -1.3863$, so $\tfrac{1}{2}(1 + 0.25 - 1 + 1.3863) = \tfrac{1}{2}(1.6363) = \mathbf{0.8181}$.

4. Total KL = 0.1681 + 0 + 0.8181 = **0.986 nats**.

The third unit dominates: a code that is both shifted ($\mu = -1$) and over-confident ($\sigma = 0.5$, far sharper than the prior) is expensive for both reasons. This single scalar, summed over the batch, *is* the entire regularization term of the VAE.

The same three-unit sum in NumPy reproduces the hand computation to the printed digits:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
mu    = np.array([0.5, 0.0, -1.0])     # per-unit posterior means
sigma = np.array([0.8, 1.0,  0.5])     # per-unit posterior std devs
kl_per_unit = 0.5 * (mu**2 + sigma**2 - 1.0 - np.log(sigma**2))
print("per-unit KL :", kl_per_unit.round(4))          # [0.1681 0.
0.8181]
print("total KL    :", round(float(kl_per_unit.sum()), 4), "nats")   #
0.9863
```

Assumptions & limitations

A plain autoencoder's code is not organized for sampling, so it reconstructs but cannot reliably generate. This is the reason the VAE exists. VAEs in turn produce *blurry* samples. This is the fault of the pixel-wise MSE loss: when several outputs are equally plausible, squared error returns their smeared average (Chapter 21's GANs and diffusion address exactly this). Anomaly detection assumes the “normal” training data is truly clean. A rare but harmless input also reconstructs poorly and raises a false alarm. And a contaminated training set quietly teaches the model to accept the very anomalies you are trying to find. And undercompleteness is what matters: if the code is too wide or the decoder too powerful, the pair can approximate the identity and learn nothing. This is why denoising or sparsity is added as a safeguard.

Why & when this matters

Use an autoencoder when you have plenty of unlabeled data and want structure from it: unsupervised **pretraining** and **representation learning** (Chapter 5), self-supervised objectives (Chapter 20), nonlinear **dimensionality reduction** on curved manifolds PCA cannot follow, lossy **compression** and **denoising**, and **anomaly detection** (the subject of the case study in Chapter 34) in fraud, medicine, and industrial sensors where anomalies are too rare or varied to label. The VAE is the conceptual link to Chapter 21, where a network stops rebuilding what it is shown and starts inventing plausible data it has never seen. Its probabilistic latent code is a first look at the uncertainty modeling of Chapter 24.

In the real world

Autoencoders run quietly inside many production systems. Credit-card networks and cloud-security platforms train them on normal transactions or normal server telemetry and raise an alert when reconstruction error spikes. This is unsupervised fraud and intrusion detection that needs no catalog of every possible attack. Variational autoencoders are used in molecular design. Firms encode known drug-like molecules into a smooth latent space, then decode new points in it to propose candidate compounds. And the VAE's core idea, a learned, compressed latent space, reappears at the heart of Stable Diffusion (Chapter 21), which runs its entire diffusion process inside a pretrained autoencoder's latent space rather than on raw pixels, cutting the computation by roughly an order of magnitude.

Exercises

1. An autoencoder maps $x = [1, 0, 1, 0]$ to reconstruction $\hat{x} = [0.9, 0.1, 0.8, 0.2]$. Compute the total and the mean squared reconstruction error. MATH

2. For a single VAE latent unit compute the KL to the N(0,1) prior for ($\mu = 1, \sigma = 1$) and for ($\mu = 0, \sigma = 2$), using $KL = \tfrac{1}{2}(\mu^2 + \sigma^2 - 1 - \ln \sigma^2)$. Interpret which drift costs more. MATH

3. Shrink the linear autoencoder's code from d = 2 to d = 1 and rerun on the same 2-D-plane data. Predict and then verify what happens to the final reconstruction error. CODE

4. Why can a plain autoencoder not be used as a generator by simply feeding a random z into its decoder, and what specifically does the VAE change to fix that? CONCEPT

5. A denoising autoencoder can have a code as wide as its input yet still not collapse to the identity. Why does corrupting the input prevent that trivial solution? CONCEPT

6. Given a batch of six new points with reconstruction errors [0.02, 0.03, 0.015, 0.50, 0.025, 0.90], flag anomalies using a threshold of 3× the median error. Which points are flagged? CODE

7. A linear autoencoder recovers the same subspace as PCA. If the data covariance has eigenvalues [5, 3, 1, 0.5], what is the minimal mean reconstruction error a code of size 2 can achieve, and why? MATH

8. Verify the reparameterization trick empirically. For $\mu = [0.5, -1, 2]$ and $\sigma = [0.8, 1, 0.5]$, draw many samples $z = \mu + \sigma \odot \varepsilon$ with $\varepsilon \sim N(0, I)$ and check that their mean and standard deviation match μ and σ. Why is this construction differentiable in μ and σ when sampling $z \sim N(\mu, \sigma^2)$ directly is not? CODE

9. An **overcomplete** autoencoder has a code at least as wide as its input. Explain why, unregularized, it can drive the reconstruction error to zero while learning nothing useful, and name three regularizers that rescue it. CONCEPT

10. Show that for a VAE with a Gaussian decoder $p(x|z) = N(\hat{x}, \sigma^2 I)$, maximizing the ELBO's reconstruction term $\mathbb{E}_q[\log p(x|z)]$ is the same as minimizing squared reconstruction error. MATH

Self-supervised & contrastive learning

> *Every chapter so far has assumed that a teacher stands beside each input x with the answer y. But labels are the most expensive thing in machine learning. You may need a radiologist to circle a tumor, or a crowd worker to click "cat" a million times. Meanwhile, the raw data itself is nearly free and nearly infinite.* ***Self-supervised learning*** *asks the obvious question: can the data label itself? Hide part of an example and predict it from the rest. Call two changed copies of one photo "the same" and every other photo "different." The task is invented, but the representation it forces the network to build is real. Once it is pretrained on a huge amount of unlabeled data, it transfers to task after task. This is where the autoencoder's label-free idea (Chapter 19) grows into the engine behind BERT, SimCLR, and the modern foundation model.*

Fundamental concept

Cover the last word of a sentence and a fluent reader can usually guess it: "the cat sat on the ___." Nobody had to *label* that blank. The text supplied both the question and its own answer the moment we hid a word. That is the whole trick of self-supervision. We take unlabeled data x, apply a known transformation that splits it into an artificial input and an artificial target, and then train an ordinary supervised network on the pair. The label was made from the data itself, so it costs nothing and there is an unlimited supply.

This invented task is called a **pretext task**, and here is the key point: *we do not care about the pretext task*. Guessing the missing word, or the rotation angle of an image, is only a tool, not the goal. What we keep is the **encoder**. This is the network's internal map from input to a representation h. To solve the pretext well, the encoder had to discover the real structure of the data: grammar, objects, textures, and the factors of variation that the autoencoder of Chapter 19 also looked for. Once pretrained, that encoder is frozen or lightly fine-tuned for a real

downstream task with only a few labels. Pretrain on the large pool of free data, then transfer to the small set of labeled data. This pretrain-then-transfer recipe (Chapter 22) is why self-supervision reshaped the field.

An autoencoder is in fact the simplest self-supervised model. Its pretext task is "reconstruct your own input," and the code z is the representation we keep. This chapter generalizes that idea in the two directions that are most common in practice: **contrastive learning**, which never reconstructs but only asks the encoder to tell examples apart, and **masked modeling**, which hides part of the input and predicts it. Both invent a supervised signal from raw data. They differ only in what signal they invent.

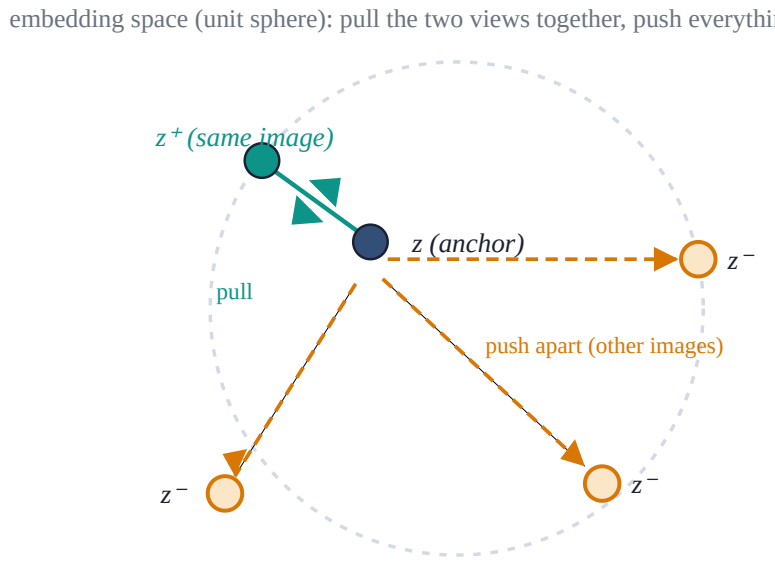
Pretext tasks: manufacturing a label

Before contrastive learning became popular, researchers hand-designed many pretext tasks, each a small puzzle whose answer can be computed from the raw input. Their variety is worth seeing, because it shows the one design rule that matters.

Predict rotation. Rotate an image by 0°, 90°, 180° or 270° and name the angle. This is a free four-way classification whose answer requires knowing which way is up (skies, faces, horizons). **Colorization.** Convert a photo to grayscale and paint the colors back. The target is the original, and coloring grass green or skin plausibly means recognizing what each region *is*. **Jigsaw.** Cut an image into a 3×3 grid, shuffle the tiles using one of a fixed set of permutations, and predict which one was used. This requires understanding how the parts fit together. **Context / inpainting.** Cut a hole and fill it from the surrounding context, a spatial cloze test for what belongs where. **Masked prediction.** Hide a fraction of the tokens or patches and predict them from the visible part. This is the cloze test made central, and it turned out to scale better than all the rest.

The rule these tasks teach, often through failure, is that *the shortcut must be harder than the concept*. If a simple cue solves the puzzle, the encoder learns the cue and nothing else. For example, early patch-position models were found to use chromatic aberration at the lens edges

instead of understanding the content. A good pretext task is one whose only easy solution is real understanding. Contrastive learning and masked modeling pass this test at scale, which is why they dominate the rest of the chapter.



The contrastive objective as geometry. Every embedding lives on the unit sphere (only direction matters, because vectors are L2-normalized). Two augmented views of one image (the anchor z and its positive z^+) are pulled together. The embeddings of all other images (the negatives z^-) are pushed away. Training arranges the sphere so each image has its own small neighborhood.

Contrastive learning: the InfoNCE loss

Contrastive learning drops reconstruction completely and builds its labels by **augmentation**: apply two independent random transformations (crop, color jitter, blur) to one example and call the two results a **positive pair**. They must map to nearby embeddings, because they are the same thing seen twice. Every other example in the batch is a **negative**, to be pushed away. The encoder thus learns the one invariance we asked for, while keeping examples different enough to tell apart. This is **instance discrimination**: treating every image as its own class.

The loss that expresses this is **InfoNCE**, also called **NT-Xent** (the normalized temperature-scaled cross-entropy). For a positive pair (i, j) it is

$$L_{i,j} = -\log \frac{\exp(\operatorname{sim}(z_i, z_j) / \tau)}{\Sigma_{k \neq i} \exp(\operatorname{sim}(z_i, z_k) / \tau)}$$

where the similarity is the **cosine** of the angle between two embeddings, which after L2-normalization is just their dot product:

$$\text{sim}(u, v) = \frac{u^T v}{\|u\| \cdot \|v\|} = u^T v \quad \text{(unit vectors)}$$

z_i, z_j	embeddings of the two views of the same example (the positive pair we want to keep close)
z_k	every other embedding in the batch ($k \neq i$). In the sum, the terms that are not the positive are the negatives
$\text{sim}(u, v)$	cosine similarity, in the range $[-1, 1]$. It equals $u^T v$ once both vectors are normalized to unit length
τ	the **temperature**, a positive scalar (typically 0.05–0.5) that sharpens (small τ) or softens (large τ) the distribution
$L_{i,j}$	the loss for one anchor i with positive j. The full loss averages this over all 2N views in a batch of N pairs

Look at the fraction and you will see it is a **softmax**. The numerator rewards a high similarity between the two views. The denominator sums the anchor's similarity to everything else. Minimizing −log of that ratio drives the positive term up and the negative terms down. This is exactly the pull-and-push of the figure. In fact the expression is simply the cross-entropy of a classifier whose job is to pick the one positive out of all the candidates, with the scaled similarities acting as the logits. Contrastive learning is a huge classification problem that the data labels for free.

Temperature, cosine, and the projection head

Three design choices in this loss each have a good reason. Take the simplest possible case first: one anchor, one positive, and one negative. This case shows how it works. With a single negative, the loss reduces to a clean closed form:

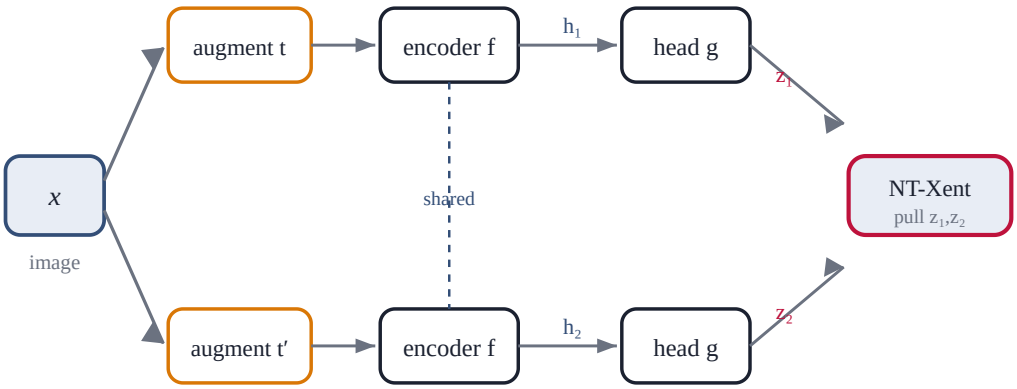
$$L = -\log \frac{e^{s^+}}{e^{s^+} + e^{s^-}} = \log\left(1 + e^{-(s^+ - s^-)}\right), \quad s^+ = \text{sim}_{pos}/\tau, \; s^- = \text{sim}_{neg}/\tau$$

This is the softplus of the negated **gap** ($s^+ - s^-$). It is the same logistic loss from Chapter 6. The loss depends only on how much the positive similarity exceeds the negative one, measured in units of τ. Raise the positive similarity or lower the negative one, and the gap widens and the loss falls. The two moves have the same effect. The worked example below turns this into numbers.

Why temperature. The gap is the raw similarity difference *divided by* τ, so τ acts like a magnifying glass on similarity differences. A small τ (say 0.1) turns a small 0.2 similarity gap into a logit gap of 2.0. This sharpens the softmax so it focuses on the hardest, closest negatives, the ones most likely to be confused with the anchor. If τ is too small, training becomes unstable and over-sensitive to noise. If τ is too large, the loss barely distinguishes near from far, and the representation stays fuzzy. It is the same softmax-sharpness control you saw when scaling attention scores by $\sqrt{d_k}$ (Chapter 16), used here on purpose.

Why cosine. Normalizing every embedding to unit length puts them all on a sphere, so only *direction* carries meaning, and the raw magnitude of an activation cannot cheat the loss. It also bounds every similarity in [−1, 1]. This keeps the logits in a predictable range that τ can then rescale sensibly. This would be impossible if dot products could grow without limit.

Why a projection head. The contrastive loss is not applied to the representation h that we ultimately want. Instead, a small MLP called the **projection head** g maps h to a lower-dimensional z. The loss is computed on z, and downstream we throw g away and keep h. The reason is subtle. The loss demands invariance to the augmentations, but color jitter and cropping destroy information (such as color and position) that a downstream task may need. If the loss acted directly on h, it would remove that information from the representation we keep. The projection head acts as a sacrificial buffer: it becomes invariant in z, while h, one layer upstream, keeps what the loss wanted to remove. In practice, a linear classifier on h outperforms one on z by ten points or more. That is a large gap for a component we discard.



The SimCLR pipeline. One image is augmented two ways. A shared encoder f maps each view to a representation h. A projection head g maps h to z. The NT-Xent loss pulls the two z's together against the batch's negatives. Only h survives to downstream tasks, and g is thrown away after pretraining.

SimCLR and MoCo

SimCLR is the pipeline of the figure, reduced to its essentials: augment, encode, project, apply NT-Xent. Two lessons stand out. First, **augmentation strength** is very important. Random cropping combined with color distortion matters more than the architecture, because a positive pair is only as hard as the transformation that relates it. Second, it needs **many negatives**, drawn simply from the rest of the mini-batch, so more negatives means a bigger batch. The strongest results used batches of 4096, which is roughly 8190 negatives per anchor, and this demands serious hardware. That dependence on batch size is SimCLR's main practical drawback.

MoCo (Momentum Contrast) removes it with two ideas. Instead of drawing negatives from the current batch, it keeps a **queue**, which is a rolling memory bank of embeddings from recent batches. This makes tens of thousands of negatives available regardless of batch size. But those queued embeddings came from older versions of the encoder, and if the encoder changes too fast, they become out of date. MoCo's fix is a **momentum encoder**: the keys come from a slowly-moving copy of the network whose weights are an exponential moving average of the trained encoder,

$$\theta_k \leftarrow m \cdot \theta_k + (1 - m) \cdot \theta_q, \quad m \approx 0.999$$

so the key encoder changes slowly enough to keep the whole queue mutually comparable. The result is SimCLR-quality representations with ordinary batch sizes. This separates the number of negatives from the memory budget. Both methods, though, still rely on negatives, and collecting good negatives is tricky. The next family asks whether they are needed at all.

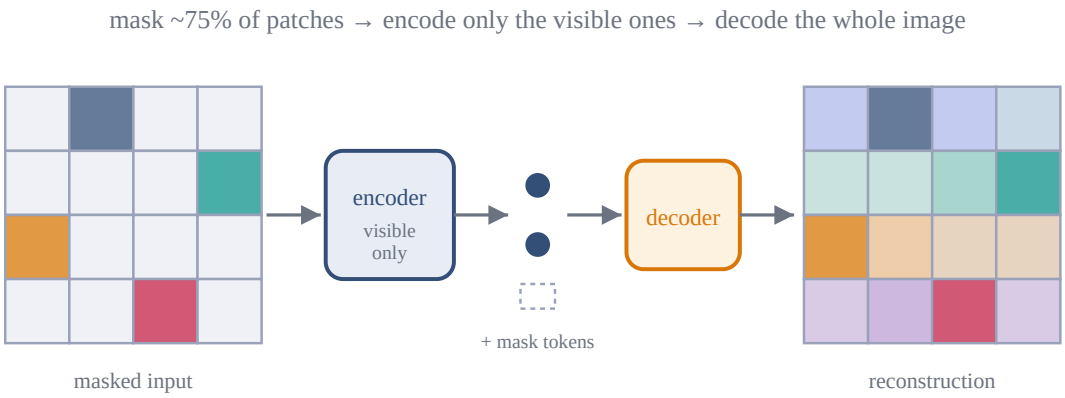
Learning without negatives: the collapse problem

Drop the negatives and a serious problem appears immediately. If the loss only pulls positive pairs together, the encoder can succeed by mapping *every* input to the same constant vector. Then all positives have cosine similarity 1 and the loss is zero. This degenerate solution is called **representational collapse**, and the main challenge of non-contrastive learning is preventing it without a repulsive term.

BYOL (Bootstrap Your Own Latent) surprised the field by doing exactly this without collapsing. It runs an **online** network and a **target** network whose weights are a momentum average of the online one. The online branch passes one view through an extra **predictor**. It is trained to predict the target's embedding of the *other* view, with a **stop-gradient** so no gradient flows into the target. This asymmetry is what prevents collapse. **SimSiam** then showed that the momentum target is not even necessary: two views go through the *same* encoder, with a predictor on one branch and a stop-gradient on the other. The stop-gradient is the essential part: remove it and SimSiam collapses within a few steps.

A second approach shapes the **statistics of the features** rather than the architecture. **Barlow Twins** pushes the cross-correlation matrix between the two views' features toward the identity matrix: ones on the diagonal (each feature agrees across views, which gives invariance) and zeros off the diagonal (the features are decorrelated, which removes redundancy). A constant embedding cannot make the diagonal equal to one, so collapse is impossible by construction. **VICReg** makes the forces explicit as Variance, Invariance, and Covariance terms. Its **variance** term forces each feature's standard deviation across the batch to stay above a minimum value. This is the direct tool against collapse, since a collapsed feature has zero variance. Finally, **DINO** treats self-supervision as **self-distillation**: a student matches the output distribution of a momentum-averaged teacher over two views, and the teacher's outputs are **centered and**

sharpened to prevent collapse. On Vision Transformers (Chapter 16), DINO's attention maps segment objects even though no segmentation labels were ever provided. This is a sign that the representation captured real structure.



A masked autoencoder (MAE). Roughly three-quarters of the image patches are hidden. The encoder, which is the part we keep, processes only the visible quarter, and this makes it cheap to train. Lightweight mask tokens are inserted and a small decoder rebuilds the missing pixels. The high masking ratio is what makes the task hard enough to force real understanding.

Masked modeling: BERT and MAE

The other main family predicts rather than contrasts, and it grew from the cloze test we started with. **BERT** pretrains a bidirectional transformer (Chapter 16) on **masked-language-modeling**: about 15% of the tokens in a sentence are replaced by a [MASK] symbol and the network must recover each original token from the words on both sides. Because it sees left and right context at once, BERT learns representations far richer than those of a left-to-right language model. And because the label is just the hidden word, the entire internet becomes training data. Fine-tuned with a small labeled set, one BERT encoder went on to lead question-answering, sentiment, and inference benchmarks at the same time. It is the template for the pretrain-then-transfer workflow of language models (Chapter 17).

The **Masked Autoencoder** (MAE) brings the same idea to vision. An image is cut into patches, and a *large* fraction (around **75%**, far more than BERT's 15%) is masked. The high ratio is on purpose. Images are locally redundant, so hiding a few patches leaves the answer easy to guess

by interpolation. Hiding most of them forces the network to reason about whole objects. MAE is also **asymmetric**. The encoder we keep sees only the visible 25% of patches, so it is small and fast to train. Shared **mask tokens** and a lightweight decoder reconstruct the missing pixels under an MSE loss, and are then discarded. Seen this way, MAE is exactly the denoising autoencoder of Chapter 19, with masking as the corruption and transformer patches as the base. It combines self-supervision and representation learning in one architecture.

The two families have different trade-offs. Contrastive methods give embeddings whose geometry is immediately useful (nearest neighbors already work), but they rely on heavy, carefully-tuned augmentation and on negatives. Masked modeling needs almost no augmentation and scales easily to enormous models, but its raw features often need fine-tuning before they work well. Both, at heart, use the same trick: create a supervised target from the data and keep the encoder.

Evaluating a representation

Since the pretext task is not the goal, its own accuracy tells us little. The representation is judged downstream, in three standard ways along a spectrum of how much the encoder may move. The **linear probe** freezes the encoder and trains only a single linear classifier on its features. High accuracy means the representation is *linearly separable*: it already exposes concepts as directions in space. This makes the linear probe the strictest and most common self-supervised benchmark, because it isolates the frozen features from any further learning. The **kNN** evaluation trains nothing at all: it labels a point by a majority vote of its nearest neighbors in embedding space. This is a cheap, classifier-free test of whether similar inputs really land nearby. **Fine-tuning** sits at the other end: you unfreeze the encoder and train end-to-end. It gives the best accuracy, at the cost of more labels and compute. It measures not the frozen features but how good an *initialization* the weights provide, which is the transfer-learning question of Chapter 22. A good model wins under all three: pretrain once on unlabeled data, and then a probe, a few neighbors, or a light fine-tune solves many labeled tasks.

Self-supervision in code

The NT-Xent loss is short in NumPy. This cell builds a small batch of N = 4 examples, makes two augmented views of each (base vector plus noise), L2-normalizes all 2N embeddings, forms the scaled cosine-similarity matrix, and computes the InfoNCE loss as a pick-the-positive cross-entropy. Press **Run**:

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)
np.set_printoptions(precision=3, suppress=True)

N, d, tau = 4, 8, 0.5              # 4 examples, 2 views each -> 2N = 8 embeddings
base = np.random.randn(N, d)       # one hidden factor per example
v1 = base + 0.4 * np.random.randn(N, d)    # view 1 = example + augmentation noise
v2 = base + 0.4 * np.random.randn(N, d)    # view 2 (a different augmentation)
Z  = np.vstack([v1, v2])           # (2N, d); row i is paired with row i+N
Z  = Z / np.linalg.norm(Z, axis=1, keepdims=True)   # L2-normalize -> cosine

S = (Z @ Z.T) / tau                # (2N, 2N) scaled cosine-similarity matrix
np.fill_diagonal(S, -1e9)          # an example is never its own positive

pos = (np.arange(2*N) + N) % (2*N)         # positive of row i is its other view
S  -= S.max(axis=1, keepdims=True)         # softmax stabilization (subtract row max)
P   = np.exp(S) / np.exp(S).sum(axis=1, keepdims=True)
loss = -np.log(P[np.arange(2*N), pos]).mean()      # InfoNCE = pick-the-positive CE

print("mean positive cos-sim:", round(float((Z[:N] * Z[N:]).sum(1).mean()), 3))
print("NT-Xent loss         :", round(float(loss), 4))   # expected: 0.9685
```

The positive views are at cosine ≈ 0.86 while the loss is ≈ 0.97. It is not zero, because six negatives still surround each anchor. The second cell makes the pull-together mechanism concrete with *no gradients at all*: it computes the loss, then moves a single positive pair 30% toward their shared mean (exactly the move the gradient would make), and recomputes. The loss drops, confirming that increasing a positive's similarity lowers InfoNCE:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.random.seed(1)

def nt_xent(Z, N, tau=0.5):                # Z is (2N, d); row i pairs with i+N
    Z = Z / np.linalg.norm(Z, axis=1, keepdims=True)
    S = (Z @ Z.T) / tau
    np.fill_diagonal(S, -1e9)
    S -= S.max(axis=1, keepdims=True)
    P = np.exp(S) / np.exp(S).sum(axis=1, keepdims=True)
    pos = (np.arange(2*N) + N) % (2*N)
    return -np.log(P[np.arange(2*N), pos]).mean()

N, d = 4, 8
Z = np.random.randn(2*N, d)
before = nt_xent(Z, N)

# gradient-free nudge: pull ONE positive pair (rows 0 and 4) 30% toward their mean
mid = 0.5 * (Z[0] + Z[4])
Z[0] = 0.7 * Z[0] + 0.3 * mid
Z[4] = 0.7 * Z[4] + 0.3 * mid
after = nt_xent(Z, N)

print("loss before nudge:", round(float(before), 4))   # expected: 2.4591
print("loss after  nudge:", round(float(after), 4))    # expected: 2.2902
print("loss dropped?    :", bool(after < before))       # expected: True
```

In real code the loss is exactly the cross-entropy of a classifier that must find each view's partner among all others. It is one line once the similarities are logits. Here it is in PyTorch, the form used inside every SimCLR training step (here f is the encoder and g is the projection head, and z1, z2 are g(f(view1)) and g(f(view2)) for a batch):

PyTorch copy → Colab

```
import torch
import torch.nn as nn
import torch.nn.functional as F

def nt_xent(z1, z2, tau=0.5):
    # z1, z2: (N, d) projection-head outputs for the two augmented views
    N = z1.size(0)
    z = F.normalize(torch.cat([z1, z2], dim=0), dim=1)   # (2N, d) unit vectors
    sim = (z @ z.t()) / tau                               # (2N, 2N) cosine / temperature
    sim.fill_diagonal_(float('-inf'))                     # forbid an example matching itself
    targets = (torch.arange(2 * N) + N) % (2 * N)         # positive of i is i+N (mod 2N)
    return F.cross_entropy(sim, targets)                  # InfoNCE = CE over candidates

z1 = torch.randn(256, 128)          # g(f(view1)) for a batch of 256 images
z2 = torch.randn(256, 128)          # g(f(view2))
loss = nt_xent(z1, z2, tau=0.5)
print(loss.item())                  # a single scalar; ~5.5 at random init, falls during training

# one optimizer step would then do:  loss.backward(); opt.step()
```

In this app's engine

This chapter has no dedicated lab, but its core computation is one you can already see. The InfoNCE loss is a softmax over similarity scores. It is the same normalization $e^{s_i}/\Sigma_k\, e^{s_k}$ that the Playground's softmax output uses to turn class logits into probabilities, and that attention (Chapter 16) uses to turn scores into weights. Open the Math Log on a multi-class run, and you are reading the operation that here pulls positive pairs together. It produces exactly the learned embeddings of Chapter 5, now shaped by a label the data invented for itself.

Worked example — one InfoNCE loss, and why the positive and negative matter

Take three unit embeddings: an anchor z_i, its positive z_j, and one negative z_k. Suppose the cosine similarities are $\text{sim}(z_i, z_j) = 0.5$ (the positive) and $\text{sim}(z_i, z_k) = 0.4$ (a deliberately *hard*, nearby negative). Use temperature $\tau = 0.1$. With a single negative the denominator has just the positive and negative terms.

1. Scale by temperature: $s^+ = 0.5 / 0.1 =$ **5.0**, $s^- = 0.4 / 0.1 =$ **4.0**.
2. Exponentiate: $e^{5.0} = 148.413$, $e^{4.0} = 54.598$.
3. Softmax on the positive: $P = 148.413 / (148.413 + 54.598) = 148.413 / 203.011 =$ **0.7311**.
4. Loss: $L = -\ln(0.7311) =$ **0.3133** nats. (Equivalently $L = \ln(1 + e^{-(s^+ - s^-)}) = \ln(1 + e^{-1.0}) = 0.3133$, which is the softplus of the gap.)
5. **Raise the positive similarity** to 0.7, keeping the negative at 0.4. Now $s^+ = 7.0$, the gap is 3.0, and $L = \ln(1 + e^{-3.0}) =$ **0.0486**. The loss fell sixfold.
6. **Instead lower the negative similarity** to 0.2, keeping the positive at 0.5. Again the gap is 3.0, and $L = \ln(1 + e^{-3.0}) =$ **0.0486**, which is the same.

What the numbers mean: with one negative the InfoNCE loss depends only on the **gap** $(\text{sim}_{pos} - \text{sim}_{neg})/\tau$, so pulling the positive closer and pushing the negative away are worth exactly the same per unit of similarity. This is the figure's pull and push made quantitative. The hard negative at 0.4 kept the initial loss at a high value of 0.31. An easy negative at −0.5 would make the gap 10 and the loss essentially zero, which is why *hard* negatives carry almost all the learning signal.

Assumptions & limitations

Contrastive learning relies on augmentations that preserve identity while changing appearance, and those are **domain-specific**: the crop-and-color recipe tuned for natural images is wrong for medical scans, audio, or tabular data, and choosing it badly builds in the wrong invariances. For example, throw away color and the model becomes color-blind permanently. It also relies on negatives, and a positive pair can be accidentally paired against a true semantic match (a **false negative**: two different photos of the same breed of dog), which the loss wrongly pushes apart. Non-contrastive methods avoid negatives but gain a new weakness: remove the stop-gradient or the variance term, and they collapse silently to a constant. Masked modeling avoids most of this, but its raw features often lag behind contrastive ones until fine-tuned. And in all cases, evaluation is subtle: a high linear-probe score does not guarantee good fine-tuning behavior, and the pretext task's own loss is nearly meaningless as a progress signal. Self-supervision reduces the need for labels. It does not remove the need for careful, labeled *evaluation*.

Why & when this matters

Use self-supervision whenever unlabeled data is plentiful and labels are scarce or costly, which is almost always. Pretrain an encoder on the large pool of raw data, then linear-probe, kNN, or fine-tune it on your small labeled set (Chapter 22). This routinely matches a fully-supervised model with a fraction of the labels. It is the default first step in modern vision and language pipelines. Prefer masked modeling when you can afford large models and want maximum scalability. Prefer contrastive or non-contrastive methods when you need strong features from a frozen encoder and can design trustworthy augmentations.

In the real world

Self-supervised pretraining is behind the systems people actually use. Every large language model (GPT, Claude, Llama) is pretrained by next-token or masked prediction on web-scale text before any human feedback. That stage is where nearly all of their knowledge is formed. In vision, OpenAI's CLIP contrasts images against their captions to learn a joint embedding that powers zero-shot recognition and text-to-image models (Chapter 21), while Meta's DINOv2 and MAE-pretrained backbones are provided as general-purpose feature extractors. Speech systems such as wav2vec 2.0 pretrain contrastively on unlabeled audio, then fine-tune to state-of-the-art transcription with a fraction of the labeled hours once required. And in biology and medicine, where every label is an expensive expert judgment, encoders trained on unlabeled scans, cells, and protein sequences have become the standard foundation to fine-tune from.

Exercises

1. An anchor has cosine similarity 0.8 to its positive and 0.3 to its single negative. With temperature $\tau = 0.2$, compute the InfoNCE loss. MATH

2. Keep $\text{sim}_{\text{pos}} = 0.8$ and $\text{sim}_{\text{neg}} = 0.6$. Compute the softmax weight the loss places on the positive at $\tau = 0.5$ versus $\tau = 0.1$, and explain what temperature does. MATH

3. Reproduce the worked example in NumPy: write a function for the InfoNCE loss of one positive against a list of negatives at $\tau = 0.1$, and confirm the base loss and both improvements. CODE

4. Why is the contrastive loss applied to the projection z rather than directly to the representation h, given that we discard the projection head afterwards? CONCEPT

5. In a non-contrastive method with positives only, what is the trivial "collapsed" solution, and precisely what prevents it in SimSiam? CONCEPT

6. MAE masks about 75% of image patches while BERT masks about 15% of text tokens. Why does vision need a far higher masking ratio? CONCEPT

7. Show that for L2-normalized embeddings, maximizing cosine similarity is the same as minimizing squared Euclidean distance. What does this say about the relationship between contrastive learning and metric learning? MATH

8. You have 1,000,000 unlabeled images and 2,000 labeled ones. Sketch a self-supervised pipeline, and name the three ways you would evaluate the learned representation. CONCEPT

Generative models: GANs to diffusion

> *A discriminative model answers a question about data it is shown. A generative model does something stranger. It creates data that never existed but plausibly could have. This chapter covers the two main methods behind modern image, audio and video synthesis: the adversarial game of GANs, and the noise-then-denoise process of diffusion that now powers systems such as Stable Diffusion and DALL·E.*

Fundamental concept

Everything so far in this manual has been **discriminative**: a classifier learns the conditional p(y|x), the boundary that separates cats from dogs. A **generative** model aims higher. It learns the data distribution p(x) itself, well enough to draw a brand-new sample from it. The difference is the gap between an art critic who can spot a forgery and an artist who can paint a convincing new canvas. Capturing a whole distribution is far harder than drawing one boundary. This is why generation had to wait for deep networks and large datasets.

$$\text{discriminative: } p(y \mid x) \qquad \text{generative: } p(x) \ \text{(then sample new } x\text{)}$$

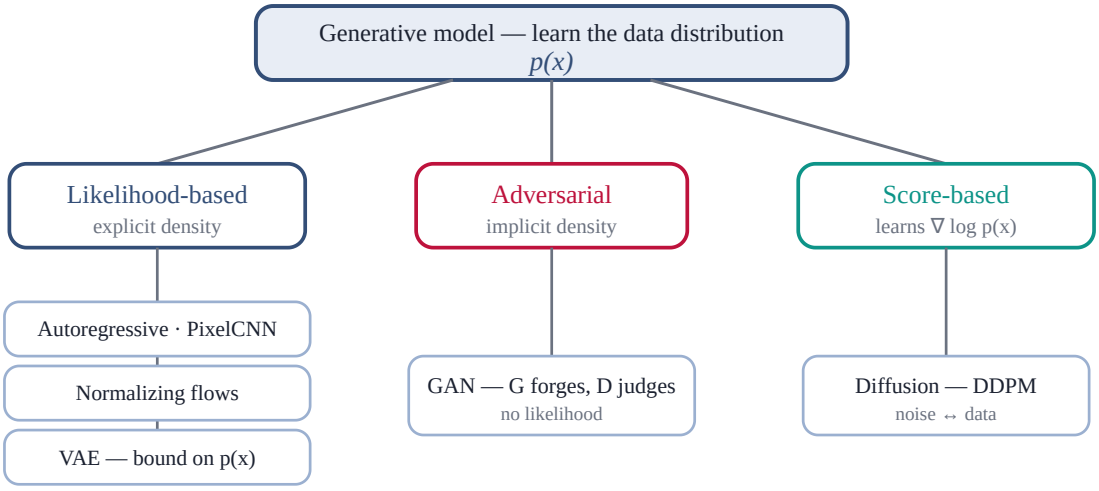
Two ideas dominate. A **generative adversarial network** (GAN) sets two networks against each other. A generator makes fake samples, and a discriminator looks for fakes. The contest between them drives realism. A **diffusion model** takes the opposite approach: it slowly destroys an image by adding noise in many small steps, then trains a network to undo each step, so that starting from pure noise it can build a clean sample. Neither writes p(x) down explicitly. Both learn to *produce* from it.

The generative landscape

Before the two main methods, it helps to see the whole map, because "learn p(x)" splits into families by *how* a model holds that distribution. The first split is **explicit** versus **implicit** density. An explicit-density model commits to a formula for p(x), or a workable bound on it,

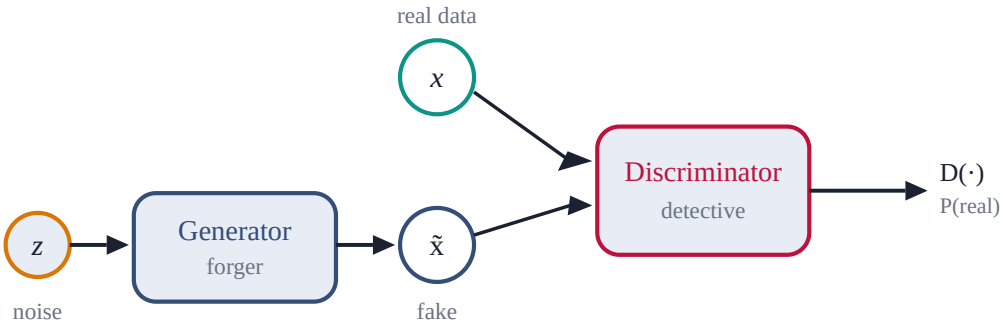
and can therefore *score* a sample, telling you how probable it is (the probability background is collected in Appendix C). An implicit-density model never forms p(x) at all. It offers only a procedure that *emits* samples, and we judge it by what comes out.

That split sorts the field into three training principles. **Likelihood-based** models maximize the data's probability, or a bound on it, directly: autoregressive models and normalizing flows do so *exactly*, while the VAE's ELBO only bounds log p(x) (Chapter 19, autoencoders). **Adversarial** models, such as the GAN, are implicit: they learn to sample without ever naming a probability. **Score-based** models learn neither the density nor the sampler first. Instead they learn the density's gradient $\nabla_x \log p(x)$, the direction of increasing plausibility. This view turns out to unify score matching with diffusion. Much of this chapter is the story of that last idea overtaking the first two for image synthesis.



A map of the families. Every generative model learns the data distribution p(x), but they differ in how they do it. Likelihood-based methods (blue) hold an explicit density and can score a sample: exactly, for autoregressive models and flows, or through a bound, for the VAE (Chapter 19). The adversarial GAN (violet) only learns to emit samples, with no density to evaluate. Score-based diffusion (teal) learns the gradient ∇ log p(x) that points toward realistic data. This chapter walks left to right.

We start with the adversarial game, which reached photorealism first. Then we turn to the noise-and-denoise process of diffusion that now leads it. Finally we complete the map with the likelihood-based families.



The adversarial game. The generator (blue) turns noise z into a fake x̃. The discriminator (violet) sees real and fake mixed together and outputs the probability that each is real. Each network's gain is the other's loss.

Governing equations

A GAN's generator turns a random seed z into a sample. Its discriminator outputs a probability that the sample is real:

$$\tilde{x} = G(z), \quad D(x) \in (0, 1)$$

They are trained on one shared objective, the **value function** V(D, G). The generator minimizes exactly what the discriminator maximizes:

$$\min_G \max_D \; V(D, G) = \mathbb{E}_{x \sim \text{data}}[\, \log D(x) \,] + \mathbb{E}_z[\, \log(1 - D(G(z))) \,]$$

A diffusion model instead fixes a **forward** noising process and learns to reverse it. Each forward step adds a small amount of Gaussian noise, a conditional Gaussian q. Its sample form, together with the closed-form jump straight to step t, are:

$$q(x_t \mid x_{t-1}) = N(\, x_t; \sqrt{1 - \beta_t}\, x_{t-1}, \; \beta_t I \,)$$

$$x_t = \sqrt{1 - \beta_t}\, x_{t-1} + \sqrt{\beta_t}\, \varepsilon$$

$$x_t = \sqrt{\alpha_t}\, x_0 + \sqrt{1 - \alpha_t}\, \varepsilon$$

$$\alpha_t = \Pi_{s=1..t} (1 - \beta_s) = (1-\beta_1)(1-\beta_2)\cdots(1-\beta_t)$$

A network ε_θ is then trained to predict the noise that was added, with a simple squared-error loss:

$$L = \mathbb{E}_{x_0,\, t,\, \varepsilon} \,\|\, \varepsilon - \varepsilon_\theta(x_t, t) \,\|^2$$

z	random latent seed for a GAN, e.g. z ∼ N(0, I); different z give different samples
G, D	generator and discriminator networks; G: z → sample, D: sample → probability of being real
x, $\tilde{x}$	a real training sample and a generated (fake) one. Both live in the same data space
β_t	noise schedule — the small fraction of variance added at forward step t ($0 < \beta_t \ll 1$)
α_t	cumulative signal kept by step t: the product of $(1 - \beta_s)$; shrinks from ≈1 toward 0
ε, ε_θ	the true Gaussian noise ε ∼ N(0, I), and the network's prediction of it
t, T	diffusion step index and total steps (often hundreds to a thousand)

How it works

The GAN game. Training alternates: hold G fixed and push D to label reals as 1 and fakes as 0. Then hold D fixed and push G to make D call its fakes real. For a fixed G the best possible discriminator is

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_G(x)}$$

Substituting it shows the generator is driven to match the data distribution. At the global optimum the two distributions become equal and the detective is reduced to a coin flip. Fakes become impossible to tell from real data:

$$p_G = p_{data} \;\Rightarrow\; D^*(x) = ½$$

Putting D* back into V(D, G) reveals what the generator is really minimizing: up to a constant, the value becomes the **Jensen–Shannon divergence** between p_{data} and p_G — a symmetric relative of the KL divergence (Appendix D, information theory) —

$$\max_D V(D, G) = 2 \cdot JSD(\, p_{data} \,\|\, p_G \,) - \log 4$$

which reaches its lowest value at $-\log 4 = 2 \ln ½ \approx -1.386$, exactly when $p_G = p_{data}$ and the divergence is zero. This is the numerical floor the code cell below moves down to.

Saturating vs non-saturating loss. How G is actually trained hides a subtle problem. The minimax form asks the generator to minimize $\mathbb{E}_z[\log(1 - D(G(z)))]$. Early in training the discriminator rejects the obvious fakes with confidence, $D(G(z)) \approx 0$. There the curve $\log(1 - D)$ is nearly flat, so its gradient *vanishes* exactly when the generator is worst and most needs a signal. Goodfellow's fix is the **non-saturating** loss. Instead of minimizing $\log(1 - D)$, the generator *maximizes* the log-probability of being called real,

$$\text{maximize}_G \ \mathbb{E}_z[\log D(G(z))] \quad (\text{equivalently, minimize } -\log D(G(z)))$$

which shares the same fixed point but has a steep gradient exactly where the fake is confidently rejected, so training actually moves. This is the loss every practical GAN uses.

Where GANs struggle. The game is fragile. In **mode collapse** the generator discovers a few outputs that reliably fool D and produces only those, ignoring the diversity of the data. Training can **oscillate or diverge** because no single loss is decreasing. The two players chase each other. And a GAN gives **no likelihood**: you cannot ask how probable a sample is, so quality is judged by proxy metrics. A **conditional GAN** controls the output by feeding a label or image to both G and D. This enables class-controlled and image-to-image generation.

Stabilizing the game. A decade of engineering reduced these failures. **DCGAN** settled on an architecture that trains reliably: strided convolutions, batch normalization (Chapter 11), and no fully-connected layers. This made GANs practical for images. **WGAN-GP** changed the objective itself. It swaps the Jensen–Shannon game for the Wasserstein (earth-mover) distance, and enforces the required 1-Lipschitz discriminator with a gradient penalty. This gives a loss that tracks sample quality and rarely collapses. These, with the non-saturating loss above, are most of the reason GANs work at all in practice.

The diffusion process. The forward pass needs no learning at all. It just adds a little Gaussian noise at each of T steps until x_T is pure noise, with β_t setting how fast. Because Gaussians compose, the closed-form jump lets you land at any step t in one shot. This is essential, because it means training can sample a random t cheaply instead of simulating the whole chain. The

reverse pass is where the learning happens. The network $\varepsilon_\theta(x_t, t)$ looks at a noisy image and predicts the noise inside it (denoising score matching, Ho et al. 2020). Predicting the noise is equivalent to pointing back toward the clean data. To generate, start from $x_T \sim N(0, I)$ and repeatedly subtract a piece of the predicted noise, stepping $t = T \to 0$:

$$x_{t-1} = \frac{1}{\sqrt{1-\beta_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\alpha_t}} \varepsilon_\theta(x_t, t) \right) + \sigma_t z$$

Why diffusion overtook GANs. The reverse process is trained with a stable regression loss rather than a fragile minimax game, so it rarely collapses. It covers all modes of the data, and it reaches state-of-the-art fidelity. The price is speed. Sampling repeats many sequential denoising steps, far slower than a GAN's single forward pass.

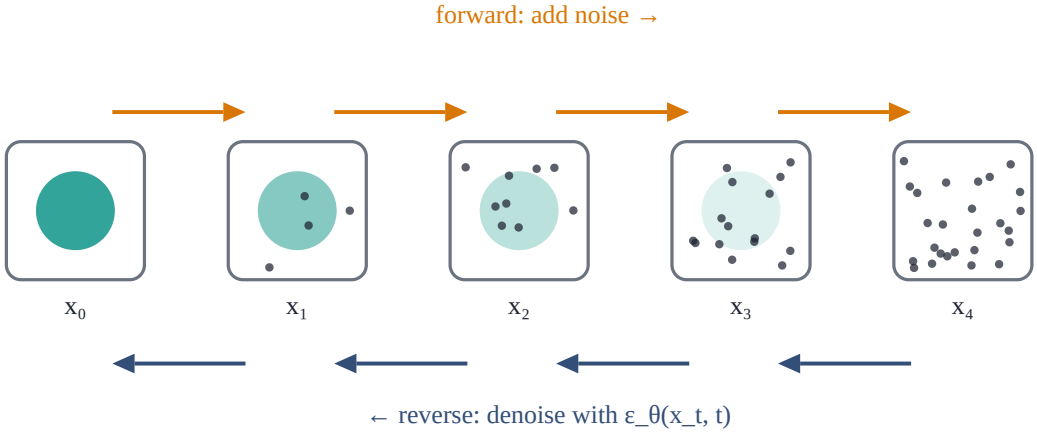
From words to pictures. Text-to-image systems condition the denoiser on a text embedding c, typically produced by a model such as CLIP, so every denoising step is guided toward images that match the prompt:

$\varepsilon_\theta(x_t, t, c)$ — the noise predictor now also reads the prompt embedding c

But conditioning alone produces images that only loosely follow the prompt. **Classifier-free guidance** tightens this control. During training the prompt is randomly dropped part of the time, so the single network learns both a conditional prediction $\varepsilon_\theta(x_t, t, c)$ and an unconditional one $\varepsilon_\theta(x_t, t, \varnothing)$ with the null prompt $\varnothing$. At sampling time, the two are combined and *extrapolated* in the direction the prompt adds, amplified by a guidance scale w:

$$\tilde{\varepsilon}_\theta = \varepsilon_\theta(x_t, t, \varnothing) + w \cdot \left(\varepsilon_\theta(x_t, t, c) - \varepsilon_\theta(x_t, t, \varnothing) \right)$$

Here $w = 1$ recovers the plain conditional model, and $w > 1$ (typically 3–8) trades a little diversity for much tighter prompt adherence. This is the control behind how literally a text-to-image model follows you. This conditioning is how Stable Diffusion and DALL·E turn a sentence into a picture. The same photoreal power drives **deepfakes**, so provenance, watermarking and consent are now serious concerns, not afterthoughts.



Diffusion, both directions. Left to right, the fixed forward process (amber) covers a clean sample x_0 with noise. Right to left, the learned network ε_θ (blue) predicts and removes noise, so sampling turns pure noise back into a fresh image.

The reverse process, step by step

Generation is the forward chain run backward. Sampling starts from pure Gaussian noise $x_T \sim N(0, I)$ and applies the learned denoiser T times. Each step moves the sample a little closer to the data manifold:

$$\text{for } t = T, T{-}1, \ldots, 1: \quad x_{t-1} = \frac{1}{\sqrt{1-\beta_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\alpha_t}} \varepsilon_\theta(x_t, t) \right) + \sigma_t z$$

Each step does two things. The bracketed part is a *denoising* guess: subtract a measured piece of the predicted noise ε_θ to recover a cleaner estimate. The trailing $\sigma_t z$ then adds back a little fresh noise ($z \sim N(0, I)$, with $\sigma_t = 0$ only on the final step), which keeps the reverse chain a genuine stochastic process rather than a deterministic average. This is the source of sample diversity. Predicting the noise ε is mathematically equivalent to predicting the clean image x_0, because the closed-form forward equation $x_t = \sqrt{\alpha_t}\, x_0 + \sqrt{1-\alpha_t}\, \varepsilon$ ties the three quantities together. The network simply uses the parameterization that trains most stably.

The obvious cost is the T sequential steps: up to a thousand network evaluations to make one image, versus a GAN's single forward pass. Much recent work targets exactly this. **DDIM** makes the reverse process deterministic (drop the $\sigma_t z$) and lets you skip steps, cutting a thousand down to twenty or fifty with little quality loss. **Distillation** trains a student to take one

step where the teacher took many, approaching real-time generation. The very same sequential-refinement structure is diffusion's strength. Unlike a GAN's single jump, each step needs only a small correction. This is why the process is so stable and covers the full data distribution instead of collapsing to a few modes.

Denoising is score matching. There is a second way to read the noise-predictor, and it is the bridge to the score-based corner of the map. For the forward marginal $q(x_t \mid x_0)$ the gradient of the log-density — the **score** $\nabla_{x_t} \log q$ — equals $-\varepsilon / \sqrt{1-\alpha_t}$, so a network trained to predict ε is, up to that scale, estimating the score of the noised data. Sampling then works as noisy gradient ascent on log-probability (Langevin dynamics): each reverse step moves toward denser regions and adds back a little noise. Song & Ermon arrived at the identical algorithm from this direction, which is why "denoising diffusion" and "score-based generative models" describe one idea.

DDPM and DDIM are two samplers over one model. The stochastic sampler above (keep σ_t z, and step through every step from T down to 0) is the original **DDPM** of Ho et al. **DDIM** reuses the very same trained ε_θ but defines a non-Markovian reverse process that is deterministic ($\sigma_t = 0$) and can skip over steps, turning a thousand evaluations into twenty. Nothing is retrained. You fit ε_θ once and choose the sampler (and the quality-versus-speed trade-off) at generation time.

Autoregressive models and normalizing flows

The two main methods are not the whole map. Two **likelihood-based** families give what GANs and diffusion cannot: an *exact* probability for any sample. Each one earns it in a different way.

Autoregressive models factor the joint distribution into an ordered chain of conditionals, one variable at a time, by the product rule of probability (Appendix C):

$$p(x) = \Pi_i \, p(x_i \mid x_1, ..., x_{i-1})$$

Each factor is a network that sees only what came before. For images, **PixelCNN** paints a picture pixel by pixel, top-left to bottom-right, every pixel conditioned on those already drawn. A masked convolution enforces the "earlier pixels only" rule. The training signal is just the

likelihood of the next value, a self-supervised objective (Chapter 20). Here is the connection worth remembering: it is exactly the objective of a language model, which factors a sentence into p(next token | previous tokens) (Chapter 17, large language models). An autoregressive image model is a language model whose tokens are pixels. The reward is a clean, exact log-likelihood and stable training. The price is sampling speed, since an N-pixel image needs N sequential network passes, one per pixel.

Normalizing flows reach an exact likelihood in a different way. A flow builds x by pushing a simple base sample $z \sim N(0, I)$ through a chain of *invertible* layers f, and scores any x by running the chain backward and applying the change-of-variables formula:

$$\log p(x) = \log p(z) - \log \left| \det \left(\partial f / \partial z \right) \right|, \quad z = f^{-1}(x)$$

Every layer must be invertible with an easily computed Jacobian determinant. This is a real architectural constraint. In return, the flow gives exact likelihoods and fast one-pass sampling, valued wherever a calibrated density matters more than raw realism: density estimation, variational inference, and uncertainty modeling (Chapter 24).

Evaluating generative models

Without an explicit likelihood a GAN cannot report how probable its samples are, so image models are judged by proxy metrics computed in the feature space of a pretrained classifier. This is a learned representation (Chapter 5) that captures semantic content far better than raw pixels. Two are standard. The **Inception Score** (IS) runs generated images through an Inception classifier and rewards two things at once: each image should draw a confident, low-entropy label (it looks like *something*), while the labels across many samples should be diverse (the model is not making one thing). Higher is better. Its weakness is that it never looks at the real data. The **Fréchet Inception Distance** (FID) fixes this: it fits a Gaussian to the Inception features of real images and another to those of generated images, and reports the Fréchet (Wasserstein-2) distance between them,

$$\mathrm{FID} = \| \mu_r - \mu_g \|^2 + \mathrm{tr}\left(\Sigma_r + \Sigma_g - 2\left(\Sigma_r \Sigma_g \right)^{1/2} \right)$$

so a lower FID means the generated set matches the real set in both the mean and the covariance of features. It captures fidelity *and* diversity in one number, and punishes mode collapse: a generator that drops modes shifts its feature covariance, so its FID climbs. FID is the field's default, but it inherits the Inception network's weaknesses and needs thousands of samples to be stable. Both metrics are convenient, not fundamental. They track human judgment only loosely, so sample grids and human-preference studies still accompany every serious result.

Generative models in code

The forward diffusion arithmetic of the worked example is a two-line loop. This cell runs both noising steps and confirms the closed-form jump: landing at step 2 directly matches stepping through it, using a single equivalent noise draw:

Python · NumPy ▶ runnable in the app

```python
import numpy as np

x0 = 2.0
betas = [0.2, 0.5]
eps   = [0.3, -1.1]

x = x0
for step, (b, e) in enumerate(zip(betas, eps), start=1):
    x = np.sqrt(1 - b) * x + np.sqrt(b) * e     # one forward noising step
    print(f"step {step}: x_{step} = {x:.4f}")

abar = np.prod([1 - b for b in betas])          # cumulative kept signal
print(f"abar_2                = {abar:.4f}")
print(f"kept  sqrt(abar)      = {np.sqrt(abar):.4f}")
print(f"noise sqrt(1 - abar)  = {np.sqrt(1 - abar):.4f}")
eps_eq = (x - np.sqrt(abar) * x0) / np.sqrt(1 - abar)
print(f"equivalent single eps = {eps_eq:.4f}")
```

The GAN objective is just as concrete, and needs no training to examine. We can evaluate the value function V(G, D) at each generator's *best-response* discriminator $D^*(x) = p_{data}/(p_{data} + p_G)$. As a 1-D generator moves its output distribution toward the real data, the game value falls toward its floor of $2 \ln 0.5 \approx -1.386$. This is the point where the discriminator is reduced to a coin flip:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)

def gauss(x, mu, s=1.0):
    return np.exp(-(x - mu) ** 2 / (2 * s * s)) / np.sqrt(2 * np.pi * s * s)

# real data ~ N(3,1). Generator emits N(gmu,1); slide gmu toward 3.
def game_value(gmu, n=200000):
    real = rng.normal(3, 1, n)
    fake = rng.normal(gmu, 1, n)
    # best-response discriminator for THIS generator: D*=p_data/(p_data+p_G)
    Dr = gauss(real, 3) / (gauss(real, 3) + gauss(real, gmu))
    Df = gauss(fake, 3) / (gauss(fake, 3) + gauss(fake, gmu))
    return np.log(Dr).mean() + np.log(1 - Df).mean()

for gmu in [0.0, 1.0, 2.0, 3.0]:
    print(f"generator N({gmu:.0f},1):  max_D V = {game_value(gmu):+.3f}")
print(f"floor  2*ln(0.5)      = {2 * np.log(0.5):+.3f}")
```

The value drops monotonically as the fake distribution approaches the real one: the generator wins by making the discriminator's task impossible. In PyTorch the two adversaries are ordinary modules trained in alternation. A diffusion model needs no adversary at all, only a stable noise-prediction regression step:

PyTorch copy → Colab

```python
import torch, torch.nn as nn
import torch.nn.functional as F

# ---- a tiny GAN: two networks, two optimizers, one alternating update ----
G = nn.Sequential(nn.Linear(64, 128), nn.ReLU(), nn.Linear(128, 784),
nn.Tanh())
D = nn.Sequential(nn.Linear(784, 128), nn.LeakyReLU(0.2), nn.Linear(128, 1))
optG = torch.optim.Adam(G.parameters(), 2e-4, betas=(0.5, 0.999))
optD = torch.optim.Adam(D.parameters(), 2e-4, betas=(0.5, 0.999))
bce = nn.BCEWithLogitsLoss()

real = torch.rand(32, 784)
z = torch.randn(32, 64); fake = G(z)
lossD = bce(D(real), torch.ones(32, 1)) + bce(D(fake.detach()),
torch.zeros(32, 1))
```

```
optD.zero_grad(); lossD.backward(); optD.step()      # D: reals->1, fakes->0
lossG = bce(D(fake), torch.ones(32, 1))              # G: fool D into saying
1
optG.zero_grad(); lossG.backward(); optG.step()

# ---- one diffusion training step: predict the noise added at a random t --
--
eps_net = nn.Sequential(nn.Linear(784 + 1, 256), nn.ReLU(), nn.Linear(256,
784))
x0 = torch.rand(32, 784); t = torch.randint(1, 1000, (32, 1)).float()
abar = torch.cos(t / 1000 * 1.5708) ** 2             # a valid schedule: 1 -
> 0
eps = torch.randn_like(x0)
xt = abar.sqrt() * x0 + (1 - abar).sqrt() * eps      # forward jump to step
t
loss = F.mse_loss(eps_net(torch.cat([xt, t / 1000], 1)), eps)
loss.backward()
```

In this app's engine

This chapter has no interactive tool. Realistic generation needs image-scale networks well beyond the Playground's tabular sandbox. But two familiar pieces reappear here: the discriminator is exactly a binary classifier trained with cross-entropy (Chapter 6, loss functions), and the diffusion denoiser is trained with the simple MSE regression loss you have already seen in the Playground. What is new is not the loss but the machinery built around it: the adversarial pairing, and the forward-and-reverse noise chain.

Worked example — two steps of forward diffusion

Take a one-dimensional signal $x_0 = 2.0$ and a two-step schedule $\beta_1 = 0.2$, $\beta_2 = 0.5$, with noise draws $\varepsilon_1 = 0.3$ and $\varepsilon_2 = -1.1$. Using $\sqrt{0.8} = 0.8944$, $\sqrt{0.2} = 0.4472$, $\sqrt{0.5} = 0.7071$:

1. Step 1: $x_1 = \sqrt{1-0.2}\cdot 2.0 + \sqrt{0.2}\cdot 0.3 = 0.8944\cdot 2 + 0.4472\cdot 0.3 = 1.7889 + 0.1342 = \mathbf{1.9231}$.

2. Step 2: $x_2 = \sqrt{1-0.5}\cdot 1.9231 + \sqrt{0.5}\cdot(-1.1) = 0.7071\cdot 1.9231 + 0.7071\cdot(-1.1) = 1.3598 - 0.7778 =$ **0.5820**.
3. Closed-form check: $\alpha_2 = (1-0.2)(1-0.5) = 0.8\cdot 0.5 =$ **0.40**, so the signal kept is $\sqrt{0.40} = 0.6325$ and the noise is $\sqrt{0.60} = 0.7746$.
4. Consistency: solving $x_2 = \sqrt{0.40}\cdot 2 + \sqrt{0.60}\cdot\varepsilon$ for the equivalent single draw gives $\varepsilon = (0.5820 - 1.2649)/0.7746 =$ **−0.882** — one N(0,1)-sized draw, exactly as the closed form promises.

After only two steps the signal has faded to ≈63% of x_0 while noise contributes ≈77% of the spread. Push t toward T and $\sqrt{\alpha_t} \to 0$, leaving pure noise. The reverse network's whole job is to reverse this fading, one denoise at a time.

Worked example — the closed-form jump and the noise-prediction target

Training never simulates the whole chain. It jumps straight to a random step with the closed form and asks the network to name the noise. Take a clean signal $x_0 = 3.0$, a step whose cumulative signal is given as $\alpha_t = 0.36$, and one noise draw $\varepsilon = 0.5$. Using $\sqrt{\alpha_t} = \sqrt{0.36} = 0.60$ and $\sqrt{1-\alpha_t} = \sqrt{0.64} = 0.80$:

1. Forward jump: $x_t = \sqrt{\alpha_t}\,x_0 + \sqrt{1-\alpha_t}\,\varepsilon = 0.60\cdot 3.0 + 0.80\cdot 0.5 = 1.80 + 0.40 =$ **2.20**.
2. The training target is the noise itself, $\varepsilon =$ **0.5**. The network sees only ($x_t = 2.20$, t) and must recover it. Suppose it predicts $\varepsilon_\theta = 0.7$. The per-sample loss is $\|\varepsilon - \varepsilon_\theta\|^2 = (0.5 - 0.7)^2 =$ **0.04**.
3. A noise guess *is* a clean-image guess. Inverting the closed form, $\hat{x}_0 = (x_t - \sqrt{1-\alpha_t}\,\varepsilon_\theta)/\sqrt{\alpha_t}$. With the imperfect $\varepsilon_\theta = 0.7$: $\hat{x}_0 = (2.20 - 0.80\cdot 0.7)/0.60 = (2.20 - 0.56)/0.60 =$ **2.733**.

4. With a *perfect* guess $\varepsilon_\theta = 0.5$: $\hat{x}_0 = (2.20 - 0.40)/0.60 = 1.80/0.60 = \mathbf{3.00} = x_0$ exactly — predicting the noise recovers the original signal.

What the numbers mean: the entire training objective is step 2, a squared error between the true noise and the network's guess, at a randomly chosen noise level. The 0.04 loss and the 0.267-off estimate $\hat{x}_0 = 2.733$ shrink together as ε_θ improves, and at $\varepsilon_\theta = \varepsilon$ the clean signal returns exactly. This is why the noise-prediction and clean-image parameterizations are the same model in different forms.

Assumptions & limitations

GANs give no explicit likelihood, so you cannot score how probable a sample is or compare models cleanly. Quality is judged by proxies like FID, and training can collapse to a few modes or diverge outright. Diffusion trades that instability for *slowness*: sampling repeats many sequential steps, which is compute-heavy, and it assumes a fixed Gaussian noise schedule that is a modeling choice, not a law. Both models are only as fair as their data. Trained on skewed samples, they reproduce and amplify the skew, and their photorealism raises real deepfake, consent and misinformation risks. Our worked example is a 1-D toy with hand-picked noise. Genuine diffusion is high-dimensional with a *learned* reverse process and a schedule of hundreds of steps.

Why & when this matters

Use a generative model when you need to create, augment or imagine data rather than label it: image, audio and video synthesis, data augmentation for rare classes, super-resolution, inpainting and denoising, physical simulation, and molecule or drug design. Prefer **diffusion** when fidelity and full coverage matter and you can afford slow sampling. Prefer a **GAN** when you need fast, single-shot generation. All three families in

these two chapters — the VAE of Chapter 19 (autoencoders) and the GANs and diffusion here — share one goal: learn the shape of the data well enough to draw something new from it.

In the real world

The text-to-image systems that reached the public in 2022 — OpenAI's DALL·E 2, Google's Imagen, and the open-source Stable Diffusion — are all conditional diffusion models. Stable Diffusion made the approach practical by running diffusion in the compressed latent space of a pretrained autoencoder (Chapter 19) rather than on megapixel images, so it fits on a consumer GPU. A text prompt is encoded by a CLIP-style model into the embedding c that steers every denoising step. The same family now generates video (OpenAI's Sora), audio, and 3-D assets, and drives professional tools across design, film, and advertising. The downside is real: the identical technology produces convincing **deepfakes**. This is why watermarking, provenance standards such as C2PA, and consent have become active areas of law and research, not afterthoughts.

Exercises

1. Run one forward diffusion step on $x_0 = 1.0$ with $\beta_1 = 0.1$ and noise $\varepsilon = 0.5$. What is x_1? MATH

2. For the schedule $\beta = [0.1, 0.2, 0.3]$, compute the cumulative signal $\bar{\alpha}_3$ and the resulting signal- and noise-scaling factors $\sqrt{\bar{\alpha}_3}$ and $\sqrt{(1-\bar{\alpha}_3)}$ for the closed-form jump to step 3. MATH

3. Simulate a 10-step increasing noise schedule and print how the cumulative kept-signal factor $\sqrt{\bar{\alpha}_t}$ decays with t. CODE

4. Diffusion models have largely displaced GANs for high-fidelity image generation. Give the two main reasons — and name the price diffusion pays. CONCEPT

5. At a point x, the true data density is $p_{data}(x) = 0.6$ and the generator's is $p_G(x) = 0.2$. Give the optimal discriminator's output $D^*(x)$, and its value when the generator perfectly matches the data. MATH

6. What is mode collapse in a GAN, why does the adversarial objective invite it, and name one mitigation. CONCEPT

7. With a 1-D toy, evaluate the GAN value function for a generator that mismatches the data (N(0,1) vs real N(3,1)) and one that matches it (N(3,1)). Confirm the matched case hits the $2 \ln 0.5$ floor. CODE

8. Early in training a discriminator confidently rejects the generator's fakes, $D(G(z)) = 0.02$. Compare the gradient the generator receives under the **saturating** objective $\log(1 - D)$ and the **non-saturating** objective $-\log D$ by evaluating the derivative of each with respect to D at that point. Which one moves the generator? MATH

9. Of GANs, autoregressive models, normalizing flows, and diffusion, which give an *exact* likelihood for a sample, which give only a bound, and which give none? And why is an autoregressive image model slow to sample? CONCEPT

10. A text-to-image denoiser returns an unconditional noise estimate and a prompt-conditioned one. Implement classifier-free guidance $\tilde{\varepsilon} = \varepsilon_{uncond} + w \cdot (\varepsilon_{cond} - \varepsilon_{uncond})$ and show what guidance scales $w = 0, 1, 3$ do. CODE

PART IV

Beyond supervised learning

When labels run out or a model must act: reusing pretrained networks through transfer learning and fine-tuning, learning from reward with deep reinforcement learning, and knowing what a network does not know through uncertainty, calibration and Bayesian methods.

22 Transfer learning & fine-tuning

23 Deep reinforcement learning

24 Uncertainty, calibration & Bayesian nets

Transfer learning & fine-tuning

> *Training a modern network from scratch is something almost nobody can afford. It takes a huge amount of labeled data and a large amount of computing power. Yet the network that recognizes a rare skin lesion from three hundred photographs, or classifies legal clauses from a few thousand contracts, was never trained from scratch at all. It inherited a backbone that had already learned what edges, textures, shapes, and grammar look like. It learned this from a million unrelated images or a trillion words of text. It then adapted that head start to the task at hand. This is* ***transfer learning****: reuse the representation a network built on a large source task (Chapter 5), and spend your scarce data and compute only on the small gap between that and what you actually need.*

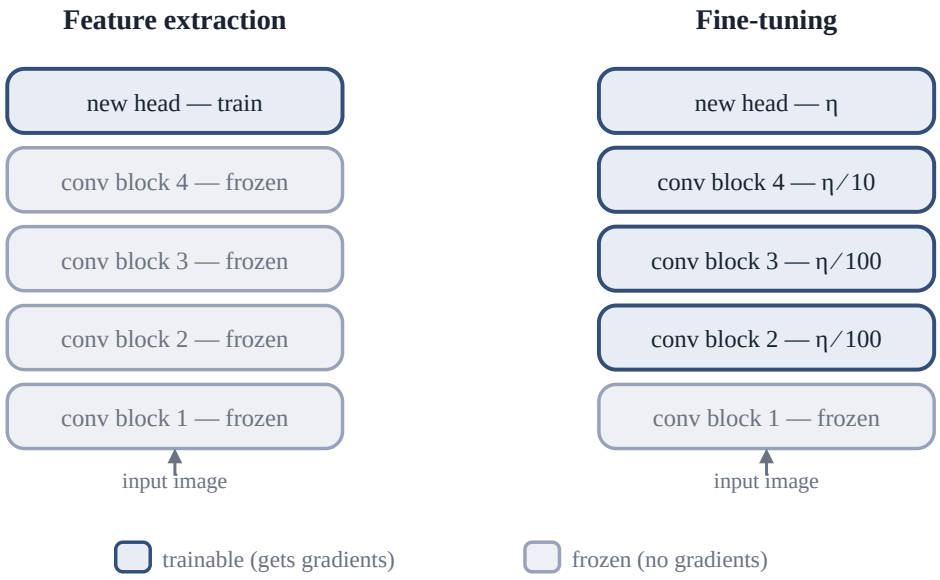
Fundamental concept

A medical student who has spent years learning anatomy can be taught to spot a new disease from a handful of examples. A person with no training who is shown the same handful learns almost nothing, because they lack the representation that the examples are meant to refine. Neural networks are the same. A convolutional network (Chapter 13) trained on ImageNet discovers edge and color detectors in its early layers, and texture and object-part detectors in its middle layers. These features are general enough to help on radiographs, satellite tiles, and artwork it never saw. Transfer learning keeps that valuable representation and aims it at a new target.

The method has two phases. **Pretraining** runs once, on a large *source* dataset. This can be supervised (ImageNet classification) or, increasingly, self-supervised on unlabeled data (Chapter 20). It produces a **backbone**: the stack of layers that maps an input to a representation. **Downstream adaptation** then fits that backbone to a *target* task with far less data. It attaches a fresh output **head** and decides how much of the backbone to move. Everything in this chapter

answers that second question, *how much to move, and how*. The options range from touching only the new head, to changing the whole network, to editing a few thousand carefully chosen parameters, to distilling one network's knowledge into another.

Why it works is simple to explain. Learning a good representation is the expensive part. Fitting a boundary on top of one is cheap. Pretraining pays that cost once and spreads it over every task that reuses the backbone. This shifts the data–accuracy curve dramatically. A task needing a hundred thousand labels from scratch may need a few hundred when fine-tuned, because the network no longer has to rediscover vision or language. It only has to learn the final step.



The two classic approaches share one pretrained backbone and differ only in which layers receive gradients. Feature extraction freezes everything but a new head. Fine-tuning unfreezes the upper layers using small, layer-wise learning rates, while often leaving the most general early layer frozen. The choice trades the ability to adapt against the risk of overwriting what pretraining learned.

Two ways to transfer: feature extraction vs fine-tuning

Both approaches start from the same pretrained parameters θ_{pre} and run ordinary gradient descent (Chapter 7), $\theta \leftarrow \theta - \eta \nabla_\theta L$, on the target loss. What differs is *which* parameters carry a gradient and how large η is.

$$\text{feature extraction: } \theta_{\text{head}} \leftarrow \theta_{\text{head}} - \eta\, \nabla L, \quad \theta_{\text{backbone}} \text{ frozen } (\nabla \text{ blocked})$$

$$\text{fine-tuning: } \theta \leftarrow \theta - \eta_\ell \nabla L \text{ from } \theta = \theta_{pre}, \ \eta_\ell \text{ small (10–100× below scratch)}$$

- θ_{pre} — the pretrained parameters used as the *initialization*. This is the single most important difference from training a fresh network, whose θ starts random (Chapter 10)
- θ_{head} — the new output layer added for the target task (e.g. a fresh linear classifier over the target classes)
- $\theta_{backbone}$ — the reused representation layers. "Frozen" means their `requires_grad` is off, so the backward pass never updates them
- η_ℓ — a per-layer learning rate. In fine-tuning it is kept small so the pretrained weights are changed slightly, not overwritten

Feature extraction freezes the backbone and trains only the head. Since no gradient flows into the frozen layers, you can push every image through the backbone *once*, cache the feature vectors, and train the head on them as a plain logistic regression. This is cheap, and it cannot overfit the backbone because the backbone cannot move. It suits a small target dataset in a domain close to the source, where the pretrained features are already almost what you need. Famously, a frozen off-the-shelf ImageNet backbone with a linear classifier on top beat many specially built vision systems of that time (Chapter 14).

Fine-tuning unfreezes some or all of the backbone and trains end-to-end, but at a learning rate one to two orders of magnitude below what you would use from scratch. The small step is the key idea. The pretrained weights already sit in a good region, and a large gradient step on the first noisy target batch would move them out of it before they help. Fine-tuning adapts more than feature extraction. It is better when the target set is larger, or when its domain differs enough that late features must genuinely change. This comes at the cost of more compute and data, and the two risks of overfitting and forgetting discussed below. For a sense of scale, freezing a ResNet-50's ~23 million backbone parameters and adding a ten-class head over its 2048-dim features costs only $2048 \cdot 10 + 10 = 20{,}490$ trainable parameters, under 0.1% of the network.

How transferable is a layer?

Feature extraction and fine-tuning both rest on a measured fact about *where* in a network reusable knowledge lives. Yosinski and colleagues (2014) measured it. They trained on one half of ImageNet's classes, transplanted the first k layers into a network for the other half, and varied

k from 1 to 7. This finding is now a core rule of the field.

Early layers are general. Late layers are specific. The first convolutional layers of essentially every vision network converge to the same thing, regardless of task or dataset: oriented edge detectors and color blobs, which are close relatives of Gabor filters. Because of this, they are not tied to any specific task and transfer almost perfectly. Higher up the stack, features become specialized to the source task's classes. For example, a late-layer unit that fires for "a specific dog breed's fur" is worthless for classifying galaxies. Transferability therefore *decays with depth*. The deeper the layer, the more of the source task it has built into it, and the less it carries over.

The study also revealed two more subtle effects. First, **co-adaptation is fragile**: neighboring layers are tuned together, so transferring a layer but re-initializing its partner breaks those connections and hurts performance even when source and target are identical. This is a fragility of optimization, not of meaning. Second, and more encouragingly, **transfer-plus-fine-tune beats random initialization even on the same task**: weights that are pretrained and then fine-tuned generalize measurably better than training from scratch, a lasting benefit of starting in a good region of weight space. The practical result is a control you can adjust: the more similar source and target, the more layers you can safely freeze. Freeze the general early layers, and retrain the specific late ones.

Discriminative learning rates and gradual unfreezing

If early layers should barely move and late layers should move more, a single global learning rate is the wrong choice. **Discriminative** (layer-wise) learning rates give each layer group its own η, decaying toward the input. Howard and Ruder's ULMFiT (2018), which brought transfer learning to NLP, set η for each layer to a fixed factor below the layer above it. A decay of about 2.6 per layer worked well, so the head learns fastest and the earliest layers change very slowly. The idea follows the transferability curve exactly: general features need only a small adjustment, specific features need real retraining, and the new head, starting from random weights, needs the most learning of all.

Even at small rates, unfreezing the whole backbone at once is dangerous. The newly-initialized head produces large, meaningless gradients on the first batches. These flow back and corrupt the backbone before the head stabilizes. **Gradual unfreezing** prevents this. Keep the backbone frozen until the head settles, then unfreeze the last block, then the next, opening the network from the top down over several epochs. A **warmup** ramps the learning rate up from near zero before decaying it. This protects the pretrained weights during the noisiest opening part of training. Together, discriminative rates, gradual unfreezing, and warmup are the standard method for fine-tuning without destroying what you started from.

When transfer fails: domain shift, negative transfer and forgetting

Transfer is not free, and three failure modes deserve an honest discussion. The first is **domain shift**: source and target are drawn from different distributions, and the further apart they are, the less the pretrained features help. Natural-image features transfer poorly to grayscale X-rays or satellite imagery, and English-text features transfer poorly to source code or protein sequences. Often only the lowest-level features survive a large shift. Edges are still edges on an X-ray. But everything above the first block must be relearned. This is why a big domain gap pushes you toward more fine-tuning and fewer frozen layers.

Push the gap far enough and you reach **negative transfer**: the pretrained backbone does *worse* than training from scratch, because its features are not just unhelpful but actively misleading, and the optimizer wastes its effort unlearning them. This is why "always pretrain" is not a rule. When source and target are unrelated, a random start can be the better choice.

The third failure mode is **catastrophic forgetting**: while fine-tuning on the target task, stochastic gradient descent sees only target batches and steadily overwrites the weights that encoded the source knowledge. The model improves on the new task and gets worse on the old one, sometimes destroying source accuracy entirely. In a *continual-learning* stream this is fatal, because each task erases the ones before it. The classic solutions all keep weights near where pretraining left them. Small learning rates and freezing do this in a crude way. **Rehearsal** replays a buffer of old examples. And **elastic weight consolidation** (Kirkpatrick et al. 2017)

adds a penalty $\lambda \Sigma_i F_i(\theta_i - \theta_{i,pre})^2$, anchoring each weight to its pretrained value in proportion to its importance F_i (its Fisher information). The cleanest fix, though, is never to touch the pretrained weights at all. That is exactly what the next family of methods does.

Parameter-efficient fine-tuning

Full fine-tuning has a hidden cost that becomes serious at scale: it produces a *complete new copy* of the parameters for every task. So fine-tuning a 7-billion-parameter language model (Chapter 17) on ten tasks means storing and serving ten 7-billion-parameter models. **Parameter-efficient fine-tuning** (PEFT) avoids this. It freezes the entire pretrained network and trains only a small set of new or selected parameters, typically well under 1%. So each task is a tiny, swappable change on top of one shared frozen backbone. Four ideas dominate.

Adapters (Houlsby et al. 2019) insert a small bottleneck inside each transformer block: down-project from width d to a tiny m, apply a nonlinearity, then up-project back to d, wrapped in a residual connection so it starts near the identity. Only the adapters train, a few percent of the total. They work well, but because they sit *in series* in the forward path, they add a little inference latency.

LoRA, or Low-Rank Adaptation (Hu et al. 2021), removes even that latency with a sharp observation: the *update* that a weight matrix goes through during fine-tuning has low intrinsic rank, so it can be written as a product of two thin matrices. Freeze the pretrained weight W and represent its change as $\Delta W = B A$, where B and A are much thinner than W:

$$h = W x + \frac{\alpha}{r} B A x, \quad W \in \mathbb{R}^{d \times d} \text{ frozen}, \; B \in \mathbb{R}^{d \times r}, \; A \in \mathbb{R}^{r \times d}, \; r \ll d$$

W the pretrained weight matrix, kept frozen. No gradient ever touches it

A, B the only trainable parameters: a down-projection A to rank r and an up-projection B. A is initialized to small random values and B is initialized to zero, so $\Delta W = BA$ starts at exactly 0 and the network begins as the untouched pretrained model

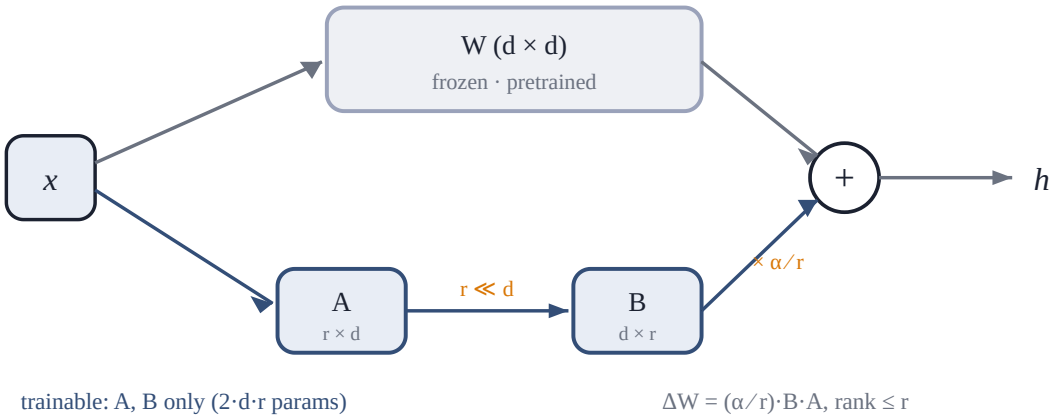
r the rank of the update, typically 4–64. The update $\Delta W = BA$ can have rank at most r, which forces the change to live in a tiny r-dimensional subspace

α a fixed scaling constant. The factor α / r keeps the update's magnitude stable as r changes, so the learning rate does not need to be re-tuned for each rank

Because $\Delta W = BA$ is just a matrix, after training you can **merge** it, $W' = W + (\alpha/r)BA$, and ship the exact original architecture with *zero* added inference cost. This is LoRA's decisive advantage over series adapters. And because W is never modified, LoRA cannot catastrophically forget the base model. You can keep dozens of task-specific adapters and swap them over one shared, untouched backbone.

Prompt and prefix tuning change *no* weights at all. Prompt tuning (Lester et al. 2021) freezes the model and learns a small number of continuous "soft prompt" vectors: trainable embeddings with no corresponding words. These are added to the front of the input and shaped by gradient descent until the frozen model behaves as the task requires. Prefix tuning (Li & Liang 2021) adds learned vectors to the front of the keys and values at *every* attention layer (Chapter 16), which gives more control. Both can train under 0.1% of the parameters. **BitFit** (Ben-Zaken et al. 2021) is the most minimal: freeze every weight and train only the *bias* terms. The fact that this is often competitive on small tasks shows how little you must change in a strong pretrained model to guide it.

frozen backbone weight W, plus a trainable low-rank detour B·A



LoRA freezes the pretrained weight W and learns its change as a low-rank detour: x is projected down to rank r by A, back up by B, scaled by α/r, and added to Wx. Only A and B train, which is 2·d·r parameters instead of d². And because ΔW = BA is a plain matrix, it can be merged back into W after training, leaving inference cost unchanged.

Worked example: the LoRA parameter budget on a 1024×1024 weight

Take one weight matrix W of a transformer, of shape d×d with d = 1024, and compare full fine-tuning against LoRA at rank r = 8 with scale α = 16.

1. **Full fine-tuning** makes every entry of W trainable: $d^2 = 1024 \cdot 1024 =$ **1,048,576** parameters for this one matrix.
2. **LoRA** adds A of shape r×d = 8×1024 = 8,192 parameters and B of shape d×r = 1024×8 = 8,192 parameters, for $2 \cdot d \cdot r = 2 \cdot 1024 \cdot 8 =$ **16,384** trainable parameters. W itself contributes none, because it is frozen.
3. **Fraction:** 16,384/1,048,576 = 2r/d = 16/1024 = 1/64 = **1.5625%**. LoRA trains about one parameter in sixty-four.
4. **Rank of the update.** ΔW = BA is a 1024×1024 matrix, but its rank is bounded by the shared inner dimension: rank(BA) ≤ min(rank B, rank A) ≤ r = **8**. So while the pretrained W may have rank up to 1024, the change LoRA can make lives in an 8-dimensional subspace. This is a strong, deliberate constraint.
5. **Scale.** The effective update is (α/r)·BA = (16/8)·BA = 2·BA. Fixing α and varying r keeps this magnitude in range, so you can change the rank without re-

tuning the learning rate.

What the numbers mean: storing a full fine-tune of this layer costs a fresh million parameters per task. LoRA costs sixteen thousand, a 64× saving that compounds across every weight matrix and every task. A hundred LoRA-adapted tasks share one frozen backbone and add 1.6 million parameters between them, versus a hundred million for a hundred full copies. The assumption LoRA makes, that the useful fine-tuning update is low-rank, is exactly why 1.56% of the parameters can recover most of full fine-tuning's accuracy.

Knowledge distillation as transfer

The methods so far transfer a representation between *tasks*. **Knowledge distillation** (Hinton, Vinyals & Dean 2015) transfers knowledge between *models*: a large, accurate **teacher** (or ensemble) trains a small, fast **student** to imitate it, so the student inherits much of the teacher's accuracy at a fraction of the size. It is transfer in the purest sense: the teacher's learned function is the source, and the student is the target. It is the standard way to get a deployable model (Chapter 14) from a strong but bulky one.

The insight is that a teacher's *soft* output carries far more than the hard label. A digit classifier shown a "3" might put 0.9 on class 3, 0.08 on class 8, and near-zero elsewhere. That 0.08 is not noise. It is the teacher saying that a 3 resembles an 8 far more than a 1. Hinton called this the **dark knowledge** in the wrong-class probabilities: a similarity structure over classes that a one-hot label discards. To reveal it, soften the softmax with a **temperature** T:

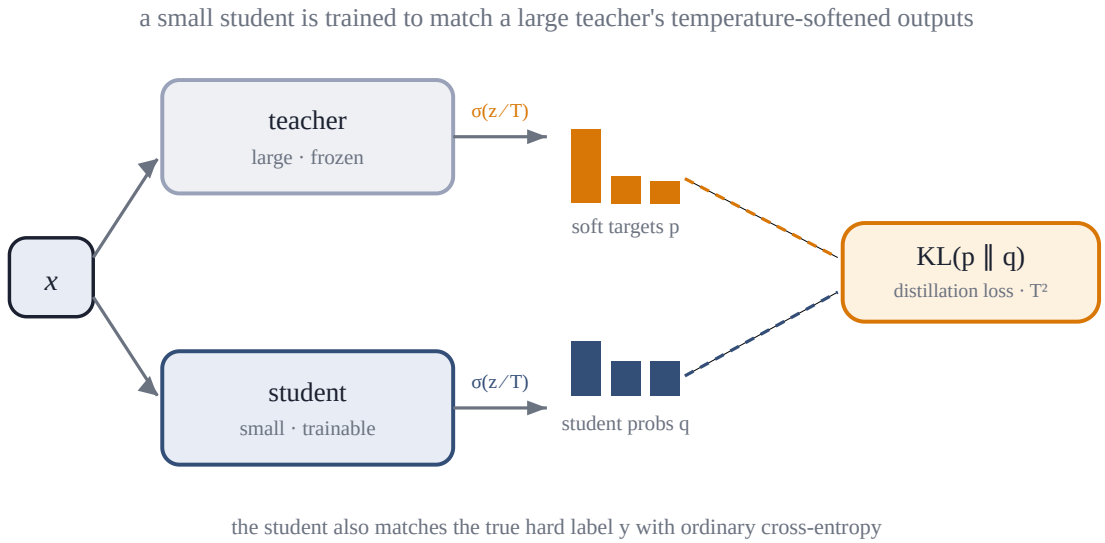
$$p_i(T) = \frac{\exp(z_i / T)}{\Sigma_j \exp(z_j / T)}$$

At T = 1 this is the ordinary softmax. Raising T above 1 flattens the distribution and lifts the small probabilities into view, while $T \to \infty$ approaches the uniform distribution. The student is then trained to match the teacher's softened distribution through a Kullback–Leibler divergence, along with the usual hard-label term:

$$L = (1 - \lambda)\cdot CE(y, p_s(1)) + \lambda\cdot T^2\cdot KL(\, p_t(T) \,\|\, p_s(T)\,)$$

$p_t(T)$, $p_s(T)$	teacher and student class distributions, each softened at temperature T. The teacher's is a fixed target (no gradient flows into the teacher)
$KL(p \,\|\, q)$	$\Sigma_i\, p_i \log(p_i / q_i)$, the relative entropy. It is zero when the student matches the teacher, and positive otherwise
$CE(y, p_s)$	the ordinary cross-entropy against the true hard label y, which keeps the student faithful to the ground truth
λ	a mixing weight balancing imitation against the hard label. The T^2 factor rescales the distillation gradient, which otherwise shrinks like $1/T^2$, so the two terms stay comparable as T varies

Taken together, the objective says: *be right about the true label, and match the teacher's shades of doubt.* As T grows large, the KL term approaches a squared error between teacher and student logits, so distillation moves from matching probabilities to matching raw scores. The idea appears everywhere. DINO's self-distillation matches a momentum-averaged teacher (Chapter 20), and instruction-tuned language models are often distilled from larger ones. This works because imitating a soft distribution is a remarkably rich signal.



Knowledge distillation. The same input goes to a frozen teacher and a trainable student. Each set of logits is softened at temperature T into a probability distribution, and the student is pulled toward the teacher's soft targets by a KL divergence (scaled by T^2), while a separate cross-entropy keeps it faithful to the true label. The teacher's non-zero "wrong-class" probabilities, the dark knowledge, are the extra signal that a hard label lacks.

Worked example: softening a teacher and its KL to a student

Three classes. The teacher's logits are $z_t = [3.0, 1.0, 0.5]$. The student's are $z_s = [2.0, 1.0, 1.0]$. Use temperature $T = 2$.

1. **Teacher, T = 1 (sharp):** softmax([3, 1, 0.5]) = [**0.8214**, 0.1112, 0.0674]. Almost all the mass is on class 1.
2. **Teacher, T = 2 (softened):** divide logits by T → [1.5, 0.5, 0.25], exponentiate → [4.4817, 1.6487, 1.2840], sum 7.4144, so p_t = [**0.6045**, 0.2224, 0.1732]. The peak fell from 0.82 to 0.60 and the two "wrong" classes rose. This is the dark knowledge, now visible.
3. **Student, T = 2:** [2, 1, 1]/2 = [1.0, 0.5, 0.5] → exp [2.7183, 1.6487, 1.6487], sum 6.0157, so p_s = [0.4519, 0.2741, 0.2741].
4. **KL divergence** $KL(p_t \| p_s) = \Sigma\, p_{t,i} \ln(p_{t,i}/p_{s,i}) = 0.6045 \cdot \ln(1.3378) + 0.2224 \cdot \ln(0.8114) + 0.1732 \cdot \ln(0.6319) = 0.1759 - 0.0465 - 0.0795 =$ **0.0499** nats.

What the numbers mean: the student already leans toward class 1 but is under-confident there and over-spreads to classes 2 and 3, so the KL is small but non-zero. Its gradient pushes the student's logit gaps toward the teacher's. It teaches the student not just "class 1" but "class 1, and by the way class 3 is a little less likely than class 2," which is the relational knowledge that a hard label could never convey. Multiplying by $T^2 = 4$ restores the gradient magnitude that the $1/T^2$ softening removed, so this term pulls with a weight comparable to the hard-label cross-entropy.

Foundation models

Transfer learning is the economic engine behind the **foundation model** (Bommasani et al. 2021): a single large network, pretrained on broad data at enormous scale (almost always self-supervised, Chapter 20), then adapted to a wide range of downstream tasks. GPT and BERT for language (Chapter 17), CLIP and ViT for vision, and their multimodal successors are all

foundation models, and every technique in this chapter is one way to adapt them. The economics are the key point. Pretraining costs millions of dollars and runs once. Each downstream use is cheap: a frozen linear probe, a light fine-tune, a LoRA adapter, or, most striking, **in-context learning**. In in-context learning, a language model performs a completely new task from a few examples in its prompt with *no weight updates at all*. Here, transfer is accomplished entirely at inference time.

This concentration has two sides. Spreading the cost of one pretraining run across thousands of applications is why a small team can build a strong classifier in an afternoon. But **homogenization** means every downstream system inherits the same backbone's blind spots. So a flaw in the foundation spreads everywhere, a systemic risk the field is still learning to manage. Foundation models are transfer learning taken to its extreme: build the representation once, at great cost, and let everyone build on it.

Transfer learning in code

The two runnable cells make the two quantitative claims of the chapter concrete in NumPy. The first builds a LoRA layer and shows three things at once: that a zero-initialized update leaves the network's output exactly unchanged, that the update's rank never exceeds r, and that the parameter count is a tiny fraction of the full matrix. Press **Run**:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.random.seed(0)
np.set_printoptions(precision=4, suppress=True)

d, r, alpha = 1024, 8, 16          # weight is d x d; LoRA rank r; scale alpha
x  = np.random.randn(d)            # an input activation vector
W  = 0.02 * np.random.randn(d, d)  # frozen pretrained weight (never updated)
A  = 0.01 * np.random.randn(r, d)  # down-projection  (r x d)
B  = np.zeros((d, r))              # up-projection init to ZERO  (d x r)

# LoRA forward: (W + (alpha/r) B@A) @ x   vs   the base   W @ x
y_base = W @ x
y_lora = (W + (alpha / r) * (B @ A)) @ x
```

```
print("at init  max|y_lora - y_base| =", np.max(np.abs(y_lora - y_base)))  # exactly 0.0

# after a few training steps B is small but non-zero:
B  = 0.01 * np.random.randn(d, r)
dW = (alpha / r) * (B @ A)                       # the effective weight update
print("rank(B@A) =", np.linalg.matrix_rank(B @ A), " (<= r =", r, ")")   # 8
print("||dW|| / ||W|| =", round(np.linalg.norm(dW) / np.linalg.norm(W), 4))
# small: 0.0281

full = d * d
lora = 2 * d * r
print("full params  =", full)                    # 1048576
print("LoRA params  =", lora, "=", round(100 * lora / full, 2), "% of full")
# 16384 = 1.56 %
```

The update starts at exactly zero (so fine-tuning begins from the untouched pretrained function), its rank is 8 even though it is a 1024×1024 matrix, and its whole parameter count is 1.56% of the dense weight. This is the worked example, reproduced by machine. The second cell demonstrates temperature softening and the distillation KL. It softmaxes the teacher's logits at T = 1 and T = 2, softmaxes the student at T = 2, and computes the KL divergence between teacher and student:

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)
np.set_printoptions(precision=4, suppress=True)

def softmax_T(logits, T):                    # temperature-scaled softmax
    z = np.asarray(logits) / T
    z = z - z.max()                          # stabilize before exp
    e = np.exp(z)
    return e / e.sum()

teacher = np.array([3.0, 1.0, 0.5])          # teacher logits over 3 classes
student = np.array([2.0, 1.0, 1.0])          # student logits (being trained)

pt1 = softmax_T(teacher, 1.0)                # teacher at T = 1 (sharp)
pt2 = softmax_T(teacher, 2.0)                # teacher at T = 2 (softened)
```

```
print("teacher T=1:", pt1, " max =", round(float(pt1.max()), 4))   # 0.8214
print("teacher T=2:", pt2, " max =", round(float(pt2.max()), 4))   # 0.6045
-> T>1 softens

# KL( teacher_soft || student_soft ) at T = 2  (the distillation objective)
T = 2.0
p = softmax_T(teacher, T)
q = softmax_T(student, T)
kl = np.sum(p * np.log(p / q))
print("student T=2:", q)
print("KL(teacher||student) at T=2 =", round(float(kl), 4), "nats")  #
0.0499

# raising T raises the entropy of the target -> a softer, more informative
signal
H1 = -np.sum(pt1 * np.log(pt1)); H2 = -np.sum(pt2 * np.log(pt2))
print("entropy T=1 =", round(float(H1), 4), " entropy T=2 =",
round(float(H2), 4))  # 0.5876 -> 0.9423
```

Softening drops the teacher's peak probability from 0.82 to 0.60 and raises its entropy from 0.59 to 0.94 nats, putting more mass on the informative wrong classes. The PyTorch code below shows the two workflows you would actually run: freezing a pretrained backbone behind a fresh head with discriminative learning rates, and a minimal `LoRALinear` module that trains only its low-rank factors.

PyTorch copy → Colab

```
import torch
import torch.nn as nn
import torchvision.models as models

# ---- 1. Feature extraction: freeze the backbone, train a NEW head ----
net = models.resnet50(weights="IMAGENET1K_V2")     # pretrained backbone
for p in net.parameters():
    p.requires_grad = False                        # freeze every backbone
weight
net.fc = nn.Linear(net.fc.in_features, 10)         # fresh head: 2048 -> 10
classes
# only net.fc.weight and net.fc.bias now have requires_grad = True
opt = torch.optim.Adam(
    filter(lambda p: p.requires_grad, net.parameters()), lr=1e-3)
```

```
# ---- 2. Fine-tuning with DISCRIMINATIVE (layer-wise) learning rates ----
opt = torch.optim.Adam([
    {"params": net.layer1.parameters(), "lr": 1e-5},   # early / general:
tiny LR
    {"params": net.layer4.parameters(), "lr": 1e-4},   # late  / specific:
larger
    {"params": net.fc.parameters(),     "lr": 1e-3},   # new head: largest
])

# ---- 3. A minimal LoRA-adapted linear layer ----
class LoRALinear(nn.Module):
    def __init__(self, base: nn.Linear, r=8, alpha=16):
        super().__init__()
        self.base = base                                # pretrained Linear,
frozen
        self.base.weight.requires_grad = False
        if base.bias is not None:
            self.base.bias.requires_grad = False
        d_in, d_out = base.in_features, base.out_features
        self.A = nn.Parameter(torch.randn(r, d_in) * 0.01)  # down-
projection  (r x d_in)
        self.B = nn.Parameter(torch.zeros(d_out, r))        # up-projection
init 0 (d_out x r)
        self.scale = alpha / r
    def forward(self, x):                               # W x  +  (alpha/r)
B A x
        return self.base(x) + self.scale * ((x @ self.A.t()) @ self.B.t())

layer = LoRALinear(nn.Linear(1024, 1024), r=8, alpha=16)
trainable = sum(p.numel() for p in layer.parameters() if p.requires_grad)
print(trainable)          # 16384  =  2 * 1024 * 8   (the frozen 1,048,576
weights train nothing)
```

In this app's engine

This chapter has no dedicated lab, but its core ideas are ones the app already lets you watch. Freezing a backbone is nothing more than switching off the backward pass for a set of layers. Turn off the amber δ flow of the Backprop lab (Chapter 8) into every layer but the last, and you are doing feature extraction. The gradients simply stop at the frozen boundary. And the temperature-softened softmax of distillation is the very same normalization $e^{z_i/T} / \Sigma_j e^{z_j/T}$ the

Playground's softmax output uses to turn logits into class probabilities. Set its temperature above one in the Math Log, and you can watch the teacher's dark knowledge emerge as the distribution flattens.

Assumptions & limitations

Transfer learning assumes source and target share structure. When they do not, the pretrained features are useless or worse, and negative transfer makes a random start the better option. Fine-tuning assumes you can afford the careful use of a small learning rate. Too large a rate, or unfreezing everything at once, destroys the pretrained weights or catastrophically forgets the source. PEFT assumes the useful adaptation is small: LoRA assumes the update is low-rank, and on tasks needing a large, high-rank change it does worse than full fine-tuning. Distillation assumes a good teacher whose soft targets are trustworthy. A confidently wrong teacher transfers its errors faithfully. Distillation also still needs data (often unlabeled) to run through it. None of these methods invents capability the backbone lacks. They redirect and compress what is already there, and inherit whatever biases the pretraining data carried. So evaluate on the target distribution: a backbone that transfers very well to one domain can fail silently on a neighboring one.

Why & when this matters

Use transfer learning whenever labels or compute are scarce relative to the task. This is almost always the case, outside the largest labs. If your target data is small and close to a known pretraining domain: freeze the backbone and train a head (feature extraction). If it is larger, or in a different domain: fine-tune with small, discriminative learning rates and gradual unfreezing. If you need many tasks from one model, or you have a huge model: use PEFT (LoRA by default), so each task is a tiny swappable change on top of one frozen backbone. If you need a fast deployable model from a strong but bulky one:

distill. The decision is always the same trade-off: how much of the pretrained representation to keep fixed versus let move. This is set by how much data you have and how far your task sits from the source.

In the real world

Transfer is the default, not the exception, in deployed deep learning. Nearly every production vision system fine-tunes an ImageNet- or self-supervised-pretrained backbone (Chapter 14) rather than training from scratch. Medical-imaging teams adapt such backbones to X-rays and pathology slides with a few thousand labels. In language, the whole post-2018 workflow is pretrain-then-adapt. BERT and GPT backbones (Chapter 17) are fine-tuned or LoRA-adapted for search, moderation, and support, and open-weight models are customized on a single GPU because LoRA shrinks the job to megabytes. Distillation is used wherever latency and memory matter. DistilBERT keeps most of BERT's accuracy at 40% of the size, and on-device assistants run students distilled from server-scale teachers. Foundation models push this further: one pretrained network, adapted a thousand ways, now supports vision, language, speech, code, and biology.

Exercises

1. You freeze a ViT-B backbone (about 86 million parameters) and attach a new linear head mapping its 768-dimensional feature to 100 target classes. How many trainable parameters does feature extraction introduce, and what fraction of the model is that? MATH

2. A large transformer applies LoRA at rank r = 16 to 96 weight matrices, each of shape 4096×4096. Compute the trainable-parameter count for full fine-tuning of those matrices versus LoRA, and the fraction. MATH

3. In NumPy, build a LoRA update at d = 512, r = 4 and confirm three facts: the parameter count is $2 \cdot d \cdot r$, the rank of BA is at most r, and the update is small when the initialization is small. CODE

4. Prove that $\Delta W = BA$ with $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times d}$ has rank at most r, and give a concrete case where the rank is strictly less than r. MATH

5. In NumPy, vary the distillation temperature T over $\{1, 2, 5, 100\}$ for the logits $[4, 2, 1, 0]$ and show that the softened distribution approaches the uniform as T grows. CODE

6. The distillation loss multiplies its KL term by T^2. Given that the gradient of the softened cross-entropy with respect to the student logits scales like $1/T^2$, explain what the T^2 factor does and why it matters when mixing with the hard-label term. MATH

7. You fine-tune a pretrained model on task B and its accuracy on the original task A collapses. Name this failure, give two ways to reduce it, and explain why LoRA is naturally resistant to it. CONCEPT

8. Your source is ImageNet (color natural photos). Your target is grayscale chest X-rays with a few thousand labels. Which layers would you freeze versus fine-tune, and what learning-rate strategy would you use? Justify each choice with the transferability of features. CONCEPT

Deep reinforcement learning

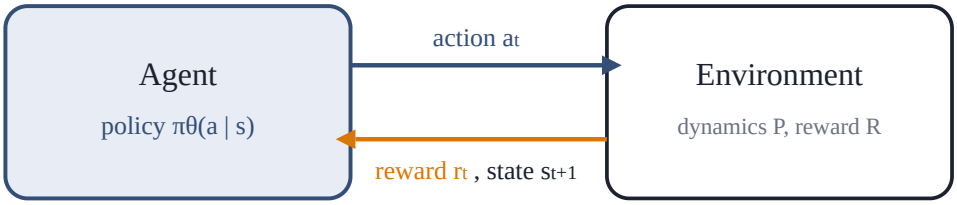
> *Every network so far has learned from a teacher who already knew the answer: a labeled image, a translated sentence, a target value to regress.* ***Reinforcement learning*** *removes that teacher. An agent is placed in a world it does not understand and is allowed to act. It is told only whether things are going well, through a single scalar called* ***reward****. This signal comes only sometimes, and often much later. From that thin signal it must discover which actions were worth taking. This chapter builds the theory that makes such learning possible (Markov decision processes, returns, value functions, the Bellman equation). It then builds the deep-learning methods that scale it: Deep Q-Networks and policy gradients. It ends where this now matters most for this book: aligning a language model (Chapter 17) to human preferences.*

The agent–environment loop

Picture learning to ride a bicycle. Nobody hands you the correct handlebar angle for every instant. You push, you wobble, you sometimes fall, and a vague sense of "that felt stable" or "that hurt" slowly builds a skill. Reinforcement learning formalizes this trial-and-error loop. At each discrete time step t the **agent** observes the **state** s_t, chooses an **action** a_t, and the **environment** responds with a scalar **reward** r_t and a new state s_{t+1}. Then the cycle repeats. Nothing tells the agent the *right* action. It only ever sees the consequences of the actions it took. This difference separates reinforcement learning from the supervised training of every earlier chapter.

Two features make the problem hard and interesting. First, reward is often **delayed**: the move that loses a chess game may have been the mistake twenty moves earlier. So the agent faces a **credit-assignment** problem across time. This is the time-based version of the credit assignment that backpropagation solved across layers (Chapter 8). Second, the agent's actions **change the**

data it sees next. A cautious policy never visits the states a bold one would, so the training distribution is not fixed. Instead, it is created by the agent's own behavior. Learning and data collection are linked in a way they never were for a static dataset.



the perception–action cycle, repeated at every time step t

The loop that defines the whole field. The agent picks an action from its policy. The environment returns a reward and the next state, and the agent does not know how the environment works. All of reinforcement learning is a method for turning the amber reward stream, and nothing else, into a better policy.

Markov decision processes

The standard mathematical framework for this loop is the **Markov decision process** (MDP), the tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$. Its defining assumption is the **Markov property**: the next state and reward depend only on the current state and action, not on the entire history. In other words, "the future is independent of the past given the present." This is less limiting than it sounds, because we are free to pack whatever history matters into the state itself (the last four game frames, the full board position). Given the Markov property, everything the agent needs to make a good decision is contained in s_t.

$$P(s' \mid s, a) = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a) \qquad R(s, a) = \mathbb{E}[\, r_t \mid s_t = s, a_t = a \,]$$

The agent's behavior is a **policy** π, a mapping from states to a distribution over actions, $\pi(a \mid s) = \Pr(a_t = a \mid s_t = s)$. A policy may be deterministic (always the same action in a state) or stochastic (a distribution, essential for exploration and for the policy-gradient methods below). The goal is not to maximize the next reward but the **return**, which is the total discounted reward from now on:

$$G_t = r_t + \gamma\, r_{t+1} + \gamma^2\, r_{t+2} + \cdots = \Sigma_{k\geq 0}\, \gamma^k\, r_{t+k}$$

s, a, r	state (in $\mathcal{S}$), action (in $\mathcal{A}$), and scalar reward. s′ is the next state s_{t+1}
P(s′|s,a)	transition dynamics: the environment's (usually unknown) probability of landing in s′
R(s,a)	expected immediate reward for taking a in s. This is the only feedback the agent ever receives
π(a|s)	the policy: the agent's action distribution in each state. This is the object we ultimately want to improve
G_t	the return: the discounted sum of all future rewards from step t. This is the quantity to be maximized
$\gamma \in [0, 1)$	discount factor: how much a reward k steps away is worth now (γ^k)

The **discount factor** γ is valuable for three reasons. Mathematically, for bounded rewards it keeps the infinite sum finite (a geometric series), so the return is always well defined. Behaviorally, it tunes the agent's horizon: γ = 0 makes it short-sighted, caring only about r_t. As γ → 1 it weighs the distant future almost as heavily as the present. And economically, it mirrors the real preference for reward sooner rather than later: a pound today beats a pound next year. Typical values sit at γ = 0.9 to 0.99. The effective horizon is roughly $\frac{1}{1-\gamma}$ steps, so γ = 0.99 plans about a hundred steps ahead.

Value functions and the Bellman equation

The return G_t is a property of a whole trajectory and is random. It depends on every action and every stochastic transition still to come. To reason about states before acting, we take its expectation under a policy. The **state-value function** $V^\pi(s)$ is the expected return from starting in s and following π afterward. The **action-value function** $Q^\pi(s, a)$ fixes the first action a and then follows π:

$$V^\pi(s) = \mathbb{E}_\pi[\, G_t \mid s_t = s \,] \qquad Q^\pi(s, a) = \mathbb{E}_\pi[\, G_t \mid s_t = s, a_t = a \,]$$

These two are the main tools of the field. V^π answers "how good is this situation?" Q^π answers the more useful question "how good is this move?" Their difference, which is how much better one action is than the state's average, is the **advantage** $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$. It reappears

at the heart of policy gradients. Because $V^{\pi}(s)$ averages Q^{π} over the policy's actions, $V^{\pi}(s) = \Sigma_a \pi(a|s)\, Q^{\pi}(s, a)$.

The single most important fact about value functions is that they are **self-referential**. Split the return into the immediate reward and the rest: $G_t = r_t + \gamma\, G_{t+1}$. Taking expectations gives the **Bellman expectation equation**, which links a state's value to the values of the states that follow it:

$$V^{\pi}(s) = \Sigma_a\, \pi(a|s)\, \Sigma_{s'}\, P(s'|s,a)\, [\, R(s,a) + \gamma\, V^{\pi}(s')\,]$$

The best any policy can do is described by the **optimal** value functions $V^*(s) = \max_{\pi} V^{\pi}(s)$ and $Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$. They satisfy the **Bellman optimality equation**, which is the same recursion but with the policy average replaced by a maximum over actions:

$$Q^*(s, a) = \mathbb{E}_{s'}[\, r + \gamma \max_{a'} Q^*(s', a')\,] \qquad V^*(s) = \max_a Q^*(s, a)$$

Once Q^* is known the optimal policy is simple: in each state take $\pi^*(s) = \operatorname{argmax}_a Q^*(s, a)$. The Bellman optimality equation is therefore the goal: solve it, and the control problem is solved. Its derivation is short but worth seeing, because the chapter's algorithms are all approximate ways of solving it.

1. By definition $Q^{\pi}(s, a) = \mathbb{E}[G_t \mid s_t{=}s, a_t{=}a]$ and the return unrolls one step: $G_t = r_t + \gamma\, G_{t+1}$.
2. Linearity of expectation splits it: $Q^{\pi}(s,a) = \mathbb{E}[r_t \mid s,a] + \gamma\, \mathbb{E}[G_{t+1} \mid s,a] = R(s,a) + \gamma\, \mathbb{E}_{s'}[\, V^{\pi}(s')\,]$, since after the transition the agent is in s' and follows π.
3. Substituting $V^{\pi}(s') = \Sigma_{a'}\, \pi(a'|s')\, Q^{\pi}(s', a')$ gives the Bellman *expectation* backup for Q^{π}.
4. The optimal policy would never average over a'. It would pick the best. Replacing $\Sigma_{a'}\, \pi(a'|s')(\cdot)$ with $\max_{a'}(\cdot)$ gives the Bellman *optimality* equation $Q^*(s,a) = R(s,a) + \gamma\, \mathbb{E}_{s'}[\, \max_{a'} Q^*(s',a')\,]$.

Turning the equation into an assignment gives the **Bellman backup**, the basic operation of nearly every value-based method: replace Q(s, a) with the right-hand side computed from the current estimates of the states that follow, $Q(s,a) \leftarrow R(s,a) + \gamma\, \mathbb{E}_{s'}[\max_{a'} Q(s',a')]$. Sweeping this backup over all states is called **value iteration**. It provably converges to Q* when P and R are known, because the backup is a γ-contraction: each sweep shrinks the worst-case error by a factor of γ toward the unique fixed point. But in almost every real problem P and R are *unknown* and $\mathscr{S}$ is far too large to sweep. The rest of the chapter backs up values from *sampled experience* instead. It then replaces the table Q(s, a) with a neural network.

Exploration versus exploitation

An agent estimating values from its own experience faces a dilemma with no equivalent in supervised learning. To act well it should **exploit**: take the action its estimates say is best. But those estimates are only as good as the data behind them, and the only way to improve them is to **explore**: try worse-looking actions in case they are secretly better. Pure exploitation settles on whatever looked good first, which may be far from optimal. Pure exploration learns about the world but never uses the knowledge. Every method must balance the two.

The simplest balance, and a remarkably long-lasting one, is **ε-greedy** selection: with probability $1 - \varepsilon$ take the greedy action, and with probability ε take a uniformly random one. A small ε (say 0.1) spends most steps exploiting while keeping a small amount of exploration to keep the estimates accurate and escape early mistakes. A common improvement slowly *reduces* ε over time, exploring broadly at first and committing as confidence grows. The clearest test case for the trade-off is the **multi-armed bandit**: a single state with k actions ("arms"), each paying a noisy reward from an unknown distribution. Here there is nothing to plan. There is only the question of how to split pulls between finding the best arm and exploiting it. The runnable bandit below shows ε-greedy resolving exactly this tension, with its average reward climbing as it closes in on the top arm.

Temporal-difference learning and Q-learning

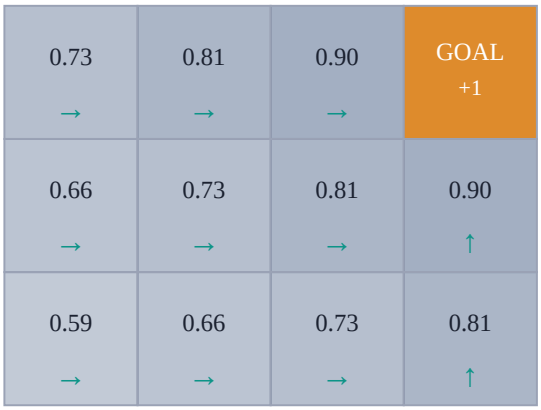
How can an agent improve Q from experience when it does not know P and so cannot compute the expectation $\mathbb{E}_{s'}$? The answer, and arguably the field's central idea, is **temporal-difference (TD) learning**: use the *observed* next state s′ as a single-sample estimate of that expectation, and move the old value toward the one-step target. After taking a in s and observing r and s′, form the **TD target** $r + \gamma \max_{a'} Q(s', a')$. This is the real reward blended with the agent's own estimate of what follows. Then move Q(s, a) a fraction of the way toward it. This is **Q-learning**:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

$\alpha \in (0, 1]$	step size (learning rate): the fraction of the TD error absorbed per update
$r + \gamma \max_{a'} Q(s', a')$	the TD target: a one-step, bootstrapped estimate of Q*(s,a)
$\delta = \text{target} - Q(s,a)$	the TD error, or surprise. Positive means the outcome beat expectation

Three properties make this update remarkable. It is **model-free**: it never needs P or R, only sampled transitions (s, a, r, s′). It **bootstraps**: the target uses the current estimate Q(s′, ·) rather than waiting for the episode to end, so learning happens online, at every step. And it is **off-policy**: because the target maximizes over a′, Q-learning estimates the *optimal* Q* no matter how exploratory the behavior that generated the data. So the agent can learn the best policy while acting ε-greedily. Its fixed point, where the TD error vanishes in expectation, is exactly the Bellman optimality equation. With a decaying step size and enough exploration to visit every state–action pair, tabular Q-learning provably converges to Q*. The first runnable cell shows this on a chain of states, with the learned Q reproducing the closed-form optimum to three decimals.

learned values $V(s) = \gamma^n$ (shade) and greedy policy (arrows) → goal



deeper blue = higher value; every greedy arrow climbs the value gradient to the goal

A gridworld after learning. Each cell holds its state value $V(s) = \gamma$ raised to the number of steps to the goal (reward +1, $\gamma = 0.9$), and the teal arrow is the greedy action $\text{argmax}_a Q(s, a)$. Following the arrows always climbs the value surface toward the goal. This is a picture of a solved Bellman equation, and the two-dimensional version of the chain solved in code below.

Deep Q-Networks

Tabular Q-learning stores one number per state–action pair. This is hopeless when a state is an Atari screen or a robot's continuous joint angles: the table is extremely large and, worse, learns nothing about a state from its neighbors. The fix is the theme of the whole book: replace the table with a **function approximator**. A **Deep Q-Network** (DQN) is a network $Q(s, a; \theta)$, a convolutional net (Chapter 13) on pixels or an MLP on features, that *generalizes* across states and outputs a Q-value for every action from the state alone. It is trained by turning the TD update into a regression: minimize the squared TD error between the network's prediction and the bootstrapped target.

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right]$$

$Q(s,a;\theta)$	the online Q-network with parameters θ, trained by gradient descent
θ^-	parameters of the **target network**, a periodically-frozen copy of θ
$\mathcal{D}$	the **replay buffer**: a large memory of past transitions (s,a,r,s')

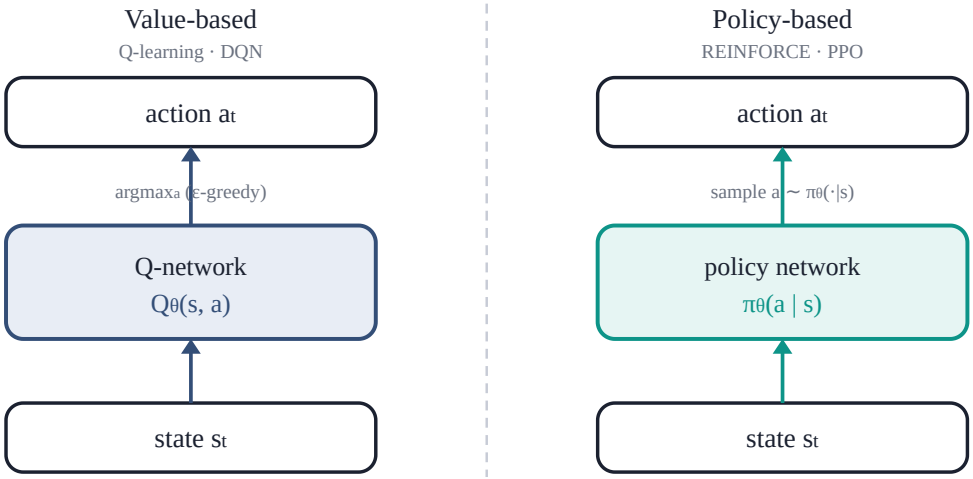
Simply fitting a network to a moving TD target is unstable, and DQN's contribution (Mnih et al., 2015, the first to reach human-level play on dozens of Atari games from raw pixels) was two simple stabilizers. **Experience replay** stores every transition in a buffer $\mathcal{D}$ and trains on *random minibatches* drawn from it. This breaks the severe temporal correlation between consecutive frames. Gradient descent (Chapter 7) assumes roughly independent samples, and a sequential stream badly violates this. Replay also lets each experience be reused many times, improving sample efficiency. The **target network** addresses a more subtle problem: because the same weights θ appear in both the prediction and the target, an update chases a target that moves the instant you step. This is a feedback loop that can diverge. Freezing the target's weights at θ^- and refreshing them only every few thousand steps gives the regression a *stationary* target. This is exactly the setting supervised learning works well in. The gradient itself is ordinary backpropagation (Chapter 8) through $Q(s, a; \theta)$, the target treated as a constant (no gradient flows through θ^-).

Two improvements are worth knowing by name. **Double DQN** fixes a systematic *over-estimation*: because $\max_{a'} Q(s', a'; \theta^-)$ both selects and evaluates the next action with the same noisy network, it favors actions over-estimated by chance. Separating the two reduces the bias measurably: the online network picks the argmax action, and the target network scores it. **Dueling DQN** splits the network into two heads that estimate the state value $V(s)$ and the per-action advantage $A(s, a)$. These recombine as $Q(s, a) = V(s) + (A(s, a) - \text{mean}_a A(s, a))$. This learns which states are valuable without having to learn the effect of every action, and it helps most when actions barely differ. Both are small changes on the same TD-error loss, and both appear in essentially every modern value-based agent.

Policy-gradient methods

Value-based methods learn a value function and read a policy out of it by argmax. This indirect approach has costs: the argmax is awkward in continuous action spaces (imagine maximizing over every torque), it gives a deterministic policy where a stochastic one may be needed, and a tiny change in a value can flip the chosen action suddenly. **Policy-gradient** methods take the direct approach: parameterize the policy as a network $\pi_\theta(a \mid s)$, which is a softmax over logits

for discrete actions or the mean and variance of a Gaussian for continuous ones. Then adjust θ by gradient *ascent* on the expected return $J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]$. No table, no argmax, and the policy can be as stochastic as the problem demands.



Two approaches. A value-based agent learns Q(s, a) and derives its actions by maximizing over them. A policy-based agent learns the action distribution $\pi_\theta(a \mid s)$ and samples from it directly. The first works best with discrete actions and reuses data off-policy. The second handles continuous actions and stochastic policies naturally. Actor–critic methods, next, learn both at once.

The problem is that the thing we want the gradient of is an expectation over trajectories whose distribution *itself* depends on θ. You cannot simply differentiate a sample. The **log-derivative trick** (also called the score-function or REINFORCE estimator) turns it into an expectation we *can* sample, using the identity $\nabla_\theta p_\theta = p_\theta \nabla_\theta \log p_\theta$:

1. Write the objective as an integral over trajectories: $J(\theta) = \mathbb{E}_{\tau\sim p_\theta}[R(\tau)] = \int p_\theta(\tau)\, R(\tau)\, d\tau$.
2. Differentiate and push the gradient inside: $\nabla_\theta J = \int \nabla_\theta\, p_\theta(\tau)\, R(\tau)\, d\tau$.
3. Apply the identity $\nabla_\theta p_\theta = p_\theta \nabla_\theta \log p_\theta$ to recover an expectation: $\nabla_\theta J = \int p_\theta(\tau)\, \nabla_\theta \log p_\theta(\tau)\, R(\tau)\, d\tau = \mathbb{E}_\tau[\, \nabla_\theta \log p_\theta(\tau) \cdot R(\tau)\,]$.
4. The trajectory's log-probability is $\log p_\theta(\tau) = \Sigma_t \log \pi_\theta(a_t|s_t) + \Sigma_t \log P(s_{t+1}|s_t,a_t)$. The dynamics term is independent of θ, so its gradient is zero. The **unknown environment drops out**.

5. What remains, using that an action at time t can only affect rewards from t onward (causality), is the **REINFORCE** estimator:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\Sigma_t \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot G_t \right]$$

Read this and the algorithm becomes clear. Run the policy to sample a trajectory. For every action taken, compute the gradient of its log-probability and scale it by the return G_t that followed. Then average and take a step. The effect is intuitive: $\nabla_\theta \log \pi_\theta(a|s)$ points in the direction that makes action a *more* likely in state s, and the multiplier G_t pushes up the probability of actions followed by high return and pushes down those followed by low return. The policy learns to repeat what worked. There is no environment model and no value table, just weighted maximum likelihood on its own good behavior.

Pure REINFORCE works but learns slowly, because its gradient has very high **variance**: G_t is a noisy sum of many random rewards, and scaling every gradient by such an unstable number makes the update jump around. The standard fix is a **baseline**: subtract from G_t any state-only function b(s), which leaves the gradient unbiased (since $\mathbb{E}[\nabla_\theta \log \pi_\theta(a|s)] = 0$ for any such b) while shrinking its variance. The best baseline is V(s), and the multiplier then becomes the **advantage** $A(s, a) = Q(s, a) - V(s)$: "how much better was this action than the state's average?" Actions are now reinforced in proportion to how much they *beat* the baseline rather than by the raw return. This centers the signal and stabilizes learning.

Actor–critic and PPO

Using the value function as a baseline invites an obvious idea: learn it too. **Actor–critic** methods run two networks together: an **actor** $\pi_\theta(a|s)$ that chooses actions, and a **critic** $V_\varphi(s)$ that estimates their value and supplies the baseline. The critic is trained by TD, exactly as in DQN. The actor is trained by the policy gradient with the advantage estimated from the critic. The simplest advantage estimate is the one-step TD error itself, which is an unbiased estimate of the advantage:

$$A(s_t, a_t) \approx \delta_t = r_t + \gamma V_\varphi(s_{t+1}) - V_\varphi(s_t) \qquad \nabla_\theta J = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \delta_t \right]$$

This is **A2C** (Advantage Actor–Critic), the standard on-policy algorithm: the critic bootstraps, so the actor gets a low-variance signal at every step instead of waiting for the episode's full return. The actor supplies exploration and the differentiable policy. The critic supplies a stable target. This combines the two approaches of the previous figure into one loop.

On-policy gradients share one dangerous setting: the step size. Because the data was collected by the current policy, a single too-large update can push the policy into a region where its own samples are useless and training collapses. There is no replay buffer to fall back on. **Proximal Policy Optimization** (PPO) is the now-standard solution, cautious by design: improve the policy but do not let it move too far from the version that collected the data. Writing the probability ratio $\rho_t(\theta) = \dfrac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, PPO maximizes a **clipped** objective:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(\rho_t A_t,\ \text{clip}(\rho_t, 1-\varepsilon, 1+\varepsilon)\, A_t\right)\right]$$

The idea is easier than it looks. When an action was good ($A_t > 0$), the objective rewards raising its probability. But the clip limits the ratio at $1 + \varepsilon$, so once the action is already $1 + \varepsilon$ times more likely than before, the gradient vanishes and the policy stops rushing toward it. When an action was bad ($A_t < 0$), the symmetric clip at $1 - \varepsilon$ limits how fast its probability is driven down. Either way the policy is held in a trust region near $\pi_{\theta_{\text{old}}}$, making each update safe enough to reuse for several gradient steps on the same batch. This is a large efficiency win over basic policy gradients. That combination of simplicity and robustness is why PPO became the default for continuous control and, as we are about to see, for aligning language models.

Reinforcement learning in code

Two runnable NumPy cells make the tabular theory concrete. Both import only NumPy, are seeded, and print numbers you can check against the equations. The first runs **Q-learning** on a six-state chain: state 5 is the goal, stepping into it pays +1, and the optimal policy is "always go right." After a few hundred episodes the learned Q-values for the right action reproduce the closed form $Q^*(s, \text{right}) = \gamma^{4-s}$, the discounted distance to the goal, and the greedy policy is right everywhere.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.random.seed(0)

# ---- a 6-state chain (corridor). State 5 is the goal (terminal). ----
# actions: 0 = left, 1 = right.  Reward +1 only on stepping INTO the goal.
N, GOAL = 6, 5
def step(s, a):                                  # deterministic dynamics
    s2 = min(s + 1, N - 1) if a == 1 else max(s - 1, 0)
    r  = 1.0 if s2 == GOAL else 0.0
    return s2, r, (s2 == GOAL)

Q = np.zeros((N, 2))                             # value table, one row per
state
alpha, gamma, eps = 0.5, 0.9, 0.1
for episode in range(400):
    s = np.random.randint(0, GOAL)               # random non-goal start
state
    for t in range(50):                          # cap the episode length
        a = np.random.randint(2) if np.random.rand() < eps else
int(np.argmax(Q[s]))
        s2, r, done = step(s, a)
        target = r + (0.0 if done else gamma * Q[s2].max())   # TD target
        Q[s, a] += alpha * (target - Q[s, a])    # tabular Q-learning
update
        s = s2
        if done: break

opt = 0.9 ** (4 - np.arange(GOAL))               # closed form Q*(s, right)
= gamma^(4-s)
print("Q(s, right) =", np.round(Q[:GOAL, 1], 4))  # -> [0.6559 0.729  0.81
0.9    1.    ]
print("optimal     =", np.round(opt, 4))         # -> [0.6561 0.729  0.81
0.9    1.    ]
greedy = np.argmax(Q, axis=1)[:GOAL]             # greedy action in each
non-goal state
print("greedy path :", "".join("R" if a == 1 else "L" for a in greedy))  # -
> RRRRR
```

The second cell is the **ε-greedy bandit**: five arms with fixed, deliberately close true values, each paying a reward with unit-variance noise. Estimating each arm's value by a running sample average and acting ε-greedily, the agent's average reward climbs from near the random-pick baseline toward the best arm's value of 1.0 as its estimates improve. This is exploration paying for exploitation.

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(2)

# ---- 5-armed bandit with fixed, deliberately close true values ----
q_star = np.array([0.2, -0.3, 0.5, 0.0, 1.0])     # arm 4 is best (value 1.0)
k, best = len(q_star), int(np.argmax(q_star))

Q  = np.zeros(k); Nc = np.zeros(k)                 # value estimates and pull counts
eps, steps = 0.1, 2000
rewards = np.zeros(steps)
for t in range(steps):
    a = np.random.randint(k) if np.random.rand() < eps else int(np.argmax(Q))
    r = q_star[a] + np.random.randn()              # noisy reward, unit variance
    Nc[a] += 1
    Q[a]  += (r - Q[a]) / Nc[a]                    # incremental sample-average update
    rewards[t] = r

run = np.cumsum(rewards) / (np.arange(steps) + 1)  # running average reward
for c in [10, 50, 200, 500, 2000]:
    print("after %4d steps: avg reward %.3f" % (c, run[c - 1]))
#   -> 10:-0.007  50:0.482  200:0.752  500:0.791  2000:0.889  (rises toward 1.0)
print("random baseline mean(q*) = %.3f, best value = %.3f" % (q_star.mean(), q_star[best]))
print("final estimates Q =", np.round(Q, 3), " greedy arm =", int(np.argmax(Q)))  # arm 4
```

Finally, a PyTorch listing (shown, not run) gives one training step of a **Deep Q-Network** with a target network and the Double-DQN target. It is the concrete version of the squared-TD-error loss above, ready to drop into a training loop with a replay buffer.

PyTorch copy → Colab

```
import torch, torch.nn as nn, torch.nn.functional as F

def q_net(obs_dim, n_act):                          # a small MLP Q-network
    return nn.Sequential(nn.Linear(obs_dim, 128), nn.ReLU(),
                         nn.Linear(128, 128), nn.ReLU(),
                         nn.Linear(128, n_act))     # one Q-value per
action

online = q_net(4, 2)                                # e.g. CartPole: 4 obs,
2 actions
target = q_net(4, 2)
target.load_state_dict(online.state_dict())         # start the target as a
copy
opt = torch.optim.Adam(online.parameters(), lr=1e-3)
gamma = 0.99

def dqn_step(batch):                                # batch from the replay
buffer
    s, a, r, s2, done = batch                       # shapes: (B,4)(B,)(B,)
(B,4)(B,)
    q = online(s).gather(1, a[:, None]).squeeze(1)    # Q(s,a) for the
actions taken
    with torch.no_grad():                           # no gradient through
the target
        a_star = online(s2).argmax(1)               # Double DQN: online
picks a'
        q_next = target(s2).gather(1, a_star[:, None]).squeeze(1)  # target
scores it
        y = r + gamma * q_next * (1.0 - done)         # TD target (0 past a
terminal s')
    loss = F.smooth_l1_loss(q, y)                     # Huber loss on the TD
error
    opt.zero_grad(); loss.backward(); opt.step()      # one gradient step
(Chapter 8)
    return loss.item()
```

```
# every C steps, refresh the frozen target:
target.load_state_dict(online.state_dict())
```

From games to alignment: RLHF

Reinforcement learning's most famous successes are games, where the reward is clear and the environment is cheap to simulate. **TD-Gammon** reached expert backgammon by TD learning in 1992. **DQN** learned dozens of Atari games from pixels with the single loss above. And **AlphaGo** (2016), then **AlphaZero** (2017), mastered Go, chess, and shogi at superhuman level by pairing a policy-and-value network with Monte-Carlo tree search and training purely by **self-play**, generating its own curriculum with no human games at all. Beyond games, the same methods drive **robotics and control**: locomotion and manipulation policies, data-center cooling, chip floorplanning, and tokamak plasma control are all reinforcement-learning controllers, usually trained in simulation and transferred to hardware (the "sim-to-real" problem, related to transfer learning, Chapter 22).

But the application most relevant to this book is **alignment**. A pretrained language model (Chapter 17) is fluent, yet by default it is not helpful or harmless. And "a good, safe, honest answer" is exactly the kind of goal that is easy to *judge* but impossible to write down as a labeled target. This is precisely the situation reinforcement learning was built for. **RLHF** (reinforcement learning from human feedback) casts it as an MDP in three stages. First, collect human **preference** comparisons: annotators see two responses to a prompt and record which they prefer. Second, fit a **reward model** r_ψ(prompt, response) to those comparisons, so a learned network supplies the scalar reward no human could specify by hand. Third, treat the language model as a **policy**: the prompt is the state, and each generated token is an action. Optimize it with **PPO** to maximize that reward, with a KL penalty holding it close to the original so it does not "reward-hack" into low-quality text. The clipped objective, the advantage, and the actor–critic loop of the previous sections are not analogies here. They are the literal algorithm that turned a next-token predictor into an assistant you can talk to. The simpler **DPO** reaches a similar result by optimizing the preferences directly, skipping the reward model and RL loop. But the framing it replaces is this chapter's.

Worked example: returns and one Q-learning update, by hand

Part A: returns. Take the reward sequence $r = [r_0, r_1, r_2] = [0, 0, 1]$ with discount $\gamma = 0.9$, using $G_t = \Sigma_{k \geq 0} \gamma^k r_{t+k}$. Work backward from the last step:

1. $G_2 = r_2 = \mathbf{1}$ (nothing follows the last reward).
2. $G_1 = r_1 + \gamma r_2 = 0 + 0.9 \cdot 1 = \mathbf{0.9}$.
3. $G_0 = r_0 + \gamma r_1 + \gamma^2 r_2 = 0 + 0 + 0.81 \cdot 1 = \mathbf{0.81}$.

The single unit of reward is worth less the further you are from it in time: full value at the step it lands, 0.9 one step before, and 0.81 two steps before. This is the discount γ^k in action. Notice this is exactly the value profile of the chain in the code, where the goal reward decays by 0.9 per step of distance.

Part B: one Q-learning update. Suppose the agent is in state s (say chain state 3), takes the "right" action, receives $r = 0$, and lands in s' (state 4) whose best action currently has estimate $\max_{a'} Q(s', a') = 1.0$. Its current estimate is $Q(s, a) = 0.20$. Use $\alpha = 0.5$, $\gamma = 0.9$.

1. TD target: $r + \gamma \max_{a'} Q(s', a') = 0 + 0.9 \cdot 1.0 = \mathbf{0.90}$.
2. TD error: $\delta = \text{target} - Q(s, a) = 0.90 - 0.20 = \mathbf{0.70}$.
3. Update: $Q(s, a) \leftarrow 0.20 + 0.5 \cdot 0.70 = 0.20 + 0.35 = \mathbf{0.55}$.

What the numbers mean: the update moves $Q(s, a)$ from 0.20 a fraction $\alpha = 0.5$ of the way toward the TD target 0.90, landing at 0.55. It closes half the gap δ. It moves *toward* the target because the target $r + \gamma \max Q(s', \cdot)$ is a fresh, partly real estimate of the true $Q^*(s, a)$: it contains the actual reward r plus a discounted look at the best the agent believes it can do next. Repeating such updates drives $Q(s, a)$ to the fixed point where $\delta = 0$, namely $Q(s, a) = r + \gamma \max_{a'} Q(s', a') = 0.9$. This is the Bellman optimality equation, and exactly the value $Q^*(3, \text{right}) = \gamma^1 = 0.9$ that the runnable chain converges to.

Assumptions & limitations

The clean theory depends on the **Markov property** and on observing the state. When the state is only partially observed, the guarantees weaken and the problem becomes a far harder POMDP. Reinforcement learning is famously **sample-inefficient**: DQN needed tens of millions of frames per Atari game, because reward is sparse and the agent must stumble onto it before it can learn. This is why so much training happens in simulation. It is **unstable**: bootstrapping, function approximation, and off-policy data form the "deadly triad" that can make value estimates diverge. Also, tabular Q-learning's convergence proofs do *not* carry over to deep networks. And it is only as good as the **reward**: a poorly specified reward is optimized aggressively and literally, producing **reward hacking**. Examples include a boat-racing agent that spins in circles for bonus points instead of finishing, or a model that games its reward with flattering filler. Specifying reward correctly is the field's deepest practical and safety problem.

Why & when this matters

Use reinforcement learning when the problem is genuinely **sequential** and **evaluative**: decisions build on each other over time, actions change the world, and you can score outcomes but cannot demonstrate the optimal action to imitate. This describes control (robots, vehicles, energy systems), games and simulations, recommendation and bidding as sequential decisions, and preference-based fine-tuning of generative models. Prefer supervised learning when correct labels exist: if you can show the right answer, imitation is faster, more stable, and cheaper than learning from reward. And whenever a real reward is hard to define, work on the reward first, because an RL agent will optimize exactly what you wrote, not what you meant.

In the real world

The algorithms in this chapter run at the frontier of AI. Every major aligned language model, the assistants of Chapter 17, is fine-tuned with PPO-style RLHF or a preference method derived from it. This makes it arguably the most important real-world use of reinforcement learning on Earth. AlphaZero's self-play approach led to AlphaFold-adjacent search and to superhuman play in Go, chess, StarCraft, and Diplomacy. DeepMind's controllers manage data-center cooling and have stabilized plasma in a real tokamak. And robotics increasingly trains locomotion and manipulation in massively parallel simulators before transferring to hardware. The common thread is always the same loop of this chapter, namely state, action, reward, scaled up by a neural network that stands in for the value function or the policy.

Exercises

1. For the reward sequence r = [1, 0, 2, 0] with $\gamma = 0.5$, compute the returns G_3, G_2, G_1, G_0 using $G_t = \Sigma_{k \geq 0} \gamma^k r_{t+k}$. MATH

2. Carry out one Q-learning update for a non-terminal transition with $Q(s, a) = 1.0$, reward $r = 2$, $\max_{a'} Q(s', a') = 4.0$, $\alpha = 0.1$, $\gamma = 0.9$. Report the TD target, the TD error, and the new $Q(s, a)$. Then say what a smaller α would have done. MATH

3. Explain why Q-learning is called **off-policy**, and why that is possible even though the agent explores with ε-greedy actions. CONCEPT

4. DQN uses two stabilizers, experience replay and a target network. Explain the specific instability each one addresses. CONCEPT

5. Modify the chain Q-learning cell to add a step penalty of −0.05 on every non-goal transition (keep +1 for reaching the goal). Predict qualitatively what happens to the learned values, and confirm the greedy policy is still all-right. CODE

6. Derive why subtracting a state-only baseline b(s) from the return in the policy gradient leaves it **unbiased**. MATH

7. In the PPO clipped objective, an action had a *positive* advantage $A_t > 0$ and the new policy already makes it $1 + \varepsilon$ times as likely as the old one (so $\rho_t = 1 + \varepsilon$). What gradient does the objective provide for pushing ρ_t higher, and why is that the intended behavior? CONCEPT

8. Frame RLHF for a language model (Chapter 17) explicitly as an MDP: identify the state, the action, the policy, and the reward, and explain why reinforcement learning, rather than ordinary supervised learning, is the natural tool. CONCEPT

Uncertainty, calibration & Bayesian nets

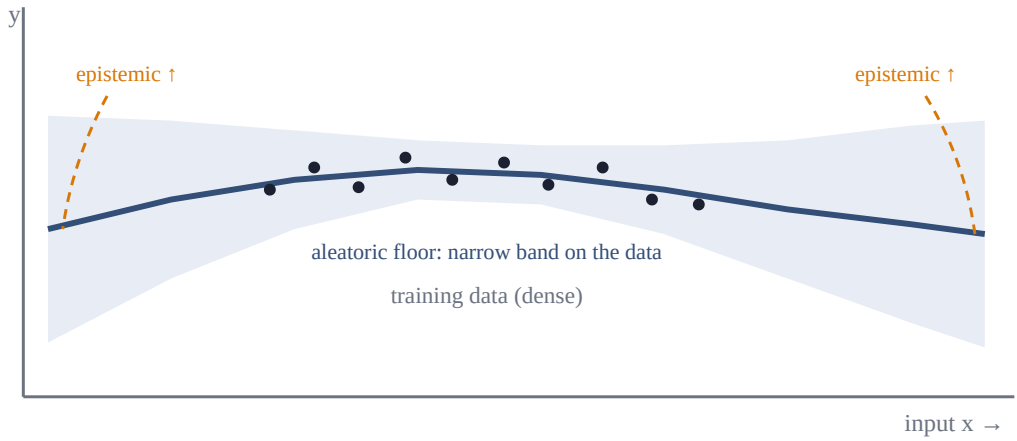
> *A network that answers "99% cat" and is wrong one time in three has told you two things: a label, and a lie about how sure it is. For medicine, self-driving, and science, the second number matters as much as the first. A model must know what it does not know. This chapter separates the two kinds of not-knowing, **aleatoric** uncertainty (noise in the world) and **epistemic** uncertainty (gaps in the model), shows why a raw softmax probability is an untrustworthy confidence, and builds the practical toolkit — calibration, temperature scaling, Bayesian and ensemble approximations, and distribution-free prediction sets — that turns a point prediction into an honest one.*

Fundamental concept

Ask a good weather forecaster for tomorrow's rain and you get "70% chance", and over many such forecasts it rains on about 70% of the "70%" days. That is a *calibrated* probability: the stated confidence matches the observed frequency. Now ask the same forecaster about weather on a planet they have never seen. The honest answer is not an exact number but a shrug — "I have no idea". These are two different kinds of uncertainty, and confusing them is the main cause of most over-confident predictions.

Aleatoric uncertainty (from *alea*, a die) is the irreducible randomness in the data itself: sensor noise, overlapping classes, a coin that is truly fair. No amount of extra data removes it, because the label truly is random given the input. It is the σ^2 noise floor of the loss chapter (Chapter 6) — the smallest error any model can achieve. **Epistemic uncertainty** (from *epistēmē*, knowledge) is uncertainty about the *model*: which weights, which function, are right. It is large where training data is scarce and shrinks as you collect more, because more evidence narrows down the model. A useful summary: aleatoric uncertainty is a property of the *data*, epistemic uncertainty is a property of the *model*.

This difference is not just theoretical. If a prediction is uncertain for aleatoric reasons, gathering more data of the same kind will not help — the task is naturally noisy, and you should widen your error bars and move on. If it is uncertain for epistemic reasons, more data (or data from the right region) will sharpen it. And, importantly, a test input that lies far from anything seen in training will have *high epistemic uncertainty*. This is exactly the signal you want for detecting out-of-distribution inputs and for deciding when to abstain and defer to a human. A standard network trained with cross-entropy gives you neither number reliably. The rest of this chapter is about recovering them.



*Aleatoric versus epistemic uncertainty on a 1-D regression. The blue curve is the mean prediction. The shaded band is its predictive standard deviation. Over the training data the band has a small, roughly constant width — the irreducible **aleatoric** noise. Away from the data, on both sides, the band widens: many functions fit the points but disagree in the gaps, so **epistemic** uncertainty grows. Collecting more data would shrink the widening but never the central floor.*

Governing equations

The Bayesian view provides the key equations. Instead of a single weight vector, treat the weights $\mathbf{w}$ as uncertain, with a **posterior** distribution given the data D obtained from Bayes' rule:

$$p(\mathbf{w} \mid D) = \frac{p(D \mid \mathbf{w})\, p(\mathbf{w})}{p(D)}, \quad p(D) = \int p(D \mid \mathbf{w})\, p(\mathbf{w})\, d\mathbf{w}$$

A prediction is then an **average** over all reasonable weights, weighted by how well each explains the data — the **predictive distribution**, approximated by drawing T weight samples:

$$p(y \mid x, D) = \int p(y \mid x, \mathbf{w})\, p(\mathbf{w} \mid D)\, d\mathbf{w} \approx \frac{1}{T} \sum_{t=1..T} p(y \mid x, \mathbf{w}_t), \quad \mathbf{w}_t \sim p(\mathbf{w} \mid D)$$

To measure whether the resulting probabilities are honest, we use the **Expected Calibration Error**. Sort predictions into M confidence bins B_m and compare each bin's accuracy to its average confidence:

$$\mathrm{ECE} = \sum_{m=1..M} \frac{|B_m|}{n} \cdot |\, \mathrm{acc}(B_m) - \mathrm{conf}(B_m) \,|$$

The cheapest recalibration divides the logits by a single scalar **temperature** T before the softmax. The total uncertainty of the averaged prediction is its **predictive entropy** H, and the epistemic part is the **mutual information** I between the label and the weights:

$$p_k(T) = \frac{e^{z_k/T}}{\sum_j e^{z_j/T}}, \quad H[p] = -\sum_k p_k \ln p_k$$

$$I(y, \mathbf{w} \mid x, D) = H[\, \mathbb{E}_{\mathbf{w}}\, p(y|x,\mathbf{w}) \,] - \mathbb{E}_{\mathbf{w}}[\, H[\, p(y|x,\mathbf{w}) \,] \,] = (\text{total}) - (\text{aleatoric})$$

w, D — the network's weights (now a random variable) and the training set; p(**w**) is the prior, p(D|**w**) the likelihood (Chapter 6)

p(w|D) — the posterior over weights; p(D) is the marginal likelihood (evidence), the intractable normalizing integral

p(y|x,D) — predictive distribution — the model's answer after averaging over weight uncertainty

w_t, T — the t-th of T posterior weight samples (or forward passes); T also denotes the softmax temperature — the context makes clear which is meant

B_m, n, M — the m-th confidence bin, total number of predictions n, number of bins M; acc/conf are the bin's mean accuracy and mean confidence

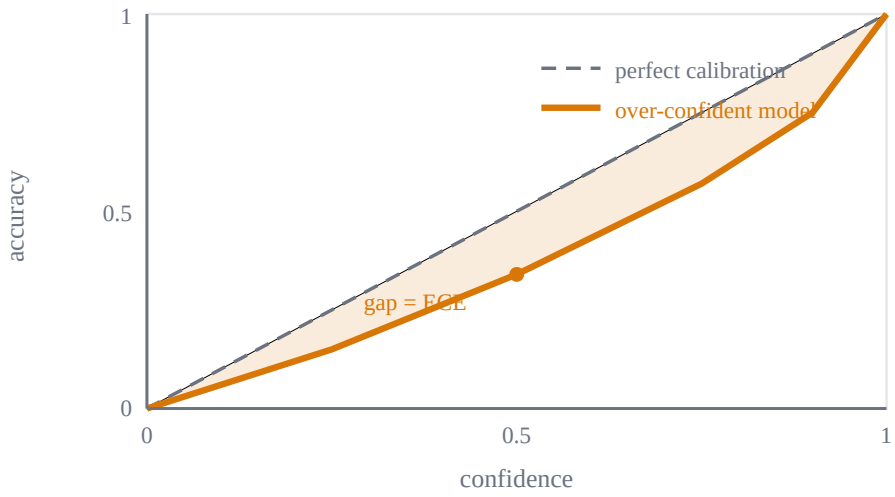
z_k — the k-th logit (pre-softmax score); temperature T>1 flattens the softmax, T<1 sharpens it

p — the predictive (ensemble-averaged) probability vector; H is its Shannon entropy in nats, I the mutual information (epistemic uncertainty)

Why a softmax score is not confidence

It is tempting to read the largest softmax output as "the probability the answer is right". It is not. The softmax is a normalized exponential of unbounded logits. Nothing in cross-entropy training forces its output to match an empirical frequency. Training pushes the true-class probability toward 1 on every example it gets right, so with enough capacity the network learns to produce near-1.0 scores almost everywhere — including where it is wrong. The result, documented on a large scale by Guo et al. (2017), is that **modern networks are systematically over-confident**: a batch of predictions the model calls "99% sure" might be only 80% accurate. Interestingly, the shallow LeNet-era networks of the 1990s were roughly calibrated. The depth, width, and reduced regularization that improved accuracy also broke calibration, because larger models push their logits harder to minimize cross-entropy and overshoot into over-confidence. Label smoothing and weight decay (Chapter 12) reduce but do not cure it.

To *see* miscalibration, draw a **reliability diagram**. Group predictions by their confidence into bins, and plot each bin's observed accuracy against its average confidence. A perfectly calibrated model lands on the diagonal — 70%-confident predictions are right 70% of the time. An over-confident model drops *below* the diagonal: its accuracy stays below its stated confidence at every level. The area of that drop, weighted by how many predictions fall in each bin, is exactly the Expected Calibration Error above — a single number in [0, 1] (often reported as a percentage) summarizing how much the probabilities can be trusted. A closely related worst-case score, the **Maximum Calibration Error** (MCE), takes the largest bin gap instead of the weighted average, and matters when any single over-confident bin is dangerous.



A reliability diagram. The dashed diagonal is perfect calibration (accuracy = confidence). The amber curve is a typical over-confident network: at every confidence level its accuracy is lower, so the curve drops below the line. The shaded area between them — the average vertical gap, weighted by how many predictions sit at each confidence — is the Expected Calibration Error. Temperature scaling pulls the amber curve up toward the diagonal.

Temperature scaling

The good news is that over-confidence is often a one-parameter disease with a one-parameter cure. **Temperature scaling** leaves the trained network untouched and simply divides every logit by a single learned scalar T before the softmax. Because dividing all logits by the same positive T is a monotonic transformation, it cannot change which class is largest — so *accuracy is exactly preserved* — but it does change how peaked the distribution is. With $T>1$ the logits are squeezed toward zero, the exponentials toward one, and the softmax toward the uniform distribution: the model becomes less sure. With $T<1$ it becomes more sure. The two limits make this clear: as $T \to \infty$ every prediction collapses to the uniform $1/K$ (maximum uncertainty), and as $T \to 0$ it hardens to a one-hot argmax (maximum confidence).

The single parameter T is fit *after* training on a held-out validation set by minimizing the negative log-likelihood — the same cross-entropy of Chapter 6, now optimized over one number instead of millions. For an over-confident network the minimizer is some $T>1$ (values of 1.5–3 are typical), which softens the probabilities until the reliability curve lies close to the diagonal and the ECE drops, often by an order of magnitude, at zero cost to accuracy. Temperature scaling is the default first thing to try, and it is a special case of a broader family

(Platt scaling, vector and matrix scaling) that fits a small post-hoc transformation of the logits. Its limitation is exactly its strength: because it is a single global scalar, it fixes *average* calibration but cannot make an individual out-of-distribution input honest, and it adds no epistemic uncertainty — for that we need the model itself to express doubt.

Bayesian neural networks

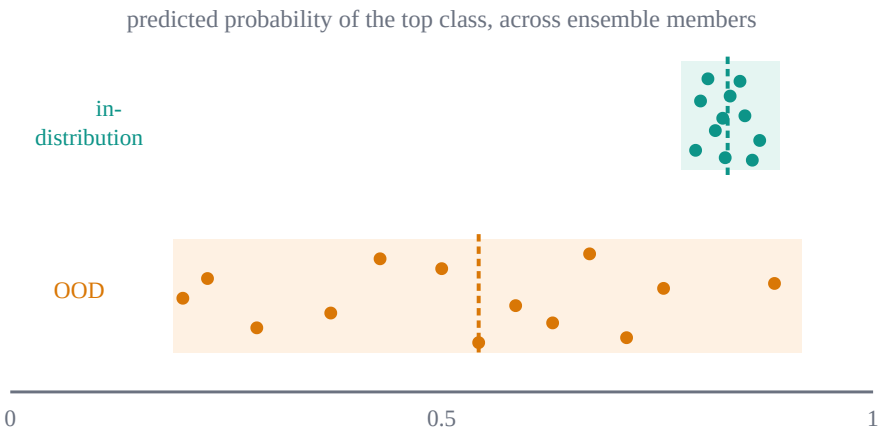
Where does proper epistemic uncertainty come from? From refusing to commit to one weight vector. A **Bayesian neural network** places a prior $p(\mathbf{w})$ over the weights, updates it to the posterior $p(\mathbf{w}|D)$ by Bayes' rule, and predicts by integrating the network's output over that posterior — the predictive integral above. Intuitively, every setting of the weights that fits the training data casts a vote. Where the votes agree the prediction is confident, and where they disagree — typically far from the data — the spread of votes *is* the epistemic uncertainty. This is precisely the widening band of the opening figure, now given a mechanism.

The problem is the denominator. The evidence $p(D) = \int p(D|\mathbf{w})\, p(\mathbf{w})\, d\mathbf{w}$ is an integral over the entire weight space — millions of dimensions — with no closed form, so the exact posterior is **intractable**. Every practical Bayesian method is therefore an approximation to this integral. The **Laplace approximation** is the simplest: train normally to a maximum-a-posteriori weight $\mathbf{w}^*$, then fit a Gaussian around that mode whose covariance is the inverse Hessian of the loss, $p(\mathbf{w}|D) \approx N(\mathbf{w}^*, H^{-1})$. It is cheap — a curvature estimate at a point you already found — and it turns any pre-trained network into an approximate Bayesian one after the fact, but it is a purely *local* picture and only as good as a single-mode Gaussian around one solution. **Variational inference** instead assumes a simple family $q(\mathbf{w})$ and fits it to the true posterior by maximizing a lower bound (the ELBO), while **Markov-chain Monte Carlo** samples weights directly and is asymptotically exact but expensive for deep networks. All three target the same intractable integral. The next two methods approximate it so cheaply that they have become the practical default.

MC dropout and deep ensembles

Two approximations dominate practice because they are almost free to add to an ordinary network. **Monte-Carlo dropout** (Gal & Ghahramani, 2016) reuses the dropout of Chapter 12 with one change: instead of switching dropout *off* at test time, you keep it *on* and run the network T times. Each pass randomly turns off a different subset of units, so each is effectively a different thinned sub-network — a different weight sample — and averaging their softmax outputs approximates the predictive integral. The remarkable result is that a network already trained with dropout needs no retraining at all: the same masks that regularized it during training now sample an approximate posterior at test time. The spread across the T passes is a usable epistemic signal, and it costs T forward passes and nothing else.

Deep ensembles (Lakshminarayanan et al., 2017) are even simpler and, empirically, the strongest of all. Train N copies of the same architecture from different random initializations (Chapter 10) and different data shuffles. At test time, average their predictions. Because the loss landscape has many good but distinct minima, different seeds converge to truly different functions that all fit the training data yet disagree away from it — exactly the disagreement that measures epistemic uncertainty. On an in-distribution input the N members agree and the averaged prediction is sharp. On an out-of-distribution input they scatter, the average flattens, and the variance across members rises sharply. Ensembles cost N× training and N× inference, which is their only real drawback. MC dropout is the cheaper alternative when a single network is all you can afford. Both turn a point predictor into a distribution, and both are, in effect, rough but effective posterior samplers.



Predictive spread of an ensemble (or MC-dropout) on two inputs. Each dot is one member's top-class probability. For the in-distribution input (teal) the members agree closely around a high value — low epistemic uncertainty. For the out-of-distribution input (amber) they scatter across the whole range — high epistemic uncertainty. The spread, not any single pass, is the uncertainty signal. The runnable ensemble cell computes exactly this variance.

Scoring uncertainty: entropy, mutual information, OOD

Given an ensemble's averaged prediction p, how do we turn it into a number to act on? The natural total-uncertainty score is the **predictive entropy** $H[p] = -\Sigma_k\, p_k \ln p_k$, maximized ($\ln K$ nats) when the average is uniform and zero when it is a confident one-hot. But entropy alone cannot tell *why* the model is unsure. The neat fix is to split it. The **mutual information** between the label and the weights,

$$I = H[p] - \frac{1}{T}\,\Sigma_t\, H[\, p(y \mid x, \mathbf{w}_t)\,] = (\text{total uncertainty}) - (\text{aleatoric}) = \text{epistemic}\,,$$

subtracts the *average entropy of the individual members* (their own aleatoric confusion) from the entropy of their *average* (the total). What remains is disagreement *among* members — pure epistemic uncertainty, sometimes called the BALD score. The two cases are worth remembering: if every member is individually unsure but they all agree (a truly ambiguous input, e.g. a blurry 3-or-8), total entropy is high but the members agree, so mutual information is near zero — the uncertainty is aleatoric. If every member is individually confident but they confidently *disagree* (a new input each net extrapolates differently), the average is flat, the individual entropies are low, and mutual information is large — the uncertainty is epistemic. Exercise 4 computes both numbers by hand.

This decomposition enables **out-of-distribution detection**. An input drawn from a class or domain the model never trained on should cause high *epistemic* uncertainty — the ensemble members, forced to extrapolate, disagree. Setting a threshold on predictive entropy or mutual information (or, more cheaply, the maximum softmax probability, though it is the least reliable) flags such inputs so the system can abstain, route to a human, or fall back to a safe default. It is not perfect — deep networks can still be confidently wrong on cleverly constructed OOD inputs — but ensemble disagreement is one of the strongest simple signals available.

Conformal prediction: coverage without distributions

All of the above quantifies uncertainty. **Conformal prediction** instead *guarantees* it. Choose a miscoverage rate α (say 0.1 for 90% coverage). On a held-out calibration set, score each example by a **nonconformity** measure — for classification, a simple choice is $s = 1 - p(\text{true class})$, large when the model is surprised by the truth. Take the empirical quantile $\hat{q}$ = the $\lceil (n+1)(1-\alpha) \rceil$/n-th smallest calibration score. At test time, output the *set* of all labels whose nonconformity is at most $\hat{q}$ — for the score above, every class with predicted probability at least $1 - \hat{q}$. Under the only assumption that calibration and test data are **exchangeable** (a slightly weaker condition than i.i.d.), this set is guaranteed to contain the true label with probability at least $1 - \alpha$, in finite samples, for *any* model and *any* data distribution — no Gaussian, no asymptotics.

The cost of this distribution-free guarantee is that the output is a set, not a point, and its *size* is where the uncertainty appears: easy inputs give single-label sets, hard or new inputs give large sets. Conformal prediction combines well with everything above — feed it a temperature-scaled or ensembled probability and the sets get smaller — and it is the rigorous partner to calibration: calibration makes the probabilities honest on average, conformal prediction makes the coverage exact.

Uncertainty in code

Two NumPy cells make the chapter concrete. The first computes the Expected Calibration Error from arrays of confidences and correctness, matching the worked example's ECE = 0.11, then temperature-scales a logit vector and shows the maximum softmax probability fall from 0.836 to 0.619 — the same over-confident prediction, softened. Press **Run**:

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)

# ten predictions: their confidence (max softmax prob) and whether each was correct
conf    = np.array([0.6, 0.6, 0.8, 0.8, 0.8, 0.9, 0.9, 0.9, 0.9, 0.9])
correct = np.array([1,   0,   1,   1,   0,   1,   1,   1,   1,   0  ], float)

def ece(conf, correct, M=10):                 # Expected Calibration Error, M equal bins
    idx = np.minimum((conf * M).astype(int), M - 1)   # bin index of each prediction
    e = 0.0
    for b in range(M):
        sel = idx == b
        k = sel.sum()
        if k:                                 # weighted gap between accuracy and confidence
            e += k / len(conf) * abs(correct[sel].mean() - conf[sel].mean())
    return e

print("ECE =", round(ece(conf, correct), 4))  # expected: 0.11

def softmax(z):
    z = z - z.max()                           # numerical stability (Chapter 6)
    e = np.exp(z)
    return e / e.sum()

z  = np.array([3.0, 1.0, 0.2])                # an over-confident logit vector
p1 = softmax(z)                               # T = 1
```

```
p2 = softmax(z / 2.0)                          # temperature scaling, T = 2
(softer)
print("T=1  max prob =", round(p1.max(), 4))  # 0.836  <- over-confident
print("T=2  max prob =", round(p2.max(), 4))  # 0.619  <- calibrated down,
class unchanged
```

The second cell simulates a small ensemble: it slightly changes a base logit vector many times (each change is one "member"), averages their softmaxes into a predictive mean, and reports the total variance and predictive entropy. For a confident in-distribution input the members agree, so both are small. For an ambiguous out-of-distribution input the spread — the epistemic uncertainty — is far larger:

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)

def softmax_rows(Z):
    Z = Z - Z.max(axis=-1, keepdims=True)
    E = np.exp(Z)
    return E / E.sum(axis=-1, keepdims=True)

def predictive(base, noise, K=500):           # K ensemble members = noisy
logit copies
    Z = base + noise * np.random.randn(K, base.size)
    P = softmax_rows(Z)                        # K x C member probabilities
    pbar = P.mean(0)                           # predictive mean
    H = -np.sum(pbar * np.log(pbar + 1e-12))   # predictive entropy (nats)
    return pbar, P.var(0).sum(), H             # mean, total variance,
entropy

pbar_id,  var_id,  H_id  = predictive(np.array([3.0, 1.0, 0.2]), 0.4)  # in-
distribution
pbar_ood, var_ood, H_ood = predictive(np.array([0.3, 0.1, 0.0]), 2.5)  #
OOD-like input

print("ID : mean", np.round(pbar_id, 3),  " var", round(var_id, 4),  "
entropy", round(H_id, 3))
print("OOD: mean", np.round(pbar_ood, 3), " var", round(var_ood, 4), "
entropy", round(H_ood, 3))
print("epistemic spread larger for OOD:", bool(var_ood > var_id),
      "  entropy higher:", bool(H_ood > H_id))
```

In PyTorch the same two ideas are a few standard lines. MC dropout keeps the dropout layers active at evaluation and averages many random passes. Temperature scaling learns one scalar T by minimizing validation NLL. This listing is for reading and copying into Colab — it is not run here:

PyTorch copy → Colab

```
import torch, torch.nn as nn, torch.nn.functional as F

# ---- MC Dropout: keep Dropout ON at test time, average T stochastic passes ----
@torch.no_grad()
def mc_dropout_predict(model, x, T=50):
    model.eval()                                # BatchNorm etc. in eval mode ...
    for m in model.modules():                   # ... but re-enable Dropout (Chapter 12)
        if isinstance(m, nn.Dropout):
            m.train()
    probs = torch.stack([F.softmax(model(x), dim=-1) for _ in range(T)])  # T x N x C
    mean = probs.mean(0)                        # predictive distribution
    var  = probs.var(0)                         # epistemic spread, per class
    entropy = -(mean * mean.clamp_min(1e-12).log()).sum(-1)               # total uncertainty
    return mean, var, entropy

# ---- Temperature scaling: fit ONE scalar T on a held-out validation set ----
class TemperatureScaler(nn.Module):
    def __init__(self):
        super().__init__()
        self.log_T = nn.Parameter(torch.zeros(1))     # parametrize T = exp(log_T) > 0
    def forward(self, logits):
        return logits / self.log_T.exp()

def fit_temperature(logits_val, labels_val):
    scaler = TemperatureScaler()
    opt = torch.optim.LBFGS(scaler.parameters(), lr=0.01, max_iter=100)
    def closure():
        opt.zero_grad()
        loss = F.cross_entropy(scaler(logits_val), labels_val)   # NLL on
```

```
validation logits
        loss.backward()
        return loss
    opt.step(closure)
    return scaler.log_T.exp().item()          # learned T, typically > 1 for
over-confident nets
```

Worked example — ECE from ten predictions, then a temperature fix

Ten test predictions are grouped by their confidence (the max softmax probability) into three bins. Within each bin we record the average confidence and the observed accuracy.

BIN (CONFIDENCE)	COUNT $\lvert B_M \rvert$	CONF	ACCURACY	$\lvert$ACC − CONF$\rvert$	WEIGHTED GAP
≈ 0.60	2	0.60	1/2 = 0.50	0.10	(2/10)(0.10) = 0.020
≈ 0.80	3	0.80	2/3 = 0.667	0.133	(3/10)(0.133) = 0.040
≈ 0.90	5	0.90	4/5 = 0.80	0.10	(5/10)(0.10) = 0.050

1. Every bin is over-confident: its accuracy is below its confidence, so the model claims more certainty than it earns — the reliability curve of the figure.
2. Sum the weighted gaps: ECE = 0.020 + 0.040 + 0.050 = **0.11**. On average the stated confidence exceeds the true accuracy by 11 percentage points. (The worst single bin gives the Maximum Calibration Error, MCE = **0.133**.)
3. Now soften one over-confident prediction with logits z = [3.0, 1.0, 0.2]. At T = 1 the softmax is p = [0.836, 0.113, 0.051], a confidence of **0.836**.
4. Apply temperature T = 2: divide the logits, z/T = [1.5, 0.5, 0.1], and re-softmax to p = [0.619, 0.228, 0.153], a confidence of **0.619**.
5. The maximum probability fell from 0.836 to **0.619** while the ranking (class 0 wins) is unchanged — accuracy is preserved, over-confidence is not.

What the numbers mean: an ECE of 0.11 says the probabilities are too high, and the 0.90-bin — 90% confident but only 80% accurate — is the problem that temperature scaling targets. Pulling that 0.836 down to 0.619 moves the confident predictions toward their true hit-rate. Refit on a validation set, a single T shrinks the whole ECE with no change to which class is predicted.

Assumptions & limitations

ECE depends on the binning — too few bins hide miscalibration, too many leave each bin noisy — and a low ECE confirms only *average* calibration, not correctness on any single input. Temperature scaling assumes over-confidence is a global scaling issue that can be fixed on a validation set drawn from the *same* distribution as deployment. It does nothing for distribution shift and adds no epistemic uncertainty. MC dropout and ensembles only *approximate* the posterior — their uncertainty is as good as the diversity of the samples, and a badly-specified model can be confidently and uniformly wrong (every member agreeing on the same error), which no amount of averaging detects. Conformal prediction's guarantee holds only under exchangeability, which distribution shift breaks, and it guarantees *coverage*, not small sets. None of these methods creates knowledge the data lacks: they make ignorance visible, not smaller.

Why & when this matters

Use this chapter the moment a prediction's *confidence* will drive a decision — to treat a patient, to brake a car, to auto-approve a loan, or to mark a scientific measurement as unusual. If you only need the top-1 label, raw softmax is fine. If you need to know when to trust it, defer to a human, or reject an input as out-of-distribution, you need calibrated probabilities and an epistemic signal. The cheap step-by-step plan is: temperature-scale on a validation set for honest average confidence. Add MC dropout or a small deep

ensemble for epistemic uncertainty and OOD detection. Wrap the result in conformal prediction when you need a coverage guarantee. Each step costs more compute and gives more trustworthiness.

In the real world

Calibration is now standard practice. Guo et al.'s temperature scaling (2017) is used in production classifiers because it is a free post-hoc fix. Deep ensembles are the reliable method behind many uncertainty-critical systems, from weather and climate emulators to the OOD detectors that guard autonomous-driving perception stacks. MC dropout gave the first cheap Bayesian estimates for medical imaging — diabetic-retinopathy and tumor-segmentation models use predictive entropy to refer uncertain scans to a doctor. Conformal prediction has quickly moved into real use precisely because its guarantee is distribution-free: it is now applied to drug-discovery screens, LLM answer-selection, and any pipeline that must promise "the true answer is in this set 90% of the time". The common theme is a shift from asking a network only *what* it predicts to demanding *how much* it should be believed.

Exercises

1. A classifier's predictions fall into four confidence bins with counts 10, 20, 30, 40; average confidences 0.55, 0.65, 0.85, 0.95; and accuracies 0.50, 0.55, 0.70, 0.80. Compute the ECE and the MCE. MATH

2. Prove that dividing all logits by a temperature $T>0$ never changes the predicted class, and find the softmax in the limits $T \to \infty$ and $T \to 0$. MATH

3. Extend the ECE cell to also report the MCE and to vary the number of bins $M \in \{5, 10, 15, 20\}$ on the same ten predictions. CODE

4. An ensemble of two members predicts, for a binary problem, the class probabilities [0.9, 0.1] and [0.9, 0.1] on input A, and [0.99, 0.01] and [0.01, 0.99] on input B. For each input compute the total, aleatoric, and epistemic (mutual-information) uncertainty in nats. MATH

5. Given an array of K member softmax rows, write NumPy that decomposes the predictive uncertainty into total, aleatoric, and epistemic, and check it on agreeing versus disagreeing members. CODE

6. Explain why a single softmax probability is not a calibrated confidence, and state precisely what temperature scaling fixes and what it cannot fix. CONCEPT

7. Compare deep ensembles and MC dropout as epistemic-uncertainty estimators: how each obtains its weight samples, their relative cost, and why different random seeds capture model uncertainty. CONCEPT

8. You have nine calibration nonconformity scores $s = 1 - p(\text{true class})$: {0.05, 0.10, 0.15, 0.20, 0.30, 0.40, 0.55, 0.70, 0.90}. For 90% coverage, find the conformal threshold $\hat{q}$, give the rule for the prediction set, and state the guarantee and its one assumption. CONCEPT

PART V

In practice

Turning the ideas into a working project: the end-to-end training recipe, a data-centric view of quality, making models small and fast enough to ship, and the interpretability, fairness and safety that decide whether they should.

25 Training in practice: the complete recipe

26 Data-centric deep learning

27 Efficiency & deployment

28 Interpretability & explainability

29 Ethics, fairness & safety

CHAPTER 25 · IN PRACTICE

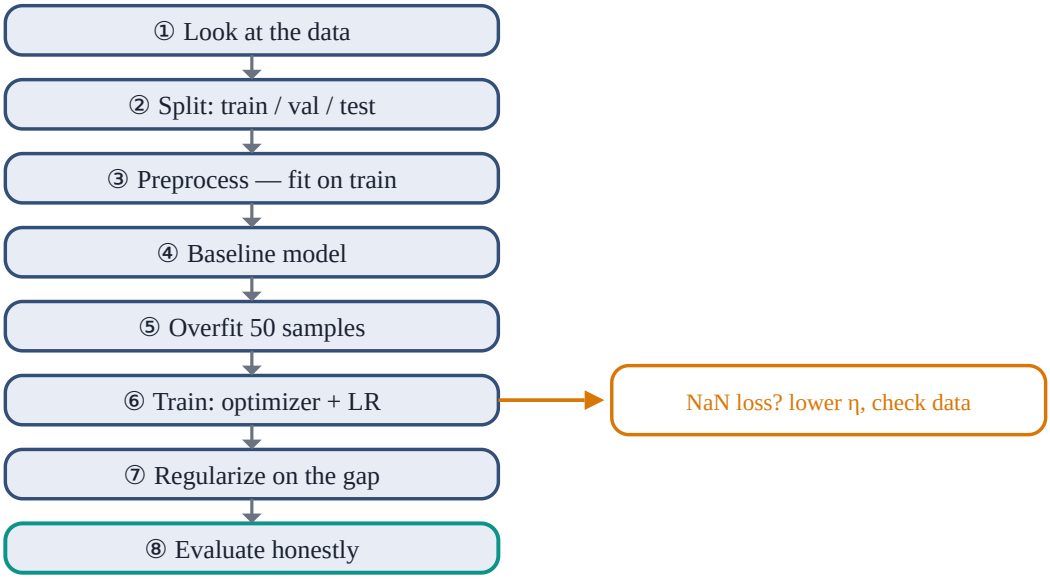
Training in practice: the complete recipe

> *A working network is rarely lost because of the mathematics of earlier chapters. It is lost because of process. Data leaks, a test set is peeked at, accuracy makes a useless model look good, or a run cannot be reproduced. This final chapter is the checklist that prevents those problems: a numbered, end-to-end recipe you can run on any supervised problem, and the honest metrics that tell you whether it actually worked.*

Fundamental concept

Think of training a model the way a pilot thinks of a pre-flight checklist. The individual actions are not hard. The value is in doing them *in order*, every time, so that a predictable mistake never becomes a crash. Most machine-learning failures are not unusual. They are the same few process errors repeated: mixing test data into training, letting a stray feature reveal the answer, judging an imbalanced problem by raw accuracy, or changing five things at once and not knowing which one helped.

The recipe below is arranged so that failures happen cheaply and early. You look at the data before you build anything, split it before you touch a model, build a simple baseline before a large model, and prove the pipeline can memorize a few points before you trust it on millions. Each step exists to catch a specific, common way of fooling yourself. The single most important habit is to treat your **test set** as a sealed exam that you take exactly once.



The recipe as a flowchart. Follow the main line from top to bottom. The amber branch is the first thing to try when a run breaks down. Every step is designed to catch a mistake before it reaches the sealed test set (⑧).

Governing equations

Preprocessing statistics come from the **training set only**. Reusing them on validation and test is what keeps the evaluation honest:

$$x' = \frac{x - \mu_{\text{train}}}{\sigma_{\text{train}}}$$

$$\text{images: } x' = x / 255 \;\rightarrow\; [0, 1]$$

Training is the familiar parameter update, with the learning rate η as the main control. You watch the **generalization gap** to decide when to regularize:

$$\theta \leftarrow \theta - \eta \, \nabla_\theta L$$

$$\text{gap} = L_{\text{val}} - L_{\text{train}}$$

On imbalanced problems raw accuracy misleads, so evaluation is based on the four cells of a confusion matrix:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

$$\text{precision} = \frac{TP}{TP + FP}$$

$$\text{recall} = \frac{TP}{TP + FN}$$

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

$$\text{specificity} = \frac{TN}{TN + FP}$$

$$\text{balanced acc} = \tfrac{1}{2}\,(\text{recall} + \text{specificity})$$

TP, TN	true positives / negatives — cases the model labeled correctly as positive / negative
FP, FN	false positives (false alarms) / false negatives (misses) — the two kinds of error
μ_{train}, σ_{train}	per-feature mean and standard deviation, estimated on the training split only
η	learning rate — the step size of the optimizer, and the single most sensitive hyperparameter
L_{train}, L_{val}	average loss on the training and validation splits. Their gap signals overfitting
precision	of everything flagged positive, the fraction truly positive (trustworthiness of an alarm)
recall	of everything truly positive, the fraction caught (coverage of the rare class)

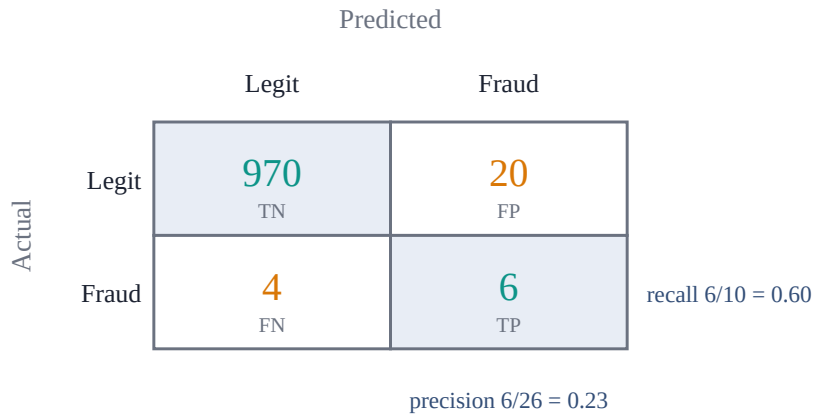
The recipe, step by step

1. **Look at the data first.** Plot feature distributions and the class balance before modeling. Look for **leakage** — a feature or split that secretly carries in the answer. The classic case: the same patient's records end up in both train and test, so the model recognizes the *patient*, not the disease, and scores very well in the lab but fails in the clinic. Split by patient, and drop any feature recorded after the outcome.

2. **Split train / validation / test — then seal the test set.** Decide the three splits before touching a model. Validation tunes your choices. The test set is opened exactly once, at the very end, to report a number you can trust.
3. **Preprocess using training statistics only.** Standardize with train μ and σ, one-hot categoricals, scale images to [0, 1]. Computing μ and σ over the whole dataset leaks test information backward and inflates your score.
4. **Build a baseline first.** Fit logistic regression or a tiny network. It sets a baseline score: if a deep model cannot beat it, the extra complexity is not worth it.
5. **Deliberately overfit 50 samples.** Take a tiny subset and drive its loss toward zero. A correct model *must* be able to memorize this. If it cannot, you have a bug — wrong loss, a broken data pipeline, a detached gradient — and no amount of data will fix it.
6. **Choose architecture by data type.** Tabular data → 1–3 ReLU layers; images → a CNN (Chapter 13); sequences → an RNN or a transformer (Chapters 15–16). Match the inductive bias to the structure of the input.
7. **Set optimizer and learning rate.** Adam at $\eta = 10^{-3}$ is a strong default. Run a quick learning-rate range test — raise η and watch the loss — to find the largest stable value, then choose the batch size based on available memory. The learning rate matters more than any other setting.
8. **Regularize only once a gap appears.** When L_{val} rises above L_{train}, add early stopping first, then L2, dropout, or data augmentation (Chapter 12, regularization). Regularizing a model that is still underfitting only makes it worse.
9. **Change one thing at a time.** Adjust one hyperparameter per run, log every configuration and its metrics, and fix the random seed *last* — after the method works. A seed chosen to make a result look good is just hiding variance.
10. **Evaluate honestly.** On imbalanced data accuracy lies (see the worked example). Read precision, recall, F1 and the full confusion matrix, and check calibration — do predicted probabilities of 0.7 come true about 70% of the time?
11. **Debug by symptom.** NaN or Inf loss → η too high or unscaled/bad data; flat loss → η too low, bad initialization, or a bug (run a gradient check); train good but test bad →

overfitting, so regularize; both bad → underfitting or a bug, so add capacity or features.

12. **Make it reproducible, and fair.** Record seeds, data versions and library versions so a run can be repeated. Remember that biased data produces a biased model: a hiring model trained on past hires learns past prejudice, so audit performance across subgroups before anyone relies on it.



The fraud confusion matrix. Precision reads down the "predicted fraud" column (how many alarms were real). Recall reads across the "actual fraud" row (how many frauds were caught). Accuracy, dominated by the 970 correct legitimate cases, hides both.

Framing the problem and choosing the metric

Before step 1 there is a step zero: decide *what you are predicting and how success will be measured*. Is one class rare, and is a miss (false negative) more costly than a false alarm? The answer sets the loss you train and the metric you report. They need not be the same: you might *train* smooth cross-entropy but *report* recall at a fixed precision, because that is what the radiologist or fraud desk acts on. Pick the metric that punishes the errors that hurt, fix it **before** you see results, and pair every score with a simple reference (majority-class rate, a coin flip, last year's system) so "92%" reads as an *improvement over a baseline*. When a decision uses the model's confidence, not just its label, calibration becomes a target of its own (Chapter 24, uncertainty). Which errors you trade off is an ethical judgment, not only a statistical one (Chapter 29, ethics).

Diagnosing the fit: the iterate loop

Once the pipeline runs, training is a short diagnostic loop you repeat, not a single step. Each pass asks one question: *underfitting or overfitting?* You read the answer from two numbers, the training and validation loss, using the bias–variance decomposition of Chapter 12 (regularization & generalization). **High training loss** is high bias, **underfitting**: the model cannot fit even what it has seen, so add power — a bigger or deeper model, more training, better features, better initialization (Chapter 10), and *less* regularization. **Low training loss but a wide validation gap** is high variance, **overfitting**: it has memorized noise, so close the gap — more data, augmentation, stronger L2 or dropout, early stopping, or a smaller model (all Chapter 12). The two moves that are always wrong are to regularize while underfitting or to add capacity while overfitting. Each one adjusts the wrong thing.

One check earns its own name because it catches the most bugs: **overfit a single batch** — a few dozen examples, shuffling and regularization off, driven to near-zero loss, the easiest thing a network can do. If it cannot, the fault is a wiring bug (wrong loss, misaligned labels, a detached gradient, a loader returning the same batch), caught in seconds rather than after a GPU-week.

Finding the learning rate, and searching the rest

The learning rate η is the most sensitive hyperparameter (Chapter 7, gradient descent), and there is a low-cost way to set it: the **learning-rate range test**. Start from a tiny η and raise it geometrically over a few hundred mini-batches, plotting loss against η on a log axis. The curve goes down, reaches a minimum, then turns sharply up as η grows large enough to diverge. The best rate is just below that upturn, and a safe choice is about ten times below the point where it becomes unstable. Adam (Chapter 9) works over a wider range than plain SGD, but even Adam has an upper limit. The cell below runs this idea on a one-feature regression, where gradient descent is stable only for $\eta < 2/\text{curvature}$, so you can watch the sweep cross it:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)
X = rng.normal(0, 1, 200)                    # one feature
y = 2.0 * X + 0.1 * rng.normal(0, 1, 200)  # target: slope 2, small noise
```

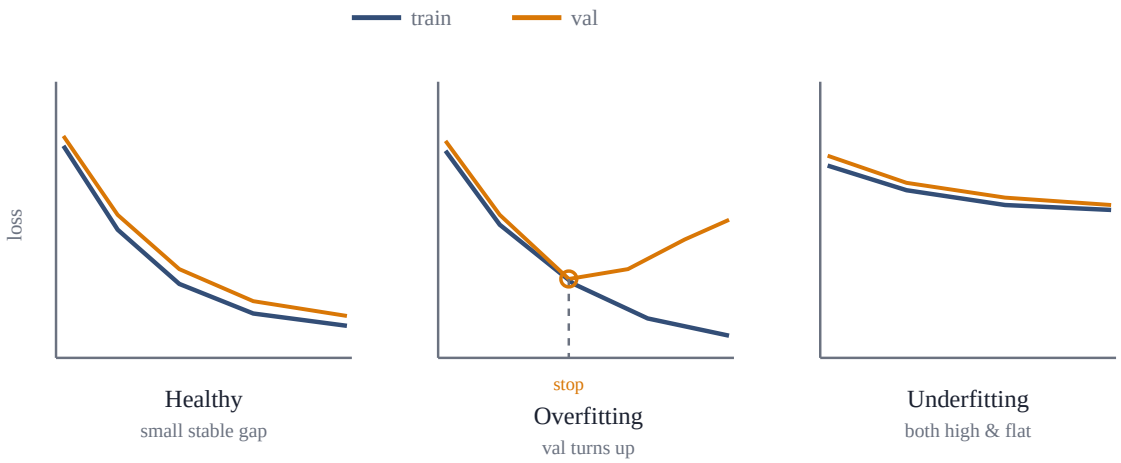
```
def sweep(lr, steps=80):                    # GD on MSE from the same start
    w = 0.0
    for _ in range(steps):
        w -= lr * (2/len(y)) * (X * (w*X - y)).sum()
    return ((w*X - y)**2).mean()

curv = (2/len(y)) * (X*X).sum()             # GD is stable only for lr < 2/curv
print(f"stability ceiling 2/curv = {2/curv:.3f}")
for lr in [1e-3, 1e-2, 3e-2, 1e-1, 3e-1, 1.0]:
    L = sweep(lr)
    tag = "diverges" if L > 1e3 else ("too slow" if L > 0.05 else "good")
    print(f"lr={lr:<6} final loss={L:12.4f}   {tag}")
```

The upper limit prints as 0.954: below it the loss settles near zero (fast at $\eta \approx 0.1$, slow at $\eta \approx 0.001$), and at $\eta = 1.0$ it explodes — the shape the range test reveals on any network. For the other settings — width, depth, weight decay, dropout, batch size — prefer **random search over grid search**: a grid wastes its budget changing the hyperparameters that do not matter, while random sampling covers many values of the *few* that do (the learning rate first, then regularization strength and model size). When trials are costly, **Bayesian optimization** does even better, fitting a cheap surrogate of "hyperparameters → score" and sampling where it expects the largest gain.

Reading the loss curves

The training and validation losses against epochs are the most informative single picture in training. A glance diagnoses most runs. **Healthy:** both fall and flatten close together — a small, stable gap means the model is learning *and* generalizing, so stop when validation stops improving. **Overfitting:** training keeps falling while validation bottoms out and turns up. That minimum is where early stopping fires, and the widening gap tells you to add data or regularization (Chapter 12). **Underfitting:** both plateau while still high and never separate — the model or η cannot fit even the training set, so add capacity or raise η. **Unstable:** the loss

spikes or, worst case, jumps to **NaN/Inf** — an exploding gradient or corrupt, unscaled data. Lower η, clip gradients, check normalization (Chapter 11), and look for a divide-by-zero or a mislabeled row.



*Loss versus epoch, training in blue and validation in amber. Left to right: a **healthy** run (small stable gap); an **overfitting** run (validation turns up — the circle marks where to early-stop); an **underfitting** run (both plateau high). A fourth shape, large spikes or a jump to NaN, is read from the same plot and signals η too high or bad data.*

K-fold cross-validation

A single train/validation split wastes data and gives a noisy estimate: on a small dataset the score you read from one held-out slice can swing several points depending on which rows happened to end up in it. **K-fold cross-validation** removes that luck. Split the data into k equal folds. Train k times, each run holding out a different fold for validation and training on the other k−1. Then average the k scores and report their spread:

$$\mathrm{CV} = \frac{1}{k} \Sigma_{i=1..k} \, \mathrm{score}_i \;\pm\; \mathrm{std}(\mathrm{score}_i)$$

Every example is used for validation exactly once and for training k−1 times, so the estimate uses all the data, and the ± tells you how much to trust it. The cell below does a manual 5-fold on a small example problem — note that no library is needed, only some index tracking:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)

# toy 1-D binary data with 15% label noise
X = rng.normal(0, 1, 200)
y = (X > 0).astype(int)
flip = rng.rand(200) < 0.15
y = np.where(flip, 1 - y, y)

k = 5
idx = rng.permutation(200)
folds = np.array_split(idx, k)
accs = []
for i in range(k):
    val = folds[i]
    train = np.concatenate([folds[j] for j in range(k) if j != i])
    thr = X[train].mean()                       # 'fit' a threshold on train
    pred = (X[val] > thr).astype(int)
    accs.append((pred == y[val]).mean())        # score on the held-out fold

accs = np.array(accs)
print("per-fold accuracy:", accs.round(3))
print(f"CV accuracy: {accs.mean():.3f} +/- {accs.std():.3f}")
```

The per-fold accuracies range from about 0.65 to 0.88 with a mean near 0.79 — and that ±0.08 spread is exactly the uncertainty a single split would have hidden. Common choices are k = 5 or k = 10. The extreme case k = N is **leave-one-out**. The costs are real: k-fold trains the model k times. That is why it is standard for small tabular datasets but rare for large deep networks, where one split is affordable and fold-to-fold variance is already small. The **test set stays sealed** throughout — cross-validation splits only the train-plus-validation data. The final honest number still comes from the test set opened once, at the very end.

The recipe in code

Two of the recipe's steps are single lines of code worth seeing exactly. First, honest evaluation on imbalanced data — the fraud confusion matrix turned into precision, recall and F1, reproducing the worked example's numbers:

Python · NumPy ▶ runnable in the app

```python
import numpy as np

# fraud detector on 1000 transactions (fraud = the positive class)
TP, FP, FN, TN = 6, 20, 4, 970
N = TP + FP + FN + TN

accuracy  = (TP + TN) / N
precision = TP / (TP + FP)
recall    = TP / (TP + FN)
f1        = 2 * precision * recall / (precision + recall)

print(f"accuracy : {accuracy:.4f}")
print(f"precision: {precision:.4f}")
print(f"recall   : {recall:.4f}")
print(f"F1       : {f1:.4f}")
# a lazy 'always legit' model beats it on accuracy while catching zero fraud
print(f"baseline (predict all legit): {(TN + FP) / N:.4f}")
```

The 97.6% accuracy is beaten by a do-nothing “always legit” baseline at 99.0%, while F1 = 0.333 tells the truth about the minority class. Second, step 3's rule — **fit preprocessing statistics on the training split only**. The mean and standard deviation come from train and are then reused exactly on test, so no test information leaks backward:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)

# two features on very different scales
Xtrain = np.column_stack([rng.normal(50, 10, 8), rng.normal(0.5, 0.1, 8)])
Xtest  = np.column_stack([rng.normal(50, 10, 4), rng.normal(0.5, 0.1, 4)])

mu = Xtrain.mean(axis=0)          # statistics from TRAIN ONLY
sd = Xtrain.std(axis=0)
Xtr = (Xtrain - mu) / sd
Xte = (Xtest  - mu) / sd          # reuse train mu, sd -- no peeking at test

print("train mean after scaling:", Xtr.mean(0).round(3))
```

```
print("train std  after scaling:", Xtr.std(0).round(3))
print("test  mean after scaling:", Xte.mean(0).round(3))
print("test  std  after scaling:", Xte.std(0).round(3))
```

The scaled training features have mean 0 and standard deviation 1 by design. The test features, standardized with the *same* train statistics, land near but not exactly at 0 and 1 — which is correct, because the test set must not influence its own scaling. Finally, the train/validation loop that every project reuses, in standard PyTorch with early stopping on the validation loss:

PyTorch copy → Colab

```
import torch, torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset

def make_loader(X, y, bs=64, shuffle=False):
    return DataLoader(TensorDataset(X, y), batch_size=bs, shuffle=shuffle)
# train_loader = make_loader(Xtr, ytr, shuffle=True)   # build from your
splits
# val_loader   = make_loader(Xva, yva)

model = nn.Sequential(nn.Linear(20, 64), nn.ReLU(), nn.Linear(64, 2))
opt   = torch.optim.Adam(model.parameters(), lr=1e-3)   # Adam @ 1e-3:
strong default
lossf = nn.CrossEntropyLoss()

def run_epoch(loader, train):
    model.train() if train else model.eval()
    tot = correct = 0; loss_sum = 0.0
    with torch.set_grad_enabled(train):
        for xb, yb in loader:
            logits = model(xb); loss = lossf(logits, yb)
            if train:
                opt.zero_grad(); loss.backward(); opt.step()
            loss_sum += loss.item() * len(xb)
            correct  += (logits.argmax(1) == yb).sum().item(); tot +=
len(xb)
    return loss_sum / tot, correct / tot

best, wait, patience = 1e9, 0, 5
for epoch in range(100):
    tr_loss, tr_acc = run_epoch(train_loader, True)
    va_loss, va_acc = run_epoch(val_loader, False)
```

```
    print(f"epoch {epoch}: train {tr_loss:.3f}/{tr_acc:.3f}  val
{va_loss:.3f}/{va_acc:.3f}")
    if va_loss < best:                        # early stopping on validation
loss
        best, wait = va_loss, 0
        torch.save(model.state_dict(), "best.pt")
    else:
        wait += 1
        if wait >= patience:
            break
```

When it won't train — and when to stop

When a run simply refuses to learn, work through the symptoms in order instead of guessing. This checklist resolves most cases:

1. **Loss is NaN or Inf.** Lower η, clip gradients, and look for unscaled inputs, a log or divide by zero, or a corrupt label — the sign of an exploding gradient.
2. **Loss is flat from the first step.** The learning rate is too low, the gradient is not flowing (dead ReLUs or bad initialization, Chapter 10), or nothing is actually connected. Run a gradient check and confirm the optimizer has the parameters.
3. **Cannot overfit a single batch.** A wiring bug: wrong loss, labels misaligned with inputs, a detached gradient, or a loader returning the same batch every step.
4. **Trains well but validation is far worse.** Overfitting — regularize or gather more data (Chapter 12).
5. **Loss falls very slowly.** Raise η, switch to Adam (Chapter 9), or normalize the inputs and activations (Chapter 11).

Knowing when to stop is its own skill: stop when the validation metric has not improved for a chosen number of epochs, when the remaining gains are smaller than the seed-to-seed noise, or when the small extra accuracy no longer justifies the computing cost — a diminishing-returns decision, not a fixed epoch count. What you hand off then is not just a weights file: **reproducibility is the deliverable**. Record the seed, the config, the code commit, and the data version, so the number can be regenerated and audited (data versioning is the subject of Chapter

26; the subgroup-fairness audit, of Chapter 29). From there the model passes to deployment and a new set of concerns — quantization, distillation, latency, drift monitoring (Chapter 27, efficiency & deployment) — and, when its decisions must be explained, the tools of Chapter 28 (interpretability).

In this app's engine

The Playground is built around this recipe. Its **split slider** sets the validation and test fractions. The **seed control** makes a run reproducible, so you can change one thing and trust the comparison. The **divergence detector** flags the NaN/exploding-loss case of step 11 the moment it happens. The **Math Log** prints per-step gradients and losses, so a flat or exploding run can be diagnosed rather than left a mystery. The **Overfitting lab** is step 8 made visible — increase capacity or remove regularization and watch the train/validation gap open in real time.

Worked example — why accuracy lies on imbalanced data

A fraud detector is tested on 1000 transactions: 990 legitimate, 10 fraudulent. The model catches 6 frauds and raises 20 false alarms. That fills the confusion matrix: TP = 6, FP = 20, FN = 10 − 6 = 4, TN = 990 − 20 = 970.

1. Accuracy = (TP + TN)/N = (6 + 970)/1000 = 976/1000 = **97.6%** — which sounds excellent.
2. But a lazy model that predicts “legit” for *everything* scores 990/1000 = **99.0%**, beating it while catching zero fraud. Accuracy is meaningless here.
3. Precision = TP/(TP + FP) = 6/(6 + 20) = 6/26 = **0.2308**: only 23% of alarms are real fraud.
4. Recall = TP/(TP + FN) = 6/(6 + 4) = 6/10 = **0.60**: the model catches 60% of actual fraud.
5. F1 = 2·TP/(2·TP + FP + FN) = 12/(12 + 20 + 4) = 12/36 = **0.3333** — the harmonic mean that refuses to be fooled by the majority class.

The single number F1 = 0.33 tells the truth that "97.6% accurate" hid: on the class that actually matters, this model is weak. Whether to raise precision (fewer false alarms) or recall (fewer missed frauds) is a business decision — but you can only make it once you stop reading accuracy alone.

Worked example — the iterate loop on a churn classifier

A telecom predicts customer churn from 20 tabular features. Churn is 18% of rows, so **F1 on churn** is chosen up front, not accuracy. The loop of the previous sections, run with real numbers (the full project is Chapter 31's tabular case study):

1. **Baseline.** Logistic regression scores validation F1 = **0.41** — the score to beat.
2. **Single-batch check.** A 3-layer ReLU MLP drives one 32-row batch to loss **0.003** in 50 steps: the pipeline is connected correctly.
3. **First real run.** Training loss settles at 0.28, validation at 0.52 — a wide gap. Diagnosis: **overfitting**.
4. **Change one thing.** Dropout ($p = 0.3$) and weight decay ($\lambda = 10^{-4}$) narrow it to train 0.34 / val 0.44 and raise val F1 to **0.48** — past the baseline.
5. **Still a gap.** Augment the minority class and add data. Val F1 reaches **0.52**, and the curves now close — a healthy fit.
6. **Tune the rate.** A range test finds divergence near $\eta = 10^{-2}$, so training uses $\eta = 10^{-3}$ with Adam. A short random search over dropout and width adds a point, val F1 = **0.55**.
7. **Seal-test.** Opened exactly once, the test set returns F1 = **0.53** — close to validation, so the estimate holds.

No single step was clever. The win came from *reading train-versus-val each pass and changing one thing*. The 0.53 is trustworthy precisely because the test set stayed sealed until that line.

Assumptions & limitations

This recipe assumes your train, validation and test sets are drawn from the *same* distribution as production. If the world changes after deployment, yesterday's honest test score quietly becomes out of date, and monitoring must replace it. A single fixed split also gives a noisy estimate on small datasets, where k-fold cross-validation is safer. F1 weights precision and recall equally, which is rarely what a real application wants — a cancer screen values recall, a spam filter values precision — so choose the operating point carefully instead of just using F1. And no metric measures fairness: a model can show excellent overall scores while failing a subgroup badly, which is why the audit in step 12 is not optional.

Why & when this matters

Run this checklist on every supervised project, from a first Playground experiment to a deployed system — the discipline scales down to small problems and up to production. Rely on it most when stakes are high and classes are imbalanced: fraud, disease screening, defect detection, safety alerts. The steps are ordered so the cheapest checks (look, split, baseline, overfit-fifty) catch the most common and most embarrassing failures before you have spent a GPU-week — and so the number you finally report is one you have earned the right to trust.

In the real world

The discipline in this chapter is what separates a demo from a deployable system, and its most well-known failures are process failures, not modeling ones. A widely reported clinical model that “detected pneumonia” had in fact learned to recognize which hospital took the X-ray — a leaked confounding feature (the machine tag) correlated with disease rate — and fell apart on new hospitals. Amazon abandoned an internal résumé-screening

tool after audits showed it penalized the word “women's” because it was trained on a decade of male-dominated hires. Regulated industries now write this chapter's steps — sealed test sets, subgroup fairness audits, documented data and model versions (“model cards” and “datasheets”) — into standards and law such as the EU AI Act, and the entire MLOps set of tools for experiment tracking, data versioning, and drift monitoring exists to make step 12's “reproducible and fair” enforceable at scale.

Exercises

1. A classifier gives TP = 40, FP = 10, FN = 5, TN = 45. Compute accuracy, precision, recall and F1. MATH

2. Show why F1 is called a harmonic mean that “refuses to be fooled”: compute it for a model with precision = 1.0 but recall = 0.01, and compare with the plain average of the two. MATH

3. Write NumPy that turns a 2×2 confusion matrix [[TN, FP], [FN, TP]] = [[45, 5], [8, 42]] into precision, recall and F1. CODE

4. Why must standardization statistics (μ, σ) be computed on the training split alone, and what specifically leaks if you compute them over the whole dataset before splitting? CONCEPT

5. The training feature is x = [2, 4, 6, 8]. Compute its mean and (population) standard deviation, then standardize a new test value x = 10 with those training statistics. MATH

6. Step 5 of the recipe says to deliberately overfit ~50 samples before any real training. Why is this a powerful early sanity check, and what does failure tell you? CONCEPT

7. Show the “fit on train only” rule: standardize a 4-row training matrix, then apply the *same* statistics to a new test point. CODE

8. On a dataset that is 95% negative, a model scores 95% accuracy but 0% recall on the positive class. Explain what happened and which metric you would report instead. CONCEPT

9. During training the curves show training loss falling to 0.05 and still dropping, while validation loss reaches 0.30 by epoch 20 and then climbs to 0.45 by epoch 60. Name the problem, say which epoch you should have stopped at, and give two fixes. CONCEPT

10. Show the underfitting-to-overfitting transition directly: fit polynomials of increasing degree to noisy data and confirm that training error falls monotonically while validation error is U-shaped. CODE

CHAPTER 26 · IN PRACTICE

Data-centric deep learning

> *For most of deep learning's history, the way to a better model went through a better model — a new layer, a smarter optimizer, another hundred million parameters. The **data-centric** view, argued most strongly by Andrew Ng, turns that idea around: hold the architecture fixed and improve the data instead, step by step. On many real problems that is where the largest, cheapest gains hide. This chapter is about treating the dataset as the thing you build — collecting, labeling, cleaning, splitting, balancing and augmenting it. It is also about the quiet, expensive mistakes (leakage, label noise, imbalance, distribution shift) that no amount of modeling can undo.*

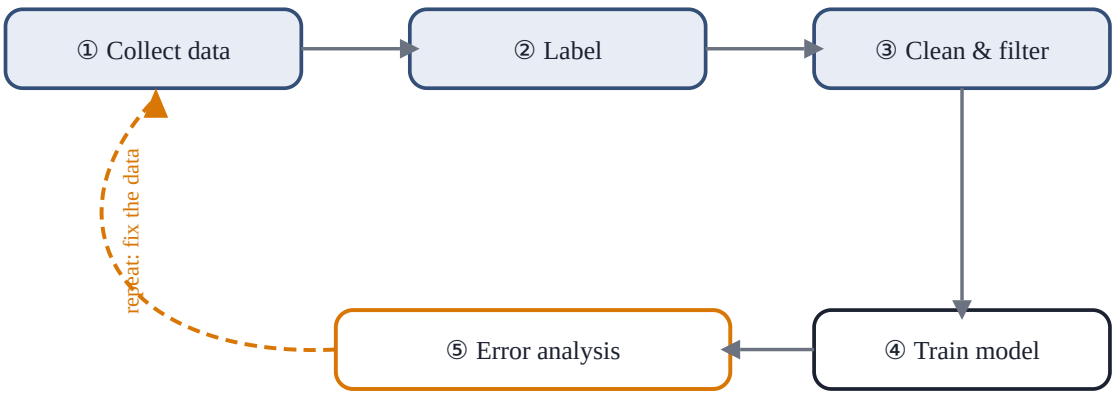
Fundamental concept

Imagine two teams given the same benchmark. The **model-centric** team freezes the data and searches over many architectures and hyperparameters. The **data-centric** team freezes a solid, ordinary model — a ResNet, a small transformer — and puts its effort into the dataset: finding mislabeled examples, defining the labeling task more clearly, collecting more of the cases the model gets wrong. From years of industrial projects, Ng observed that once you have a reasonable baseline, the second team usually wins. The reason is that production datasets are full of inconsistency, and this inconsistency limits accuracy long before model capacity does. A model can only be as good as the signal in its labels. If two annotators disagree on 15% of images, no network can be right more than 85% of the time on those images, no matter how deep it is.

The slogan is **data quality over quantity**. Ten thousand consistently-labeled examples usually beat a hundred thousand noisy ones, because noise does not average out the way beginners hope. Systematic label errors teach the model the wrong function, and random ones raise the irreducible noise floor of Chapter 12's bias–variance decomposition. Quality is not one property

but several: labels that are correct and consistent, coverage of the situations the model will actually face, balance across the classes and subgroups that matter, and freedom from features that secretly encode the answer. Improving any of these is data-centric work.

The work is not a one-off task. It is a **loop**. You collect data, label it, clean it, and train a model. Then comes the step beginners skip: **error analysis**. You look at the specific examples the model gets wrong, find the pattern (a whole category under-represented, a class of labels systematically flipped, a corner case never collected), and feed that diagnosis back into the next round of collection and labeling. The architecture barely changes, but the dataset gets better with each loop. This loop is the core of the whole chapter and of the training recipe in Chapter 25, whose very first step is "look at the data."



The data-centric loop. Unlike a pipeline that runs only once, this loop closes: error analysis (⑤) reads the model's mistakes and guides the next round of collection and labeling. The model architecture is kept fixed on purpose, so that every measured gain can be credited to the data.

The data pipeline and honest labels

A supervised dataset is built, not found, and the pipeline that builds it has stages that each fail in their own way. **Collection** gathers raw inputs from some source. The first question is whether that source matches deployment. For example, a defect detector trained only on daytime factory video has silently decided that nights do not exist. **Labeling** attaches targets, usually by humans. This is where most quality is won or lost. **Cleaning** removes duplicates, corrupt inputs,

and near-duplicates that would otherwise land on both sides of the splits. **Preprocessing** — normalization (Chapter 11), tokenization, resizing — turns raw inputs into tensors. Only then does modeling begin.

Human labels are noisier than they look. Annotators mislabel because they are tired, because a case is unclear, or because the task definition never clearly settled the hard cases: is a partly hidden stop sign a "stop sign"? Is a mixed review "positive"? This **label noise** comes in two kinds. *Random* noise (independent mistakes) mostly increases variance and can be partly averaged away with more data. *Systematic* noise, where every annotator applies the same wrong rule, is far more dangerous. It teaches the network a confidently wrong mapping, and more data only reinforces it. Northcutt and colleagues found that even trusted test sets such as ImageNet and MNIST contain thousands of label errors, enough to change how models rank.

You measure labeling quality by having several annotators label the same items and computing their **inter-annotator agreement**. Raw percent agreement is misleading, because two people who both guess the majority class will often agree just by luck. **Cohen's κ** corrects for that chance:

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

Here p_o is the observed fraction of items on which two annotators agree, and p_e is the agreement expected if each labeled independently at their own base rates. $\kappa = 1$ is perfect agreement, $\kappa = 0$ is chance-level, and negative κ is worse than chance. If two annotators agree on 90 of 100 items ($p_o = 0.90$) but one calls 30% "positive" and the other 40%, chance agreement is $p_e = 0.3 \cdot 0.4 + 0.7 \cdot 0.6 = 0.54$, so $\kappa = (0.90 - 0.54)/(1 - 0.54) = 0.78$. On the usual scale this is "substantial", not the near-perfect agreement that 90% implied. A low κ is a sign that you should fix the *instructions*, not hire more annotators: make the label definition clearer, add examples of the hard cases, and measure again. Agreement is a property of the task, and a task humans cannot agree on is a task a network cannot learn.

Splits and the many forms of data leakage

Chapter 25 sealed the test set and insisted on splitting before touching a model. This chapter focuses on the single most common way that discipline is quietly broken. **Data leakage** is any path by which information from the validation or test set (or from the future) reaches the model during training. It produces the most deceptive failure in machine learning: a model that scores very well offline but fails badly in production. The high offline score was measuring the leak, not real skill. Three forms cause most of these disasters.

Preprocessing-before-split leakage. You standardize the whole dataset with a global mean and standard deviation, *then* split. Every training feature has now taken in a statistic computed partly from the test rows. So the model is tuned on inputs that were shaped by data it will later be judged on. The fix, from Chapter 25, is to fit all preprocessing (scalers, vocabularies, imputation values, PCA bases) on the training split only, then reuse those frozen numbers everywhere. The runnable cell below shows the leak with numbers: standardizing with global statistics leaves a nonzero mean in the training set, because the test rows pulled the center.

Temporal leakage. When the task is to predict the future, a random shuffle is cheating: it lets the model train on tomorrow to predict today. A credit-risk team once built a churn model with an offline AUC near 0.95 that was useless when live. Several of its features had been aggregated over a window that included the label period, so the model was reading the answer. The fix is a **time-based split**: train on the past, then validate and test on strictly later data. Check every feature for look-ahead: anything recorded at or after the moment of prediction must go.

Group leakage. When examples come in correlated groups (multiple images of the same patient, several recordings from the same speaker, near-duplicate photos), a random split spreads members of a group across train and test. The model then scores well by recognizing the *group* rather than the pattern. The well-known case is a medical model that "detected pneumonia" but had actually learned which hospital's scanner produced each X-ray. This happened because a patient's images landed on both sides of the split (a story revisited in Chapter 25). The fix is a **grouped split**: split by patient, speaker, or source so that no group crosses the boundary, and remove duplicates first. In each case the warning sign is the same: an offline score that is too good to be true. The habit to build is to be very careful about how information can travel.

Class imbalance: resampling, weights and focal loss

Real classification targets are rarely balanced. Fraud, disease, defects, and clicks are all rare, and a dataset that is 99% negative encourages a lazy model that predicts "negative" for everything. It reaches 99% accuracy while catching nothing. This is the trap that Chapter 25 exposed with precision, recall and F1. The fixes act on the data, the loss, or the decision threshold. The key equations are these:

$$w_c = \frac{N}{K \cdot n_c} \quad \text{— inverse-frequency class weight}$$

$$L_w = -\frac{1}{m} \Sigma_i \, w_{y_i} \log p_{i,y_i} \quad \text{— reweighted cross-entropy}$$

$$FL(p_t) = -(1 - p_t)^{\gamma} \log p_t \quad \text{— focal loss } (\alpha\text{-balanced: } -\alpha_t(1 - p_t)^{\gamma} \log p_t)$$

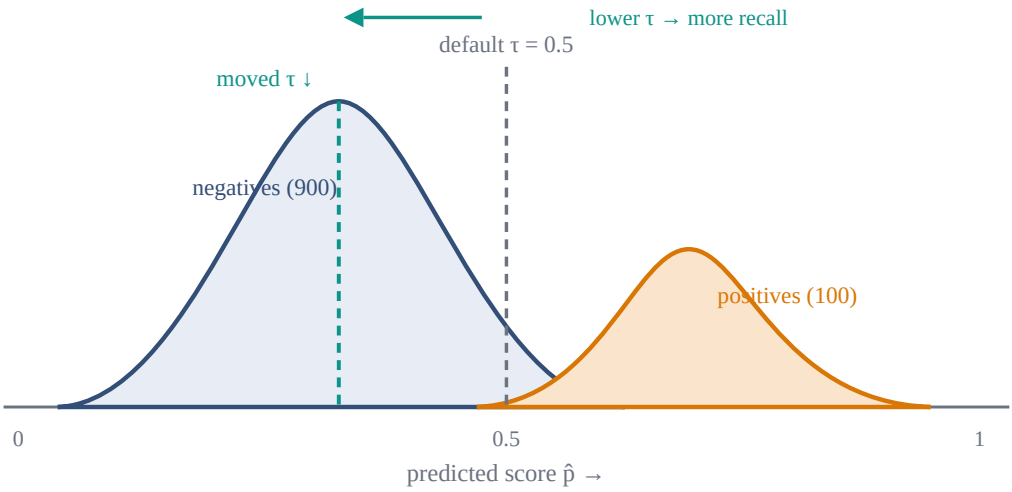
$$\tau^* = \frac{c_{FP}}{c_{FP} + c_{FN}} \quad \text{— cost-optimal decision threshold}$$

n_c, N, K	number of training examples of class c, the total count, and the number of classes
w_c	weight applied to class c in the loss. The form above makes the average weight per example exactly 1
p_t	the probability the model assigns to the *true* class of an example (so a confident-correct example has $p_t \approx 1$)
γ	focal-loss focusing parameter ($\gamma \geq 0$). $\gamma = 0$ is ordinary cross-entropy, and larger γ suppresses easy examples more strongly
α_t	optional per-class balancing factor combined with the focal term ($\alpha = 0.25$ for the rare class is common)
τ, c_{FP}, c_{FN}	the decision threshold (predict positive when $\hat{p} \geq \tau$) and the costs of a false positive / false negative

Resampling rebalances the data directly. *Oversampling* repeats or creates new minority examples (SMOTE interpolates between neighbors). *Undersampling* discards majority examples. Oversampling risks memorizing the duplicated rare cases. Undersampling throws away real information. **Class weighting** leaves the data alone and rescales the loss so each class

contributes equally to the gradient. With $w_c = N/(K \cdot n_c)$, the total weight mass on every class is N/K. So a 900:100 split, which would otherwise let the majority dominate the gradient nine-to-one, is pulled back to balance. This is exactly the reweighting the worked example computes.

Focal loss, introduced by Lin et al. for dense object detection, is more subtle: instead of reweighting by class, it reweights by *difficulty*. The modulating factor $(1 - p_t)^\gamma$ is near zero for a well-classified example ($p_t \to 1$) and near one for a badly-classified one ($p_t \to 0$). So the loss automatically stops paying attention to the easy majority and focuses on the hard, usually rare, cases, without any resampling. At $\gamma = 2$ a confidently-correct example contributes only 1% of its cross-entropy loss, while a hard example keeps nearly half of its own. Finally, **threshold moving** separates the decision from training. A model trained on rebalanced data produces distorted probabilities, so you tune the operating threshold τ on validation data. You move it toward the class prior, or, when errors have different costs, to the cost-optimal τ^* above. Lowering τ raises recall (catch more positives) at the price of precision.



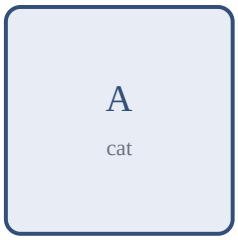
Imbalance and the moving threshold. The rare positive class (amber) is far outnumbered by the negatives (blue), and its scores tend to be low, so the default $\tau = 0.5$ misses most positives. Sliding τ left (teal) catches more of them, trading precision for recall. This is why the threshold is tuned after training, not fixed at 0.5.

Augmentation: encoding the invariances you know

Data augmentation is the bridge between this chapter and regularization (Chapter 12), where it appeared as a variance-reducing regularizer. The data-centric view is sharper: augmentation is how you **inject prior knowledge** about which transformations leave the label unchanged. Every time you flip an image of a cat and keep the label "cat," you are telling the network that class identity does not change under left–right reflection. Otherwise it would have to learn this fact from scratch, using capacity and data. Augmentation creates labeled examples for free by applying label-preserving transformations. On limited data it is often the single most powerful tool available. It is more effective than any weight penalty, because it encodes real structure rather than a generic preference for smoothness.

In **vision** the common methods are geometric (random crop, horizontal flip, small rotations and scalings) and photometric (brightness, contrast, and color jitter). These encode invariance to position, framing and lighting. Automated policies such as **RandAugment** combine several such operations at a randomly chosen magnitude, so you do not have to tune each one by hand. Two mixing methods go further by blending examples. **mixup** trains on convex combinations of two inputs *and* their labels, $\tilde{x} = \lambda x_A + (1-\lambda)x_B$ and $\tilde{y} = \lambda y_A + (1-\lambda)y_B$, with λ drawn from a Beta distribution. It forces the network to behave linearly between classes and greatly improves calibration (Chapter 24). **CutMix** instead pastes a rectangular patch from one image onto another and mixes the labels in proportion to the patch area. Note that a horizontal flip is safe for cats but not for handwritten digits (a flipped "2" is nonsense). The invariances you encode must be true of the task, or augmentation adds false knowledge.

Other data types have their own label-preserving methods. In **text**, **back-translation** rewrites a sentence by translating it to another language and back. This keeps the meaning while changing the wording. **EDA** (Easy Data Augmentation) does simple synonym replacement, random insertion, swap and deletion. In **audio**, **SpecAugment** masks bands of time and frequency directly on the spectrogram. This teaches a speech model to cope with missing or corrupted regions. The common idea is always the same question: what can I change about the input that must not change the label?



y

$_A$ = [1,

0] B dog y_B = [0, 1] $\lambda x_A+(1-\lambda)x_B$ $\tilde{y}$ = [0.6, 0.4] λ = 0.6 1 − λ = 0.4

mixup interpolating two examples and their labels. The blended input is $\lambda x_A + (1-\lambda)x_B$ and its soft target is the same mixture of the one-hot labels, $\tilde{y}$ = [0.6, 0.4]. Training on such interpolations linearizes the model between classes and improves calibration. It is an augmentation that is also a regularizer.

Getting more from less: synthetic data, curriculum and active learning

When real labeled data is scarce, three strategies get more out of what you have. **Synthetic data** generates examples from a simulator or a generative model (Chapter 21): rendered scenes for self-driving perception, physics engines for robotics, and GAN- or diffusion-sampled images to fill out rare classes. Synthetic data is unlimited and comes with perfect labels. But it carries a **sim-to-real gap**: a model trained only in simulation over-fits the simulator's oddities and fails on real inputs. So synthetic data is usually mixed with real examples, or fine-tuned on them, rather than trusted alone.

Curriculum learning, proposed by Bengio et al., orders the training examples from easy to hard instead of showing them in random order. This mirrors how humans are taught. Presenting clean, clear cases first can give the optimizer a smoother path to a good basin before it meets the confusing examples. This sometimes improves both convergence speed and final accuracy. But the ordering must be chosen with care, since a poor curriculum can also lock in early biases.

Active learning works directly on the labeling budget and connects straight back to uncertainty (Chapter 24). Instead of labeling data at random, you label the examples the model is *least sure about*. The loop works like this: train on a small labeled seed, run the model over a large

unlabeled pool, and score how informative each point is. The simplest measure is predictive uncertainty, such as a low margin between the top two class probabilities or high entropy. Then label the most uncertain handful, add them, and repeat. Because a confident-correct example teaches almost nothing, while an example near the boundary is highly informative, active learning can reach a target accuracy with a fraction of the labels that random sampling would need. One honest warning is that uncertainty estimates from a half-trained network are themselves unreliable (Chapter 24's calibration problem). So practitioners combine uncertainty with diversity, to avoid labeling a cluster of near-identical hard cases.

Documentation, bias and distribution shift

A dataset without documentation is a risk. Two documents, now standard, make its assumptions clear. **Datasheets for datasets** (Gebru et al.) record how the data was collected, who labeled it, what it does and does not represent, and which uses are not appropriate. This is the background a downstream engineer needs to judge whether the data fits. **Model cards** (Mitchell et al.) do the same for a trained model. They report its intended use and its performance broken down across subgroups. Both exist because **biased data yields a biased model**. A résumé screener trained on a decade of male-dominated hires learns to penalize signals of being a woman. A face system trained mostly on light-skinned faces fails on dark-skinned ones. Overall accuracy hides these failures. Only a subgroup audit reveals them. This is why fairness is a data property before it is a modeling one, a topic Chapter 29 takes up in full.

Finally, all of supervised learning rests on one assumption: that training and deployment data are drawn from the same distribution. In the long run this is usually false, and its failure is called **distribution shift**. It comes in three kinds. **Covariate shift**: the input distribution $p(x)$ changes while the labeling rule $p(y|x)$ stays the same. Examples: a medical model meeting a new hospital's scanner, or a vision model meeting winter after training on summer. **Label (prior) shift**: the class base rates $p(y)$ change (for example, a disease grows more common) while $p(x|y)$ stays stable. Threshold moving can partly correct this. **Concept shift**: the relationship $p(y|x)$ itself changes. For example, spammers invent a new trick, so inputs that used to be safe are now malicious. Each one quietly makes a model worse, because the sealed test set

from training time no longer resembles today's inputs. The defenses are data-centric: monitor input and prediction distributions in production, detect drift, and retrain on fresh, correctly-labeled data. This closes the loop of the opening figure once more.

Data-centric work in code

Three of this chapter's ideas come down to a few lines of NumPy. The first cell builds the inverse-frequency class weights for the 900:100 set, checks that they rebalance each class's contribution to the gradient, and makes a table of focal loss against cross-entropy across a range of p_t. Press **Run**. The printed weights and the 100×-versus-2× down-weighting reproduce the worked example exactly.

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)

# --- inverse-frequency class weights, normalized so the per-example average
is 1 ---
counts = np.array([900, 100])          # [negatives, positives]
N, K = counts.sum(), len(counts)
w = N / (K * counts)                   # w_c = N / (K * n_c)
print("class weights         =", w.round(4))            # [0.5556 5.    ]
print("ratio w[pos]/w[neg]   =", round(float(w[1]/w[0]), 3))   # 9.0 ==
900/100

avg = (counts * w).sum() / N           # average weight per example
print("avg weight per example =", round(float(avg), 4))  # 1.0  (that is the
normalization)

mass = counts * w                      # total weight each class puts on the
loss/gradient
print("weighted mass per class =", mass.round(1))        # [500. 500.] ->
rebalanced

# --- focal loss vs cross-entropy across a range of p_t (gamma = 2) ---
gamma = 2.0
pt = np.array([0.9, 0.7, 0.5, 0.3, 0.1])
ce = -np.log(pt)                       # cross-entropy of the true class
fl = (1 - pt)**gamma * ce              # focal loss FL = -(1-pt)^gamma * log
```

```
pt
print("p_t          :", pt)
print("cross-entropy:", ce.round(4))
print("focal (g=2)  :", fl.round(4))
print("modulator    :", ((1 - pt)**gamma).round(4))      # 0.01 at pt=0.9,
0.49 at pt=0.3
```

The equal 500/500 mass is the whole point of class weighting: the majority no longer outweighs the minority in the gradient. In the focal columns the modulating factor is 0.01 for the easy example ($p_t = 0.9$) but 0.49 for the hard one ($p_t = 0.3$). So focal loss shrinks the easy example's contribution a hundredfold, while barely touching the hard case. The second cell shows two more one-liners. The first is a mixup interpolation. The second is a live demonstration of preprocessing-before-split leakage:

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(1)

# --- mixup: convex-combine two examples AND their one-hot labels ---
lam = np.random.beta(0.2, 0.2)                 # mixing coefficient ~
Beta(a, a)
xa, xb = np.array([0., 1., 2., 3.]), np.array([4., 3., 2., 1.])
ya, yb = np.array([1., 0.]), np.array([0., 1.])    # one-hot: class 0 vs
class 1
xm = lam * xa + (1 - lam) * xb
ym = lam * ya + (1 - lam) * yb
print("lambda   =", round(float(lam), 4))
print("mixed x  =", xm.round(4))
print("mixed y  =", ym.round(4), " (soft label sums to",
round(float(ym.sum()), 3), ")")

# --- leakage: fit the scaler on ALL data (wrong) vs on the training split
only ---
feat = np.random.normal(10.0, 3.0, 200)        # one feature, full dataset
train, test = feat[:150], feat[150:]
mu_all = feat.mean()                           # WRONG: this mean saw the
test rows
mu_tr  = train.mean()                          # RIGHT: training statistics
only
print("mean(all) =", round(float(mu_all), 4),
      "  mean(train) =", round(float(mu_tr), 4),
```

```
      "  leaked shift =", round(float(mu_all - mu_tr), 4))
print("centered-train mean, GLOBAL stats =", round(float((train -
mu_all).mean()), 4))  # != 0
print("centered-train mean, TRAIN  stats =", round(float((train -
mu_tr).mean()), 4))   # == 0
```

The soft label [0.06, 0.94] is a genuine target, not a rounding error. Here mixup asks the network to predict a blend. In the leakage half, centering the training set with the *global* mean leaves a nonzero mean (here about −0.18), because the test rows shifted the center. Centering with the training mean leaves exactly 0. That leftover value is the clear sign of test information leaking backward into training. In PyTorch the imbalance and augmentation tools are all provided. Here is a focal-loss module, a `WeightedRandomSampler` that draws class-balanced batches, and a torchvision augmentation pipeline that encodes the invariances of the task:

PyTorch copy → Colab

```
import torch, torch.nn as nn, torch.nn.functional as F
from torch.utils.data import DataLoader, WeightedRandomSampler
from torchvision import transforms

# --- 1. focal loss as a drop-in module (Lin et al., 2017) ---
class FocalLoss(nn.Module):
    def __init__(self, gamma=2.0, alpha=None):        # alpha: optional per-
class weights
        super().__init__()
        self.gamma, self.alpha = gamma, alpha
    def forward(self, logits, target):
        logp  = F.log_softmax(logits, dim=1)
        logpt = logp.gather(1, target[:, None]).squeeze(1)   # log p_t of
the true class
        pt    = logpt.exp()                                  # p_t
        loss  = -(1 - pt) ** self.gamma * logpt              # FL = -(1-
pt)^gamma * log pt
        if self.alpha is not None:
            loss = loss * self.alpha[target]                 # class-
balanced variant
        return loss.mean()

# --- 2. WeightedRandomSampler: oversample the minority into every mini-
batch ---
counts   = torch.bincount(train_labels)                 # e.g. tensor([900,
```

```
100])
class_w  = 1.0 / counts.float()                          # inverse frequency
sample_w = class_w[train_labels]                         # one weight per
training example
sampler  = WeightedRandomSampler(sample_w, num_samples=len(sample_w),
replacement=True)
loader   = DataLoader(train_ds, batch_size=64, sampler=sampler)   # (no
shuffle= with a sampler)

# --- 3. torchvision augmentation: label-preserving transforms only ---
train_tf = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.6, 1.0)),   # position /
scale invariance
    transforms.RandomHorizontalFlip(),                     # left-right
invariance
    transforms.ColorJitter(0.4, 0.4, 0.4),                 # lighting
invariance
    transforms.RandAugment(num_ops=2, magnitude=9),        # an automated
augmentation policy
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],            # ImageNet stats
(fit on train!)
                         [0.229, 0.224, 0.225]),
])
test_tf = transforms.Compose([                             # NO random ops
at test time
    transforms.Resize(256), transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
])
```

In this app's engine

The Playground makes several of these ideas concrete. Its **split slider** sets the train / validation / test fractions, and its **seed control** fixes the shuffle, so you can watch how a fragile single split (Chapter 25) moves the score. The **dataset cards** show class balance up front, and the confusion matrix and precision/recall panels reveal the imbalance trap the moment a rare class

appears. The Overfitting lab's augmentation and noise controls are this chapter's variance argument on a small scale. Add label noise and watch the achievable accuracy ceiling drop, exactly as inter-annotator disagreement predicts.

Worked example — reweighting and focal loss on a 900:100 set

A binary training set has 900 negatives and 100 positives (N = 1000, K = 2). We compute inverse-frequency class weights, then compare focal loss ($\gamma = 2$) with cross-entropy on one easy and one hard positive.

1. Raw inverse frequency: $1/n_0 = 1/900$, $1/n_1 = 1/100$. To normalize so the average weight per example is 1, scale by N/K = 500, giving $w_c = N/(K \cdot n_c)$: $w_0 = 500/900 =$ **0.556**, $w_1 = 500/100 =$ **5.000**.
2. Check the normalization: per-example mean weight = (900·0.556 + 100·5.000)/1000 = (500 + 500)/1000 = **1.000**. Each class now carries equal total mass 500, so the 9:1 gradient imbalance is erased. The minority's weight is larger by exactly the imbalance ratio: 5.000/0.556 = **9**.
3. Easy positive, $p_t = 0.9$. Cross-entropy: CE = −ln 0.9 = **0.1054**. Focal modulator: $(1 - 0.9)^2 = 0.1^2 =$ **0.01**. Focal loss: FL = 0.01 · 0.1054 = **0.001054**. So the focal loss keeps just 1% of the cross-entropy (a 100× down-weight).
4. Hard positive, $p_t = 0.3$. Cross-entropy: CE = −ln 0.3 = **1.2040**. Focal modulator: $(1 - 0.3)^2 = 0.7^2 =$ **0.49**. Focal loss: FL = 0.49 · 1.2040 = **0.5899**. So the focal loss keeps 49% (barely a 2× down-weight).
5. Relative emphasis. Under cross-entropy the hard example's loss is 1.2040/0.1054 = **11.4×** the easy one's. Under focal loss it is 0.5899/0.001054 = **560×**. Focal loss has multiplied the hard example's relative pull by 0.49/0.01 = **49×**.

What the numbers mean: class weighting and focal loss attack imbalance in two different ways. The weights rebalance *classes* so the rare positives get attention. Focal loss rebalances by *difficulty*, almost erasing the confident-correct examples (down 100×)

while sparing the hard ones (down only 2×). The net effect is that a network's gradient stops being dominated by the thousands of easy negatives and flows to the handful of cases that actually decide the model. And it does all this with no change to the data at all.

Assumptions & limitations

Data-centric methods are not without risk. Class weighting and focal loss assume your *labels* are trustworthy. On a noisy minority class they amplify the noise, obediently chasing the mislabeled rare examples. Oversampling can memorize the duplicated points, and heavy augmentation can encode a false invariance (flipping a digit, color-jittering a medical stain) that quietly corrupts the labels. mixup's soft targets help calibration but can hurt tasks where hard boundaries are real. Active learning's uncertainty scores are only as good as the model's calibration (Chapter 24), and a biased seed set biases every later query. Above all, no reweighting or augmentation creates information that was never collected. If a subgroup is missing from the data, the model is blind to it, and the honest fix is to collect more data, not to change the loss function.

Why & when this matters

Turn to the data-centric loop first whenever a model performs poorly and the architecture is already reasonable. On real problems, this is most of the time. Check labels and inter-annotator agreement when accuracy levels off below what the task should allow. Look for leakage the instant an offline score looks too good. Apply class weights, focal loss or resampling the moment a class is rare and matters. Add augmentation whenever data is limited and you can name a true invariance. The ordering matches Chapter 25's recipe: look at the data, split it honestly, then spend your effort where the signal actually is. A day spent fixing labels often beats a week spent tuning a network.

In the real world

Industrial machine learning is largely data work. Andrew Ng's data-centric effort came out of manufacturing projects where the datasets were small and inconsistent, and the fastest wins came from relabeling, not remodeling. Tesla's Autopilot team is organized around a data engine that mines fleet video for the rare cases the model fails on and sends them back for labeling. This is the loop of the opening figure on a continental scale. Focal loss was invented to train dense object detectors (RetinaNet) that were flooded with easy background boxes. It now appears across imbalanced medical and fraud pipelines. Regulated fields increasingly require the documentation in this chapter: datasheets and model cards are appearing in procurement and in laws such as the EU AI Act. And the entire MLOps discipline of drift monitoring exists because distribution shift, not model error, is what most often ends a deployed model's useful life.

Exercises

1. A three-class set has counts [700, 200, 100]. Compute inverse-frequency class weights normalized so the average weight per example is 1, and state the largest weight ratio. MATH

2. For focal loss with $\gamma = 2$, compute the modulating factor $(1 - p_t)^\gamma$ at $p_t = 0.99$ and $p_t = 0.5$, and compare $\gamma = 1$ with $\gamma = 2$ at $p_t = 0.9$. Interpret. MATH

3. Two annotators label 100 items. They agree on 82. Annotator A marks "positive" 25 times, B marks it 35 times. Compute observed agreement, expected agreement, and Cohen's κ, and interpret. MATH

4. Name the leakage type in each scenario and give the fix: (a) you standardize features using the whole dataset's mean and standard deviation, then split; (b) predicting next-quarter churn, you shuffle rows randomly into train and test; (c) a diagnosis model puts several scans of the same patient into both train and test. CONCEPT

5. Verify the class-weight rebalancing in code: for counts [900, 100], compute inverse-frequency weights and show each class contributes equal weight mass to the loss. CODE

6. Make a table of the focal modulating factor $(1 - p_t)^\gamma$ for γ in $\{0, 1, 2, 5\}$ at $p_t = 0.9, 0.5, 0.1$, and describe how increasing γ reshapes the loss. CODE

7. You have 1,000,000 unlabeled examples and budget to label only 10,000. Describe an active-learning loop that spends the budget well, and explain why it beats labeling 10,000 at random. Connect your answer to uncertainty (Chapter 24). CONCEPT

8. Define covariate, label (prior), and concept shift. A spam filter that worked for months suddenly lets a brand-new campaign through. Which shift is this most likely to be, and what is the cheapest lasting fix? CONCEPT

CHAPTER 27 · IN PRACTICE

Efficiency & deployment

> *A model that wins a benchmark is only half a system. The other half is whether it can answer in ten milliseconds, fit in a phone's memory, run on a battery, and cost less per query than it earns.* ***Efficiency*** *is the engineering that turns a trained network into a deployable one — trimming its arithmetic, shrinking its numbers, and matching it to real hardware — usually while giving up almost none of its accuracy. This chapter is the set of tools: how to measure cost honestly, and the few techniques — quantization, pruning, distillation, efficient architectures, and compilation — that make big models small and fast.*

Fundamental concept

Think of a Formula 1 car and a delivery van. Both are "good vehicles", but they are optimized for different quantities: one for lap time on a closed track, the other for cost per parcel on real roads. A research model is the race car — it maximizes accuracy on a leaderboard, with a datacenter of GPUs behind it and no time limit. A **deployed** model is the van: it must meet a latency budget, stay inside a memory limit, use little energy, and be cheap enough to run millions of times a day. The recipe of Chapter 25 gets you a model that is *correct*. This chapter gets you one that is *affordable*.

Five costs drive every deployment decision. **Latency** is the time to answer one request — the number a user notices, and a strict limit in interactive settings (a keyboard's autocomplete, a car's pedestrian detector). **Throughput** is how many requests a server handles per second, which sets the size of your GPU fleet and therefore the bill. **Memory** — the model's weights plus its working activations — decides whether it fits on the target device at all. **Energy** matters twice: as battery life on a phone or drone, and as the electricity and carbon of a datacenter serving billions of queries. And **cost**, in dollars, is the quantity a business ultimately optimizes, which depends on the other four.

These costs matter most at the **edge** — on phones, cameras, watches, cars, and microcontrollers, where there is no datacenter to rely on. On-device inference gives privacy (data never leaves the device), works offline, and answers with no network round-trip, but it must run in a few hundred megabytes and a watt or two. The techniques below all serve one goal: use less compute and memory for nearly the same predictions. Importantly, efficiency comes *after* accuracy in the workflow — first build a model that works (Chapter 25), then make it cheap. Optimizing a model that is not yet correct only makes a broken model run faster.

What to measure — and why FLOPs are not latency

Before optimizing anything, measure the right numbers. A layer's **parameter count** sets its memory footprint. Its **MAC** count (multiply–accumulate operations) sets its arithmetic. For a linear layer mapping n_{in} inputs to n_{out} outputs, and a convolution with a k×k kernel over $C_{in} \to C_{out}$ channels producing an H×W map:

$$\text{params}_{\text{linear}} = n_{in} \cdot n_{out} + n_{out} \qquad \text{MACs}_{\text{linear}} = n_{in} \cdot n_{out}$$

$$\text{MACs}_{\text{conv}} = k \cdot k \cdot C_{in} \cdot C_{out} \cdot H \cdot W \qquad \text{FLOPs} \approx 2 \cdot \text{MACs}$$

One MAC is a multiply and an add, so a FLOP (floating-point operation) count is about twice the MAC count — so be careful which one a paper reports. Memory is a parameter count times the bytes per number, the quantity every compression method reduces:

$$\text{bytes} = \text{params} \cdot b, \quad b = 4\ (\text{fp32}),\ \ 2\ (\text{fp16 / bf16}),\ \ 1\ (\text{int8})$$

A common mistake is to treat FLOPs as a substitute for time. They are not. Real latency is set by the **roofline** model: a kernel is either *compute-bound*, limited by the chip's peak arithmetic rate π, or *memory-bound*, limited by how fast bytes move from memory at bandwidth β. Which regime you are in depends on the **arithmetic intensity** I — useful FLOPs performed per byte moved:

$$I = \frac{\text{FLOPs}}{\text{bytes moved}} \qquad \text{attainable rate} = \min(\,\pi,\ \ I \cdot \beta\,)$$

params	number of learned weights (and biases) in a layer or model; times b gives its size in bytes
MAC	one multiply–accumulate, a·b + c; the natural unit of neural-network arithmetic
FLOP	one floating-point operation; FLOPs ≈ 2·MACs since each MAC is a multiply plus an add
b	bytes per stored number: 4 for fp32, 2 for fp16/bf16, 1 for int8 — the setting quantization changes
I	arithmetic intensity, FLOPs per byte of memory traffic; low I ⇒ memory-bound, high I ⇒ compute-bound
π, β	the hardware's peak compute rate (FLOP/s) and memory bandwidth (bytes/s); their ratio π/β is the roofline "ridge"

This is why FLOPs mislead. An operation with low arithmetic intensity — a depthwise convolution, an elementwise activation, the tiny matrix-vector products of a batch-size-one language model — spends most of its time waiting for memory, so cutting its FLOPs does nothing. Only cutting its *memory traffic* helps. Two models with identical FLOP counts can differ 3× in latency because one keeps the hardware's matrix units fully busy and the other waits. Kernel-launch overhead, parallelism, cache behavior, and how well an operation maps to the chip's tensor cores all come between a FLOP count and a millisecond. The rule is simple: **measure latency and peak memory on the target hardware**, at the batch size and sequence length you will actually serve, and treat params and FLOPs as design-time estimates, not promises.

Numeric precision: floats and INT8 quantization

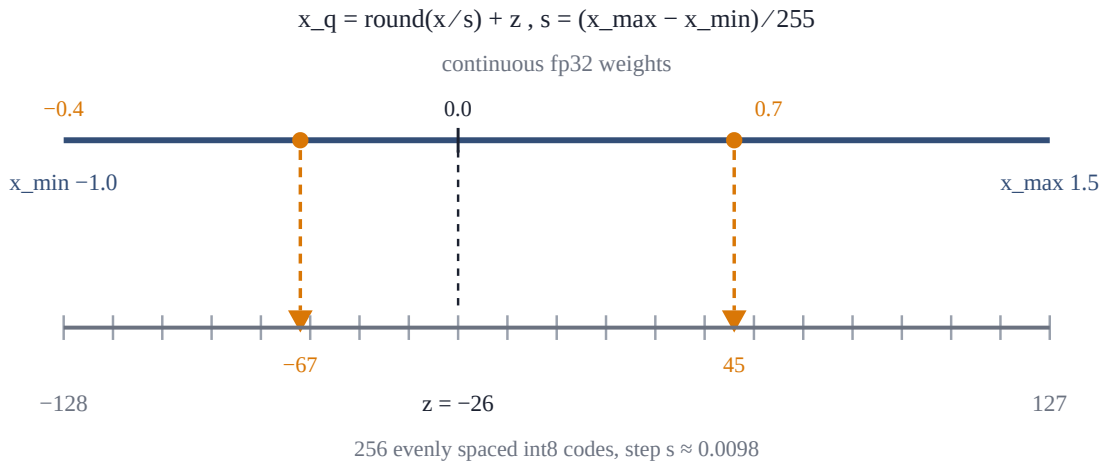
The cheapest thing to change is the size of each number. Training normally uses 32-bit floats (fp32), but most of that precision is wasted — networks are robust to noise, so their weights and activations still work with far fewer bits. **Mixed-precision training** keeps a master copy of the weights in fp32 but runs the heavy matmuls in a 16-bit float, roughly doubling throughput and halving memory. Two 16-bit formats compete. **fp16** has a 5-bit exponent and 10-bit mantissa: precise, but its narrow exponent range underflows small gradients to zero, so it needs **loss scaling** (multiply the loss by a large constant before backprop, divide it out of the gradients afterward). **bf16** keeps fp32's full 8-bit exponent and gives up mantissa bits instead, so it has the same dynamic range as fp32 and trains stably with no loss scaling — which is why modern accelerators prefer it.

Inference can go further, all the way to 8-bit integers. **INT8 quantization** maps a floating-point range onto the 256 codes of a signed byte with an **affine** (uniform) rule. Given the range $[x_{min}, x_{max}]$ of the tensor and integer limits $[q_{min}, q_{max}] = [-128, 127]$, choose a **scale** s and integer **zero-point** z so that the real value 0 falls exactly on a code:

$$s = \frac{x_{max} - x_{min}}{q_{max} - q_{min}} \qquad z = q_{min} - \text{round}(x_{min} / s)$$

$$x_q = \text{clip}(\text{round}(x / s) + z,\ q_{min}, q_{max}) \qquad \hat{x} = s \cdot (x_q - z)$$

Quantizing then dequantizing is not lossless: every value is moved to its nearest grid point, so the round-trip error is at most half a step, $|x - \hat{x}| \le s/2$. But the benefit is large and cheap — 8-bit weights are 4× smaller than fp32 and run on fast integer units — and for most networks the accuracy drop is under a percentage point. Real ranges are asymmetric (a ReLU output is never negative), which is exactly why the zero-point exists: it lets the grid cover an uneven range without wasting codes.



Affine INT8 quantization. A continuous fp32 range is split into 256 integer codes. Each weight moves to its nearest code (dashed amber), so the round-trip error is at most half a step, s/2. The zero-point z = −26 is the code that real 0.0 maps to, letting an asymmetric range use the grid without waste.

Two workflows produce a quantized model. **Post-training quantization (PTQ)** takes a finished fp32 network and converts it directly, using a small *calibration* set of a few hundred examples to estimate each tensor's range. It is fast, needs no labels and no retraining, and is the usual first choice. The cost is a small accuracy drop that grows as you push below 8 bits. When that drop is

unacceptable, **quantization-aware training (QAT)** simulates the rounding *during* training — inserting "fake-quant" operations in the forward pass and letting a straight-through estimator pass gradients around the non-differentiable round — so the weights learn to be robust to their own quantization. QAT recovers most of the lost accuracy and is what makes aggressive 8-bit and 4-bit deployment possible, at the price of a training run. The choice is the usual one: use PTQ first, and move to QAT only when the accuracy budget demands it.

Pruning and sparsity

A trained network has more capacity than it needs: many weights end up near zero and contribute almost nothing. **Pruning** removes them. The simplest and most reliable method is **magnitude pruning** — rank the weights by absolute value and set the smallest fraction to zero — based on the idea that a weight near zero barely moves any activation, so deleting it barely changes the output. The fraction of weights that are zero is the **sparsity**. A network pruned to 90% sparsity keeps only a tenth of its connections. Pruning is often followed by a short fine-tune so the remaining weights make up for the removed ones, and the prune-then-fine-tune cycle can be repeated to reach high sparsity with little accuracy loss.

The key question is the *shape* of the zeros. **Unstructured pruning** removes individual weights wherever they are smallest, which preserves accuracy best because it is as flexible as possible. But it leaves a matrix that is mostly zeros scattered at random, and general-purpose hardware runs a randomly sparse matmul no faster than a dense one. Getting the speed-up needs specialized sparse kernels or hardware. **Structured pruning** instead removes whole units — entire rows of a weight matrix, entire channels or filters of a convolution — so the layer becomes a truly *smaller dense* layer that every GPU runs faster immediately. Structured pruning is less flexible, so it usually costs more accuracy at the same sparsity, but it is the one that reliably cuts latency on ordinary hardware. This trade-off is the recurring theme of this chapter: unstructured wins on the metric of accuracy-per-weight, structured wins on the metric of actual wall-clock speed.

Two ways to prune a weight matrix. Unstructured pruning (left) deletes the smallest individual weights wherever they fall, keeping accuracy but leaving an irregular pattern that ordinary hardware cannot use. Structured pruning (right) deletes whole rows — here two of the four output neurons — giving a smaller dense matrix that every GPU runs faster immediately.

Pruning also carries a surprising scientific result, the **lottery-ticket hypothesis** (Frankle & Carbin, 2019): inside a large, randomly initialized network there exists a small sparse subnetwork — a "winning ticket" — that, if trained in isolation *from the original initialization*, can match the full network's accuracy. The problem is that the ticket is found only after training the dense network and pruning it. You cannot easily guess which connections will win. The hypothesis describes over-parameterization not as waste but as buying many lottery tickets at once, and it helps explain why pruning works so well afterward.

Distillation, low-rank factorization & efficient architectures

If pruning and quantization shrink a given model, three more ideas build a small model from the start. **Knowledge distillation** (Hinton et al., 2015) trains a compact *student* to imitate a large *teacher* — not just its hard labels but its softened probability distribution, produced by dividing the logits by a temperature T > 1. Those "soft targets" carry the teacher's dark knowledge (that a 7 looks a little like a 1), a far richer signal than a one-hot label, so the student reaches accuracy it could never learn from the raw data alone. Distillation is a form of transfer between models and is developed fully in Chapter 22 (transfer learning & fine-tuning). Here it is a compression tool — DistilBERT, for instance, keeps ~97% of BERT's accuracy at 40% of its size.

Low-rank factorization works on a big weight matrix directly. A dense W of shape m×n costs m·n numbers, but if it is approximately low rank it can be written as a product of two thin matrices, W ≈ U V with U of shape m×r and V of shape r×n, costing only m·r + r·n:

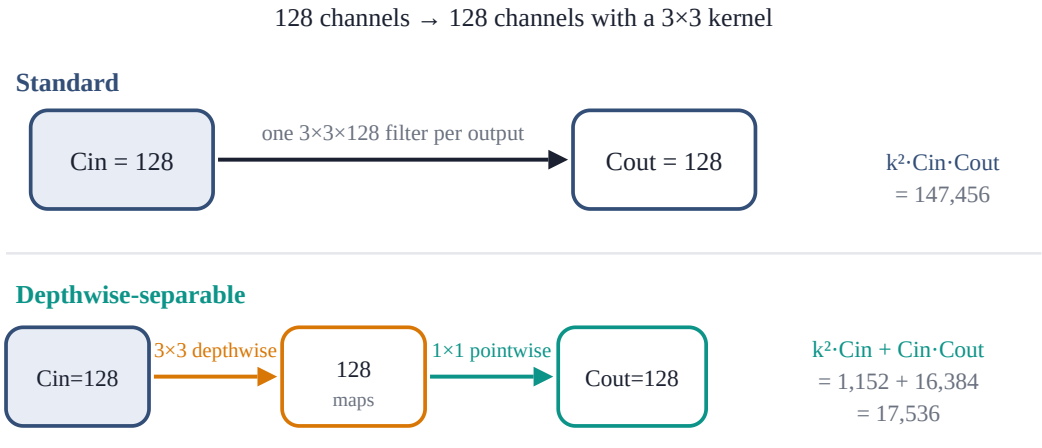
$$W \approx U\,V, \quad \text{params: } m{\cdot}r + r{\cdot}n \ll m{\cdot}n \quad \text{when } r \ll \frac{m{\cdot}n}{m+n}$$

For a 1024×1024 layer, rank r = 64 cuts the parameters from ~1.05M to ~131K, an 8× saving, at the cost of limiting the map to rank 64. The same factorization is the basis for the LoRA adapters of parameter-efficient fine-tuning (Chapter 22), where only the small U and V are trained.

The most influential efficient-architecture idea changes the convolution itself. A standard 3×3 convolution does two jobs at once — it filters spatially *and* mixes channels — at a cost of $k^2 \cdot C_{in} \cdot C_{out}$ MACs per pixel. The **depthwise-separable convolution** of MobileNet (Howard et al., 2017) separates these: a **depthwise** step applies one k×k filter per input channel (spatial filtering, no channel mixing, $k^2 \cdot C_{in}$ MACs), then a **pointwise** 1×1 convolution mixes channels ($C_{in} \cdot C_{out}$ MACs). The ratio of the two costs is the main reason MobileNet fits on a phone:

$$\frac{k^2 C_{in} + C_{in} C_{out}}{k^2 C_{in} C_{out}} = \frac{1}{C_{out}} + \frac{1}{k^2}$$

With a 3×3 kernel this is about $1/C_{out} + 1/9$, so for any reasonable channel count the reduction is roughly $k^2 = 9\times$. The worked example works out the exact numbers for $C_{in} = C_{out} = 128$ and gives 8.4×. This factorization — spatial and channel mixing done in separate cheap steps — turns the convolution of Chapter 13 (convolutional neural networks) into the mobile computer-vision backbones of Chapter 14 that run on-device.



Standard vs depthwise-separable convolution. The standard conv (top) filters space and mixes channels in one expensive step. The separable version (bottom) splits the work into a cheap per-channel 3×3 depthwise step and a 1×1 pointwise mixing step, cutting the arithmetic ~8.4× here — though its low arithmetic intensity means the wall-clock speed-up is often smaller.

Compilers, runtimes and serving

A trained network in PyTorch is a Python program. A deployed one is a compiled artifact. The bridge is usually **ONNX**, an open interchange format that captures the computation graph so it can be passed to a specialized **runtime**: **TensorRT** for NVIDIA GPUs, **TensorFlow Lite** (and Core ML) for phones, ONNX Runtime for servers. These compilers apply optimizations a general framework cannot. The most important is **operator fusion**: a convolution followed by a batch-norm (Chapter 11) followed by a ReLU is three passes over memory in eager mode, but fuses into a single kernel that reads the input once, does all three operations in registers, and writes the output once. Because so many layers are memory-bound, reducing this traffic — not reducing FLOPs — is often where the real speed-up comes from. Compilers also fold constants, pick hardware-specific kernels, and plan memory in advance. PyTorch's own `torch.compile` and XLA do much of this in-framework.

At serving time the main tool is **batching**. A GPU is a wide parallel machine that sits idle on a single input. Stacking many requests into one batch raises arithmetic intensity and keeps the matrix units busy, increasing throughput several times over. The cost is latency. A request may wait for the batch to fill, so production systems use *dynamic batching* with a short timeout to

balance the two. Batching is the clearest example of the latency/throughput tension: the configuration that serves the most requests per dollar is rarely the one that answers any single request fastest.

Autoregressive language models (Chapter 17) add their own serving method, the **KV cache**. If done in the simple way, generating text one token at a time would re-attend over the entire prefix at every step — $O(T^3)$ work for a length-T sequence. Because a past token's keys and values never change, they are cached. Each new token computes only its own query, key and value, turning generation into $O(T^2)$ total. The cost moves from compute to memory: the cache holds two vectors per layer per past token, on the order of $2 \cdot L \cdot T \cdot d$ numbers per sequence, which for a long context is gigabytes and is what actually limits a served model's batch size and context length. Quantizing the KV cache and sharing keys/values across attention heads (grouped-query attention) are, for exactly this reason, active areas of inference-efficiency work — the deployment side of the architectures in Chapter 17.

Efficiency in code

The two methods you can see in a few lines of NumPy are quantization and pruning. The first cell affine-quantizes a small fp32 weight vector to signed INT8 and back, printing the scale, zero-point, round-trip error and the 4× compression — the exact numbers of the worked example below. Press **Run**.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.set_printoptions(suppress=True)

w = np.array([-1.0, -0.4, 0.0, 0.3, 0.7, 1.5])   # a small fp32 weight vector
qmin, qmax = -128, 127                            # signed int8 grid (256 codes)

s = (w.max() - w.min()) / (qmax - qmin)           # scale = range / 255
z = int(np.clip(np.round(qmin - w.min()/s), qmin, qmax))   # integer zero-point

q   = np.clip(np.round(w/s) + z, qmin, qmax).astype(np.int8)   # quantize
```

```
deq = s * (q.astype(np.float64) - z)                          # dequantize

print("scale       =", round(s, 6))          # 0.009804
print("zero_point  =", z)                     # -26
print("int8 codes  =", q)                     # [-128 -67 -26  5  45 127]
print("dequantized =", deq.round(4))          # close to w, snapped to the
grid
print("max abs err =", round(np.max(np.abs(w - deq)), 6))   # 0.003922  (<=
s/2)
print("compression =", 32 // 8, "x  (fp32 -> int8)")        # 4 x
```

Every value returns within half a step (s/2 = 0.0049) of its original, and the tensor is four times smaller. The second cell magnitude-prunes the smallest 30% of a weight vector to zero and measures what that does to a dot product — the basic operation inside every layer. Because the pruned weights were the smallest, the output barely moves:

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)

W = np.random.randn(1000)         # a layer's weights (fp32)
x = np.random.randn(1000)         # an input activation vector
p = 30                             # prune the smallest 30% by magnitude

thresh = np.percentile(np.abs(W), p)   # magnitude cutoff at the 30th
percentile
mask   = np.abs(W) >= thresh            # keep the large-magnitude weights
Wp     = W * mask                       # smallest 30% set exactly to zero

y_full, y_pruned = W @ x, Wp @ x
print("achieved sparsity  =", round(np.mean(Wp == 0), 3))          # 0.3
print("dense  dot product =", round(float(y_full), 4))            # -30.7258
print("pruned dot product =", round(float(y_pruned), 4))          # -31.4206
print("relative change    =", round(abs(y_pruned - y_full) / abs(y_full),
4))  # 0.0226
```

Thirty percent of the connections are gone, yet the output changed by about 2% — the observed fact that makes pruning safe. In practice you would fine-tune briefly to remove even that. Frameworks provide both tools as one-liners. The PyTorch listing below dynamically quantizes

a small model's linear layers to INT8 and magnitude-prunes them with `torch.nn.utils.prune` — the production versions of the two cells above:

PyTorch copy → Colab

```
import torch, torch.nn as nn
import torch.nn.utils.prune as prune

model = nn.Sequential(nn.Linear(784, 256), nn.ReLU(), nn.Linear(256, 10))

# (1) unstructured magnitude pruning: zero the smallest 40% of each layer's
weights
for m in model:
    if isinstance(m, nn.Linear):
        prune.l1_unstructured(m, name="weight", amount=0.4)   # add a 0/1
mask
        prune.remove(m, "weight")                             # bake the
zeros in
# structured alternative — drop whole output rows (neurons):
# prune.ln_structured(m, name="weight", amount=0.25, n=2, dim=0)

# (2) post-training dynamic quantization: fp32 -> int8 weights for the
Linear layers
qmodel = torch.quantization.quantize_dynamic(
    model, {nn.Linear}, dtype=torch.qint8)   # activations quantized on the
fly

x = torch.randn(1, 784)
print(qmodel(x).shape)          # torch.Size([1, 10]) -- same API, int8
matmuls, ~4x smaller
```

Worked example — quantize a weight vector, then count a separable convolution

Part 1 — INT8 quantization. Take the fp32 weight vector w = [−1.0, −0.4, 0.0, 0.3, 0.7, 1.5], so x_{min} = −1.0 and x_{max} = 1.5, targeting signed int8 with [q_{min}, q_{max}] = [−128, 127].

1. Scale: s = (1.5 − (−1.0)) / (127 − (−128)) = 2.5 / 255 = **0.009804**.

2. Zero-point: $z = q_{min} - \text{round}(x_{min}/s) = -128 - \text{round}(-1.0/0.009804) = -128 - (-102) = \mathbf{-26}$.
3. Quantize 0.7: $x_q = \text{round}(0.7/0.009804) + (-26) = \text{round}(71.4) - 26 = 71 - 26 = \mathbf{45}$. Dequantize: $\hat{x} = 0.009804 \cdot (45 - (-26)) = 0.009804 \cdot 71 = \mathbf{0.6961}$. Round-trip error 0.0039.
4. Quantize −0.4: $x_q = \text{round}(-0.4/0.009804) - 26 = \text{round}(-40.8) - 26 = -41 - 26 = \mathbf{-67}$. Dequantize: $0.009804 \cdot (-67 + 26) = 0.009804 \cdot (-41) = \mathbf{-0.4020}$. Error 0.0020.
5. The largest round-trip error over the vector is **0.0039**, safely under the bound $s/2 = 0.0049$. Storage drops from $6 \cdot 4 = 24$ bytes to $6 \cdot 1 = 6$ bytes — a **4× memory saving**.

Part 2 — depthwise-separable convolution, with $C_{in} = C_{out} = 128$ and a 3×3 kernel. A standard convolution costs $k^2 \cdot C_{in} \cdot C_{out} = 9 \cdot 128 \cdot 128 = \mathbf{147{,}456}$ MACs per output pixel. The separable version costs a depthwise $9 \cdot 128 = \mathbf{1{,}152}$ plus a pointwise $128 \cdot 128 = \mathbf{16{,}384}$, totaling **17,536**. The reduction is 147,456 / 17,536 = **8.41×**, matching the formula $1/C_{out} + 1/k^2 = 1/128 + 1/9 = 0.1189$, whose inverse is 8.41. So swapping one convolution for its separable form does the same channel transformation for roughly one-eighth of the arithmetic. This is the trade-off MobileNet makes throughout. Remember that its low arithmetic intensity means the measured latency gain is usually smaller than 8×.

Assumptions & limitations

Every technique here trades a little accuracy for a lot of efficiency, and the trade-off is not free. Aggressive quantization or pruning lowers accuracy unevenly — often worst on rare classes and edge cases, exactly where a metric like top-1 accuracy hides it. So a compressed model needs re-evaluation on the slices that matter, not just the aggregate. FLOP and parameter reductions do not translate one-to-one into latency: a technique that

halves FLOPs but lowers arithmetic intensity (depthwise convs, unstructured sparsity) can leave wall-clock time almost unchanged on hardware that was memory-bound to begin with. Speed-ups are hardware- and runtime-specific: INT8 needs integer tensor cores, sparsity needs sparse kernels, and a method that works well on a server GPU may do nothing on a phone. And every method adds a calibration or fine-tuning step to the pipeline, so the engineering cost is real — which is why you compress only after Chapter 25's recipe has given you a model worth deploying.

Why & when this matters

Turn to this chapter the moment a working model meets a real constraint: a latency SLA, a memory ceiling, a battery, or a cloud bill. The usual order is to start with the cheapest, safest improvements — INT8 post-training quantization and operator fusion through a compiled runtime, which together often give 2–4× with negligible accuracy loss and no retraining — then move on to pruning, distillation, or a redesigned efficient architecture if the budget is still not met. On-device and high-volume serving are where the benefit is largest: saving a millisecond or a megabyte, multiplied over billions of inferences, is the difference between a feature that ships and one that cannot.

In the real world

Efficiency is what puts deep learning in your pocket. Phones run quantized MobileNet and its descendants on dedicated neural accelerators for face unlock, live photo enhancement, and offline speech, all within a couple of watts. Voice assistants keep a tiny always-on keyword-spotting network on a microcontroller so the large model wakes only on "Hey —". Autonomous-driving stacks compile their perception networks with TensorRT to hit hard per-frame deadlines. On the server side, the entire economics of large language models (Chapter 17) depends on inference efficiency — INT8/INT4

weights, paged and quantized KV caches, speculative decoding, and continuous batching — because at billions of queries a day, a 2× efficiency gain is a 2× cut in a very large electricity and hardware bill. The research frontier that produced these models is measured in accuracy. The industry that deploys them is measured in the quantities of this chapter.

Exercises

1. A weight tensor has range $[x_{min}, x_{max}] = [-2.0, 6.0]$ and is quantized to *unsigned* int8 with $[q_{min}, q_{max}] = [0, 255]$. Compute the scale and zero-point, then quantize and dequantize the value $x = 1.0$. MATH

2. Compute the depthwise-separable FLOP reduction for a 3×3 convolution with $C_{in} = C_{out} = 512$, then for a 5×5 kernel with the same channels. What does the kernel size tell you about where separable convolutions help most? CODE

3. Explain, using the roofline model, how a model with *fewer* FLOPs can be *slower* than one with more. CONCEPT

4. Contrast post-training quantization (PTQ) with quantization-aware training (QAT): what does each cost, and when would you choose QAT? CONCEPT

5. Contrast unstructured and structured pruning at the same sparsity. Why does unstructured usually preserve more accuracy, yet structured more reliably reduce latency? CONCEPT

6. A 1024×1024 linear layer is replaced by a low-rank factorization $W \approx U V$ of rank $r = 128$. How many parameters does this use, and what is the compression ratio? What accuracy risk does it introduce? MATH

7. Estimate the INT8 model size and the fp16 KV-cache memory for a served language model with 7 billion parameters, $L = 32$ layers, hidden size $d = 4096$, at a context of $T = 4096$

tokens (single sequence). MATH

8. Push the body's quantization to INT4 (16 levels) on the same vector and compare the max round-trip error with INT8. Why is quantization-aware training more important at 4 bits? CODE

CHAPTER 28 · IN PRACTICE

Interpretability & explainability

> *A trained network is a decision without a reason attached. It labels the X-ray "pneumonia", denies the loan, flags the transaction. By default, it offers nothing but the number.* ***Interpretability*** *is the field of finding a reason. It turns the millions of weights that produced an answer into something a human can inspect, trust, challenge, or debug. This chapter reviews how it is done: gradients that ask "which input mattered?", perturbations that ask "what if this were absent?", and the game theory that makes one of those answers provably fair. It is also honest about a harder truth the field keeps rediscovering: an explanation that looks plausible is not automatically a faithful one.*

Why open the black box

Four separate pressures push us to explain a model, and confusing them is the first mistake. **Trust**: a radiologist will not act on a network's diagnosis unless she can see it looked at the lung, not at the scanner's brand text in the corner. **Debugging**: when accuracy is high on the test set but collapses in deployment, an explanation is often the fastest way to find that the model learned a shortcut rather than the signal. **Fairness and regulation**: the EU's GDPR grants a limited "right to an explanation" for automated decisions, credit-scoring law requires adverse-action reasons, and you cannot audit for discrimination you cannot see. **Scientific insight**: a network that predicts protein structure or plays Go at superhuman level has discovered *something*, and reading it back out is itself a discovery.

These goals want different things. Trust and regulation want a *local* reason for one decision ("why was *my* loan denied?"). Debugging and science often want a *global* account of the model's behavior ("what has it learned to detect in general?"). Keep the goal in view. An explanation that is excellent for one is often useless for another.

A map of the field

Two axes organize almost every method. The first is **intrinsic vs post-hoc**. An *intrinsically* interpretable model is transparent by construction, such as a short decision tree, a sparse linear model, or a small set of rules. Here the mechanism *is* the explanation. A *post-hoc* method leaves a fixed, already-trained black box untouched and attaches an explanation after the fact. Deep networks are the black-box case by definition, so almost everything in this chapter is post-hoc: we probe a frozen network from the outside.

The second axis is **global vs local**. A *global* explanation describes the whole function the model computes ("this feature raises risk on average across all customers"). A *local* explanation accounts for a single prediction ("*for this customer,* the recent missed payment was decisive"). Local is usually easier and more honest, because a highly nonlinear model may have no faithful global summary at all, yet still be nearly linear in the small neighborhood around one input. Most of the main tools (saliency, LIME, SHAP) are local and post-hoc. A third practical axis is **model-agnostic vs model-specific**. LIME and Shapley treat the model as a black-box function $f : x \mapsto$ score and only need to query it. Gradient methods and Grad-CAM, in contrast, reach inside and use backpropagation (Chapter 8), so they are specific to differentiable networks.

Gradient and saliency methods

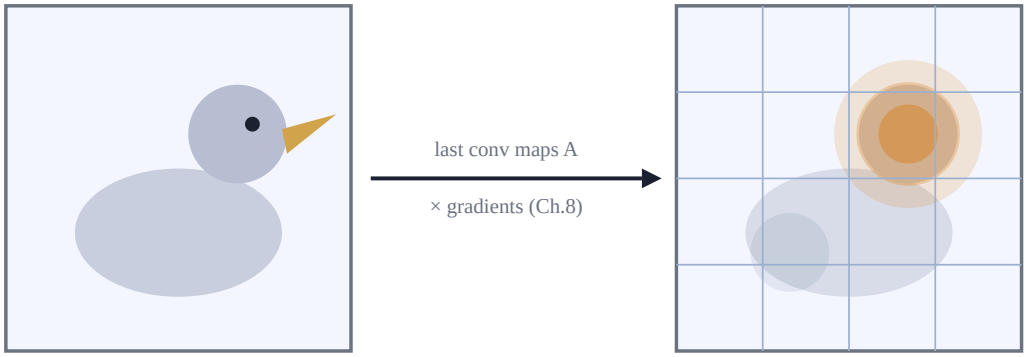
The cheapest question you can ask a differentiable model is: *which inputs, if changed slightly, would change the answer most?* That is exactly a gradient. For a classifier that outputs a score $f_c(x)$ for class c, the **vanilla saliency map** is the magnitude of the gradient of that score with respect to each input feature:

$$S(x) = \left| \nabla_x f_c(x) \right|, \quad S_i(x) = \left| \partial f_c \,/\, \partial x_i \right|$$

Nothing new is needed to compute it. The backward pass of Chapter 8 already runs the chain rule from the output back through the network. Normally it stops at the weights. But if you let it continue one step further, to the input pixels instead of the parameters, you get $\partial f_c / \partial x$ for free (Simonyan et al. 2013). Bright pixels are the ones the score is most sensitive to.

Vanilla gradients have two well-known flaws. They are **visually noisy**: a ReLU network is piecewise linear, so its gradient is a high-frequency, scattered function that changes sharply from pixel to pixel, even where the underlying evidence is smooth. **SmoothGrad** (Smilkov et al. 2017) cleans this up by averaging the saliency over many copies of the input perturbed with small Gaussian noise, $\mathbb{E}_{\varepsilon\sim N(0,\sigma^2)}[\nabla_x f_c(x+\varepsilon)]$, which cancels the jitter and keeps the signal. The second flaw is **saturation**. Suppose the score has already flattened at x, so the feature did its job and the function is now locally flat. Then $\partial f/\partial x \approx 0$ and the gradient reports "unimportant" about a feature that was in fact decisive. This is the problem that Integrated Gradients fixes below.

Grad-CAM = ReLU($\Sigma_k \alpha_k A^k$) , α_k = mean gradient of the score over map k



Grad-CAM turns the gradient of a class score into a picture of "where the network looked". Each feature map of the last convolutional layer (Chapter 13) is scaled by α_k, the average of the class score's gradient over that map. The maps are then summed and passed through ReLU. The result is coarse (it has the low resolution of the deep feature grid) but class-discriminative: the hot amber region marks the pixels most responsible for the "bird" decision.

For convolutional networks (Chapter 13) there is a much cleaner method than pixel gradients. **Grad-CAM** (Selvaraju et al. 2017) uses the fact that the last convolutional layer still keeps spatial structure: each of its feature maps A^k is a small image that lights up where some learned pattern occurs. To find which maps matter for class c, take the class score y^c (the logit that feeds the loss of Chapter 6) and average its gradient over each map to get an importance weight, then form a weighted, rectified sum of the maps:

$$\alpha_k^{\,c} = \frac{1}{Z}\,\Sigma_i\,\Sigma_j\,\frac{\partial y^c}{\partial A^k_{ij}}\,, \quad L_{\text{Grad-CAM}}^{\,c} = \text{ReLU}\Big(\,\Sigma_k\,\alpha_k^{\,c}\,A^k\,\Big)$$

The gradient tells you *how much each map helped the class*. The maps tell you *where each pattern fired*. Multiplying and summing gives a heatmap of where the class evidence is, and the ReLU keeps only regions that *raise* the score. Grad-CAM is computed at the coarse resolution of the deep feature grid (often 7×7 or 14×14) and then upsampled. It is spatially blurry but reliably class-discriminative. It is the standard check for "is the CNN looking at the object or at the background?".

Integrated Gradients (Sundararajan et al. 2017) fixes the saturation problem by not trusting the gradient at just one point. Choose a **baseline** x' that represents "absence of signal", such as a black image, a zero embedding, or the dataset mean. Then integrate the gradient along the straight path from the baseline to the actual input:

$$\mathrm{IG}_i(x) = (x_i - x'_i) \int_{\alpha=0}^{1} \frac{\partial f(\, x' + \alpha(x - x')\,)}{\partial x_i}\, d\alpha$$

Averaging the gradient over the whole path, rather than reading it only at the end point, means a feature that saturated still gets credit for the rise it caused along the way. Integrated Gradients obeys the **completeness axiom**: the attributions sum exactly to the change in output between baseline and input,

$$\Sigma_i\, \mathrm{IG}_i(x) = f(x) - f(x')$$

which in one dimension is just the fundamental theorem of calculus. Completeness makes the numbers a genuine *accounting*: every unit of output above the baseline is given to some feature, with none invented and none lost. The price is the baseline itself. A different x' gives different attributions, so the choice is a modeling decision, not a small detail.

$\mathbf{f_c,\ y^c}$	the model's score/logit for class c (before the softmax and loss of Chapter 6)
$\mathbf{A^k_{ij}}$	activation at spatial position (i,j) of the k-th feature map in the last conv layer (Chapter 13)
$\mathbf{\alpha_k{}^c}$	Grad-CAM weight for map k: how much that map, on average, raises the class score
$\mathbf{x'}$	baseline / reference input encoding "no information" (black image, zero embedding, feature mean)
$\mathbf{Z}$	number of spatial locations in a feature map (the normalizer for the average)

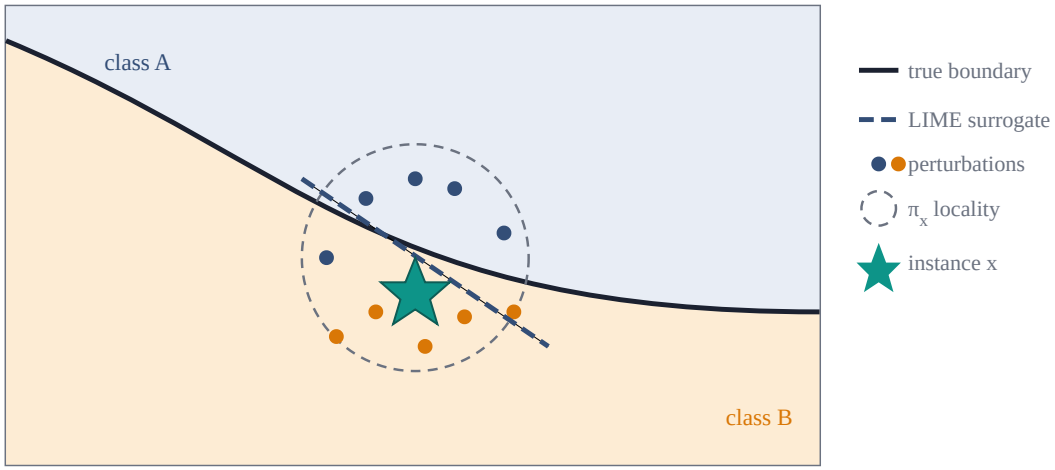
Perturbation and game-theoretic attribution

Gradient methods need to reach inside the network. The other major family treats the model as a sealed box and learns from *poking* it: change the input, watch the output move. The most basic version is **occlusion** (Zeiler & Fergus 2014): slide a gray patch across an image, and wherever hiding a region makes the correct-class probability drop sharply, that region was important. Occlusion asks a different question from the gradient. It asks not "what is the local slope?" but "what happens if this feature is removed entirely?". The two can disagree sharply, as the code section shows.

LIME, short for Local Interpretable Model-agnostic Explanations (Ribeiro et al. 2016), turns poking into a systematic local model. To explain one prediction f(x), LIME samples many perturbed points around x (switching superpixels on and off, or masking words), asks the black box to label each, weights those samples by how close they are to x through a proximity kernel π_x, and then fits a *simple, sparse* surrogate g (usually a linear model) to that weighted local dataset:

$$\xi(x) = \operatorname{argmin}_{g \in G} \mathcal{L}(f, g, \pi_x) + \Omega(g)$$

Here $\mathcal{L}$ measures how badly the surrogate g reproduces the black box f in the neighborhood defined by π_x, and $\Omega(g)$ penalizes complexity (e.g. keeping only a few nonzero coefficients). The surrogate's coefficients *are* the explanation: "for this instance, these five features, with these signs and weights, locally reproduce the model." LIME is model-agnostic and intuitive, but it is only as stable as its sampling. Rerun it with a new random seed or a different kernel width, and the explanation can shift. This is its most-cited weakness.



How LIME explains one prediction. The black box's true decision boundary (solid) is highly curved, so there is no faithful global linear summary. But near the instance (teal star) the boundary is nearly straight: LIME samples points around x, labels them with the black box, weights them by proximity (dashed circle), and fits a sparse straight-line surrogate (dashed blue). Its slope and intercept, the coefficients, are the local explanation. Trust it only inside the circle.

LIME's flexibility is also its weakness: kernel width, sample count, and surrogate family are all free choices that change the answer. **SHAP** (SHapley Additive exPlanations; Lundberg & Lee 2017) removes the arbitrariness by borrowing a result from cooperative game theory. Think of a prediction as a payout $f(x) - f(x')$ that the features must divide fairly, where the "value" of a coalition S is the model's output when only those features take their real values and the rest stay at the baseline: $v(S) = f(x_S, x'_{\bar{S}})$. The **Shapley value** of feature i is its average *marginal contribution* (how much it adds to a coalition) over every order in which the features could be added:

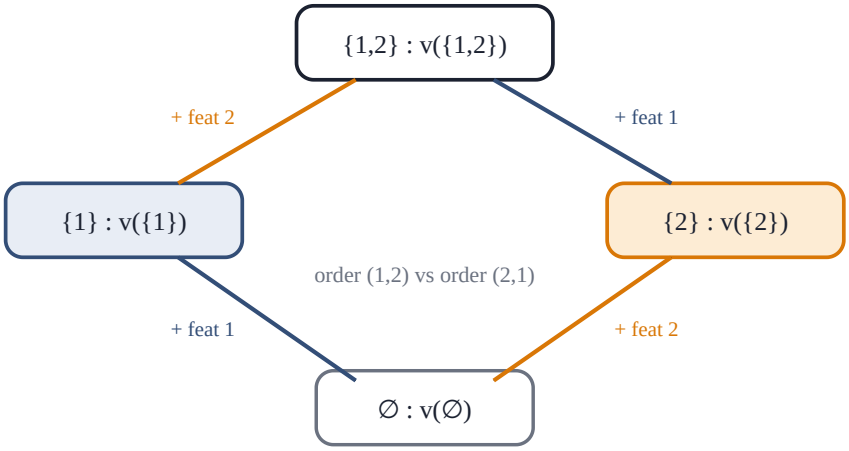
$$\varphi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\,(n - |S| - 1)!}{n!} \left[v(S \cup \{i\}) - v(S) \right]$$

The combinatorial weight is exactly the fraction of orderings in which the features of S come before i and the rest come after, so φ_i is a simple average of "what did i add when it joined?" across all n! orderings. Lloyd Shapley's 1953 theorem is why this specific formula matters: it is the **unique** attribution that satisfies four axioms at the same time.

1. **Efficiency** (completeness): the attributions sum to the whole gap, $\sum_i \varphi_i = f(x) - f(x')$. Nothing is invented or lost. This is the same guarantee Integrated Gradients gives.

2. **Symmetry**: two features that add the same amount to every coalition receive equal attribution. Identical roles, identical credit.
3. **Dummy** (null player): a feature that changes no coalition's value, $v(S \cup \{i\}) = v(S)$ for all S, gets exactly $\varphi_i = 0$. Irrelevant features score zero.
4. **Additivity** (linearity): the attribution for a sum of two games is the sum of the attributions, $\varphi(v + w) = \varphi(v) + \varphi(w)$. This is what lets SHAP explain an ensemble as the sum of its members' explanations.

No other rule satisfies all four. That uniqueness is SHAP's main advantage: where LIME offers *an* explanation, Shapley offers *the* explanation, given the axioms you accept. The drawback is cost. The sum runs over all 2^n subsets, which is exponential in the feature count. So practical SHAP is a family of approximations (KernelSHAP samples coalitions and solves a weighted regression; TreeSHAP computes exact values in polynomial time for tree ensembles). And the axioms guarantee fairness, not truth: a poor baseline still produces "fair" nonsense.



The Shapley value as an average over coalitions. Reading the lattice bottom-up, each edge adds one feature and its label is that feature's marginal contribution, $v(S \cup \{i\}) - v(S)$. There are two paths from ∅ to {1,2}, one per ordering. Feature 1's Shapley value averages its two blue edges (the bottom-left and top-right gains). Feature 2 averages its two amber edges. When features interact, an edge's gain depends on who is already present, so the two paths differ and the averaging really matters. For a purely additive model, the paths are the same.

Attention is not explanation

Transformers (Chapter 16) expose attention weights, which are coefficients saying how much each token attended to each other token. It is tempting to read them directly as an explanation: "the model focused on *these* words." Jain & Wallace (2019), in their critical paper "Attention is not Explanation," showed this is unsafe. Across many NLP models they found that attention weights correlate only weakly with gradient- and erasure-based importance. And, worse still, one can often find an *entirely different* attention distribution that leaves the model's prediction unchanged. If two different "explanations" produce the same output, neither can be the reason for it. The reply, Wiegreffe & Pinter (2019), "Attention is not not Explanation," argues the picture is more complex. Attention can be a plausible and useful explanation under stricter definitions and tests. But the lesson stands: attention is a component of the computation, not a guaranteed account of it. Treat attention maps as a hypothesis to be checked by a faithfulness test, never as ground truth about why a model decided what it did.

Concepts, probes, and circuits

Attributions answer "which inputs?". A different family asks "which *human concepts* does the network use, and where do they live inside it?" **TCAV**, short for Testing with Concept Activation Vectors (Kim et al. 2018), makes "concept" concrete: collect examples of a concept (say, "stripes") and counter-examples, fit a linear classifier separating their activations in some hidden layer, and take its normal as the **concept activation vector**, a direction meaning "more stripe-like." The concept's influence on class c is then the fraction of inputs for which moving along that direction *raises* the class score. This is a directional derivative of the logit. TCAV gives global statements in human terms ("the zebra prediction is sensitive to stripes"), not pixels.

Probing classifiers (Alain & Bengio 2016; Hewitt & Manning 2019) examine representations (Chapter 5): freeze the network, treat its activations as fixed features, and train a *simple* classifier to predict some property, such as part of speech, syntactic depth, or sentiment. High

probe accuracy means the property is *linearly decodable* there, which is evidence the network encoded it. The key limitation: a probe measures what information is *present*, not what the model actually *uses*. It shows availability, not causal reliance.

Mechanistic interpretability is the most ambitious program: reverse-engineer a network into human-readable algorithms, the way one disassembles a compiled binary. Its vocabulary is worth stating precisely. A **feature** is a direction in activation space that encodes an interpretable property. A **circuit** is a small subgraph of features and weights that implements a recognizable computation. Examples are the curve-detector and object-part circuits mapped in vision models (Olah et al. 2020), or induction heads that copy repeated patterns in transformers. The main obstacle is **superposition** (Elhage et al. 2022). A layer can represent far more useful features than it has neurons, so it packs them into *overlapping, non-orthogonal* directions. This makes a single neuron typically *polysemantic*: it fires for several unrelated concepts at once. Superposition is why "read off what each neuron does" fails, and why recent work trains **sparse autoencoders** to separate the crowded activations into many sparse, nearly monosemantic features that can be inspected one at a time. The program is early and labor-intensive, but it targets the thing the other methods only approximate: the mechanism itself, not a summary of input–output sensitivity.

Evaluating an explanation

Every method above *produces* a picture. Few, on their own, tell you whether it is **faithful**: whether it reflects the model's real computation or merely looks reasonable to a human. Faithfulness and plausibility are different properties, and optimizing for the second while assuming the first is the field's recurring error. Two kinds of test guard against this.

The first is **ablation-based faithfulness**: if an attribution is honest, deleting its top-ranked features should hurt the prediction quickly, and deleting its lowest-ranked ones should barely matter. These are the *deletion* and *insertion* curves. ROAR (Hooker et al. 2019) improves this by *retraining* the model after masking the top-attributed pixels. This defends against the trap that masking alone just pushes inputs off the data distribution. Worryingly, several popular saliency methods barely beat a random ranking under it.

The second is the **sanity check** of Adebayo et al. (2018), which every practitioner should know well. An explanation of a model must *depend on the model*. So, step by step, **randomize the network's weights** from the top layer down and recompute the map. If the model now computes garbage, its explanation should turn to garbage too. Several widely used methods, notably Guided Backpropagation, *fail*. Their maps stay almost visually unchanged after randomization. This reveals them as input-dependent edge detectors, largely independent of the function the network learned. A method that draws the same pretty heatmap for a trained model and a random one is explaining the image, not the model. The related *data-randomization* test permutes training labels for the same purpose. The lesson is a discipline, not a tool: never trust an attribution method you have not watched pass a randomization check on your own setup.

Attribution in code

The Shapley value is defined by an average over orderings, and with only NumPy we can compute it *exactly* for a small model by enumerating every ordering, with no sampling and no approximation. This cell builds a three-feature model with an interaction term, computes each feature's Shapley value, and verifies both the efficiency axiom (the values sum to f(x) − f(baseline)) and the additivity axiom (Shapley of a sum of games equals the sum of Shapley values):

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)                                # determinism
n = 3
w = np.random.randn(n)                           # linear weights
A = np.random.randn(n, n); A = (A + A.T) / 2     # symmetric interaction
matrix
x = np.random.randn(n)                           # the input we want to
explain
b = np.zeros(n)                                  # baseline / reference input

def perms(seq):                                  # all orderings, using no
imports
    if len(seq) <= 1: return [list(seq)]
    out = []
    for i in range(len(seq)):
```

```
        for p in perms(seq[:i] + seq[i+1:]):
            out.append([seq[i]] + p)
    return out

def shapley(f):                                  # exact Shapley: average over
orderings
    orders = perms(list(range(n)))
    phi = np.zeros(n)
    for order in orders:
        z = b.copy(); prev = f(z)                # start from the baseline
coalition
        for i in order:
            z = z.copy(); z[i] = x[i]            # feature i joins the
coalition
            cur = f(z); phi[i] += cur - prev     # its marginal contribution
            prev = cur
    return phi / len(orders)

lin   = lambda z: w @ z                          # game 1: the linear part
inter = lambda z: z @ A @ z                      # game 2: the interaction
part
full  = lambda z: lin(z) + inter(z)              # the whole model

phi = shapley(full)
print("Shapley values     =", np.round(phi, 4))
print("sum of Shapley      =", round(float(phi.sum()), 4))
print("f(x) - f(baseline)  =", round(float(full(x) - full(b)), 4))  #
efficiency: equal
print("additivity holds    =", np.allclose(shapley(lin) + shapley(inter),
phi))
```

The sum of the Shapley values matches f(x) − f(baseline) to machine precision. This is efficiency. Splitting the model into its linear and interaction games, explaining each, and adding the results reproduces the full explanation exactly. This is additivity. Next, here are the two perturbation ideas side by side on a tiny differentiable model. They show how a *gradient* (local slope) and an *occlusion* (remove-the-feature effect) attribute importance differently:

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)                                # determinism
def f(x):                                        # a small differentiable
```

```
toy model
    return 2*x[0] + 0*x[1] - x[2] + x[3]**2        # two linear feats, one
squared feat
x    = np.array([1.0, 5.0, 2.0, 3.0])              # the input we want to
explain
base = np.zeros(4)                                 # reference used for
occlusion

eps = 1e-5                                          # (1) vanilla-gradient
saliency
grad = np.zeros(4)                                 #     by central finite
differences
for i in range(4):
    xp = x.copy(); xp[i] += eps
    xm = x.copy(); xm[i] -= eps
    grad[i] = (f(xp) - f(xm)) / (2*eps)            # d f / d x_i  at the
input
print("f(x)              =", f(x))                 # 9.0
print("gradient saliency =", np.round(grad, 4))    # [ 2  0 -1  6 ]  local
slope
print("|gradient|        =", np.round(np.abs(grad), 4))

occ = np.zeros(4)                                  # (2) occlusion-
importance map
for i in range(4):
    xo = x.copy(); xo[i] = base[i]                 # reset feature i to the
baseline
    occ[i] = f(x) - f(xo)                          # how much the output
drops
print("occlusion map     =", np.round(occ, 4))     # [ 2  0 -2  9 ]  remove-
feature
```

Compare the two vectors. Feature 1 (a wide-open door onto x_1) is not part of the model at all, so both methods give it 0. This is the dummy property, shown in practice. The linear features agree on *sign* but not *magnitude*: the gradient reports the coefficient ($\partial f/\partial x_2 = -1$), while occlusion reports the whole contribution removed ($w_2 x_2 = -2$). The gap is widest on the squared feature: its gradient is the local slope $2x_3 = 6$, but occluding it erases its entire $x_3^2 = 9$ of output. That mismatch (local sensitivity versus total contribution) is the same distinction that separates vanilla gradients from Integrated Gradients and Shapley values, now visible in five lines.

For real convolutional networks, Grad-CAM is the standard tool, and modern autograd makes it a few lines with **hooks** that capture a layer's activations on the forward pass and its gradients on the backward pass. This PyTorch listing explains the class the network predicted, at the last convolutional layer:

PyTorch copy → Colab

```
import torch, torch.nn as nn, torch.nn.functional as F

# a small CNN; in practice this is any trained classifier (e.g. a
torchvision ResNet)
model = nn.Sequential(
    nn.Conv2d(3, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
    nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(),        # <-- last conv:
explain THIS
    nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Linear(32, 10))
model.eval()
target = model[3]                       # the second Conv2d, our feature
extractor

feats, grads = {}, {}                    # hooks stash the tensors we need
target.register_forward_hook(       lambda m, i, o: feats.__setitem__("A",
o.detach()))
target.register_full_backward_hook( lambda m, gi, go:
grads.__setitem__("dA", go[0].detach()))

x = torch.randn(1, 3, 32, 32)           # one input image, shape (1, 3, 32,
32)
logits = model(x)                        # forward pass fills feats["A"] ->
(1,32,16,16)
cls = logits.argmax(1)                   # the predicted class
model.zero_grad()
logits[0, cls].backward()                # backprop the class score -> fills
grads["dA"]

A     = feats["A"]                       # (1, 32, 16, 16) activations of
last conv (Ch.13)
dA    = grads["dA"]                      # (1, 32, 16, 16) gradients
d(score)/dA (Ch.8)
alpha = dA.mean(dim=(2, 3), keepdim=True)          # (1,32,1,1) importance
weight per map
cam   = F.relu((alpha * A).sum(1, keepdim=True))   # (1,1,16,16) weighted
sum, keep +ve
```

```
cam   = F.interpolate(cam, size=x.shape[-2:], mode="bilinear",
align_corners=False)
cam   = (cam - cam.amin()) / (cam.amax() - cam.amin() + 1e-8)   # normalise
to [0,1]
print(cam.shape)                          # torch.Size([1, 1, 32, 32]) — a
heatmap over x
```

Worked example — Shapley for an additive model, and Integrated Gradients in 1-D

Part A — Shapley of a two-feature additive model. Let $f(x) = w_1x_1 + w_2x_2$ with $w_1 = 2$, $w_2 = 3$, input $x = (1, 2)$, and baseline $x' = (0, 0)$. The coalition value $v(S)$ sets the features in S to x and the rest to the baseline:

1. $v(\varnothing) = 0$; $v(\{1\}) = 2\cdot1 = 2$; $v(\{2\}) = 3\cdot2 = 6$; $v(\{1,2\}) = 2 + 6 = 8$.
2. Feature 1's marginal contribution in each ordering. Order (1,2): it joins first, $\varnothing \to \{1\}$, gain $v(\{1\}) - v(\varnothing) = 2 - 0 = \mathbf{2}$. Order (2,1): it joins second, $\{2\} \to \{1,2\}$, gain $v(\{1,2\}) - v(\{2\}) = 8 - 6 = \mathbf{2}$. Average $\varphi_1 = (2 + 2)/2 = \mathbf{2}$.
3. Feature 2's marginal contribution. Order (1,2): $\{1\} \to \{1,2\}$, gain $8 - 2 = \mathbf{6}$. Order (2,1): $\varnothing \to \{2\}$, gain $6 - 0 = \mathbf{6}$. Average $\varphi_2 = \mathbf{6}$.
4. **Efficiency check:** $\varphi_1 + \varphi_2 = 2 + 6 = \mathbf{8} = f(x) - f(x') = 8 - 0$. The attributions divide the prediction exactly.

Both orderings gave feature 1 the same gain (2 and 2) and feature 2 the same gain (6 and 6). This is not a coincidence: an additive model has *no interaction*, so a feature's contribution never depends on who joined first, and the Shapley value simplifies to the closed form $\varphi_i = w_i(x_i - x'_i)$, here $2\cdot1$ and $3\cdot2$. Averaging over orderings matters only when features interact (Exercise 1), where it splits the shared interaction fairly by symmetry.

Part B — Integrated Gradients in one dimension. Take $f(u) = u^2$, baseline $x' = 0$, input $x = 2$. Integrated Gradients averages the derivative $f'(u) = 2u$ along the path and multiplies by the change in input:

1. IG = $(x - x') \int_0^1 f'(x' + \alpha(x - x'))\, d\alpha = 2 \int_0^1 f'(2\alpha)\, d\alpha = 2 \int_0^1 4\alpha\, d\alpha$.
2. $= 2 \cdot [2\alpha^2]_0^1 = 2 \cdot 2 =$ **4**.
3. **Completeness check:** $f(x) - f(x') = 2^2 - 0^2 =$ **4** = IG. The single feature is credited with the entire output change.
4. Now compare the naive "gradient × input" saliency: $f'(2) \cdot (2 - 0) = 4 \cdot 2 =$ **8**, twice the true value. The endpoint gradient overcounts because the slope at x = 2 (namely 4) is steeper than the average slope 2 along the way. Integrating fixes exactly this.

What the numbers mean: efficiency and completeness are the same guarantee under two names: attributions that *add up* to the prediction. Both a path integral of gradients and an average of coalition contributions achieve it, which is why Integrated Gradients and SHAP, though computed in completely different ways, are the two most trusted attribution methods.

Assumptions & limitations

Every method here rests on a key assumption. Attribution methods need a **baseline** (a black image, a zero embedding, or the feature mean), and the numbers can flip with the choice. There is no universally "neutral" input. LIME assumes local linearity and stable sampling, neither guaranteed. Exact Shapley is exponential in the feature count, so deployed SHAP is an approximation of varying quality. Most seriously, an explanation can be **plausible without being faithful**. It can look like a clear reason while having little relation to the model's actual computation. This is exactly what the Adebayo et al. (2018) checks expose. And an explanation is not a justification: showing which pixels a model used does not make the decision correct or fair, only readable. Interpretability tells you *what* a model did, rarely *whether it should have*.

Why & when this matters

Use local attribution (saliency, Grad-CAM, LIME, SHAP) to justify or audit a *single* decision, such as a denied claim or a flagged scan. Use global tools (TCAV, probing, circuit analysis) to understand the model *as a whole*, for debugging or science. Whichever you pick, run a faithfulness or randomization check before believing it. And remember which question you asked: gradients report local sensitivity, while occlusion and Shapley report the effect of removing a feature. The two can disagree by a factor of two, or even in sign. The right explanation is the one whose question matches yours, and that you have tested.

In the real world

Interpretability is now infrastructure, not decoration. SHAP is an industry default. TreeSHAP explains the gradient-boosted models that dominate tabular credit-scoring, fraud, and healthcare risk, where regulators demand per-decision reasons. Grad-CAM is built into medical-imaging tools so clinicians can see whether a model looks at the lesion or an artifact. It also exposed the pneumonia-classifier "hospital token" shortcut that broke transfer. The GDPR and the EU AI Act have turned explanation into a compliance requirement for high-risk systems. And at the frontier, mechanistic interpretability has moved from curiosity to safety practice. Anthropic and others use sparse autoencoders to pull millions of interpretable features out of production language models. They search for the circuits behind deception and unsafe behavior, trying to understand these systems before trusting them with more.

Exercises

1. Redo the Shapley worked example for a model *with interaction*: $f(x) = 2x_1 + 3x_2 + 4x_1x_2$, input $x = (1, 1)$, baseline $(0, 0)$. Compute each feature's marginal contribution in both

orderings, the Shapley values, and verify efficiency. Why do the two orderings now disagree? MATH

2. Integrated Gradients on a *saturating* function. Let $f(u) = \tanh(u)$, baseline $x' = 0$, input $x = 2$. Show completeness holds, and compare IG with the endpoint "gradient × input" attribution. What does the comparison reveal? MATH

3. Modify the occlusion cell to use the feature *mean* as the baseline instead of zero, and observe how the attributions change. Why is occlusion baseline-dependent? CODE

4. Extend the Shapley cell to test the **dummy axiom**: add a fourth feature that the model completely ignores, and confirm its Shapley value is 0. CODE

5. Why are raw input-gradient saliency maps visually noisy, and how does SmoothGrad reduce the noise? What does the need for smoothing suggest about faithfulness? CONCEPT

6. A colleague points at a transformer's attention map and says "the model focused on the word *not*, so that is the explanation for the negative sentiment." Using Jain & Wallace (2019), explain why this is unsafe. CONCEPT

7. Describe the Adebayo et al. (2018) model-parameter randomization sanity check. If a saliency method's heatmap barely changes after you randomize the network's weights, what have you learned about that method? CONCEPT

8. A CNN for 224×224 images has a last convolutional feature map of size 7×7, and Grad-CAM is computed there and upsampled to the input. What is the linear upsampling factor, and why is Grad-CAM strong on *class discrimination* but weak on *fine spatial detail*? Connect your answer to the receptive field of Chapter 13. MATH

CHAPTER 29 · IN PRACTICE

Ethics, fairness & safety

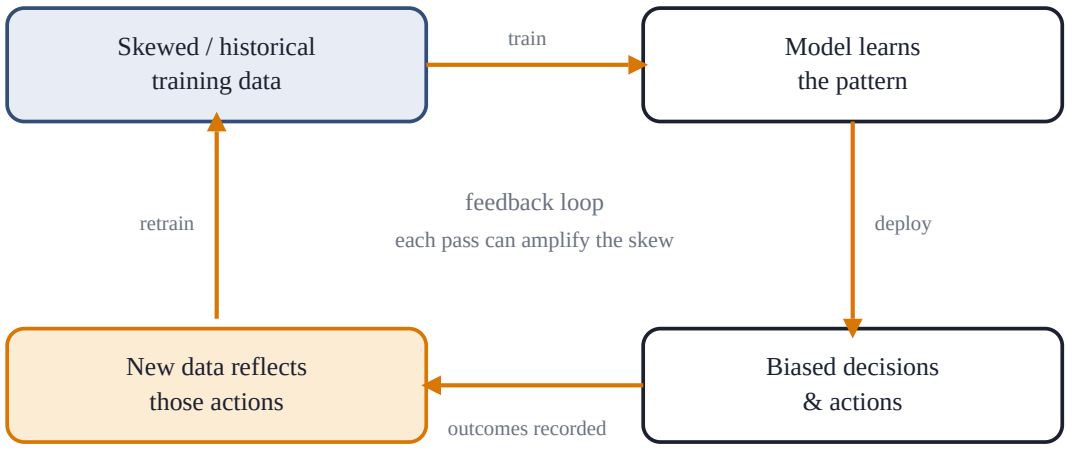
> *The same network that reads a chest X-ray can decide who is offered a loan, who is released before trial, and whose CV is ever seen by a human. Once a model makes important decisions about people, three questions come up that no loss curve can answer. Is it **fair** across the groups it affects? Does it **protect** the people whose data trained it? And is it **safe** against people who would fool or misuse it? This chapter makes those questions technical. It defines fairness clearly enough to measure, and it proves that these natural definitions cannot all hold at the same time. It then turns to privacy, adversarial security, and the alignment of powerful systems. It treats each one as engineering, with quantities, guarantees, and honest limits rather than slogans.*

Where bias comes from

A model is a mirror held up to its training data, and the mirror does not know which reflections are fair. Bias enters long before the first gradient step. It comes through at least four separate channels, and these are worth separating because they need different fixes. **Historical bias** is the hardest to remove. The data accurately records a world that is itself unequal. Take a hiring model trained on decades of decisions in a field that in the past excluded some group. It will learn that exclusion as signal, even if every label is "correct" and the sample is perfectly representative. No amount of cleaner sampling removes it, because the unfairness is in the target, not in how the data was collected.

Representation (sampling) bias is different. Some groups are simply under-sampled, so the model has less evidence about them and makes larger errors there. Face-analysis systems trained on datasets dominated by lighter-skinned faces showed clearly higher error rates on darker-skinned faces (Buolamwini & Gebru 2018) for exactly this reason. **Measurement and labeling bias** enters when the label we can observe is a biased *proxy* for the target we care about. For example, "re-arrest" stands in for "re-offense", and "was treated" stands in for "was sick". The proxy carries the distortions of the process that generated it. Finally, **feedback loops**

in deployment close the circle. A model's own decisions shape the data it is later retrained on. Send more patrols where a model predicts crime, and you record more incidents there. This teaches the next model to send even more patrols. The result is a self-fulfilling prophecy that amplifies the original imbalance (Ensign et al. 2018). Curating and documenting data to catch these effects is the subject of data-centric deep learning (Chapter 26). Here we take the biased model as given and ask how to measure and limit it.



The deployment feedback loop. A model trained on skewed data acts on the world, and its actions decide which new data is collected. So the next model is trained on evidence that the earlier model helped to shape. Unlike fixed historical bias, this loop is dynamic. Even a small starting gap can grow with every retraining cycle, unless the collection process is corrected on purpose.

What do we mean by "fair"?

To limit bias we must first define it. The field offers several formal definitions, and they *disagree with one another*. Write A for a protected attribute (say group 0 vs group 1), Y for the true outcome, and $\hat{Y} \in \{0,1\}$ for the model's decision. The definitions split into two families. **Group fairness** equalizes some statistic across groups. The four most used are:

$$\text{demographic parity:}\quad P(\hat{Y}=1 \mid A=0) = P(\hat{Y}=1 \mid A=1)$$

$$\text{equalized odds:}\quad P(\hat{Y}=1 \mid Y=y, A=0) = P(\hat{Y}=1 \mid Y=y, A=1),\ \ y \in \{0,1\}$$

$$\text{equal opportunity:}\quad P(\hat{Y}=1 \mid Y=1, A=0) = P(\hat{Y}=1 \mid Y=1, A=1)$$

$$\text{predictive parity:}\quad P(Y=1 \mid \hat{Y}=1, A=0) = P(Y=1 \mid \hat{Y}=1, A=1)$$

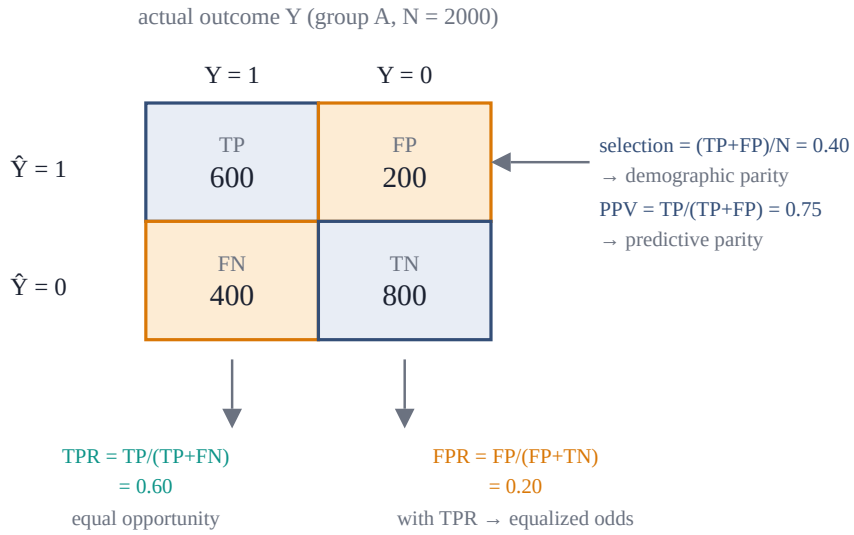
Demographic parity asks that the selection rate be identical across groups: equal shares of loans, equal shares of interviews, whatever the underlying labels are. **Equalized odds** (Hardt et al. 2016) is stricter. It asks for equal true-positive *and* equal false-positive rates. So the model is equally accurate on the qualified people and equally mistaken on the unqualified people in every group. **Equal opportunity** relaxes this to only the true-positive rate: an equal chance of a favorable decision *among those who truly qualify*. **Predictive parity** takes the other side of the confusion matrix. It asks that a positive prediction mean the same thing everywhere, that is, equal precision (positive predictive value). Predictive parity is the single-threshold version of **calibration**. Calibration means that a predicted risk of 0.7 corresponds to a 70% observed rate within every group. This is the property studied for confidence in Chapter 24.

The second family is **individual fairness** (Dwork et al. 2012). It does not trust group averages at all, and instead demands that *similar individuals be treated similarly*. The model M should be Lipschitz with respect to a task-specific similarity metric,

$$d_Y(\, M(x), M(x')\,) \le L \cdot d_X(x, x')$$

A, Y, $\hat{Y}$	protected attribute (group), true outcome, and the model's binary decision. TP/FP/FN/TN are the confusion-matrix counts
TPR, FPR	true- and false-positive rate, $P(\hat{Y}=1 \mid Y=1)$ and $P(\hat{Y}=1 \mid Y=0)$. FNR = 1 − TPR is the false-negative rate
PPV	positive predictive value (precision), $P(Y=1 \mid \hat{Y}=1)$, the quantity predictive parity equalizes
p	base rate / prevalence $P(Y=1)$ within a group, the quantity that differs and forces the trade-offs below
d_X, d_Y, L	distances on inputs and outputs, and a Lipschitz constant, defining individual fairness

Individual fairness sounds appealing, but it moves the whole difficulty onto the metric d_X. Deciding which people are "similar for this purpose" is exactly the value judgment the model was meant to help make. And a biased similarity metric brings back the very discrimination it promised to prevent.



Every group-fairness metric is a ratio read from one per-group confusion matrix. The prediction rows give the selection rate and the precision (demographic and predictive parity). The outcome columns give the true- and false-positive rates (equal opportunity and equalized odds). Because the four ratios draw on overlapping cells, fixing one usually moves the others. This is the tension that the next section makes exact.

The impossibility theorems

The definitions look like a menu from which you pick the criterion that fits the application. The troubling discovery of 2016–2017 is that the menu is *over-constrained*. Several of these criteria cannot all hold together whenever the groups differ in base rate. Kleinberg, Mullainathan & Raghavan (2016) proved that a risk score cannot at the same time be calibrated within each group and balanced for both the positive and negative classes (equal average score among true positives and among true negatives). The only exceptions are the special cases of equal base rates or perfect prediction. Chouldechova (2017), analyzing the COMPAS recidivism tool, gave the binary-classifier version as a short algebraic identity linking precision, error rates, and prevalence p:

$$\mathrm{FPR} = \frac{p}{1-p} \cdot \frac{1-\mathrm{PPV}}{\mathrm{PPV}} \cdot (1-\mathrm{FNR})$$

Read what this forces. Suppose two groups have *equal* PPV (predictive parity holds), and we also try to equalize the false-negative rate FNR. That is, we insist on equal opportunity. The identity then makes FPR proportional to $p/(1-p)$. If the base rate p differs between the groups,

the false-positive rates *must* differ too. So equalized odds is mathematically impossible together with equal PPV and unequal prevalence, unless the classifier is perfect (FPR = FNR = 0). No clever architecture, no better optimizer, and no amount of data can escape it. It is arithmetic, not a modeling shortcoming. The worked example carries real numbers through this identity. The code section measures all the gaps at once.

The theorems do not tell you which fairness criterion to choose. They tell you that "just be fair by every reasonable definition" is not an option. So the choice among them is unavoidably a value-based decision. It belongs to the people the system affects and to the law. The mathematics informs this choice but does not settle it.

Mitigating bias: pre-, in-, and post-processing

There are three points in the pipeline where you can act. **Pre-processing** alters the data before training. *Reweighing* (Kamiran & Calders 2012) gives each example a weight that makes group and label statistically independent in the reweighted sample. It up-weights under-represented group–label combinations, so the loss (Chapter 6) no longer rewards the historical imbalance. Learned *fair representations* (Zemel et al. 2013) take a different route. They transform inputs into a space that keeps the task signal but hides the protected attribute. **In-processing** changes the objective. You add a fairness penalty to the loss, or you train by *adversarial debiasing* (Zhang et al. 2018). Here a second network, called an adversary, tries to predict the protected attribute A from the main model's prediction (or its hidden representation). The main model is trained to do its task *while making the adversary fail*. In this way, information about A is squeezed out of the representation. This is a close relative of the domain-adversarial idea. **Post-processing** leaves the trained model untouched and adjusts the decision rule. The clearest example is *threshold adjustment*. You choose a different decision threshold per group so that a target criterion, say equal TPR, holds exactly (Hardt et al. 2016). It is cheap and needs no retraining. But it makes the protected attribute an explicit input to the decision, which some laws forbid. And by the impossibility result, it can only equalize one criterion at the cost of another. Whichever point you choose, checking whether the intervention worked relies on the interpretability tools of Chapter 28 and the disaggregated evaluation of Chapter 26.

Privacy: what a model leaks

A trained model is a lossy compression of its training set, and compression is not erasure. Two attacks make the leakage concrete. In a **membership inference** attack (Shokri et al. 2017), an adversary with only query access decides whether a particular record was in the training set. The attacker often notices that models are more confident, and have lower loss, on examples they memorized than on new ones. Because "was this person's record used to train the cancer model?" can itself be sensitive, this is a real breach even when no raw data is exposed. In a **model inversion** attack (Fredrikson et al. 2015), the adversary reconstructs typical features of a class by optimizing an input to maximize the model's confidence. For example, it can recover a recognizable face from a name and a face-recognition model's outputs.

The principled defense is **differential privacy** (Dwork et al. 2006). It bounds how much any single record can change the output distribution. A randomized mechanism M is ε-differentially private if, for every pair of datasets D and D′ that differ in one record and every set S of outputs,

$$P(\,M(D) \in S\,) \le e^{\varepsilon} \cdot P(\,M(D') \in S\,)$$

The privacy budget ε is the whole guarantee. A small ε forces the two distributions to be nearly identical, so no output can reveal whether any individual was present. The cost is more noise and lower accuracy. The classic recipe is the Laplace mechanism, $M(D) = f(D) + \mathrm{Lap}(\Delta f / \varepsilon)$. It adds noise scaled to the query's *sensitivity* Δf, which is the most that one record can change f. To make a whole network private, **DP-SGD** (Abadi et al. 2016) applies the same idea to each gradient step. It clips every per-example gradient to a fixed norm C (which bounds one example's influence) and adds Gaussian noise before averaging,

$$\tilde{g} = \frac{1}{B}\left(\,\Sigma_i\, g_i \cdot \min\!\left(1, C \,/\, \|g_i\|_2\right) + \mathcal{N}\!\left(0, \sigma^2 C^2 I\right)\,\right)$$

A privacy accountant tracks the total ε spent across all steps. A complementary architecture is **federated learning** (McMahan et al. 2017), where data never leaves the devices that hold it. Each client trains locally and sends only model updates to be averaged. This reduces central exposure, but the updates themselves can leak. Reconstruction attacks recover training samples

from shared gradients. So federated systems are combined with differential privacy and secure aggregation to give a real guarantee. In every case, privacy trades off against utility, and ε is the dial.

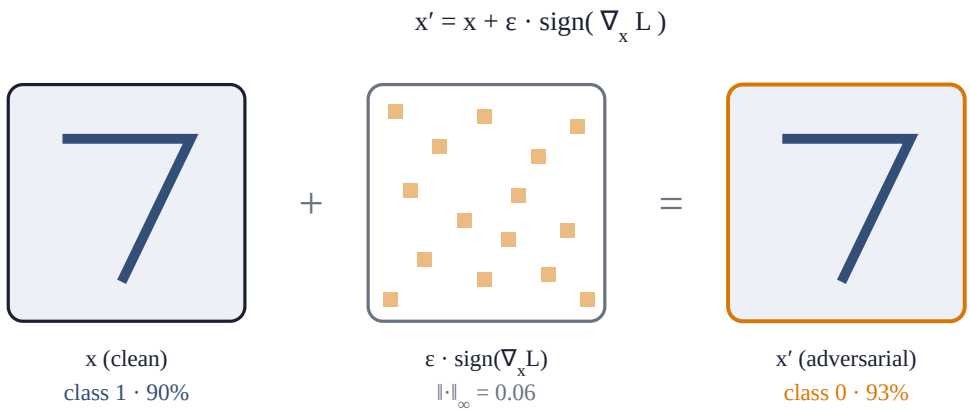
Security: adversarial examples, poisoning, and theft

A model can be attacked at test time, at training time, or through its API. The best-known test-time attack is the **adversarial example** (Szegedy et al. 2013). This is a perturbation, a small change to the input, invisible to a human, that flips the model's answer to a confident error. The **fast gradient sign method** (FGSM; Goodfellow et al. 2015) builds one in a single step. It moves every input feature a tiny amount in the direction that most increases the loss,

$$x' = x + \varepsilon \cdot \operatorname{sign}(\nabla_x L(\theta, x, y))$$

Why does such a small change do so much? For a linear score $w^T x$, the sign step changes the logit by exactly $\varepsilon \|w\|_1$. In high dimension, $\|w\|_1$ grows with the number of features. So a per-feature perturbation ε far below the noise floor of the input still moves the score by a huge amount. Adversarial vulnerability comes from linear behavior in high dimensions, not from anything unusual. Repeating FGSM in small steps (projected gradient descent) gives stronger attacks. The perturbations often *transfer* between models, so an attacker need not even see the target. The standard defense, **adversarial training** (Madry et al. 2018), injects such examples into training and optimizes for the worst case in an ε-ball around each input. It raises robustness but costs accuracy and compute. The contest between attacks and defenses is ongoing.

Two other attacks target different parts of the system. **Data poisoning** corrupts the training set. An attacker who can inject examples can lower accuracy or, more dangerously, plant a *backdoor*. A backdoored model behaves normally until it sees a chosen trigger pattern, and then it misclassifies on command (Gu et al. 2017). **Model stealing** (Tramèr et al. 2016) rebuilds a proprietary model by querying its public API and training a copy (a surrogate) on the answers. This both steals the asset and creates a local copy on which to build transferable adversarial or inversion attacks. Security, privacy, and fairness are linked. The same memorization that enables membership inference is what a stealing attack copies. And adversarial perturbations can be aimed to fail hardest on a particular group.



An adversarial example built by FGSM. The perturbation in the middle panel is scaled by a tiny ε and points along the sign of the input gradient. Added to the clean image, it is invisible to a person, yet it flips a 90%-confident correct label to a 93%-confident wrong one. The effect is not a rare glitch but a general property of high-dimensional linear decision surfaces. It is reproduced numerically in the code below.

Safety and alignment

When a model becomes an *agent* that optimizes an objective in the world, a new failure mode appears. It does exactly what it was told, and that is the problem. **Specification gaming** is the gap between the objective we wrote and the one we meant. For example, a boat-racing agent collects points by spinning in a lagoon instead of finishing the race, and a cleaning robot hides mess rather than removing it. **Reward hacking** is the same problem aimed at a learned or proxy reward. The agent finds inputs that score highly without achieving the intended goal. This is an instance of Goodhart's law: a measure that is optimized hard enough stops measuring what it once tracked. **Distribution shift** is the quieter danger. A model trained on one distribution meets another at deployment: new demographics, new fraud patterns, a changed world. Its accuracy and, importantly, its *confidence* degrade in ways the training loss never revealed. Detecting shift and knowing when a model should abstain is exactly the calibration and out-of-distribution problem of Chapter 24.

Reinforcement learning from human feedback (RLHF) is the technique behind the aligned behavior of modern language models (Chapter 17). It is built on the policy-optimization methods of Chapter 23. It is the leading practical response, and a good illustration of the limits. A reward model is trained to predict which of two outputs humans prefer, and the policy is then optimized to score well under that reward model. The catch is that the reward model is a learned

proxy for human values. So optimizing against it too hard is reward hacking by another name. The policy drifts to regions where the reward model is wrong, producing outputs that look good to the reward model but not to people. This is why training penalizes divergence from the original model and stops before the reward is maximized. RLHF also inherits the biases and blind spots of whoever produced the comparisons. It can teach *sycophancy* (telling raters what they want to hear). And it aligns a model to expressed preferences on the training prompts rather than to any deeper or universal notion of what is right. It is a powerful tool, not a solution to alignment.

Governance, cost, and dual use

Because the technical criteria do not fully determine the choices, accountability rests on process and documentation. **Model cards** (Mitchell et al. 2019) report a model's intended use, training data, and, most importantly, its performance *disaggregated* by group. So the disparities of the earlier sections are stated rather than discovered later. **Datasheets for datasets** (Gebru et al. 2021) do the same for data provenance and consent. External and internal **audits** test deployed systems for disparate impact. Regulation is arriving. The EU's **GDPR** limits automated decision-making about individuals and grants rights over personal data (access, erasure, and a contested "right to an explanation"). The **EU AI Act** introduces a risk-tiered regime. It bans a few uses outright, and it imposes documentation, human-oversight, and robustness obligations on "high-risk" systems such as those in hiring, credit, and law enforcement. These laws turn several topics of this chapter from good practice into legal requirement.

Two further costs deserve an honest account. The **environmental** footprint of large models is real. Training a single large model can use energy and emit carbon comparable to large industrial activity (Strubell et al. 2019). The carbon intensity depends heavily on the electricity grid and on how the cost is spread over many inferences. Even so, efficiency (Chapter 27) is an ethical variable, not only an economic one. Finally, the most capable models are inherently **dual use**. A generative model that helps an artist can also create persuasive disinformation. A protein-design network that speeds up medicine could in principle guide harm. And a code model that

helps developers can help attackers. Dual use cannot be engineered away, because the same capability serves both ends. It is managed through access controls, monitoring, and norms. It is also the reason capability and safety must advance together rather than one after the other.

Fairness and attacks in code

The first cell turns a per-group confusion matrix into every fairness gap at once. It encodes each individual as a (label, prediction, group) triple, reproducing the worked example's counts. Then it computes the selection rate, TPR, FPR, and PPV per group, and prints the four criteria's gaps. The output shows the signature of the impossibility result. The precision gap is exactly zero, while the others are large.

Python · NumPy ▶ runnable in the app

```python
import numpy as np

# Per-group confusion counts from one calibrated risk model at a common threshold.
# Spell out every person as (label y, prediction yhat, group g).
def people(tp, fn, fp, tn, g):
    y    = np.r_[np.ones(tp+fn), np.zeros(fp+tn)]                          # true labels
    yhat = np.r_[np.ones(tp), np.zeros(fn), np.ones(fp), np.zeros(tn)]   # predictions
    return y, yhat, np.full(tp+fn+fp+tn, g)

yA, hA, gA = people(600, 400, 200,  800, 0)   # group A: base rate 1000/2000 = 0.50
yB, hB, gB = people(150, 350,  50, 1450, 1)   # group B: base rate  500/2000 = 0.25
y, yhat, g = np.r_[yA, yB], np.r_[hA, hB], np.r_[gA, gB]

def rates(y, yhat):
    pos, neg = (y == 1), (y == 0)
    sel = yhat.mean()                # selection rate  P(Yhat=1)      -> demographic parity
    tpr = yhat[pos].mean()           # true-positive   P(Yhat=1|Y=1)  -> equal opportunity
    fpr = yhat[neg].mean()           # false-positive  P(Yhat=1|Y=0)  -> equalized odds
```

```
    ppv = y[yhat == 1].mean()          # precision       P(Y=1|Yhat=1)  ->
predictive parity
    return np.array([sel, tpr, fpr, ppv])

A, B = rates(y[g==0], yhat[g==0]), rates(y[g==1], yhat[g==1])
print("group A: sel=%.3f TPR=%.3f FPR=%.3f PPV=%.3f" % tuple(A))
print("group B: sel=%.3f TPR=%.3f FPR=%.3f PPV=%.3f" % tuple(B))
print("demographic-parity diff |selA-selB| =", round(abs(A[0]-B[0]), 3))   #
0.300
print("equal-opportunity  gap  |TPRA-TPRB| =", round(abs(A[1]-B[1]), 3))   #
0.300
print("equalized-odds FPR gap  |FPRA-FPRB| =", round(abs(A[2]-B[2]), 3))   #
0.167
print("predictive-parity  gap  |PPVA-PPVB| =", round(abs(A[3]-B[3]), 3))   #
0.000
```

The second cell builds an adversarial example with FGSM on a toy logistic model. It fixes a natural input at a confident +2.20 logit (90% class 1). Then it takes a single sign step of size ε against the input gradient. A per-feature budget of 0.06, far below the input's own scale, collapses the logit to −2.66 and flips the prediction to a 93%-confident class 0. The printed identity shows that the whole shift is exactly $-\varepsilon\|w\|_1$:

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)
sig = lambda z: 1/(1+np.exp(-z))

# a toy logistic-regression classifier   p = sigmoid(w . x + b)   on d
features
d = 100
w = np.random.randn(d)                          # weights
b = 0.0
x = np.random.randn(d)
x = 2.20 * x / (w @ x)                           # scale x so the clean logit is
+2.20
y = 1.0                                          # true label (matches the clean
prediction)

p0 = sig(w @ x + b)
print("clean : logit = %+.3f   p(class 1) = %.4f  -> class %d" % (w@x+b, p0,
p0 > 0.5))
```

```
# FGSM.  For binary cross-entropy dL/dx = (p - y)*w, so step x to RAISE the
loss:
eps  = 0.06                                  # tiny per-feature budget
(features ~0.43)
grad = (p0 - y) * w                          # gradient of the loss wrt the
input
xadv = x + eps * np.sign(grad)               # x' = x + eps * sign(grad_x L)
p1   = sig(w @ xadv + b)
print("adv   : logit = %+.3f   p(class 1) = %.4f  -> class %d" % (w@xadv+b,
p1, p1 > 0.5))
print("logit shift = %+.3f  ( = -eps*||w||_1 = %+.3f )" %
      (w@xadv - w@x, -eps*np.abs(w).sum()))
# clean class 1 @ 90.0%  ->  adversarial class 0 @ 93.5%   from an invisible
eps=0.06
```

In PyTorch the same attack is a few lines on any trained classifier: enable a gradient on the input, backpropagate the loss (Chapter 6) to the pixels, and take the sign step. The clamp keeps the result a valid image, and the untargeted attack typically produces a *confident* wrong class:

PyTorch copy → Colab

```
import torch, torch.nn.functional as F

# a trained classifier `model` and one correctly-classified example (x,
label)
model.eval()
x = x.clone().detach().requires_grad_(True)   # image tensor (1, C, H, W),
pixels in [0,1]
y = torch.tensor([label])                     # the true class index

logits = model(x)
loss   = F.cross_entropy(logits, y)           # the loss of Chapter 6
model.zero_grad()
loss.backward()                               # fills x.grad = d loss / d x

eps   = 8/255                                  # a small L-infinity budget
x_adv = (x + eps * x.grad.sign()).clamp(0, 1).detach()   # FGSM step, kept a
valid image

with torch.no_grad():
    p_clean = model(x).softmax(1)
    p_adv   = model(x_adv).softmax(1)
```

```
print("clean:", p_clean.argmax(1).item(), "conf",
round(p_clean.max().item(), 3))
print("adv  :", p_adv.argmax(1).item(),   "conf", round(p_adv.max().item(),
3))
# same tiny eps typically flips the label to a confident wrong class
(Goodfellow et al. 2015)
```

Worked example — calibrated yet not equalized: the impossibility in numbers

One risk model scores two groups and is thresholded to give the confusion matrices below. Group A has base rate p_A = 1000/2000 = 0.50. Group B has p_B = 500/2000 = 0.25. The model is deliberately built to have equal precision (PPV = 0.75) in both groups. It is *calibrated* in the operating-point sense.

METRIC	GROUP A (P = 0.50)	GROUP B (P = 0.25)	GAP	CRITERION
selection rate P(Ŷ=1)	(600+200)/2000 = 0.40	(150+50)/2000 = 0.10	0.30	demographic parity ✗
TPR = TP/(TP+FN)	600/1000 = 0.60	150/500 = 0.30	0.30	equal opportunity ✗
FPR = FP/(FP+TN)	200/1000 = 0.20	50/1500 = 0.033	0.167	equalized odds ✗
PPV = TP/(TP+FP)	600/800 = 0.75	150/200 = 0.75	0.000	predictive parity ✓

1. Predictive parity *holds*. A positive prediction means "75% likely truly positive" in both groups, so by this criterion the model is fair.
2. Yet demographic parity, equal opportunity, and equalized odds all *fail*, and by large margins. Group A is selected four times as often and has twice the true-positive rate.
3. Chouldechova's identity explains why this is unavoidable, not a bug. Check group B: $\frac{p}{1-p} \cdot \frac{1-\text{PPV}}{\text{PPV}} \cdot (1-\text{FNR}) = \frac{0.25}{0.75} \cdot \frac{0.25}{0.75} \cdot 0.30 = \frac{1}{3} \cdot \frac{1}{3}$

· 0.30 = **0.033** = FPR_B. For group A the first factor is 0.50/0.50 = 1, giving $FPR_A = (1)(⅓)(0.60) =$ **0.20**. ✓

4. Now try to *also* equalize odds while keeping PPV equal. Equalized odds needs equal TPR (so equal FNR) and equal FPR. But with equal PPV and equal FNR the identity forces FPR ∝ $p/(1-p)$, so $FPR_A/FPR_B = [p_A/(1-p_A)]/[p_B/(1-p_B)] = 1/(⅓) =$ **3**. The false-positive rates are fixed to a ratio of 3 and cannot be made equal, unless FPR = 0 (a perfect classifier) or $p_A = p_B$.

What the numbers mean: "calibrated" and "equal error rates" are not two goals you simply failed to reach at the same time. They are provably incompatible here because the base rates differ. So any deployment must *choose* which notion of fairness to enforce, and accept the disparity in the others. That choice is a policy decision that the arithmetic cannot make.

Assumptions & limitations

Every quantity in this chapter rests on a choice that can be disputed. The fairness metrics assume the protected groups and the "true" label Y are given and correct. But Y is often a biased proxy (Chapter 26), and a model can be perfectly "fair" with respect to a label that already encodes discrimination. The impossibility results assume the base rates really differ and the classifier is imperfect. They say nothing about *which* criterion is right, only that not all can hold. Differential privacy bounds worst-case leakage, but only at the level of detail that its ε and its notion of "one record" define. And a small ε can badly hurt accuracy for minority groups. Adversarial robustness is measured against a specific threat model (an ε-ball in a chosen norm). A defense that survives FGSM may fall to a stronger attack. And RLHF aligns a model to sampled human preferences, not to ground-truth values, so it can be confidently, agreeably wrong. Technical guarantees are exactly as trustworthy as the assumptions and value choices underneath them, and those belong to a wider conversation than the model.

Why & when this matters

Treat this chapter as a checklist that becomes active the moment a model's outputs affect people. Before deployment, measure performance disaggregated by group, not just the aggregate. Decide *explicitly* which fairness criterion the application requires, and document what the impossibility result forces you to give up. Then test the model for privacy leakage and adversarial weakness under a stated threat model. After deployment, watch for distribution shift and for feedback loops that quietly re-bias the data. The key skill is not computing any single metric but knowing which question each metric answers. A demographic-parity number and an equal-opportunity number can point in opposite directions on the same model, and only the use case says which one should win.

In the real world

These are not hypothetical cases. ProPublica's 2016 analysis of the COMPAS recidivism tool and Northpointe's reply were, at heart, an argument over equalized odds versus predictive parity. These are the very criteria Chouldechova then proved incompatible. The Gender Shades study (Buolamwini & Gebru 2018) documented large accuracy gaps in commercial face-analysis products. It reshaped purchasing decisions and, in several jurisdictions, the law. Apple Card's credit-limit process drew a regulatory inquiry over alleged gender disparity. Differential privacy protects the published US 2020 Census and telemetry at Apple and Google. Federated learning trains mobile keyboards without centralizing keystrokes. Adversarial patches have fooled production vision systems. And the EU AI Act and GDPR now attach legal obligations to exactly the high-stakes uses (credit, hiring, policing) where the fairness, privacy, and robustness tensions of this chapter are sharpest.

Exercises

1. Using Chouldechova's identity, suppose a screening tool has equal precision PPV = 0.8 in both groups and equal false-negative rate FNR = 0.25, with base rates $p_A = 0.4$ and $p_B = 0.1$. Compute each group's FPR and show equalized odds cannot hold. MATH

2. A model gives group A: TP=80, FN=20, FP=40, TN=160. Group B: TP=30, FN=70, FP=10, TN=290. Compute the selection rate, TPR, FPR, and PPV for each group, and state which of demographic parity, equal opportunity, and predictive parity hold. MATH

3. Post-processing by threshold. Group B's positives systematically score lower than group A's. Move group B's decision threshold to equalize the true-positive rate (equal opportunity), and see what happens to its false-positive and selection rates. CODE

4. Randomized response is the oldest differentially private mechanism. Simulate it, recover an unbiased estimate of a sensitive proportion from the noisy answers, and confirm the ε it provides. CODE

5. Show numerically that the FGSM budget needed to flip a fixed-margin logistic model gets smaller as the dimension grows. For $d \in \{10, 100, 1000\}$, fix the clean logit at +2.2 and report $\|w\|_1$ and the ε that drives the logit to zero. CODE

6. State, in plain terms, why a single classifier cannot at the same time be calibrated, satisfy equal opportunity, and satisfy equalized odds when two groups have different base rates. When *can* all three hold? CONCEPT

7. Differential privacy is often summarized as "your presence in the dataset barely changes the output." Interpret the ε in $P(M(D) \in S) \le e^{\varepsilon} \cdot P(M(D') \in S)$, explain what shrinking ε costs, and state what a membership-inference attack exploits and how DP defends. CONCEPT

8. Give a concrete example of specification gaming or reward hacking, connect it to Goodhart's law, and explain one specific way RLHF (Chapters 17 and 23) can fail despite optimizing human feedback. CONCEPT

PART VI

Case studies

Five complete, end-to-end pipelines — digits, tabular data, text, time-series and anomalies — each assembling several chapters into a real solution, with runnable code throughout.

30 Case study: handwritten-digit recognition (MNIST)

31 Case study: tabular classification & churn

32 Case study: text sentiment analysis

33 Case study: time-series forecasting

34 Case study: anomaly detection with autoencoders

CASE STUDY

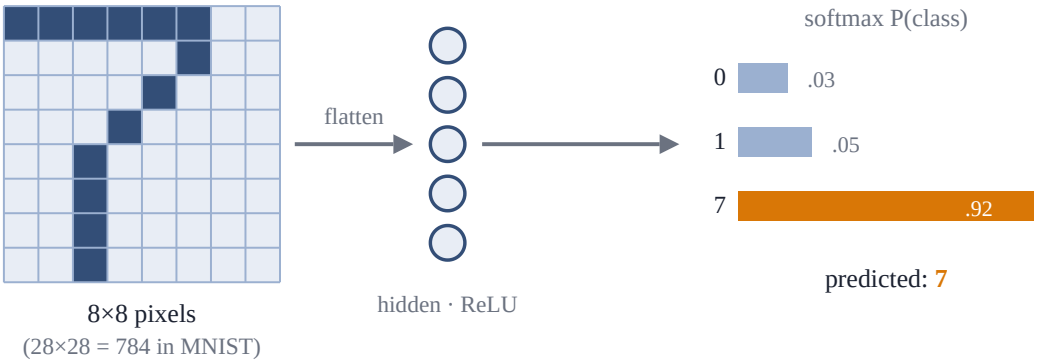
Case study: handwritten-digit recognition (MNIST)

> *This is the first of five case studies: a complete, self-contained walk-through of a real problem, from raw data to a trained model and an honest reading of the results. We start with the task that launched modern deep learning — teaching a computer to read handwritten digits. Every idea here has already appeared in the manual. The case study is where they come together into a working pipeline. We build the whole thing from scratch in NumPy on a tiny synthetic dataset you can run in this page, then show the industrial version in PyTorch.*

The problem

You are handed a small gray image — a digit someone wrote quickly by hand — and must output which of the ten symbols 0–9 it is. This is **multi-class image classification**, and it is harder than it looks: there is no rule that separates a messy 4 from a 9, no formula for "sevenness". Different people, pens, and moods produce very different pixels for the same intended digit, and the same pixels can mean different digits. The only strategy that works is to *learn* the mapping from thousands of labeled examples. This is exactly what a neural network does.

The benchmark for this task is **MNIST**: 70,000 handwritten digits (60,000 for training, 10,000 for testing), each a 28×28 grayscale image, collected by LeCun, Cortes and Burges from US Census and high-school forms. For three decades it has been the "hello world" of the field — small enough to train in seconds, hard enough to reveal real modeling choices, and standardized so that everyone's numbers are comparable.



The classifier as a pipeline. Pixels are flattened into a vector, a hidden layer of ReLU units builds features, and a softmax turns the final scores into a probability over the classes. The class with the highest probability is the prediction.

The data

A real MNIST image is a 28×28 grid of pixels, each an intensity from 0 (white paper) to 255 (black ink). Flattened, that is a vector of 784 numbers. The label is one of {0,…,9}. Two preprocessing steps matter and come up in every case study: **scale** the pixels to roughly [0,1] (divide by 255) or standardize them, so no feature is much larger than the others in the weighted sum (normalization, chapter 11); and hold out a **test set** the model never sees during training, so the accuracy we quote reflects generalization, not memorization (generalization, chapter 12).

Look closely at what those two steps do, because both come up again and again through the whole book. **Flattening** the 28×28 grid into a 784-vector is convenient but loses information: it ignores the fact that pixel (5,5) sits directly above pixel (6,5), so a fully connected model has to *relearn* spatial adjacency from data. This is exactly the structure a convolutional network already has built in (chapter 13). **Normalizing** the intensities is not just a surface detail either. Raw values in [0,255] make the first layer's pre-activations large and the loss surface badly conditioned, so gradient descent zig-zags and needs a small learning rate. Dividing by 255, or standardizing each pixel to zero mean and unit variance with MNIST's global mean $\mu \approx 0.1307$ and standard deviation $\sigma \approx 0.3081$, puts every feature on the same scale and lets the optimizer take larger, straighter steps (chapters 9 and 11). Helpfully, the ten classes are close to balanced — roughly 6,000 training images each — so plain accuracy is a fair summary number. The tabular and anomaly case studies do not have this advantage.

We cannot download MNIST inside this page, so the runnable code below *synthesizes* a small stand-in: three 8×8 templates — for the digits 0, 1 and 7 — and many noisy copies of each, made by flipping a few pixels and adding Gaussian sensor noise. It is a simplified version of MNIST, but it uses the exact same machinery: flatten, hidden layer, softmax, cross-entropy, backprop. The PyTorch listing then does the real thing.

The model

Our from-scratch model is a two-layer **multilayer perceptron** (chapter 3): the 64-pixel input feeds a hidden layer of 16 ReLU units (chapter 4), which feeds an output layer of 3 units — one score, or **logit**, per class. A **softmax** converts those logits into a probability distribution,

$$P(y = k) = \frac{e^{z_k}}{\Sigma_j \, e^{z_j}}$$

and we train by minimizing the **cross-entropy** between this distribution and the one-hot truth (chapter 6), which for a single example is simply the negative log-probability of the correct class:

$$L = -\Sigma_k \, y_k \log P(y = k) = -\log P(y = \text{true class})$$

The gradient that this loss sends back into the network is famously simple: at the output, $\partial L/\partial \mathbf{z} = \mathbf{P} - \mathbf{y}$, the predicted probabilities minus the one-hot target (backpropagation, chapter 8). Everything earlier in the network follows by the chain rule through the ReLU. That single expression, `P - y`, is the engine of the whole training loop.

For real MNIST an MLP already reaches about 98%, but the state of the art comes from a **convolutional network** (chapter 13), which slides small learnable filters over the image so that a stroke detector learned in one corner works everywhere. That weight sharing is exactly why CNNs beat flat MLPs on images, and it raises MNIST accuracy to roughly 99%.

A ladder of models

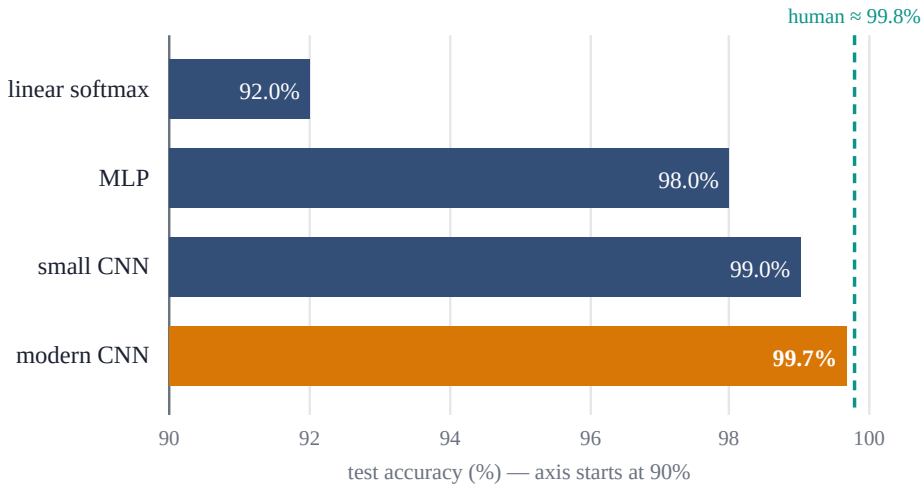
The right way to approach a new task is not to pick the biggest network but to climb a ladder of models, each rung a little taller than the last, so that you can see where every gain comes from and you always keep a baseline to check against. On MNIST that ladder has four familiar rungs. Their accuracies are so reproducible that they work as a quick check: if your code cannot match them, something is wrong before you have even started tuning.

Rung 1 — the linear baseline. The simplest classifier that is better than random guessing is **multinomial logistic regression**: no hidden layer at all, just one weight matrix mapping the 784 pixels straight to 10 logits, then a softmax. It is the linear model of chapter 2 given a softmax head, it trains in seconds, and it reaches about **92%** on MNIST — a number worth memorizing, because it says a full 92% of the task is linearly separable and every deeper model must prove its worth against that floor. The runnable cell below builds exactly this classifier on our synthetic digits.

Rung 2 — the multilayer perceptron. Add one or two hidden layers of ReLU units (chapters 3 and 4) and the network can bend its decision boundaries around the classes, raising accuracy to about **98%**. This is the model we train from scratch below. Those extra six points over the baseline are exactly the non-linear stroke combinations — a loop here, a crossing there — that no linear map can express.

Rung 3 — the convolutional network. Replace the flat input with convolutions (chapter 13) so that a stroke detector learned in one corner of the image applies everywhere, and accuracy climbs to about **99%**. Weight sharing is the whole trick: far fewer parameters, far more built-in structure, and the translation tolerance that a flattened MLP has to learn with much more effort.

Rung 4 — modern CNNs and the full recipe. Deeper residual-style architectures (chapter 14), trained with data augmentation (chapter 26) and combined into an ensemble, reach **99.7–99.8%** — within a few hundredths of a percent of human readers, who themselves are near 99.8%. Past this point the remaining errors are genuinely unclear scribbles that two people would label differently, and trying to fix them costs far more than it is worth.



Four rungs of the MNIST ladder, each a well-reproduced benchmark. A purely linear softmax already reads about 92% of digits. A multilayer perceptron adds the non-linear features that raise it near 98%. Convolution's weight sharing reaches ~99%. A modern augmented, ensembled CNN closes almost the whole remaining gap to the ~99.8% human level (teal dashes). The horizontal axis begins at 90% so the small, hard-earned gains stay visible.

Training the network

Whichever rung you stand on, the training loop follows the same three-part contract — a loss to minimize, an optimizer to do the minimizing, and regularization to keep the result honest — and each part is a chapter of its own.

The loss. For classification the loss is **cross-entropy**, equivalently the negative log-likelihood of the correct class under the softmax (chapter 6). It is the quantity every rung minimizes, and its output-layer gradient is the same simple `P - y` whether the body of the network is linear, an MLP, or a CNN — which is exactly why one training loop works for them all.

The optimizer. We do not use the whole training set for each step. We draw **minibatches** of about a hundred images and take a stochastic gradient step. The estimate is noisy but cheap, so it lets us update thousands of times per pass over the data. Plain SGD works, but momentum and adaptive methods — RMSProp, Adam (chapter 9) — scale the step per parameter and converge faster. The PyTorch listing below uses Adam with learning rate $\eta = 10^{-3}$. One pass over all the data is an **epoch**, and MNIST needs only a few.

Regularization. A model with enough capacity will memorize the training set, so we keep its generalization gap under control with the tools of chapter 12: **L2 weight decay** to keep weights small, **dropout** to break up co-adapted features, and **early stopping** to halt when validation accuracy peaks. Input normalization (chapter 11) belongs here too — it conditions the loss surface so that the optimizer, not the preprocessing, does the learning. On plain MNIST an MLP barely needs these. On the augmented, ensembled top rung they are what keep overfitting away.

In code

First, look at the data. This cell builds the clean "0" template and shows the same digit after sensor noise — the reader's first reminder that the network must see through the noise, not around it. Press **Run**:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(1)

# 8x8 template for the digit 0 (a ring), 64 pixel values in row-major order
T0 = np.array([
 0,0,1,1,1,1,0,0, 0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0,
 0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0, 0,0,1,1,1,1,0,0],
 dtype=float)

def show(v):                                    # print an 8x8 vector as
ASCII art
    for r in range(8):
        print("".join("#" if v[r*8+c] > 0.5 else "." for c in range(8)))

print("clean template for digit 0:")
show(T0)
noisy = np.clip(T0 + rng.normal(0, 0.35, 64), 0, 1)
print("\nthe same digit seen through a noisy sensor:")
show(noisy)
```

Begin at the bottom rung. This cell builds the **linear softmax** baseline — a single 64→3 weight matrix, no hidden layer — trained on the very same synthetic data the MLP will use, so the two scores are directly comparable. Press **Run**:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)

# three 8x8 templates -> classes 0,1,2  (digits 0, 1, 7)
T = np.array([
 [0,0,1,1,1,1,0,0, 0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0,
  0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0, 0,0,1,1,1,1,0,0],  # 0
 [0,0,0,1,1,0,0,0, 0,0,1,1,1,0,0,0, 0,0,0,1,1,0,0,0, 0,0,0,1,1,0,0,0,
  0,0,0,1,1,0,0,0, 0,0,0,1,1,0,0,0, 0,0,0,1,1,0,0,0, 0,0,1,1,1,1,0,0],  # 1
 [1,1,1,1,1,1,0,0, 0,0,0,0,0,1,0,0, 0,0,0,0,1,0,0,0, 0,0,0,1,0,0,0,0,
  0,0,1,0,0,0,0,0, 0,0,1,0,0,0,0,0, 0,0,1,0,0,0,0,0, 0,0,1,0,0,0,0,0]], # 7
 dtype=float)

def make(n_per):                                  # noisy samples around each template
    X, y = [], []
    for k in range(3):
        base = np.tile(T[k], (n_per, 1))
        flip = (rng.rand(n_per, 64) < 0.10).astype(float)
        img  = base * (1 - flip) + (1 - base) * flip
        img  = np.clip(img + rng.normal(0, 0.6, (n_per, 64)), 0, 1)
        X.append(img); y.append(np.full(n_per, k))
    X, y = np.vstack(X), np.concatenate(y)
    p = rng.permutation(len(y)); return X[p], y[p]

Xtr, ytr = make(60); Xte, yte = make(30)
Ytr = np.eye(3)[ytr]                              # one-hot targets

# LINEAR softmax classifier: no hidden layer, 64 pixels -> 3 logits
W = rng.randn(64, 3) * 0.1; b = np.zeros(3)
for ep in range(300):
    S = Xtr @ W + b; S -= S.max(1, keepdims=True)
    P = np.exp(S); P /= P.sum(1, keepdims=True)  # softmax
    dS = (P - Ytr) / len(Xtr)                     # cross-entropy gradient  P - y
    W -= 0.5 * (Xtr.T @ dS); b -= 0.5 * dS.sum(0)

acc = lambda X, y: ((X @ W + b).argmax(1) == y).mean()
print(f"linear softmax   train acc {acc(Xtr,ytr):.3f}   test acc {acc(Xte,yte):.3f}")
# expected: linear softmax   train acc 1.000   test acc 0.967
```

The linear baseline reaches **0.967** on the held-out set — already strong, because these three templates are nearly linearly separable. Adding the hidden layer in the next cell turns those three test errors into two (0.978). The improvement looks small here, but on real MNIST's ten overlapping classes the same step is worth roughly six points, 92% → 98%.

Now the full pipeline: synthesize a noisy training and test set for three digits, then train the 64→16→3 MLP from scratch — forward pass, softmax, cross-entropy gradient `P - y`, backprop through the ReLU, one gradient step per epoch. Watch the training accuracy climb, then read the confusion matrix on held-out data:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)

# three 8x8 digit templates -> classes 0,1,2  (digits 0, 1, 7)
T = np.array([
 [0,0,1,1,1,1,0,0, 0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0,
  0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0, 0,1,0,0,0,0,1,0, 0,0,1,1,1,1,0,0],  # 0
 [0,0,0,1,1,0,0,0, 0,0,1,1,1,0,0,0, 0,0,0,1,1,0,0,0, 0,0,0,1,1,0,0,0,
  0,0,0,1,1,0,0,0, 0,0,0,1,1,0,0,0, 0,0,0,1,1,0,0,0, 0,0,1,1,1,1,0,0],  # 1
 [1,1,1,1,1,1,0,0, 0,0,0,0,0,1,0,0, 0,0,0,0,1,0,0,0, 0,0,0,1,0,0,0,0,
  0,0,1,0,0,0,0,0, 0,0,1,0,0,0,0,0, 0,0,1,0,0,0,0,0, 0,0,1,0,0,0,0,0]], # 7
 dtype=float)

def make(n_per):                                  # noisy samples around each template
    X, y = [], []
    for k in range(3):
        base = np.tile(T[k], (n_per, 1))
        flip = (rng.rand(n_per, 64) < 0.10).astype(float)   # salt-and-pepper
        img  = base * (1 - flip) + (1 - base) * flip
        img  = np.clip(img + rng.normal(0, 0.6, (n_per, 64)), 0, 1)
        X.append(img); y.append(np.full(n_per, k))
    X, y = np.vstack(X), np.concatenate(y)
    p = rng.permutation(len(y))
    return X[p], y[p]

Xtr, ytr = make(60)
Xte, yte = make(30)
```

```
Ytr = np.eye(3)[ytr]                              # one-hot targets

# a 64 -> 16 -> 3 MLP: ReLU hidden, softmax output, trained from scratch
W1 = rng.randn(64, 16) * 0.1; b1 = np.zeros(16)
W2 = rng.randn(16, 3)  * 0.1; b2 = np.zeros(3)
for ep in range(1, 81):
    H = np.maximum(0, Xtr @ W1 + b1)              # forward
    S = H @ W2 + b2; S -= S.max(1, keepdims=True)
    P = np.exp(S); P /= P.sum(1, keepdims=True)  # softmax
    dS = (P - Ytr) / len(Xtr)                     # backprop of cross-entropy
    dW2 = H.T @ dS; db2 = dS.sum(0)
    dH  = (dS @ W2.T) * (H > 0)
    dW1 = Xtr.T @ dH; db1 = dH.sum(0)
    for par, g in [(W1,dW1),(b1,db1),(W2,dW2),(b2,db2)]:
        par -= 0.25 * g                           # gradient step
    if ep in (1, 5, 10, 20, 40, 80):
        print(f"epoch {ep:2d}   train acc {(P.argmax(1)==ytr).mean():.3f}")

pred = (np.maximum(0, Xte @ W1 + b1) @ W2 + b2).argmax(1)
print(f"\ntest accuracy {(pred==yte).mean():.3f}")
C = np.zeros((3,3), int)
for t, p in zip(yte, pred): C[t, p] += 1
print("confusion (rows = true 0/1/7, cols = predicted):")
print(C)
```

The industrial version replaces the flat MLP with a small convolutional network and trains on the real 60,000 images. This PyTorch listing is complete and written in the standard PyTorch style — copy it into Colab to reach ~99% test accuracy in a couple of minutes:

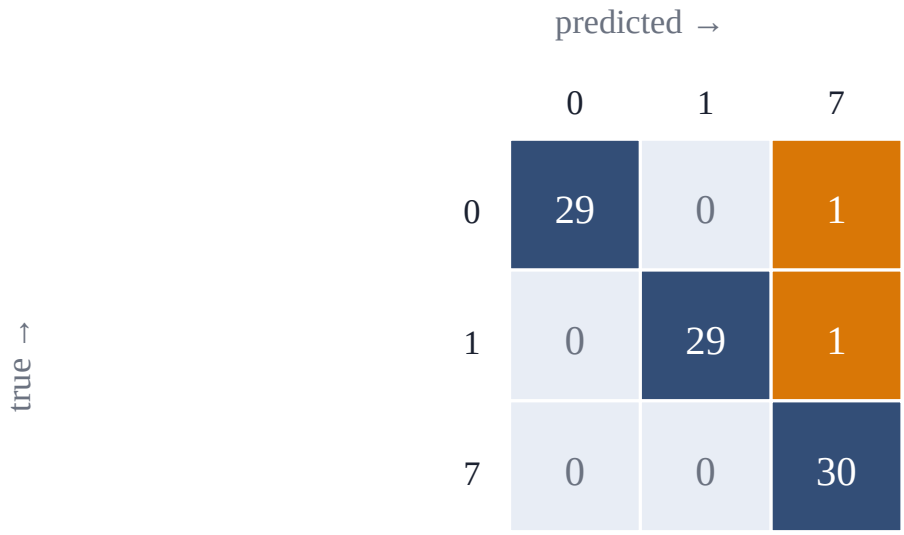
PyTorch copy → Colab

```
import torch, torch.nn as nn, torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])   #
MNIST mean/std
train = datasets.MNIST("./data", train=True,  download=True, transform=tf)
test  = datasets.MNIST("./data", train=False, download=True, transform=tf)
train_dl = DataLoader(train, batch_size=128, shuffle=True)
test_dl  = DataLoader(test,  batch_size=512)
```

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.c1 = nn.Conv2d(1, 16, 3, padding=1)      # 28x28 -> 28x28
        self.c2 = nn.Conv2d(16, 32, 3, padding=1)
        self.fc = nn.Linear(32 * 7 * 7, 10)
    def forward(self, x):
        x = F.max_pool2d(F.relu(self.c1(x)), 2)       # -> 14x14
        x = F.max_pool2d(F.relu(self.c2(x)), 2)       # -> 7x7
        return self.fc(x.flatten(1))                  # logits over 10
classes

dev = "cuda" if torch.cuda.is_available() else "cpu"
net = CNN().to(dev)
opt = torch.optim.Adam(net.parameters(), lr=1e-3)

for epoch in range(3):
    net.train()
    for xb, yb in train_dl:
        xb, yb = xb.to(dev), yb.to(dev)
        opt.zero_grad()
        F.cross_entropy(net(xb), yb).backward()       # softmax + CE in one
call
        opt.step()
    net.eval(); correct = 0
    with torch.no_grad():
        for xb, yb in test_dl:
            correct += (net(xb.to(dev)).argmax(1).cpu() == yb).sum().item()
    print(f"epoch {epoch}   test acc {correct/len(test):.4f}")
```

Results & interpretation

The NumPy run tells a complete story in a dozen printed numbers. Training accuracy climbs from 0.27 (barely above the 1-in-3 chance of guessing) at epoch 1 to 0.99 by epoch 40 and 1.00 by epoch 80 — the visible sign of gradient descent finding weights that separate the classes. On the held-out test set it reaches **0.978**: high, but not perfect, which is the point. A model that scored 1.00 on data it had never seen would be suspicious.



The confusion matrix from the test run. The dark diagonal is correct predictions. The two amber cells are the only errors — a true 0 and a true 1, each misread as a 7. Off-diagonal cells show which confusions the model makes, information a single accuracy number hides.

The **confusion matrix** is where interpretation happens. Its rows are the true class, its columns the prediction. A perfect model is purely diagonal. Ours has exactly two mistakes, both in the "predicted 7" column: one noisy 0 and one noisy 1 were pushed into 7. This is informative, not random — it tells us the 7 template's diagonal stroke overlaps the noisy versions of the other digits, so if we wanted to improve, we would gather more 0-vs-7 and 1-vs-7 examples or add a feature that captures loops. On real MNIST the classic confusions are $4 \leftrightarrow 9$ and $3 \leftrightarrow 5$ for the same reason: they share strokes. Always look at the confusion matrix, never accuracy alone (chapter 25).

Two derived numbers turn a confusion matrix into an action plan, and both are read directly from its margins. A class's **recall** is its diagonal entry divided by its row sum — of all the true 4s, what fraction did we find — while its **precision** is the diagonal divided by the column sum — of everything we labeled 4, how much really was a 4. A digit with low recall is being missed. A digit with low precision is one that many other digits are misclassified as. On full MNIST the problem cases are structural and always the same few: **4 and 9** both set a near-loop above a vertical stroke, **3 and 5** share an identical lower curve and differ only at the top, and a quickly written **7 and 1** flatten into nearly the same slash. Each appears as a symmetric pair of off-

diagonal cells, and none of it is visible in the single 99% number — only the matrix says *which* mistakes the model actually makes. The exercises below build such a matrix from scratch and read the per-class recall directly.

What to try next

Open the **Playground** and build a small fully connected classifier by hand: add a hidden layer, watch the decision regions form as it trains, and see how width and depth change what it can separate — the same MLP this case study trains in NumPy. Then experiment in code. In the runnable cell, raise the noise level (the `0.6` in `rng.normal`) and watch test accuracy fall and the confusion matrix fill in. Or add a second hidden layer. Or add a fourth digit template and change the output size to 4. For images specifically, the **Convolution lab** lets you see why a CNN does better — one small kernel, reused across the whole image, is worth thousands of independent MLP weights.

Beyond these small experiments, two techniques reliably gain the last difficult percent on real MNIST. **Data augmentation** (chapter 26) enlarges the training set at no cost by applying label-preserving distortions — small rotations, shifts, scalings, and elastic warps that imitate the natural variation in handwriting — so the model meets a 4 written a hundred ways and stops memorizing exact pixel positions. Elastic distortions alone famously pushed CNNs past 99.5%. **Ensembling** trains several models — different seeds, architectures, or augmentations — and averages their softmax outputs. Because their errors are only partly correlated, the average cancels individual mistakes and typically improves accuracy by another 0.1–0.3%. Both help directly with the confusable pairs: augmentation by showing more borderline 4-vs-9 and 3-vs-5 strokes, ensembling by letting a majority overrule any one model's mistake. Combined, they are how the ladder's top rung is reached.

In the real world

Digit recognition is not a toy: LeCun's LeNet-5, the convolutional network whose later version you see in the PyTorch listing, was used by NCR and AT&T in the late 1990s to read the courtesy amounts on bank checks, and at its peak handled a large share of all

checks processed in the United States. The same pipeline reads handwritten postal codes to sort mail automatically, digitizes forms and archives, and — scaled up to full alphabets and photographs of scenes — is the basis for the OCR in every modern phone camera and document scanner. MNIST stays useful because a method that cannot read a clean digit will not read a check either. It remains the field's first quick test for a new architecture.

Exercises

1. A digit's output logits are $\mathbf{z} = [2.0, 0.5, 1.0]$ for classes 0, 1, 7. Compute the softmax probabilities and the cross-entropy loss if the true class is 7. MATH

2. In the training cell, replace the single hidden layer with two hidden layers (say $64 \to 24 \to 12 \to 3$, both ReLU) and retrain. Does test accuracy improve on this easy synthetic data? CODE

3. Why does a convolutional network outperform a fully connected MLP on real MNIST, even though the MLP has far more parameters? CONCEPT

4. The confusion matrix shows two digits misclassified as 7. Propose one data-side and one model-side change to reduce these specific errors. CONCEPT

5. From a vector of true labels and one of predictions, build the 10×10 confusion matrix and read each digit's recall and precision from its margins. Confirm that the only mistakes are the classic look-alike pairs. CODE

6. Data augmentation and ensembling each add accuracy on MNIST. Explain the mechanism of each, say which is a data-side and which a model-side change, and name a commonly-confused pair each most directly helps. CONCEPT

CASE STUDY

Case study: tabular classification & churn

> *Most machine learning in industry is not images or text — it is **tables**: rows of customers, transactions, or patients, each described by a handful of columns. This case study takes the classic tabular task, **customer churn prediction** (will this subscriber leave next month?), and works through it from start to finish. We build a churn-like table in NumPy, standardize it, and train a classifier from scratch. We then find not just how accurate it is but which features drive the decision, and why raw accuracy can badly mislead you here.*

The problem

A subscription business (a telecom, a bank, a streaming service) loses revenue every time a customer cancels. Winning a customer back after they leave is far more expensive than encouraging them to stay before they go, so the business wants a model that, given what it knows about a customer today, outputs the probability they will **churn** (cancel) soon. Customers marked as high-risk get a retention offer. This is **binary classification** on tabular data, and it is the most common shape of applied machine learning in the world.

Two features of the problem shape everything that follows. First, the columns are *of different types*: some are numeric and continuous (monthly charge in dollars), some are counts (support calls), some are categorical (contract type). They must be turned into numbers on comparable scales before a network can use them. Second, churn is *rare*: typically only 15–30% of customers leave in a given window, so the classes are imbalanced. A model that simply predicts "nobody churns" can look good on accuracy while being useless.

The data

A real churn table has one row per customer and columns such as tenure (months as a customer), monthly charges, number of support calls, contract type, payment method, and a churn label of yes/no. The well-known Telco dataset is the standard teaching example. Before

modeling, two transformations are essential:

One-hot encoding turns a categorical column with no natural ordering into several 0/1 columns, one per category, so the model never invents a false "month-to-month < one-year < two-year" ordering. **Standardization** rescales each numeric column to mean 0 and standard deviation 1. This way a feature measured in the thousands (charges) does not quietly dominate the weighted sum over a feature measured in single digits (calls). It is the same mean-0, unit-variance method that normalization layers later apply inside the network (normalization, Chapter 11), here applied to the raw inputs. Both are shown here in pure NumPy:

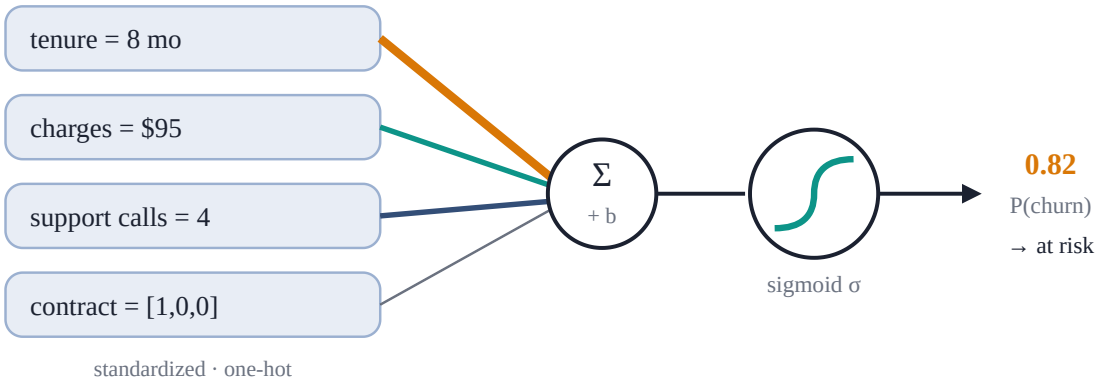
Python · NumPy ▶ runnable in the app

```
import numpy as np
# a categorical column: contract type as integer codes 0/1/2
contract = np.array([0, 2, 1, 0, 1, 2])
onehot = np.eye(3)[contract]                # 3 columns, exactly one 1 per
row
print("one-hot 'contract':\n", onehot.astype(int))
# a numeric column: standardize to mean 0, sd 1 (fit on TRAINING data only)
charges = np.array([20., 55., 95., 70., 120., 45.])
z = (charges - charges.mean()) / charges.std()
print("standardized charges:", np.round(z, 2))
print("check -> mean", round(z.mean(), 3), " sd", round(z.std(), 3))
```

One-hot encoding is fine when a categorical has a handful of levels, such as contract type or payment method. But it fails for **high-cardinality** columns. A "device model" or "zip code" field with tens of thousands of values would add tens of thousands of near-empty columns, and every level would be learned on its own, with rare ones getting almost no signal. The neural solution is an **entity embedding**: a learned lookup table that maps each category to a short dense vector (say 8–32 numbers), trained end to end with the rest of the network. Categories that behave alike end up with nearby vectors, so the model shares statistical strength across related levels and generalizes to rare ones. The network learns a *representation* of the category instead of being given an orthogonal indicator (representations, Chapter 5). Embeddings are what let neural nets use the high-cardinality categorical features where classical tabular methods struggle.

An important detail: the mean and standard deviation used for standardization must be computed on the *training set only* and then applied to the test set. Using the whole dataset's statistics leaks information about the test rows into training and makes your reported accuracy too high. This is a data-leakage trap that data-centric practice is built to prevent (data-centric machine learning, Chapter 26).

We cannot load the Telco file here, so the runnable code below creates a substitute with a known logistic relationship: churn probability rises as tenure falls, charges rise, and support calls increase, plus noise. Because we built the relationship, we know the right answer and can check whether the trained model recovers it.



A churn classifier as one sigmoid neuron. Standardized, one-hot features feed weighted connections. Edge color shows the learned sign (amber = raises churn as it falls, here short tenure; teal/blue = raises churn as it grows), edge thickness the magnitude. The sigmoid turns the evidence total into a probability, and a threshold turns that into an action.

The model

The natural first model is **logistic regression**, which, as Chapter 1 notes, is exactly a single sigmoid neuron. It computes a weighted sum of the standardized features, passes it through a sigmoid, and reads the result as P(churn):

$$\text{P(churn)} = \sigma(\mathbf{w}\cdot\mathbf{x} + b), \sigma(t) = \frac{1}{1 + e^{-t}}$$

Training minimizes **binary cross-entropy** (derived from maximum likelihood in Chapter 6). Its gradient with respect to the logit is once again the clean residual **P − y** (predicted probability minus the 0/1 label), which we descend by gradient descent (Chapter 7). A great advantage of logistic regression on standardized features is *interpretability*: because every feature has the same unit (one standard deviation), the learned weights are directly comparable, and their sizes tell you which features the model actually relies on. When a linear boundary is not enough, the same method extends to a small **MLP**, a couple of hidden ReLU layers, which the PyTorch listing uses.

When a straight-line boundary is too rigid, we replace the single neuron with a small **multilayer perceptron**: two or three hidden ReLU layers feeding the same sigmoid output. This helps because churn may depend on *interactions*, for example a high bill only mattering for short-tenure customers. Two now-standard techniques help such a network work well on tabular data. **Batch normalization** (Chapter 11) standardizes each hidden layer's pre-activations across the mini-batch. This stabilizes training and lets us use a larger learning rate. It is the same mean-0, unit-variance idea we applied to the raw inputs, now applied layer by layer. **Dropout** (Chapter 12) randomly zeros a fraction p of hidden units each step, so the network cannot rely on any single feature interaction and acts like a low-cost ensemble. Together with weight decay and early stopping, it is the difference between an MLP that memorizes 800 training rows and one that generalizes to next month's customers.

Class imbalance changes how we train and judge the model. With only ~25% churners, a network minimizing plain cross-entropy tends toward the majority class. There are two standard fixes. First, weight the rare class more heavily in the loss (a positive-class weight). Second, at decision time, do not simply threshold at 0.5. Instead choose the cut that balances the business cost of a missed churner against the cost of a wasted retention offer. And never report accuracy alone: use **precision** (of those we flagged, how many really churned) and **recall** (of those who churned, how many we caught).

Handling class imbalance

Because churn is rare, the average loss is dominated by the majority class, and a network can lower its cross-entropy simply by predicting "stays" confidently everywhere. Three groups of fixes, drawn from the data-centric toolkit (data-centric machine learning, Chapter 26), address this in different ways.

Re-weight the loss. Multiply each churner's loss term by a factor $w_+ > 1$ (and each stayer's by 1), so a missed churner hurts as much as several missed stayers. The natural choice is the inverse class frequency, $w_+ = n_- / n_+$ (with 25% churn that is about 3), which is exactly the `pos_weight` passed to `BCEWithLogitsLoss` in the listing below. **Focal loss** goes further. It down-weights the examples the model already gets right, so training focuses on the hard, usually minority, cases:

$$\mathscr{L}_{\text{focal}} = -(1 - p_t)^{\gamma} \cdot \log p_t, \; p_t = P(\text{correct class})$$

The focusing parameter γ (typically 2) shrinks the loss on confident correct predictions toward zero. At $\gamma = 0$ it reduces to ordinary cross-entropy.

Resample the data. Oversample the minority (duplicate churners, or create new ones by interpolating between nearby churn rows, as SMOTE does) or undersample the majority, so the network sees balanced data. This is a data-level rather than loss-level intervention (Chapter 26), and it can be combined with re-weighting.

Move the threshold. The three methods above change what the model *learns*. The simplest one changes only how we *act* on it. Lower the decision cut below 0.5 and recall climbs while precision falls, showing the trade-off we make explicit under evaluation. One warning applies to all of them: re-weighting, focal loss, and oversampling all distort the base rate the model sees, so its output probabilities come out miscalibrated. We return to this point shortly.

In code

First, look at the data: build the table and inspect the class balance and the mean of each feature within the churned and retained groups. If the features carry useful information, the group means should differ:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)
n = 1000
tenure  = rng.uniform(0, 60, n)          # months as a customer
charges = rng.uniform(20, 120, n)        # monthly bill, $
calls   = rng.poisson(1.5, n)            # support calls last quarter

# ground-truth log-odds of churn (short tenure, high bill, many calls -> churn)
z = -1.8 - 0.07*tenure + 0.02*charges + 0.55*calls + rng.normal(0, 0.5, n)
y = (rng.rand(n) < 1/(1+np.exp(-z))).astype(float)
X = np.stack([tenure, charges, calls], axis=1)

print(f"customers: {n}   churned: {int(y.sum())}  ({y.mean()*100:.1f}%)")
print(f"majority-class baseline accuracy: {max(y.mean(), 1-y.mean()):.3f}\n")
print(f"{'group':8s}{'tenure':>9s}{'charges':>9s}{'calls':>8s}")
for lab, name in [(0, 'stayed'), (1, 'churned')]:
    m = X[y == lab].mean(0)
    print(f"{name:8s}{m[0]:9.1f}{m[1]:9.1f}{m[2]:8.2f}")
```

Now train the classifier from scratch. We split off a test set, standardize using training statistics only, then run logistic regression: forward sigmoid, residual gradient `P - y`, gradient step. We print the rising accuracy, the true test metrics (accuracy, precision, recall), and, as the key result, the learned weights that reveal which features matter:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)
n = 1000
tenure  = rng.uniform(0, 60, n)
charges = rng.uniform(20, 120, n)
calls   = rng.poisson(1.5, n)
z = -1.8 - 0.07*tenure + 0.02*charges + 0.55*calls + rng.normal(0, 0.5, n)
y = (rng.rand(n) < 1/(1+np.exp(-z))).astype(float)
X = np.stack([tenure, charges, calls], axis=1)

# split, then STANDARDIZE using training statistics only (no leakage)
Xtr, ytr, Xte, yte = X[:800], y[:800], X[800:], y[800:]
mu, sd = Xtr.mean(0), Xtr.std(0)
```

```
Xtr = (Xtr - mu) / sd
Xte = (Xte - mu) / sd

# logistic regression from scratch (one sigmoid neuron, BCE loss)
w = np.zeros(3); b = 0.0
for ep in range(1, 401):
    p  = 1 / (1 + np.exp(-(Xtr @ w + b)))
    g  = p - ytr                                # gradient of BCE wrt the
logit
    w -= 0.3 * (Xtr.T @ g) / len(ytr)
    b -= 0.3 * g.mean()
    if ep in (1, 50, 100, 200, 400):
        print(f"epoch {ep:3d}   train acc {((p>0.5)==ytr).mean():.3f}")

pt   = 1 / (1 + np.exp(-(Xte @ w + b)))
pred = pt > 0.5
tp = np.sum(pred & (yte==1)); fp = np.sum(pred & (yte==0)); fn =
np.sum(~pred & (yte==1))
print(f"\ntest accuracy {(pred==yte).mean():.3f}   (majority baseline
{max(yte.mean(),1-yte.mean()):.3f})")
print(f"precision {tp/(tp+fp):.3f}   recall {tp/(tp+fn):.3f}")
print("\nstandardized weights (effect of a 1-sd change):")
for name, wi in sorted(zip(['tenure','charges','calls'], w), key=lambda t: -
abs(t[1])):
    print(f"  {name:8s} {wi:+.3f}")
```

In production you would use a small MLP (or, very often on tables, gradient-boosted trees). Here is the standard PyTorch tabular pipeline: a two-hidden-layer network with **batch normalization** (Chapter 11) to steady each layer's input distribution, **dropout** and **weight decay** for regularization (Chapter 12), a positive-class weight for the imbalance, a train/validation split, and **early stopping**. Early stopping halts training when validation loss stops improving, and it is the simplest and most effective regularizer of all:

PyTorch copy → Colab

```
import torch, torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader, random_split

# X_std: (N, d) standardized features; y_lbl: (N,) labels in {0., 1.}
X = torch.tensor(X_std, dtype=torch.float32)
y = torch.tensor(y_lbl, dtype=torch.float32)
ds = TensorDataset(X, y)
```

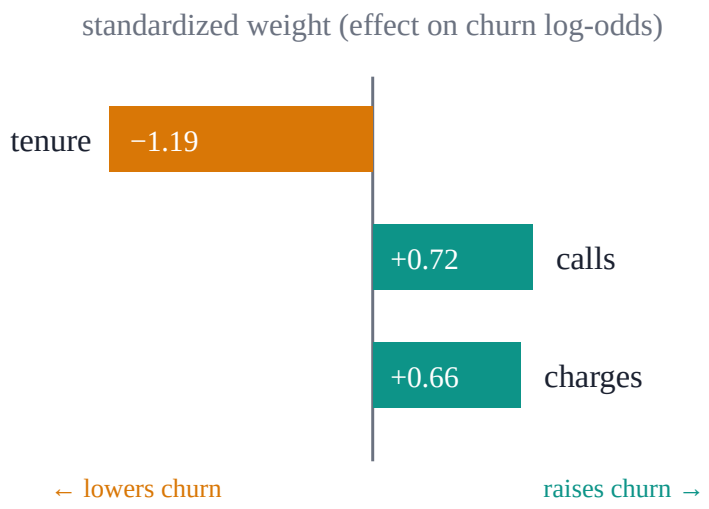
```
n_val = int(0.2 * len(ds))
train_ds, val_ds = random_split(ds, [len(ds) - n_val, n_val])
train_dl = DataLoader(train_ds, batch_size=64, shuffle=True)
val_dl   = DataLoader(val_ds,   batch_size=256)

net = nn.Sequential(
    nn.Linear(X.shape[1], 32), nn.BatchNorm1d(32), nn.ReLU(),
nn.Dropout(0.2),
    nn.Linear(32, 16),         nn.BatchNorm1d(16), nn.ReLU(),
nn.Dropout(0.2),
    nn.Linear(16, 1))                                  # a single logit

pos_weight = torch.tensor([(y == 0).sum() / (y == 1).sum()])   # up-weight
rare churners
loss_fn = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
opt = torch.optim.Adam(net.parameters(), lr=1e-3, weight_decay=1e-4)

best, wait, patience = float("inf"), 0, 5
for epoch in range(100):
    net.train()
    for xb, yb in train_dl:
        opt.zero_grad()
        loss_fn(net(xb).squeeze(1), yb).backward()
        opt.step()
    net.eval(); vloss = 0.0
    with torch.no_grad():
        for xb, yb in val_dl:
            vloss += loss_fn(net(xb).squeeze(1), yb).item() * len(xb)
    vloss /= len(val_ds)
    if vloss < best - 1e-4:
        best, wait = vloss, 0
        torch.save(net.state_dict(), "best.pt")        # checkpoint the best
model
    else:
        wait += 1
        if wait >= patience:
            print(f"early stop at epoch {epoch}, best val loss {best:.4f}")
            break
```

Results & interpretation

The exploration cell reports 254 churners out of 1000 (25.4%), a majority-class baseline of 0.746, and clearly separated group means: churners average 19.0 months of tenure versus 33.4 for those who stay, 82.2 dollars in charges versus 67.4, and 1.93 support calls versus 1.33. The signal is real and matches how the ground truth was built.

Training accuracy rises from 0.748 to about 0.800 and settles there. On the test set the model scores **0.790 accuracy, above the 0.746 baseline but only barely**, and this is the case study's central lesson. The precision is 0.656 and the recall only **0.404**: the model catches fewer than half of the customers who actually churn. Accuracy hid that failure because predicting "stays" for everyone is already right three-quarters of the time. For a churn team, recall is often what matters, because a missed churner is lost revenue. So you would lower the decision threshold below 0.5, or train with the positive-class weight in the PyTorch version, trading some precision (more wasted offers) for much higher recall.

The learned weights clearly recover the truth. On standardized features they come out roughly tenure **−1.19**, calls **+0.72**, charges **+0.66**: tenure is the strongest driver and it is negative, so longer-tenured customers are far less likely to leave, while high charges and frequent support calls both push toward churn. This is the answer the business actually wants: not just a risk score, but the knowledge that *early-tenure, high-bill, high-friction customers* are the ones to protect.



The learned weights on standardized features. Because every feature is in units of one standard deviation, bar length is directly comparable: tenure (amber, negative) is the dominant protective factor. Support calls and monthly charges (teal, positive) push customers toward leaving.

Evaluation beyond accuracy

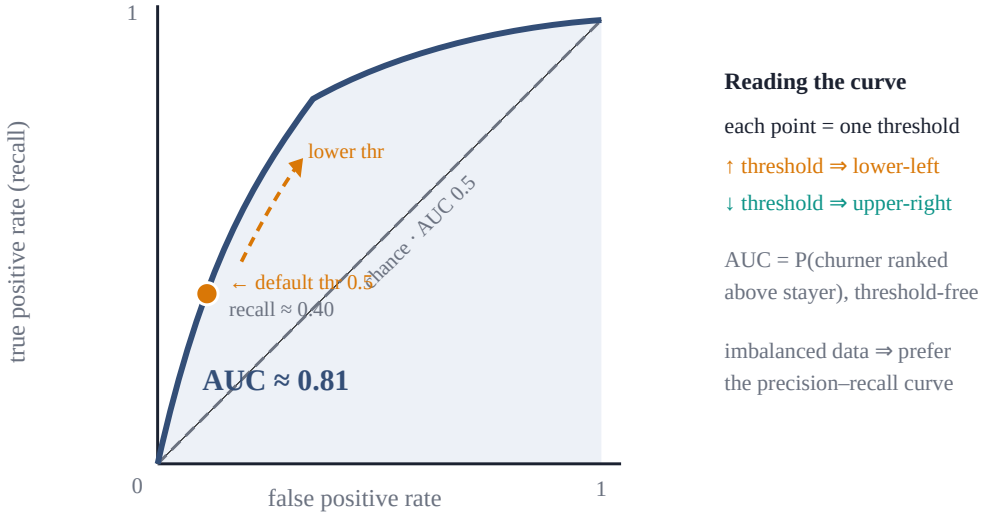
The results section already showed accuracy hiding a weak recall. A serious tabular project reports a small panel of metrics instead. **Precision** and **recall** answer two connected questions: of those we flagged, how many churned; of those who churned, how many we caught. The **F1 score**, their harmonic mean F1 = 2·P·R ⁄ (P + R), combines the pair into one number. Unlike accuracy, it stays honest under imbalance because it ignores the large number of correctly-predicted stayers. But precision, recall, and F1 all depend on a chosen threshold. To judge the model *independently* of that choice, sweep the threshold and read the **ROC curve** (true-positive rate against false-positive rate). Its area, the **ROC-AUC**, equals the probability that the model scores a random churner above a random stayer: 0.5 is a coin flip, 1.0 a perfect ranking. The runnable cell computes precision, recall, and F1 across thresholds and the AUC by that ranking rule:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(7)
n = 60
y = (rng.rand(n) < 0.35).astype(int)                    # true churn
labels
# an imperfect model's scores in [0,1]: churners bumped up, but heavy
```

```
overlap
scores = np.clip(0.55*rng.rand(n) + 0.20*y + 0.20*rng.randn(n) + 0.20, 0.02,
0.98)

def prf(y, pred):
    tp = np.sum((pred==1)&(y==1)); fp = np.sum((pred==1)&(y==0)); fn =
np.sum((pred==0)&(y==1))
    prec = tp/(tp+fp) if (tp+fp)>0 else 0.0
    rec  = tp/(tp+fn) if (tp+fn)>0 else 0.0
    f1   = 2*prec*rec/(prec+rec) if (prec+rec)>0 else 0.0
    return prec, rec, f1

print(f"churners: {int(y.sum())} of {n}  ({y.mean()*100:.0f}%)")
print("thr    prec   rec    F1")
best_thr, best_f1 = 0.0, -1.0
for thr in np.arange(0.30, 0.76, 0.05):
    pr, rc, f1 = prf(y, (scores>=thr).astype(int))
    if f1 > best_f1: best_thr, best_f1 = thr, f1
    print(f"{thr:.2f}   {pr:.3f}  {rc:.3f}  {f1:.3f}")
print(f"best-F1 threshold = {best_thr:.2f}   (F1 = {best_f1:.3f})")

# ROC-AUC = P(score(random churner) > score(random stayer)); ties count 1/2
pos, neg = scores[y==1], scores[y==0]
wins = (pos[:,None] > neg[None,:]).sum() + 0.5*(pos[:,None] ==
neg[None,:]).sum()
print(f"ROC-AUC = {wins/(pos.size*neg.size):.3f}   (0.5 coin flip, 1.0
perfect)")
```

The sweep makes the trade-off clear: at a low cut the model flags nearly everyone (recall 1.0, precision near the 25% base rate), and as the threshold rises precision improves while recall falls, with F1 peaking in between (here around 0.60). The AUC of 0.81 says the ranking is truly useful even though no single threshold is excellent. It is the number to track when you tune the model, because it does not move when you later re-choose the operating point. Under heavy imbalance the **precision–recall curve**, which ignores true negatives entirely, is often the more honest alternative to the ROC.



An ROC curve for the churn model. Each point is one decision threshold. Sliding the cut moves you along the curve, trading recall against false alarms. The shaded area is the ROC-AUC (≈ 0.81), the probability the model ranks a random churner above a random stayer, a threshold-free summary of ranking quality. The amber dot marks the default 0.5 cut, whose low recall (≈ 0.40) is the failure the results section flagged. Lowering the threshold climbs up-and-right toward higher recall.

Which point on the curve you deploy is a **business decision, not a statistical one**: it depends on the relative cost of a missed churner versus a wasted retention offer, and you pick the threshold that minimizes expected cost.

Worked example — choosing the operating threshold by cost

Take a month of 1000 customers at the 25% base rate (250 churners) and suppose a missed churner costs $200 in lost revenue while every retention offer costs $20 whether or not it was needed. Scale the sweep above to this population and calculate the cost of three thresholds:

1. Threshold 0.30 — recall 1.00, precision 0.30: TP = 250, FN = 0; offers = TP⁄precision ≈ 833. Cost = 0·$200 + 833·$20 ≈ **$16,700**.
2. Threshold 0.50 — recall 0.80, precision 0.44: TP = 200, FN = 50; offers ≈ 455. Cost = 50·$200 + 455·$20 ≈ **$19,100**.
3. Threshold 0.75 — recall 0.53, precision 0.57: TP = 133, FN = 117; offers ≈ 233. Cost = 117·$200 + 233·$20 ≈ **$28,000**.

What the numbers mean: because a false negative is 10× as expensive as a wasted offer, the cost-minimizing cut is the *low* one, so flag aggressively. If offers had been costly relative to lost revenue, the minimum would move the other way. The 0.5 default is almost never optimal. The right threshold comes from the cost matrix, not the shape of the sigmoid.

Calibration. A probability is only trustworthy if it is *calibrated*: among customers scored 0.30, about 30% should truly churn. The imbalance fixes above deliberately distort the base rate, so a class-weighted or focal-trained model usually reports inflated probabilities. Its ranking (and thus AUC) stays valid, but the raw numbers no longer mean what they say. This matters the moment a score feeds a cost calculation like the one above. Diagnose it with a **reliability diagram** (bucket predictions, then plot predicted against observed churn frequency) and correct it with Platt or temperature scaling fit on a held-out set. This is the subject of uncertainty and calibration (Chapter 24).

Model inspection & feature importance

A churn model is only useful if the business can see *why* it flags a customer. Logistic regression gives you this for free: on standardized features the weights are directly comparable, and we already read them off, with tenure the dominant protective factor and charges and calls pushing toward churn. For any model, including the MLP, a model-agnostic alternative is **permutation importance**: shuffle one feature's column across the test set, breaking its link to the label, and measure how much a metric (here ROC-AUC) drops. A feature the model relies on will hurt when scrambled. An ignored feature will not. The runnable cell retrains the logistic model and permutes each feature in turn:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)
n = 1000
tenure=rng.uniform(0,60,n); charges=rng.uniform(20,120,n);
calls=rng.poisson(1.5,n)
```

```
z=-1.8-0.07*tenure+0.02*charges+0.55*calls+rng.normal(0,0.5,n)
y=(rng.rand(n)<1/(1+np.exp(-z))).astype(float)
X=np.stack([tenure,charges,calls],1)
Xtr,ytr,Xte,yte=X[:800],y[:800],X[800:],y[800:]
mu,sd=Xtr.mean(0),Xtr.std(0); Xtr=(Xtr-mu)/sd; Xte=(Xte-mu)/sd
w=np.zeros(3); b=0.0
for _ in range(400):
    p=1/(1+np.exp(-(Xtr@w+b))); g=p-ytr
    w-=0.3*(Xtr.T@g)/len(ytr); b-=0.3*g.mean()

def auc(sc, lab):                                   # AUC by the ranking
rule
    pos,neg=sc[lab==1],sc[lab==0]
    return ((pos[:,None]>neg[None,:]).sum()+0.5*
(pos[:,None]==neg[None,:]).sum())/(pos.size*neg.size)

base=auc(1/(1+np.exp(-(Xte@w+b))), yte)
print(f"test ROC-AUC (all features) = {base:.3f}")
names=['tenure','charges','calls']; perm=np.random.RandomState(42)
for j in range(3):
    Xp=Xte.copy(); Xp[:,j]=perm.permutation(Xp[:,j])  # break feature j's
link to y
    drop=base-auc(1/(1+np.exp(-(Xp@w+b))), yte)
    print(f"  shuffle {names[j]:8s}: AUC drop = {drop:+.3f}")
```

The AUC falls most when tenure is scrambled (−0.17), then support calls (−0.11), then charges (−0.07). This is the same ranking the weights gave, now derived without looking inside the model, so it transfers unchanged to a network or a gradient-boosted tree. For per-customer explanations rather than global ones ("this account was flagged chiefly for its six support calls"), the same method underlies SHAP values and related attribution methods (interpretability, Chapter 28). A warning that chapter emphasizes: importance is not causation. The fact that short tenure predicts churn does not mean extending a contract prevents it. The model reports association, and intervening on it is a separate, causal question.

Assumptions & limitations — when a net is the wrong tool

On a medium-sized table of mixed-type columns (a few thousand to a few hundred thousand rows, mixed numeric and low-cardinality categorical), a **gradient-boosted decision-tree** ensemble (XGBoost, LightGBM, CatBoost) usually matches or beats an MLP. It trains in seconds to minutes, needs almost no preprocessing (no standardization, native categorical handling), and easily ignores uninformative columns and outliers. Trees split on thresholds, which fits the axis-aligned, non-smooth structure of real tabular data better than the smooth functions an MLP prefers (Grinsztajn et al., 2022). A neural network is worth using when the table is very large, when a high-cardinality categorical benefits from a learned **entity embedding** (representations, Chapter 5) that a tree cannot share across categories, or when tabular features must be fused with text, images, or sequences inside one differentiable model. The honest tabular pipeline fits a gradient-boosted baseline first and uses a network only when it beats that baseline. Meanwhile every preprocessing, imbalance, evaluation, and calibration habit above transfers to both.

What to try next

Open the **Playground** and train a small network on a two-feature classification problem. It is the same sigmoid-and-cross-entropy engine, with the decision boundary drawn for you as it learns. Then in the runnable cell, experiment: lower the decision threshold from 0.5 (change `pt > 0.5`) and watch recall climb while precision falls; add L2 regularization to the weight update and watch the weights shrink; or add a fourth feature with zero true effect and confirm the model correctly learns a near-zero weight for it. To see overfitting and the regularizers that reduce it (dropout, weight decay, early stopping), visit the **Overfitting lab**.

In the real world

Churn models run quietly inside almost every subscription business. Telecoms score their entire customer base monthly and route high-risk accounts to retention teams with targeted discounts. Banks and insurers model customer loss the same way. SaaS companies predict seat cancellations to trigger customer-success outreach. An honest warning that practitioners learn fast: on tabular data like this, gradient-boosted decision trees (XGBoost, LightGBM) frequently match or beat neural networks and train faster, so a strong tabular pipeline usually tries both. The neural-network skills here still transfer directly: the preprocessing, the loss, the imbalance handling, and above all the discipline of judging a model by precision and recall rather than a single accuracy number.

Exercises

1. On a 200-customer test set a churn model produces TP = 21, FP = 11, FN = 32, TN = 136. Compute accuracy, precision, and recall, and explain why accuracy looks fine while the model is weak. MATH

2. Add L2 regularization to the from-scratch logistic loop (add $\lambda \mathbf{w}$ to the weight gradient) and observe how the weights change as λ grows. CODE

3. Why must the standardization mean and standard deviation be computed on the training set only, then reused on the test set, rather than on the full dataset? CONCEPT

4. The model's recall is only about 0.40 at a 0.5 threshold. Describe two different ways to raise recall, and the cost of each one. CONCEPT

5. Six customers get churn scores with these true labels: (0.90, churn), (0.80, stay), (0.60, churn), (0.55, churn), (0.40, stay), (0.20, stay). (a) At threshold 0.5, compute precision, recall, and F1. (b) Compute the ROC-AUC by the ranking rule, the fraction of churner–stayer pairs the model orders correctly. MATH

6. A churn model has a categorical feature "device model" with 12,000 distinct values. Contrast one-hot encoding with an entity embedding for this feature: give the input width and parameter cost of each, the generalization argument for embeddings, and one risk they carry. Connect it to representations (Chapter 5). CONCEPT

CASE STUDY

Case study: text sentiment analysis

> *Language is the hardest of the three data types in these case studies. This is because meaning lives not in any single word but in which words appear and in what order. This study looks at the entry point to natural language processing — **sentiment analysis**, deciding whether a review is positive or negative. It builds this idea twice. First, a clear bag-of-words classifier written from scratch in NumPy that you can run and read word by word. Second, the modern embedding-based pipeline in PyTorch that scales to real reviews and forms the bridge toward the transformers of chapter 16.*

The problem

Given the text of a product or film review, output whether the writer liked it. This is **binary text classification**. It is difficult because the input is not a fixed vector of numbers, but a string of words with variable length. "A great and brilliant film" and "not great, actually quite terrible" share the word *great* but mean opposite things. To use any of the tools from this manual, we must first turn text into numbers. This step is called **vectorization**, and the choices we make there decide how much of the meaning is kept.

The standard benchmark is the **IMDB movie review dataset**: 50,000 reviews (25,000 for training, 25,000 for testing), each labeled positive or negative, collected by Maas et al. (2011). It is to text what MNIST is to images: small enough to work with quickly, standard enough to compare against, and rich enough that word order really matters.

The data

Turning words into numbers happens in two stages. **Tokenization** splits a string into a list of tokens (here, we simply make the text lowercase and split it on spaces. Real tokenizers handle punctuation, contractions, and sub-words). Then we build a **vocabulary**. This is the set of known tokens, and each one is given an integer id. A review becomes a sequence of ids.

The simplest way to get a fixed-length vector from those ids is **bag-of-words** (BoW): a vector as long as the vocabulary. Its entry *i* counts how often word *i* appears. Bag-of-words ignores order completely, so "dog bites man" and "man bites dog" get identical vectors. But for sentiment it is a surprisingly strong start, because the simple *presence* of "brilliant" or "awful" is highly predictive. This cell tokenizes the text, builds a vocabulary, and shows both the id sequence and the bag-of-words vector for one review:

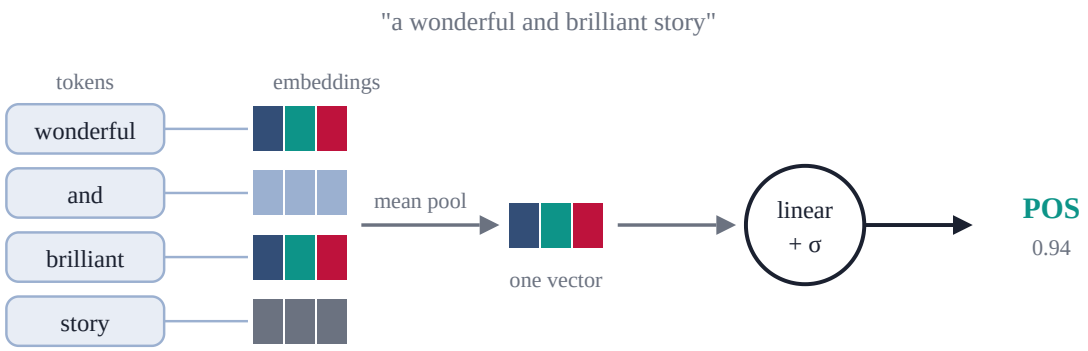
Python · NumPy ▶ runnable in the app

```python
import numpy as np

reviews = [
 "a great and brilliant movie i loved it",
 "a boring and terrible waste of time",
]
# tokenize (lowercase split) and build a vocabulary over the corpus
tokens = [r.split() for r in reviews]
vocab  = sorted(set(w for t in tokens for w in t))
idx    = {w: i for i, w in enumerate(vocab)}
print(f"vocabulary ({len(vocab)} words): {vocab}\n")

s   = reviews[0]
ids = [idx[w] for w in s.split()]
print(f"review    : {s}")
print(f"token ids : {ids}")

bow = np.zeros(len(vocab))                # bag-of-words: count each vocab
word
for w in s.split():
    bow[idx[w]] += 1
print(f"bag-of-words vector : {bow.astype(int)}")
```

Two practical notes. Real vocabularies have a size limit (say the 20,000 most frequent words) and reserve special ids for *unknown* words and for *padding* (so a batch of reviews with different lengths can be stacked into a rectangle). Also, BoW vectors are very large and mostly zero, with one nonzero value per distinct word. This is why the industrial pipeline replaces them with dense *embeddings*, shown below.

Two design choices in this stage quietly decide how much signal is kept. **Tokenization** looks simple on clean toy text, but this is where real text collections get messy: should "don't" be one token or two ("do" and "n't")? Is "state-of-the-art" one token or three? Lowercasing merges "Great" and "great", but it also turns "Apple" the company into "apple" the fruit. Removing punctuation throws away the "!" that so often signals emphasis. And emoji, hashtags, and URLs need rules of their own. Classic pipelines answer this with hand-written regular expressions. Modern systems instead learn a **subword** vocabulary. This is byte-pair encoding, as used by the transformers of chapter 17, and it splits rare words into reusable pieces ("unbelievably" → "un", "believ", "ably"), so no word is ever truly unseen and the vocabulary stays a manageable size.

Building the **vocabulary** is the second choice. We count every token across the training data, sort them by frequency, and keep the top V (say 20,000). We do this because word frequencies follow a steep Zipfian curve: a few words are very common, and a long tail of words appears once or twice, too rarely to learn from. Every token outside the kept set is **out-of-vocabulary** (OOV). Silently dropping OOV words, as the toy code above does, throws away information. For a review made entirely of rare words, it leaves nothing at all. The standard fix maps every unknown token to one reserved id, `<unk>`, so the model at least knows "an unknown word occurred here" and can learn an average sentiment for that group. A second reserved id, `<pad>`, fills out short reviews so a batch of different lengths stacks into a rectangle. It is paired with a *mask* that tells the network which positions are real. This is the same masking that the attention models of chapter 16 rely on.



The embedding pipeline. Each token is looked up in an embedding table to get a dense vector. The vectors are averaged (mean pooling) into a single fixed-length vector, no matter how long the review is. A linear layer plus a sigmoid then gives the sentiment. Gray bars mark a low-content word ("and") whose embedding the network learns to keep near zero.

A ladder of representations

Everything after this depends on one choice: how a review of variable length becomes a fixed vector that the classifier can read. That choice is like a ladder. Each rung keeps more of the meaning, but it needs more complex methods.

The bottom rung is **bag-of-words**, shown already above. It is a vector as long as the vocabulary that holds word counts. An improvement, **TF-IDF**, gives less weight to words that appear in nearly every review ("movie", "the") and more weight to rare, discriminative ones. This often lifts a linear classifier by a few points at no extra cost. But both are sparse, very large, and blind to order. They also treat every word as a separate axis: "great" and "wonderful" are as unrelated as "great" and "aardvark".

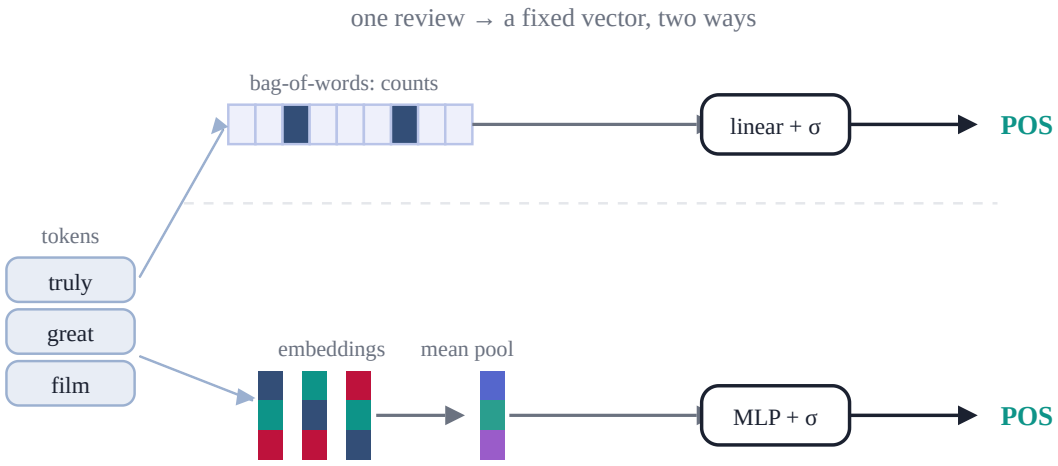
The next rung replaces counts with **learned word embeddings** (chapter 5): each word id points to a short dense vector (say 64 numbers) in a table that is trained with the rest of the model. This way, words used in similar contexts settle near one another, and the classifier can generalize from a common word to a rare synonym it barely saw. To turn a review's list of embedding vectors into one fixed vector, we **mean-pool**, that is, average them. This is a cheap operation. Like bag-of-words, it ignores order, but now it averages *meaning* rather than counting symbols. This cell builds a tiny embedding table, trains it together with a logistic layer by backpropagation on the toy reviews, and reports accuracy:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.random.seed(0)

train = [                                          # (review, label)  1 = positive
 ("a great and brilliant movie i loved it", 1),("wonderful acting and a great story", 1),
 ("brilliant fun and truly wonderful", 1),("i loved the great cast so good", 1),
 ("a boring and terrible waste of time", 0),("terrible dull and awful acting", 0),
 ("so boring i hated the dull story", 0),("awful terrible and a total waste", 0)]
vocab = sorted(set(w for s, _ in train for w in s.split()))
idx   = {w: i for i, w in enumerate(vocab)}
y     = np.array([l for _, l in train], float)

dim = 3
E   = 0.01*np.random.randn(len(vocab), dim)      # embedding table  (V x dim)
w   = np.zeros(dim); b = 0.0                     # logistic layer on the mean vector

def meanvec(s):
    ids = [idx[t] for t in s.split() if t in idx]
    return E[ids].mean(0), ids                   # mean-pool the word embeddings

for ep in range(4000):
    gE = np.zeros_like(E); gw = np.zeros(dim); gb = 0.0
    for (s, _), t in zip(train, y):
        r, ids = meanvec(s)
        d = 1/(1+np.exp(-(r@w+b))) - t           # binary cross-entropy gradient
        gw += d*r; gb += d
        for j in ids: gE[j] += d*w/len(ids)      # split the mean's gradient
    w -= 0.5*gw/len(y); b -= 0.5*gb/len(y); E -= 0.5*gE/len(y)
```

```
pred = np.array([1 if meanvec(s)[0]@w + b > 0 else 0 for s, _ in train])
print("averaged-embedding train accuracy =", (pred == y).mean())   #
expected: 1.0
```

With just three numbers per word, the averaged-embedding model separates the eight reviews perfectly. And, unlike bag-of-words, it places "great", "wonderful", and "brilliant" near one another in the learned space, so a new review using any of them lands in the same positive region. The top rungs of the ladder keep word *order* too. A small **LSTM** (chapter 15) reads the embeddings left to right and carries a hidden state, so "not" can flip what follows. A **transformer** encoder (chapter 16) lets every word attend to every other. And the modern default is to skip training from scratch and **fine-tune a pretrained transformer** such as BERT (chapters 17 and 22). This inherits a model of language learned from billions of words before it ever sees a labeled review. Every rung has the same shape — embed, (contextualize,) pool, classify — but the middle gets more powerful.



Two rungs of the representation ladder for one review. Top: bag-of-words turns the tokens into a long sparse count vector, and a single linear layer reads the sentiment. This is clear but blind to order. Bottom: each token is looked up in a small dense embedding table (chapter 5), the vectors are mean-pooled into one, and an MLP classifies. Both average away word order. Keeping it is the job of the recurrent (chapter 15) and attention (chapter 16) models.

The model

The from-scratch model is **logistic regression over bag-of-words**: one weight per vocabulary word, a bias, a sigmoid. It is again a single neuron (chapter 1), and its weights are very easy to interpret. A large positive weight marks a word that the model reads as positive sentiment, and a large negative weight marks a negative one. Training is binary cross-entropy by gradient descent, exactly as in the tabular study.

The industrial model replaces the sparse BoW vector with a learned **embedding**: each word id points to a small dense vector (say 64 numbers) in a lookup table that is trained along with everything else, so that words used in similar ways drift to nearby vectors. To classify a whole review, we **mean-pool**: we average the embeddings of its words into one fixed-length vector, then apply a linear classifier. This "embed and average" model is cheap, strong, and the basic idea behind everything that follows.

Both models are trained the same way, and it is worth naming the parts. The classifier outputs a single **logit** z. A sigmoid turns it into a probability $\hat{y} = \sigma(z)$. We fit it by minimizing **binary cross-entropy** $L = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$, averaged over the batch (chapter 6). Its gradient at the logit is the same clean $\hat{y} - y$ that drove the tabular study. The from-scratch model reduces this loss by plain gradient descent (chapter 7). The embedding model uses **Adam** (chapter 9) at a learning rate near 10^{-3}. It trains the embedding table, the hidden layer, and the output weights together by backpropagation (chapter 8). It also uses a little dropout and weight decay (chapter 12) to stop the many embedding parameters from simply memorizing the training reviews. A few passes over the data, called *epochs*, are enough on a dataset the size of IMDB.

Its weakness is the same as BoW's: averaging throws away order, so it cannot tell "good, not bad" from "bad, not good". To fix this, we must let each word's representation depend on its neighbors. A **recurrent network** (chapter 15) does this by reading left to right and carrying a hidden state. A **transformer** (chapter 16) does it with self-attention, which lets every word look directly at every other. At heart, both are upgrades to the pooling step in this same pipeline. This is why "embed, contextualize, pool, classify" describes models from this toy up to large language models.

In code

Here is the whole from-scratch sentiment classifier: a few labeled toy reviews, a vocabulary, bag-of-words vectors, logistic regression trained by hand, and then predictions on *unseen* sentences plus the words the model found most positive and negative. Press **Run**:

Python · NumPy ▶ runnable in the app

```
import numpy as np

train = [                                    # (review, 1 = positive, 0 = negative)
 ("a great and brilliant movie i loved it", 1),
 ("wonderful acting and a great story", 1),
 ("brilliant fun and truly wonderful", 1),
 ("i loved the great cast so good", 1),
 ("a boring and terrible waste of time", 0),
 ("terrible dull and awful acting", 0),
 ("so boring i hated the dull story", 0),
 ("awful terrible and a total waste", 0),
]
vocab = sorted(set(w for s, _ in train for w in s.split()))
idx   = {w: i for i, w in enumerate(vocab)}
def bow(s):
    v = np.zeros(len(vocab))
    for w in s.split():
        if w in idx: v[idx[w]] += 1
    return v
X = np.array([bow(s) for s, _ in train])
y = np.array([lab for _, lab in train], dtype=float)

# logistic regression over word counts, trained from scratch
w = np.zeros(len(vocab)); b = 0.0
for ep in range(600):
    p = 1 / (1 + np.exp(-(X @ w + b)))
    g = p - y                                # binary cross-entropy gradient
    w -= 0.3 * (X.T @ g) / len(y)
    b -= 0.3 * g.mean()

print("predictions on unseen reviews:")
for s in ["a wonderful and brilliant story",
          "an awful and boring waste",
```

```
          "great acting but a boring story"]:
    p = 1 / (1 + np.exp(-(bow(s) @ w + b)))
    print(f"  {p:.2f}  {'POS' if p > 0.5 else 'NEG'}   |  {s}")

order = np.argsort(w)
print("\nmost negative words:", [vocab[i] for i in order[:3]])
print("most positive words:", [vocab[i] for i in order[-3:]][::-1])
```

The real pipeline on IMDB uses an embedding layer and mean pooling. This PyTorch listing is complete and written in the standard style. The comments mark where a tokenizer and the IMDB data plug in, and the last line shows the one-swap upgrade to an order-aware LSTM:

PyTorch copy → Colab

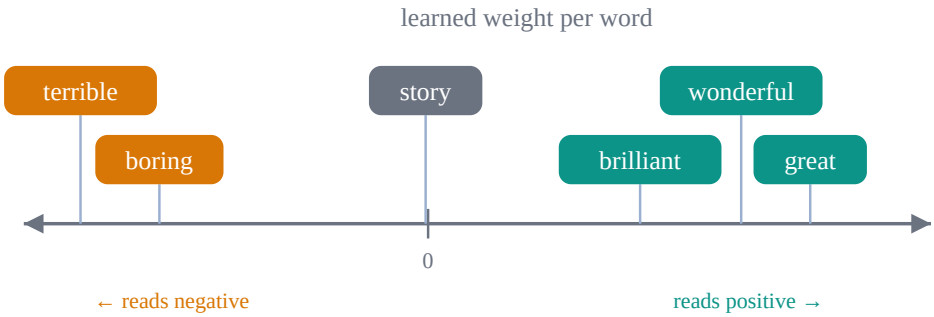
```
import torch, torch.nn as nn, torch.nn.functional as F

# Real data: the IMDB dataset, e.g.
#   from datasets import load_dataset
#   imdb = load_dataset("imdb")               # 25k train / 25k test
# Build a vocab (id 0 = <pad>, id 1 = <unk>) from training tokens, map each
# review to a list of ids, then pad each batch to a rectangle (ids, mask).

class SentimentNet(nn.Module):
    def __init__(self, vocab_size, dim=64, hidden=64, pad_idx=0):
        super().__init__()
        self.emb = nn.Embedding(vocab_size, dim, padding_idx=pad_idx)
        self.fc1 = nn.Linear(dim, hidden)
        self.fc2 = nn.Linear(hidden, 1)                  # a single logit
    def forward(self, ids, mask):                        # ids,mask: (B, L)
        e = self.emb(ids)                                # (B, L, dim)
        summed = (e * mask.unsqueeze(-1)).sum(1)         # ignore padding
        avg = summed / mask.sum(1, keepdim=True).clamp(min=1)   # mean pool
        return self.fc2(F.relu(self.fc1(avg))).squeeze(1)

net = SentimentNet(vocab_size=20000)
opt = torch.optim.Adam(net.parameters(), lr=1e-3)
loss_fn = nn.BCEWithLogitsLoss()

for epoch in range(5):
    net.train()
    for ids, mask, y in train_loader:                    # y: float in {0., 1.}
        opt.zero_grad()
```

```
        loss_fn(net(ids, mask), y).backward()
        opt.step()
    print(f"epoch {epoch} done")

# Word order matters ("good, not bad" vs "bad, not good"): swap mean pooling
# for a recurrent encoder ->  nn.LSTM(dim, hidden, batch_first=True), take
the
# final hidden state (chapter 15); or a Transformer encoder for self-
attention (chapter 16).
```

Results & interpretation

Despite training on just eight toy reviews, the from-scratch model generalizes well to sentences it never saw. "A wonderful and brilliant story" scores **0.97 → POS**. "an awful and boring waste" scores **0.02 → NEG**. Both are confident and correct, driven entirely by the sentiment words they contain. The third test is the one that teaches us the most: "great acting but a boring story" scores **0.74 → POS**. It contains both a strong positive ("great") and a strong negative ("boring"), and because bag-of-words simply *adds* their weights, the slightly heavier positive one wins. A human reads the "but" as tipping the sentence toward negative. BoW cannot do this, because it never saw the word order. That single example is the whole reason for embeddings, recurrence, and attention.

The model also explains itself. When we sort the learned weights, the **most negative words are "terrible", "dull", "boring"** and the **most positive are "great", "wonderful", "brilliant"**. With no dictionary and no supervision beyond eight labels, the classifier has discovered which words carry sentiment. This transparency is a real strength of linear text models. It is one reason they remain a strong, easy-to-debug baseline before we reach for deep networks.



The learned weights arranged on a sentiment axis (positions reflect the model's actual weights). With only eight labeled reviews, logistic regression discovers a signed vocabulary: "terrible" and "boring" pull negative, "great", "wonderful" and "brilliant" pull positive, and structural words like "story" land near zero.

Evaluation

Confident predictions on a few sentences are encouraging, but they are not evidence. To know whether the classifier works, we score it on a held-out **test set** that it never trained on, and we report numbers, not stories. The simplest summary is **accuracy**, the fraction of reviews labeled correctly. But accuracy hides *which* mistakes are made, and on skewed data it can look high while the model is useless (a filter that always guesses "positive" scores 90% when 90% of reviews are positive, yet has learned nothing). So we break the four possible outcomes of a binary decision into a **confusion matrix** and read three sharper numbers from it:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad \text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \qquad \text{F1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

TP, TN	true positives / negatives — reviews correctly called positive / negative
FP	false positives — negative reviews wrongly called positive (a false alarm)
FN	false negatives — positive reviews the model missed and called negative
precision	of the reviews it called positive, the fraction that truly were — it punishes false alarms
recall	of the truly positive reviews, the fraction it caught — it punishes misses
F1	the harmonic mean of precision and recall, high only when *both* are — one number to rank models

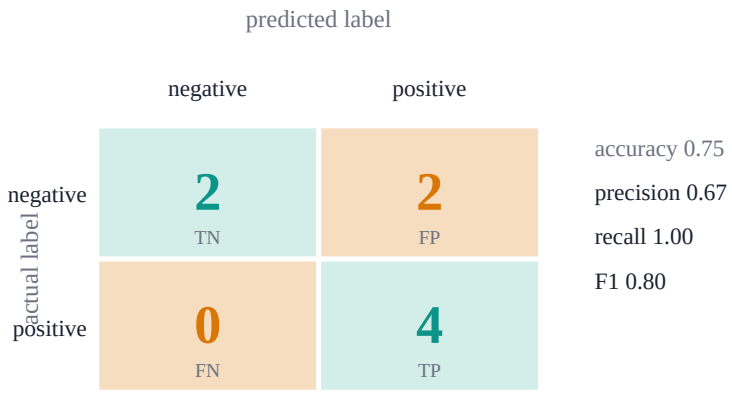
This cell trains the bag-of-words classifier on the eight toy reviews, evaluates it on eight fresh labeled reviews, and prints the confusion matrix, all four metrics, and every prediction with its errors marked. Press **Run**:

Python · NumPy ▶ runnable in the app

```
import numpy as np

train = [                                          # (review, label)  1 = positive
 ("a great and brilliant movie i loved it", 1),("wonderful acting and a great story", 1),
 ("brilliant fun and truly wonderful", 1),("i loved the great cast so good", 1),
 ("a boring and terrible waste of time", 0),("terrible dull and awful acting", 0),
 ("so boring i hated the dull story", 0),("awful terrible and a total waste", 0)]
test = [                                           # held-out reviews with human labels
 ("a wonderful and brilliant story", 1),("i loved the great fun cast", 1),
 ("great acting and a good movie", 1),("truly wonderful and so good", 1),
 ("an awful and boring waste", 0),("terrible dull and awful acting", 0),
 ("not great and not good", 0),("great story but truly terrible acting", 0)]

vocab = sorted(set(w for s, _ in train for w in s.split()))
idx   = {w: i for i, w in enumerate(vocab)}
def bow(s):
    v = np.zeros(len(vocab))
    for w in s.split():
        if w in idx: v[idx[w]] += 1
    return v
X = np.array([bow(s) for s, _ in train]); y = np.array([l for _, l in train], float)

w = np.zeros(len(vocab)); b = 0.0                  # logistic regression, from scratch
for _ in range(600):
    p = 1/(1+np.exp(-(X@w+b))); g = p - y
    w -= 0.3*(X.T@g)/len(y); b -= 0.3*g.mean()

yt   = np.array([l for _, l in test])
prob = np.array([1/(1+np.exp(-(bow(s)@w+b))) for s, _ in test])
pred = (prob > 0.5).astype(int)

TP = int(((pred==1)&(yt==1)).sum()); TN = int(((pred==0)&(yt==0)).sum())
FP = int(((pred==1)&(yt==0)).sum()); FN = int(((pred==0)&(yt==1)).sum())
acc = (pred==yt).mean(); prec = TP/(TP+FP); rec = TP/(TP+FN)
```

```
f1  = 2*prec*rec/(prec+rec)
print("confusion    pred NEG  pred POS")
print(f" actual NEG     {TN}         {FP}")
print(f" actual POS     {FN}         {TP}")
print(f"accuracy {acc:.2f}  precision {prec:.2f}  recall {rec:.2f}  F1
{f1:.2f}\n")
for (s, _), t, pp, pr in zip(test, yt, pred, prob):
    print(f"  {pr:.2f} {'POS' if pp else 'NEG'} (true {'POS' if t else
'NEG'}) {'MISS' if t!=pp else '    '} | {s}")
```

The model scores **accuracy 0.75, precision 0.67, recall 1.00, F1 0.80**. We can read this directly from the matrix: it caught all four truly-positive reviews (TP = 4, FN = 0, so recall = 4⁄4 = 1.00), but it also raised two false alarms (FP = 2). This drags precision down to 4⁄(4+2) = 0.67 and F1 to 2·(0.67·1.00)⁄(0.67+1.00) = 0.80. The gap between the perfect recall and the weak precision is the key point: the model never misses a positive review, but it too easily calls a negative one positive. And the two mistakes are not random.



The confusion matrix on eight held-out reviews. The diagonal (teal) is correct. The off-diagonal (amber) is error. Every mistake the model makes sits in a single cell (two false positives), and no true positive is missed, so recall is perfect and precision (0.67) is what holds F1 down. One bright error cell tells you exactly where to look: the negation and contrast reviews we examine next.

Where it fails: negation and sarcasm

The two misclassified reviews are worth reading one word at a time, because they are the whole reason the field moved past bag-of-words. "not great and not good" is *negative* to any human, because the writer denies both compliments. Yet the model scores it **0.96 → POS**. It sees the strong positive words "great" and "good", adds their weights, and never notices the two "not"s that flip them. This is because "not" is just another independent slot in the count vector, with no power to change its neighbors. This is the **negation** problem, and it is everywhere: "hardly worth watching", "no redeeming qualities", and "not the disaster I feared" all flip the meaning of their surface words.

The second failure, "great story but truly terrible acting" scored **0.77 → POS**, is more subtle. Here positive and negative words truly appear together. A reader uses the contrast word "but" to decide that the second clause is the main point. But bag-of-words simply *sums* "great" (+) and "terrible" (−) and lets the arithmetic decide. The same blindness makes **sarcasm** ("oh great, another two-hour snooze") nearly hopeless. Every sentiment word is positive on its surface, while the real meaning is very negative, and only tone and order reveal it. Notice, too, that both errors are false *positives*. Negation and contrast systematically make negative reviews look positive, which is exactly the precision-not-recall pattern the metrics showed.

All three problems — negation, contrast, sarcasm — share one cure: a representation in which a word's contribution can depend on the words around it. A **recurrent network** (chapter 15) reads left to right and can learn that "not" flips the sign of what follows. **Self-attention** (chapter 16) lets "but" reweight the two clauses directly. And a **pretrained transformer** fine-tuned on labeled reviews (chapters 17 and 22) brings a model of language learned from billions of words, so it recognizes these constructions before it ever meets your training set. Each one is a strictly more capable rung on the same "embed, contextualize, pool, classify" ladder.

Assumptions & limitations

Every number here rests on assumptions that are worth stating plainly. The toy dataset is tiny, and its train and test sets share almost the same vocabulary. So the 0.75 accuracy makes the model look better than it is: it would struggle on real reviews, which are full of typos, sarcasm, mixed sentiment, and tens of thousands of words. Bag-of-words *assumes* that sentiment is a sum of independent word votes, with no order, no negation, and no composition. This is why the two failures above are structural, not something that more data would cure. The labels are assumed to be correct and strictly binary, yet real sentiment is graded and often truly ambiguous. The people who label the data disagree on perhaps one review in ten, and that disagreement caps the accuracy any model can reach. Finally, a classifier trained on movie reviews will not transfer unchanged to tweets or clinical notes, where the same words carry different sentiment (chapter 22). Evaluation numbers are only valid for data drawn like the test set that produced them.

What to try next

In the runnable cell, add your own test sentence to the list and see how the model scores it. Then try one built only from words outside the vocabulary (they are skipped, so the model falls back to the bias and predicts near 0.5, the honest "I don't know"). Add a few more training reviews with new sentiment words and watch the vocabulary and the top-word lists grow. To see the limitation directly, add the training pair `("not good at all", 0)` and note that BoW still cannot use the word "not" as a flipper, because it has no idea what "not" modifies. This is the problem that chapters 15 and 16 exist to solve. This case study has no dedicated lab, but the **attention & the transformer** chapter (chapter 16) continues the same pipeline into modern models.

In the real world

Sentiment and text classification are among the most widely used NLP tasks. Companies scan product reviews, support tickets, and social media to track brand perception. Finance desks score news and filings for market signals. Platforms route or filter messages by tone and topic. The pipeline shown here — tokenize, embed, pool or contextualize, classify — is exactly the shape of production systems, only larger. Modern services fine-tune a pretrained transformer such as BERT (chapters 17 and 22) on labeled examples, inheriting language understanding from billions of words of pretraining. The bag-of-words classifier you ran is still a common, honest baseline that these heavier models must beat to justify their cost.

Exercises

1. Using the trained model, a review's sentiment logit is $z = \mathbf{w}\cdot\mathbf{x} + b$ where $\mathbf{x}$ is its bag-of-words vector. If "great" has weight +1.8, "boring" has weight −1.5, bias $b = -0.2$, and a review is exactly "great but boring" (each word once, "but" not in the vocabulary), compute the predicted probability. MATH

2. Modify the runnable classifier so it also prints each test review's *strongest* word (the in-vocabulary word with the largest absolute weight) to explain the prediction. CODE

3. Bag-of-words gives "dog bites man" and "man bites dog" identical vectors. Explain why, and name the model change that lets the two be told apart. CONCEPT

4. An embedding maps each word id to a dense 64-dimensional vector, while bag-of-words uses a sparse vector as long as the vocabulary. Give two concrete advantages of the embedding representation. CONCEPT

5. On a larger test set the classifier's predictions give TP = 80, FP = 20, FN = 40, TN = 60. Compute its accuracy, precision, recall, and F1, and say in one line what the precision–recall gap reveals about the model's behavior. MATH

6. The from-scratch `bow()` silently skips words it has never seen, so a review made only of unknown words becomes the all-zero vector, which cannot be told apart from an empty one. Add a reserved `<unk>` id and route every out-of-vocabulary token to it. Then show that an all-unknown review has a nonzero representation and predicts ≈ 0.50. CODE

CASE STUDY

Case study: time-series forecasting

> *A time series is a column of numbers indexed by time: daily sales, hourly electricity demand, a heartbeat, a stock price. Forecasting asks a hard question: given everything up to now, what comes next? This case study turns that question into a supervised-learning problem with a sliding window, trains a linear autoregressive forecaster from scratch in NumPy, shows the recurrent (LSTM/GRU) version in PyTorch, and focuses on the one mistake that quietly ruins most first attempts: leaking the future into the past.*

The problem

We want **one-step-ahead forecasting**: predict the next value x_{t+1} from a span of recent history $x_{t-L+1}, \ldots, x_t$. It looks like ordinary regression (chapter 3): inputs in, a number out. But two properties of time break the usual assumptions. First, the samples are *not independent*: today's value is highly correlated with yesterday's, so the usual "shuffle and split" of tabular data is not allowed. Second, **order carries the signal**: a model that treats the last twelve values as an unordered set loses the very trend and seasonality it is meant to capture.

Because the model may be asked to predict not just the next step but a whole **horizon** of future steps, forecasting has its own vocabulary for these difficulties. A one-step forecast is graded against the true next value. A multi-step forecast must feed its own predictions back in, so errors *compound*. And every forecaster must beat a humbling baseline (**persistence**, "tomorrow equals today") before it has earned any trust at all. Much of the work is careful, honest measurement: making sure the number you report is a number you would actually get in production, not one made to look better by information the model could never have had.

Two design choices shape the task before a single weight is trained. The **look-back** L sets how far into the past each input reaches. It should be long enough to span a seasonal cycle, but short enough that old history does not hide the signal. The **horizon** H sets how far ahead we predict. When H > 1 there are two valid strategies. A **recursive** forecaster predicts one step, feeds it

back in, and rolls forward, letting its errors compound. A **direct** forecaster trains a separate map per horizon (or one model with H outputs), so step twelve never inherits step one's mistakes. Which wins depends on how fast errors compound and how much data you can spare.

The data

Real series are messy, so we synthesize a simple one whose parts we know exactly. Each point is the sum of three ingredients that appear in almost every real signal:

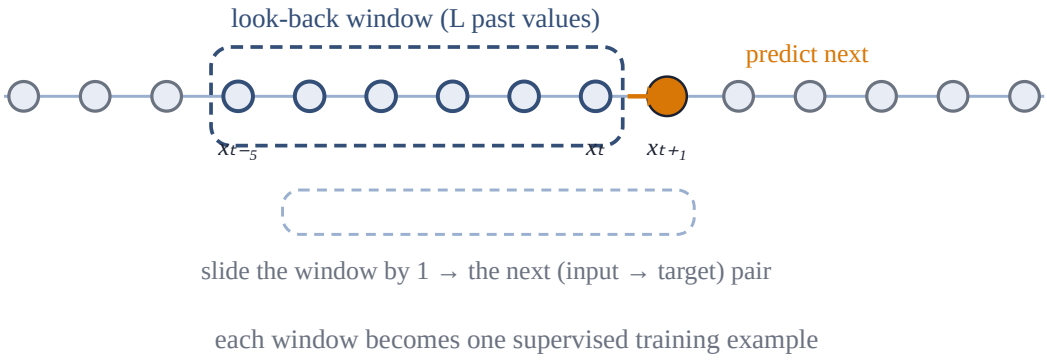
$$x_t = \sin(2\pi\, t / 24) \;+\; 0.02\, t \;+\; 0.30\, \varepsilon_t, \quad \varepsilon_t \sim N(0,1)$$

$\sin(2\pi\, t / 24)$	**seasonality** — a repeating cycle of period 24 (think hours in a day, or months across two years)
$0.02\, t$	**trend** — a slow linear drift upward. Over 360 steps it climbs by about 7 units, far larger than the season
$0.30\, \varepsilon_t$	**noise** — irreducible randomness. Its variance $0.30^2 = 0.09$ is the error floor no model can beat

To learn from it we reshape the 1-D stream into a supervised dataset with a **sliding window**. Fix a look-back length L. Then every position t gives one training pair whose input is the last L values and whose target is the next one:

$$\mathbf{x}^{(t)} = [x_{t-L+1}, \ldots, x_t] \;\longrightarrow\; y^{(t)} = x_{t+1}$$

Slide the window forward one step and read off the next pair. A series of length T gives $T - L$ such pairs. This is the single idea that lets any regressor forecast (figure 1).



Windowing (drawn with L = 6 for clarity, while the code uses L = 12). The blue bracket is the input: the last L observations. The amber point is the target: the value one step later. Slide the bracket one step to the right and you have the next example. A whole series turns into an ordinary (X, y) regression dataset.

Stationarity & normalization

Most classical forecasters, and many neural ones, behave best when the series is **stationary**: its mean and variance do not drift over time. Ours is not, because the 0.02 t trend keeps pushing the level upward. So a model predicting the raw level must *extrapolate* beyond everything it saw in training. That is exactly where forecasts fail. The classical fix is **differencing**: model the change $\Delta x_t = x_t - x_{t-1}$ instead of the level (which removes a linear trend), or a **seasonal difference** $x_t - x_{t-m}$ to remove a period-m cycle. We also **standardize** the inputs so their values are around one. But μ and σ come from the *training* block alone. Fitting them over the whole series leaks the test period's scale backward into training, the data-centric concern of Chapter 26.

Baselines before any network

Before importing a single layer, build the baselines you must beat. The **naive** (persistence) forecast sets the next value to the last, $\hat{x}_{t+1} = x_t$. It is unbeatable on a random walk and often competitive on real series. The **seasonal-naive** forecast uses the value one full season ago, $\hat{x}_{t+1} = x_{t+1-m}$, capturing a daily or weekly rhythm for free. The **moving-average** forecast returns the mean of the last k observations. A learned model earns its place only by beating the strongest of

these. A network that cannot beat seasonal-naive is only wasting power, not modeling the dynamics. That is why every result below is reported as a **skill score** against a baseline, not as a raw error.

The model

The simplest honest forecaster is a **linear autoregressive** (AR) model: predict the next value as a weighted sum of the last L values plus a bias.

$$\hat{x}_{t+1} = \Sigma_{i=1..L}\, w_i\, x_{t-L+i} + b = \mathbf{w}\cdot\mathbf{x}^{(t)} + b$$

This is exactly the neuron of chapter 1 with the window as its input, and exactly linear regression on *lagged* features. Trained with mean-squared error (chapter 5) by gradient descent (Chapter 7), it learns how much each recent lag should count. A large weight on x_t makes it rely on the most recent value. Weights that alternate in sign let it track the seasonal swing. Despite its simplicity it is a genuinely strong baseline. The classical ARIMA family is built on this backbone.

Two practical points decide whether it works. **Normalization:** we standardize each window with the mean and standard deviation of the *training* data only, then reuse those same constants on the test data. Computing the mean over the whole series would let the test set's statistics slip into training. That is a subtle leak. **The split:** because the future must never inform the past, we cut the series at a point in time and train on everything before it, validate on everything after (Chapter 26). We do not shuffle.

The linear AR is only the first rung of a ladder of capacity. One rung up is a **multilayer perceptron on lagged features**: the same window fed to a small dense network, which captures nonlinear interactions between lags that the AR cannot. Higher still, a **recurrent network** (an LSTM or GRU, Chapter 15) reads the sequence step by step and carries a hidden state, remembering in principle far beyond a hand-chosen L. Beyond that sit **temporal convolutional networks**, which slide causal 1-D filters along the series, and **transformers** (Chapter 16), whose attention lets any step attend directly to any earlier one. Every rung follows the same rule: extra capacity is useless if the evaluation leaks.

Splitting in time & backtesting

A single chronological split, even though it is honest, wastes most of the series and reports the error at only one moment in history. **Walk-forward validation** (also called **backtesting** or rolling-origin evaluation) does better: train on [0, k], forecast the next m steps, record the error, advance the origin, and repeat, grading the model repeatedly on truly unseen futures. The training window may **expand** (always start at 0) or **slide** (a fixed span that forgets the distant past, when the dynamics drift). Averaging error across the folds is the time-series analog of cross-validation. It is the only trustworthy estimate of production error, because it refuses to leak information backward across the arrow of time (Chapter 26).

In code

The whole pipeline fits in one runnable cell: synthesize, window, split by time, train a linear AR from scratch, and score. Press **Run**, then try changing the look-back `L` or the split point:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)
T = 360
t = np.arange(T)
series = np.sin(2*np.pi*t/24) + 0.02*t + 0.30*rng.randn(T)   # season +
trend + noise

L = 12                                          # look-back window length
X = np.stack([series[i:i+L] for i in range(T-L)])
y = series[L:]                                  # one-step-ahead target
n_tr = 250                                      # chronological split: past ->
future
Xtr, ytr, Xte, yte = X[:n_tr], y[:n_tr], X[n_tr:], y[n_tr:]

mu, sd = Xtr.mean(0), Xtr.std(0)                # standardize with TRAIN stats
only
Xtr, Xte = (Xtr-mu)/sd, (Xte-mu)/sd
ym = ytr.mean()                                 # center target; add back at
the end

w = np.zeros(L); b = 0.0                         # linear AR, trained from
```

```
scratch (GD)
for ep in range(3000):
    e = (Xtr@w + b) - (ytr - ym)
    w -= 0.01*(2*Xtr.T@e/len(e)); b -= 0.01*(2*e.mean())
pred = lambda Z: Z@w + b + ym
tr = np.mean((pred(Xtr)-ytr)**2); te = np.mean((pred(Xte)-yte)**2)
print(f"train MSE {tr:.3f}   test MSE {te:.3f}   (noise floor ~
{0.30**2:.3f})")
for i in range(4):
    print(f"  t={n_tr+L+i}:  actual {yte[i]:+.2f}   predicted {pred(Xte)
[i]:+.2f}")
```

The gradient-descent fit above is clear, but a linear AR also has a one-line closed form: stack the windows into a design matrix, append a column of ones for the bias, and solve the least-squares equations directly. This cell fits the weights with `np.linalg.lstsq` and scores their **RMSE** against persistence on the same time-ordered test block. This is the skill comparison that decides whether a forecaster is any good:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)
T = 400
t = np.arange(T)
series = np.sin(2*np.pi*t/24) + 0.02*t + 0.30*rng.randn(T)   # season +
trend + noise

L = 12                                          # look-back window
X = np.stack([series[i:i+L] for i in range(T-L)])
y = series[L:]                                  # one-step-ahead target
n_tr = 280                                      # chronological split: past ->
future
Xtr, ytr = X[:n_tr], y[:n_tr]
Xte, yte = X[n_tr:], y[n_tr:]

A = np.hstack([Xtr, np.ones((len(Xtr), 1))])  # design matrix + a bias
column
coef, *_ = np.linalg.lstsq(A, ytr, rcond=None)   # closed-form least-squares
AR
w, b = coef[:L], coef[L]

pred  = Xte @ w + b                             # forecast the held-out future
```

```
naive = Xte[:, -1]                              # persistence: next = last
seen value
rmse  = lambda a, f: np.sqrt(np.mean((a - f)**2))
r_ar, r_naive = rmse(yte, pred), rmse(yte, naive)
print(f"least-squares AR  RMSE = {r_ar:.3f}")      # 0.369
print(f"persistence       RMSE = {r_naive:.3f}")   # 0.452
print(f"skill vs naive    = {(1-r_ar/r_naive)*100:.0f}%   (positive => AR
wins)")  # 18%
```

Now the main point of the whole case study. We take the same windows and compare two ways of splitting them into train and test: the correct chronological cut, and the tempting-but-wrong "shuffle first, then split". A memorizing model (here a 1-nearest-neighbor forecaster, chosen because it exposes the leak most clearly) is scored under each:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)
T = 360; t = np.arange(T)
series = np.sin(2*np.pi*t/24) + 0.02*t + 0.30*rng.randn(T)
L = 12
X = np.stack([series[i:i+L] for i in range(T-L)]); y = series[L:]
n = len(X)

def nn_mse(tr, te):                             # 1-nearest-neighbour
forecaster
    d = ((X[te][:, None, :] - X[tr][None, :, :])**2).sum(-1)   # window
distances
    j = d.argmin(1)                             # nearest TRAIN window for
each test
    return np.mean((y[tr][j] - y[te])**2)     # borrow its next-step value

k = int(0.70*n)
chron = nn_mse(np.arange(k), np.arange(k, n))        # past -> future
(honest)
perm  = rng.permutation(n)
shuf  = nn_mse(perm[:k], perm[k:])                   # shuffled first
(LEAKS)
print(f"chronological split  test MSE = {chron:.3f}   <- honest")
print(f"shuffled split       test MSE = {shuf:.3f}   <- 'better', but a
```

```
lie")
print(f"leakage makes the model look {chron/shuf:.1f}x more accurate than it
is")
```

In production this is an **LSTM** or **GRU** (Chapter 15), which reads the window one step at a time and keeps a hidden state instead of flattening the lags. The PyTorch listing below is the real pipeline. Note the three details that separate a toy from a working forecaster: normalization by training statistics, the difference between *teacher forcing* during training and free-running *autoregressive rollout* at inference, and an explicit multi-step *horizon*:

PyTorch copy → Colab

```
import torch, torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader

def make_windows(series, L, H=1):             # series: 1-D tensor already
split in TIME
    xs = [series[i:i+L]        for i in range(len(series)-L-H+1)]
    ys = [series[i+L:i+L+H]    for i in range(len(series)-L-H+1)]
    X = torch.stack(xs).unsqueeze(-1)         # (N, L, 1)  -> one feature
per step
    return X, torch.stack(ys)                 # y: (N, H)

L, H = 24, 1
mu, sd = train.mean(), train.std()            # normalize with TRAIN stats
ONLY
Xtr, ytr = make_windows((train-mu)/sd, L, H)
dl = DataLoader(TensorDataset(Xtr, ytr), batch_size=32, shuffle=True)  #
shuffle windows,
#                                                                    not
time itself
class Forecaster(nn.Module):
    def __init__(self, hid=64):
        super().__init__()
        self.rnn  = nn.GRU(1, hid, batch_first=True)   # or nn.LSTM
        self.head = nn.Linear(hid, H)
    def forward(self, x):
        out, _ = self.rnn(x)                  # out: (N, L, hid)
        return self.head(out[:, -1])          # read the LAST step -> (N, H)

model = Forecaster()
opt   = torch.optim.Adam(model.parameters(), lr=1e-3)   # Adam: chapter 8
```

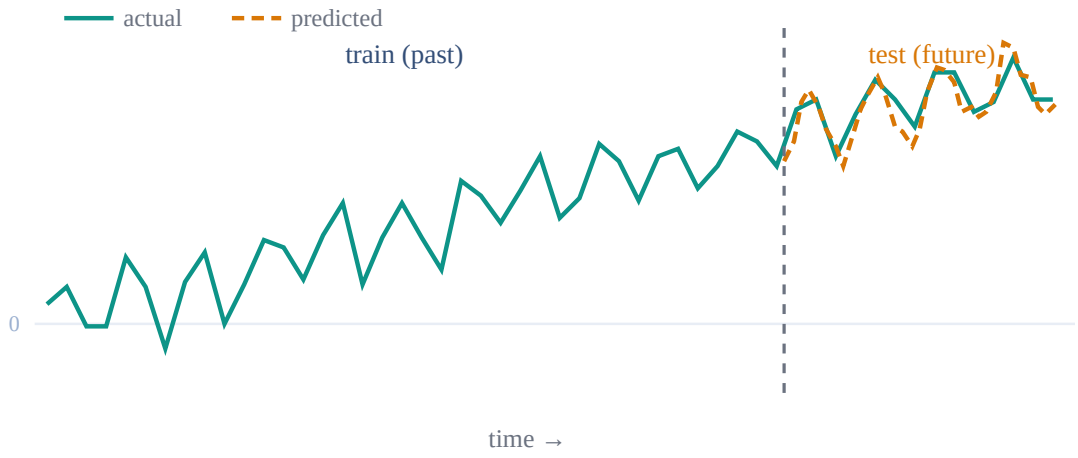
```
lossf = nn.MSELoss()
for epoch in range(30):
    for xb, yb in dl:                          # training = teacher forcing:
        opt.zero_grad()                        # the inputs are always TRUE
past values
        loss = lossf(model(xb), yb)
        loss.backward(); opt.step()

model.eval()                                   # multi-step forecast =
autoregressive rollout
with torch.no_grad():
    ctx = Xtr[-1:].clone()                     # (1, L, 1) last known window
    horizon = 12
    for _ in range(horizon):                   # feed each prediction back
in
        yhat = model(ctx)[:, :1]               # errors now COMPOUND step to
step
        ctx  = torch.cat([ctx[:, 1:], yhat.view(1, 1, 1)], dim=1)
```

Results & interpretation

The from-scratch linear AR reports **train MSE 0.141** and **test MSE 0.167**, against an irreducible noise floor of 0.090. In other words it explains almost all of the structure (season and trend), and what remains is mostly the noise we added on purpose. The four sample predictions fall within a few tenths of the truth. The small train-to-test gap is honest generalization error, plus the extra difficulty that the test block sits later in time, where the trend has carried the series into a range slightly beyond anything seen in training (extrapolation). For reference, the naive persistence baseline ("next = last") scores 0.253 on the same test block, so the AR cuts the error by about 34%. This is a real but modest win, and a forecaster must always show it.

The second cell is the lesson that matters most. The honest chronological split gives test MSE **2.127**. Shuffling the windows before splitting gives **0.683**, apparently three times better. Nothing about the model changed. Only the evaluation was corrupted. Shuffling scatters windows that are next to each other in time across both sides of the split. Because neighboring windows overlap in L−1 of their values, they are near-duplicates. So almost every test window has an almost identical twin sitting in the training set. The model looks great because it is in

effect being tested on data it has already seen. This is **data leakage**, and in a time series it hides inside the one operation you must never do: shuffling before splitting. Figure 2 shows the honest picture.



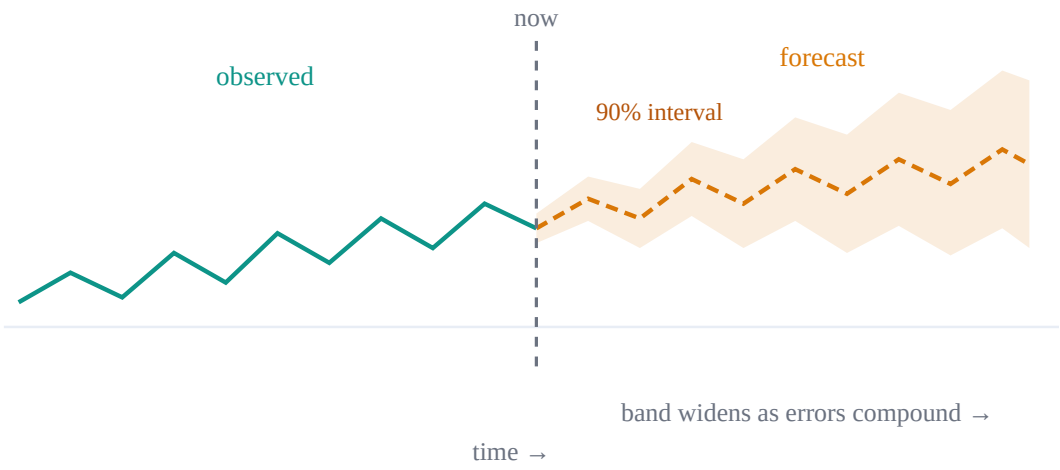
The honest forecast. The linear AR (amber, dashed) is trained only on data left of the divider, then predicts the held-out future (teal). It tracks the season and trend with test MSE 0.167. Shuffling windows across the divider would leak near-duplicate futures into training and make the score look three times better. That is the trap the second code cell exposes.

Evaluation: which error, and against what?

A single MSE hides more than it reveals, so forecasts are judged by a small set of metrics. **MAE** = $(1/n)\ \Sigma\ |x_t - \hat{x}_t|$ reports the typical miss in the series' own units and largely ignores outliers. **RMSE** = $\sqrt{((1/n)\ \Sigma\ (x_t - \hat{x}_t)^2)}$ shares those units but squares the errors, so it punishes a few large misses more harshly. **MAPE** = $(100/n)\ \Sigma\ |x_t - \hat{x}_t| / |x_t|$ is scale-free and comparable across series, but becomes very large when the truth is near zero. Never rely on any one alone: report the **skill score** $1 - \text{error}/\text{error}_{\text{naive}}$, the fraction of the baseline's error removed, and require it to be positive before you deploy.

A point forecast also hides something important: it gives one number where the honest answer is a *range*. A **prediction interval** attaches a band (say 90%) inside which the truth should fall, and it must *widen* with the horizon as errors compound (Figure 3): estimate the residual σ on validation data and report $\hat{x} \pm 1.64\ \sigma\sqrt{h}$ for an approximate 90% band h steps out, or let the

model produce its own variance or quantiles. Whether such bands are *calibrated* is the subject of Chapter 24 (uncertainty). This is what turns a forecast into a decision, because a planner sets safety stock from the width of the band, not from the point at its center.



*A forecast with honest uncertainty. Left of the divider is observed history (teal). Right of it, the forecast (amber, dashed) runs into the future inside a 90% **prediction interval** (shaded). The band is narrow one step out and widens with the horizon, because a recursive forecaster feeds its errors back and they accumulate. This is the visual sign of σ√h growth, and what Chapter 24 (uncertainty) calls a calibrated forecast.*

What to try next

Lengthen the horizon. Predict twelve steps ahead by autoregressive rollout and watch the error grow roughly like √h as mistakes compound. This is the gap between the neat one-step number and real multi-step usefulness. **Add exogenous inputs.** Real forecasts rely on calendar features (day-of-week, holidays), weather, or promotions. Concatenate them to each window. **Difference the series.** Subtracting $x_t - x_{t-1}$ removes the trend and often lets a plain MLP or RNN work far better, because it no longer has to extrapolate a level that keeps growing. **Swap the model.** Try a GRU or an LSTM (Chapter 15), a 1-D temporal convolution, or a small transformer (Chapter 16), and compare each against persistence. **Evaluate honestly.** Replace the single split with *walk-forward validation*: repeatedly train on [0, k], test on [k, k+m], then advance. This is the time-series analog of cross-validation and the only trustworthy way to estimate future error. **Predict distributions.** Output a mean and a variance (or quantiles) so the forecast comes with honest uncertainty (Chapter 24), not a single point with false precision.

Exercises

1. A persistence model predicts that the next value equals the last observed value. For three consecutive test steps the last-observed values are 3.0, 3.5, 3.2 and the true next values are 3.4, 3.1, 3.6. Compute the persistence mean-squared error by hand. MATH

2. Implement the persistence baseline in NumPy on the case-study series and confirm it is worse than the linear AR's test MSE of 0.167. CODE

3. Why must the validation set come strictly *after* the training set in time, and what real-world failure does shuffling before splitting simulate? Would standardizing the windows with the mean and standard deviation of the *whole* series (train + test) be a similar mistake? CONCEPT

4. A one-step forecaster has zero-mean prediction errors with variance σ^2. In a recursive multi-step forecast, each prediction is fed back in as the input for the next step. Explain why the h-step-ahead error variance grows at least linearly in h, and relate this to the difference between teacher forcing and autoregressive rollout. CONCEPT

5. For four test steps the true values are x = [10, 12, 11, 13] and a model predicts $\hat{x}$ = [11, 11, 9, 14]. Compute the MAE and RMSE. Then, given that the seasonal-naive baseline scores RMSE 2.0 on the same steps, compute the model's skill score. Is it worth deploying? MATH

6. Implement the **seasonal-naive** baseline (period m = 24) on the case-study series and compare its RMSE to persistence on the chronological test block. Which wins here, and what does that tell you about when seasonal-naive is the baseline to beat? CODE

In the real world

Windowed neural forecasters run much of the modern economy's planning. Amazon's **DeepAR** (Salinas et al., 2020) trains a single autoregressive LSTM across hundreds of thousands of related product series to forecast retail demand with calibrated uncertainty, and its successors power inventory decisions in AWS Forecast. Grid operators forecast

electricity load hour-by-hour to schedule generation. Ride-hailing platforms like Uber forecast trip demand per city block to pre-position drivers. And the influential **M4** and **M5** competitions (Makridakis et al.) benchmark exactly the windowing-plus-model recipe of this case study against classical statistics on tens of thousands of real series. In every one of these systems the split is chronological and the leakage rules above are not optional. A model validated by shuffling would report beautiful numbers, then lose money on its first real day.

CASE STUDY

Case study: anomaly detection with autoencoders

> *Some of the most valuable events in data are the rare ones — a fraudulent transaction, a failing bearing, an irregular heartbeat, an intruder on the network. They are too rare and too varied to label, so you cannot train a classifier to recognize them. This case study takes the opposite approach: learn what normal looks like, then flag whatever fails to fit. An autoencoder (Chapter 19) is the natural tool. We build a tiny one from scratch in NumPy, score anomalies by reconstruction error, choose a threshold from the normal error distribution, and measure precision and recall. The PyTorch listing shows the same recipe at production scale.*

The problem

Anomaly detection differs from ordinary classification in a way that changes everything: you almost never have a usable set of positive examples. Anomalies are **rare** (perhaps one in ten thousand), **diverse** (each fault fails differently), and **open-ended** (tomorrow's anomaly may look nothing like any you have seen). A supervised classifier trained on last year's frauds simply memorizes last year's frauds. Worse, the data is effectively **unlabeled**: nobody has marked which past readings were faults, and the few remembered cases are far too sparse to define a class. Supervised classification needs both classes in large numbers. Here one class is abundant and clean, while the other is rare, unlabeled, and forever changing shape. This makes supervised classification the wrong approach from the start. We therefore view the task as **one-class learning**, also called novelty detection: build an accurate model of normality from abundant normal data, then treat any input the model reconstructs, explains, or predicts poorly as suspect.

The autoencoder fits this approach exactly. Train it to copy normal data through a narrow bottleneck and it becomes an expert on the structure of normality — and *only* normality. Present a normal input and it rebuilds it almost perfectly. Present an anomaly it has never seen, and it

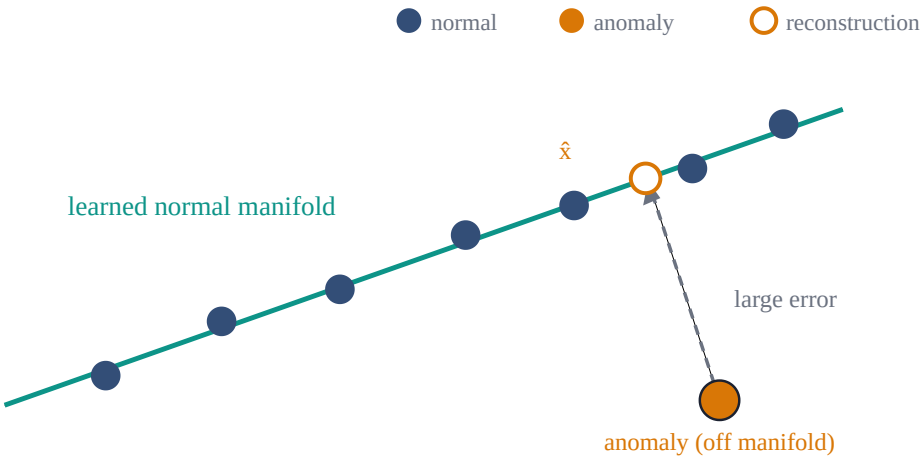
rebuilds it badly because it has no code for it. That failure is the signal. The **reconstruction error** becomes an anomaly score, and detection becomes just a matter of picking a sensible threshold on that score.

The data

Real sensor and signal data usually lives near a low-dimensional surface: eight temperature and vibration channels on a machine are highly correlated, so although each reading is a point in 8-D, the normal ones cluster near a thin 2-D sheet. We synthesize exactly that. A hidden 2-D factor $\mathbf{z}$ is drawn from a standard Gaussian and mapped into 8 dimensions by a fixed random matrix A, plus a little per-sensor noise:

$$\mathbf{x} = \mathbf{z}\,A + 0.10\,\eta, \quad \mathbf{z} \sim N(0, I_2),\ \eta \sim N(0, I_8)$$

Every normal reading therefore sits close to a 2-D plane embedded in 8-D space (figure 1). **Anomalies** are points knocked off that plane (a sensor spikes, or a new failure mode appears). We model them by adding a large, unstructured perturbation. Importantly, the autoencoder is trained on **normal data only**. The anomalies exist only to test it, and their labels are used just to compute precision and recall at the end.



The geometry of the method. The autoencoder learns the low-dimensional surface (teal) where normal readings (blue) live. A normal point sits on it and reconstructs almost exactly. An anomaly (amber) lies off the surface, so the best the decoder can do is project it back onto the manifold (hollow). That distance, the reconstruction error, is the anomaly score.

The model

We use an **undercomplete autoencoder** (autoencoders, Chapter 19): an encoder squeezes the D-dimensional input to a code of size d < D, and a decoder expands it back. With linear layers and squared-error loss this is exactly **PCA**: the code spans the top principal subspace of the normal data. This is all we need when normality is a plane:

$$\mathbf{z} = \mathbf{x}\,W_e \ \text{(encode, d numbers)}, \quad \hat{x} = \mathbf{z}\,W_d \ \text{(decode back to D)}$$

The **anomaly score** is the mean squared reconstruction error over the D features, and a point is flagged when it exceeds a threshold τ:

$$s(\mathbf{x}) = \frac{1}{D}\,\Sigma_i\,(x_i - \hat{x}_i)^2, \quad \text{flag} \iff s(\mathbf{x}) > \tau$$

$\mathbf{W_e}, \mathbf{W_d}$	encoder (D×d) and decoder (d×D) weight matrices, trained by gradient descent (Chapter 7) on normal data
d	code size, the bottleneck; here 2, matching the intrinsic dimension of the normal data
s(x)	reconstruction error, the anomaly score: small for normal inputs, large for off-manifold ones
τ	decision threshold, taken from the distribution of s on held-out normal data (a chosen percentile)

Setting τ is the one truly new decision. We never see anomaly scores while choosing it. Instead we take a high percentile of the *normal* validation errors, say the 99th. By design, this lets through only about 1% of normal points as false alarms. Raising the percentile reduces false alarms (higher precision) but risks missing subtle anomalies (lower recall). The right choice depends on the relative cost of a missed fault versus a false alarm.

Choosing the threshold

The percentile rule sets τ to a high quantile of the *normal* validation errors. It is the honest default when you have no labeled anomalies at all, and the usual starting point. It fixes the false-alarm rate by design, without needing any anomaly: the 99th percentile passes about 1% of normal points, the 99.9th about 0.1%. Pick the quantile from what the downstream team can handle. Fifty alerts a day against a million scored points means a cut near the 99.995th percentile, not the 99th.

The moment you can gather even a **handful of confirmed anomalies** (a dozen past frauds, a few labeled failures), the threshold stops being a guess. Hold them out, sweep τ across the range of normal errors (the second code cell above), and take the cut that maximizes F1, or the smallest τ that reaches a required recall ("catch 95% of known failures"), or the one that minimizes an explicit cost of a miss versus a false alarm. Those few labels are far more useful for the threshold than for training, since the detector never sees them. The one rule you cannot skip: fix τ on validation data and report on a separate test set, otherwise it overfits the very anomalies that set it.

In code

The full recipe runs in one cell: synthesize normal data, train a linear autoencoder from scratch, set a threshold from normal errors, then score a labeled test mix. It prints the key evidence: anomalies reconstruct hundreds of times worse than normal points, and the threshold cleanly separates them.

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)
D, d = 8, 2                                    # 8 sensors, true 2-D
structure
A = rng.randn(d, D)                            # fixed mixing: data
near a 2-D plane
def normal(n): return rng.randn(n, d) @ A + 0.10*rng.randn(n, D)
Xtr = normal(500); mu, sd = Xtr.mean(0), Xtr.std(0)
z = lambda X: (X - mu)/sd                      # standardize with
train stats

W1 = rng.randn(D, d)*0.1; W2 = rng.randn(d, D)*0.1   # encoder D->2, decoder
2->D
S = z(Xtr)
for ep in range(4000):                         # train AE on NORMAL
data only (GD)
    H = S@W1; G = 2*(H@W2 - S)/len(S)          # linear autoencoder (=
PCA)
    W2 -= 0.05*(H.T@G); W1 -= 0.05*(S.T@(G@W2.T))
err = lambda X: np.mean((z(X)@W1@W2 - z(X))**2, axis=1)   # reconstruction
error / row
```

```
thr = np.percentile(err(normal(300)), 99)            # threshold from NORMAL
errors only
Xn = normal(200)                                     # held-out normal
traffic
Xa = normal(40) + 3.0*rng.randn(40, D)               # anomalies: pushed off
the plane
X = np.vstack([Xn, Xa]); lab = np.r_[np.zeros(200), np.ones(40)]
flag = (err(X) > thr).astype(int)
tp = int(((flag==1)&(lab==1)).sum()); fp = int(((flag==1)&(lab==0)).sum())
fn = int(((flag==0)&(lab==1)).sum())
print(f"normal  mean recon-err {err(Xn).mean():.4f}")
print(f"anomaly mean recon-err {err(Xa).mean():.4f}
({err(Xa).mean()/err(Xn).mean():.0f}x higher)")
print(f"threshold (99th pct of normal) {thr:.4f}")
print(f"TP {tp}  FP {fp}  FN {fn}   ->  precision {tp/(tp+fp):.2f}   recall
{tp/(tp+fn):.2f}")
```

The threshold is a dial, not a constant. Sweeping the percentile shows the precision–recall trade-off directly. A stricter (higher) threshold raises precision by cutting false alarms. Because our anomalies sit so far off the manifold, recall stays perfect throughout. With subtler anomalies you would begin to lose recall as the price of higher precision.

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)
D, d = 8, 2; A = rng.randn(d, D)
def normal(n): return rng.randn(n, d) @ A + 0.10*rng.randn(n, D)
Xtr = normal(500); mu, sd = Xtr.mean(0), Xtr.std(0); z = lambda X: (X -
mu)/sd
W1 = rng.randn(D, d)*0.1; W2 = rng.randn(d, D)*0.1; S = z(Xtr)
for ep in range(4000):
    H = S@W1; G = 2*(H@W2 - S)/len(S); W2 -= 0.05*(H.T@G); W1 -= 0.05*
(S.T@(G@W2.T))
err = lambda X: np.mean((z(X)@W1@W2 - z(X))**2, axis=1)

val = err(normal(300))                               # normal error
distribution
Xn = normal(200); Xa = normal(40) + 3.0*rng.randn(40, D)
X = np.vstack([Xn, Xa]); lab = np.r_[np.zeros(200), np.ones(40)]; e = err(X)
print("percentile  threshold  precision  recall")
```

```
for pct in (90, 95, 99, 99.9):
    thr = np.percentile(val, pct); fl = e > thr
    tp = int((fl & (lab==1)).sum()); fp = int((fl & (lab==0)).sum()); fn =
int((~fl & (lab==1)).sum())
    print(f"   {pct:5}    {thr:.4f}     {tp/(tp+fp):.2f}
{tp/(tp+fn):.2f}")
```

At production scale the encoder and decoder become nonlinear MLPs so the model can follow a *curved* manifold. This is the standard setup for ECG heartbeats or machine-sensor windows. The workflow is identical: fit on normal, take a threshold from held-out normal error, flag the rest.

PyTorch copy → Colab

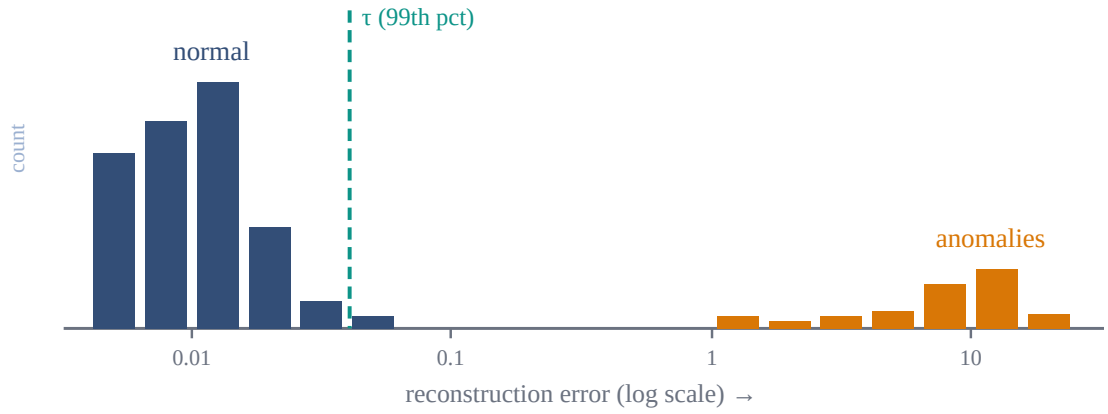
```
import torch, torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader

# X_normal: (N, D) normal windows only (e.g. ECG5000 healthy beats, or
sensor readings)
mu, sd = X_normal.mean(0), X_normal.std(0) + 1e-8    # normalize with NORMAL
stats only
Xtr = (X_normal - mu) / sd
dl  = DataLoader(TensorDataset(Xtr), batch_size=64, shuffle=True)

class AE(nn.Module):
    def __init__(self, D, code=8):
        super().__init__()
        self.enc = nn.Sequential(nn.Linear(D, 32), nn.ReLU(), nn.Linear(32,
code))
        self.dec = nn.Sequential(nn.Linear(code, 32), nn.ReLU(),
nn.Linear(32, D))
    def forward(self, x):
        return self.dec(self.enc(x))                # bottleneck: code << D

model = AE(Xtr.shape[1])
opt   = torch.optim.Adam(model.parameters(), lr=1e-3)    # Adam: Chapter 9
for epoch in range(50):
    for (xb,) in dl:
        opt.zero_grad()
        loss = ((model(xb) - xb)**2).mean()          # reconstruction loss:
Chapter 6
        loss.backward(); opt.step()
```

```
@torch.no_grad()
def score(X):                                          # anomaly score = per-
row recon error
    Xn = (X - mu) / sd
    return ((model(Xn) - Xn)**2).mean(dim=1)

thr   = torch.quantile(score(X_val_normal), 0.99)     # threshold from NORMAL
errors only
flags = score(X_test) > thr                            # above threshold  ->
anomaly
```

Results & interpretation

The numbers tell a clean story. Normal held-out data reconstructs with mean error **0.0111**. The injected anomalies reconstruct with mean error **8.67**, roughly **780 times** worse. The 99th-percentile threshold lands at **0.0400**, sitting in the wide empty gap between the two populations (figure 2). Scoring the labeled test mix gives **40 true positives, 4 false positives, 0 false negatives**, for precision **0.91** and recall **1.00**. Every anomaly is caught because each one lies far off the learned plane. The only errors are the handful of normal points that, by the definition of a 99th percentile, were always going to raise the alarm.

The sweep makes the design choice explicit: at the 90th percentile precision is just 0.58 (many false alarms), rising to 0.70 at the 95th, 0.91 at the 99th, and 0.95 at the 99.9th, with recall held at 1.00 the whole way. That flat recall is a feature of *this* easy, well-separated problem, not a law. The moment anomalies move closer to the manifold, the same dial that raises precision starts to cost recall, and the threshold becomes a real business decision about the price of a miss versus the price of a false alarm.



The reconstruction-error histogram. Held-out normal errors (blue) pile up near 0.01. The injected anomalies (amber) land 100–1000× higher, past a wide empty gap. The threshold τ (teal), the 99th percentile of the normal errors, sits in that gap, so every anomaly clears it (recall 1.00) while only about 1% of normal points raise a false alarm (precision 0.91).

Evaluation under extreme imbalance

Anomaly detection is the most imbalanced problem in this book. It is not one churner in four like the tabular case study (Chapter 31), but one event in a thousand or ten thousand. That single fact destroys **accuracy** as a metric. A detector that flags *nothing* scores 99.99% on a stream that is 0.01% anomalous. The sharper trap is that a truly useful detector often scores *worse* than that do-nothing baseline, because its handful of false alarms cost more raw errors than its catches are worth. Accuracy rewards silence, which is the opposite of the job.

The honest panel is the one built in the tabular case study (Chapter 31), now read in the imbalanced setting. **Precision** (of the points we flagged, how many were real) and **recall** (of the real anomalies, how many we caught) both ignore the vast pile of correctly-ignored normal points, so they stay meaningful. Because both move with τ, summarize the detector *across* thresholds with the **precision–recall curve** and its area, **PR-AUC** (average precision). This is preferred here over ROC-AUC, whose false-positive-rate axis is dominated by the enormous normal class and stays misleadingly high even for a weak detector. And since in practice a person or a budget reviews only the top few alerts, the most practical metric is **precision@k**: of the k highest-scoring points, what fraction are truly anomalous. This is the direct answer to "if my team investigates the 20 most suspicious cases today, how many are real?"

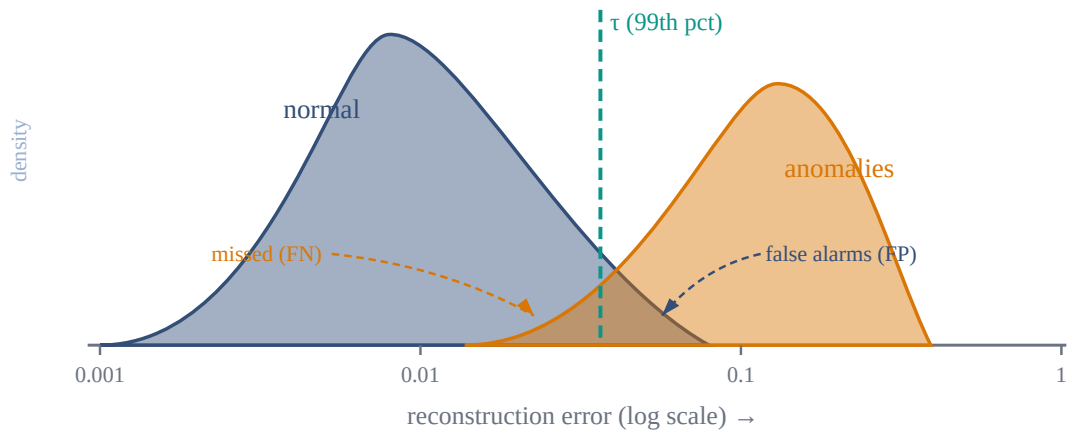
The cell below trains the detector in *closed form*. A linear autoencoder is just PCA, so the encoder is the top-d right singular vectors of the standardized normal data, with no gradient descent needed. It then scores a realistic stream of 990 normal points and 20 **subtle** anomalies (a 2% base rate, each barely off the plane so the error distributions overlap) and prints the whole panel:

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)
D, d = 8, 2                                   # 8 sensors, intrinsic dim 2
A = rng.randn(d, D)
def normal(n): return rng.randn(n, d) @ A + 0.10*rng.randn(n, D)
Xtr = normal(500); mu, sd = Xtr.mean(0), Xtr.std(0)
z = lambda X: (X - mu)/sd                     # standardize with train stats

# train the linear autoencoder in CLOSED FORM: it is just PCA (SVD, no gradient descent)
U, S, Vt = np.linalg.svd(z(Xtr), full_matrices=False)
P = Vt[:d]                                    # top-d principal axes (d x D)
err = lambda X: np.mean((z(X)@P.T@P - z(X))**2, axis=1)   # residual off the subspace

thr = np.percentile(err(normal(300)), 99)     # threshold from NORMAL errors only
Xn = normal(990)                              # a realistic 2%-anomaly stream:
Xa = normal(20) + 0.30*rng.randn(20, D)       # 20 SUBTLE anomalies, barely off-plane
X  = np.vstack([Xn, Xa]); lab = np.r_[np.zeros(990), np.ones(20)]
e  = err(X)

flag = (e > thr).astype(int)
tp = int(((flag==1)&(lab==1)).sum()); fp = int(((flag==1)&(lab==0)).sum());
fn = int(((flag==0)&(lab==1)).sum())
prec = tp/(tp+fp); rec = tp/(tp+fn)
k = 20; patk = lab[np.argsort(-e)[:k]].mean()  # precision@k: an analyst's daily budget
order = np.argsort(-e); ls = lab[order]       # PR-AUC (average
```

```
precision) by ranking
tpc = np.cumsum(ls); fpc = np.cumsum(1-ls)
Pc = tpc/(tpc+fpc); Rc = tpc/lab.sum()
ap = np.sum((Rc - np.r_[0.0, Rc[:-1]]) * Pc)

print(f"anomaly base rate            {lab.mean()*100:.1f}%")
print(f"accuracy  detector {(flag==lab).mean():.3f}   vs 'flag nothing' {1-
lab.mean():.3f}")
print(f"precision {prec:.2f}    recall {rec:.2f}    (TP {tp}  FP {fp}  FN
{fn})")
print(f"precision@{k}  {patk:.2f}     PR-AUC {ap:.3f}")
```

The result is striking. The detector catches 85% of anomalies at 49% precision, yet its **accuracy of 0.979 sits just below the 0.980 of flagging nothing**. Accuracy cannot even see it working. Precision@20 is 0.70 (fourteen of the twenty most-suspicious points are genuine). PR-AUC is 0.806, a threshold-free score that rises when the model improves and holds when you just re-pick the operating point. This is the number to track while tuning. The overlap that makes this case realistic (figure 3) is exactly what the clean-separation histogram (figure 2) could not show.



The realistic case: overlapping error distributions. Unlike figure 2's clean separation, these subtle anomalies (amber) reconstruct only slightly worse than normal points (blue), so the densities overlap. Each is normalized on its own, since 20 anomalies would vanish beside 990 normals. Now wherever τ falls it makes both errors: normal points to its right are false alarms (FP), anomalies to its left are missed (FN). This is why one accuracy number says nothing here, and precision, recall, PR-AUC and precision@k are the metrics that do.

Classical baselines to beat

The autoencoder is neither the only nor always the best way to model normality. A serious project first benchmarks three classical detectors, each a line of scikit-learn. **Isolation Forest** builds random trees that split on random features at random thresholds. A rare, different point is isolated in unusually few splits, so its average path length to isolation is the score. It needs no distance metric, scales well, and is often the strongest cheap baseline. **One-class SVM** fits a boundary that encloses most of the normal data in a kernel feature space and flags the outside. It is elegant, but roughly quadratic and sensitive to its kernel and its ν. **Density and distance methods** (kernel density estimation, the k-th nearest-neighbor distance, or its local variant LOF) score how sparse a point's neighborhood is. They are simple and interpretable, but they break down under the curse of dimensionality, where all points drift toward nearly equal distances and "density" stops meaning much.

That last weakness is where the autoencoder proves its value: when data is high-dimensional but lies near a low-dimensional, possibly *curved* manifold (images, ECG windows, sensor spectra), a nonlinear autoencoder learns that manifold and measures deviation from it, where raw-space density and distance fail. On low-dimensional or small tabular data, by contrast, Isolation Forest usually matches a network for a fraction of the effort. As with trees in the tabular study, fit the cheap baseline first, and use the autoencoder only when dimensionality and manifold structure call for it.

A probabilistic anomaly score

A raw reconstruction distance has no natural scale. Is an error of 0.04 large? It depends on the noise a sensor normally carries, which varies across features and across the manifold. A **probabilistic** score asks instead: how *unlikely* is this input under a model of normality? The variational autoencoder (autoencoders, Chapter 19; a deep generative model, Chapter 21) is the standard tool. Its encoder produces a distribution $q(z|x)$ over a latent code, a learned probabilistic *representation* of the input (representations, Chapter 5). Its decoder produces a distribution $p(x|z)$. The pair is trained to maximize a lower bound (the ELBO) on the data likelihood. The score becomes the **reconstruction probability** $\mathbb{E}_{q(z|x)}[\log p(x|z)]$, or the full

negative ELBO, which also penalizes a point for needing an unusual code. Measured in nats rather than raw squared units, it discounts inherently noisy features and flags a deviation only when it is truly improbable, not just large.

This turns detection into **density estimation** (normal inputs high-likelihood, anomalies low) and connects straight to the uncertainty material (uncertainty & calibration, Chapter 24): a Bayesian or ensemble model that is *uncertain* about an input, with high predictive entropy or wide ensemble disagreement, expresses the same "I have not seen this before" in another way, and the two cues are often combined. A cheaper relative is the **Mahalanobis distance** of the code z to the Gaussian of normal codes, which accounts for the latent covariance that a plain Euclidean error ignores.

Concept drift

Everything so far assumes "normal" is a fixed target. It is not: a machine wears, a season turns, a product ships a new version, customers change habits, and the manifold of normality drifts beneath the model. A frozen detector then suffers **concept drift**: yesterday's ordinary reading reconstructs a little worse today, the normal-error distribution slowly rises, and false alarms climb until the alerts are ignored. Drift is what quietly kills deployed detectors, and it is a data problem (data-centric deep learning, Chapter 26) more than a modeling one.

The defense is monitoring, not a cleverer network. Track the **false-alarm rate** as an early-warning signal: if the fraction of points that exceed τ drifts well above the percentile you designed for, the model (not the world) is probably out of date. Recompute τ from a recent normal-error distribution on a schedule, and retrain the autoencoder on a sliding window of recent normal data so it follows the slow drift. Keep that window clean, since a slowly growing fault absorbed into training is the contamination problem again, now arriving through time. Where a change is abrupt (a sensor swapped, a pipeline rebuilt), a formal change-point test on the score stream catches what slow retraining would miss.

What to try next

Go nonlinear. Replace the linear encoder/decoder with ReLU MLPs (the PyTorch listing) so the model can follow a curved manifold that PCA cannot. This is essential for images, ECG, or audio. **Localize the fault.** Instead of one scalar score, keep the per-feature squared error and report which sensor deviates most. On real machines this turns "something is wrong" into "bearing 3 is overheating". **Guard against contamination.** If a few anomalies sneak into the training set, the model learns to reconstruct them too. Robust variants down-weight high-error points during training, or fit iteratively. **Use a better score.** A variational autoencoder (Chapter 19) gives a reconstruction *probability* rather than a raw distance, and a Mahalanobis distance in latent space accounts for the code's own covariance. **Handle drift.** Normality changes with seasons, load, and wear, so schedule retraining and monitor the false-alarm rate as an early-warning signal. **Tune the threshold properly.** If you can gather even a small labeled validation set of anomalies, choose τ to maximize F1 or hit a target recall, rather than guessing a percentile.

Exercises

1. The per-feature reconstruction errors $|x_i - \hat{x}_i|$ for a normal sample are [0.1, 0.2, 0.1, 0.0] and for a suspected anomaly are [0.8, 1.1, 0.9, 0.6]. (a) Compute each sample's mean-squared reconstruction error. (b) With threshold $\tau = 0.1$, which is flagged? (c) By what factor does the anomaly's score exceed the normal one's? MATH

2. Extend the detector to *localize* a fault: break a single sensor on one reading and show that the largest per-feature reconstruction error points at the broken sensor. CODE

3. Why is the autoencoder trained on normal data only, and what goes wrong if the training set is contaminated with a few unlabeled anomalies? And why does this problem show perfect recall, when a real fraud or arrhythmia detector rarely does? CONCEPT

4. Explain, using the PCA view of a linear autoencoder, why an off-manifold anomaly reconstructs badly even if you widen the code from 2 up toward the input dimension, yet a

code as wide as the input (or a powerful nonlinear decoder) can destroy detection entirely. CONCEPT

5. A network monitors 100,000 transactions a day, of which about 10 are fraudulent (a 0.01% base rate). A detector flags 200 transactions and catches 8 of the 10 frauds. (a) What accuracy would a detector that flags *nothing* achieve? (b) Compute this detector's precision, recall, and accuracy. (c) Analysts can investigate only the 50 highest-scoring alerts. If 7 of those 50 are real fraud, what is precision@50, and why is it the number the fraud team actually cares about? MATH

6. You must detect anomalies in two datasets: (i) a 6-column tabular log of server metrics, a few thousand rows; (ii) 512-sample ECG windows. For each, which would you try first: Isolation Forest, a k-nearest-neighbor density score, or a nonlinear autoencoder? And why? Explain in particular why the kNN density score is a poor choice for the ECG windows. CONCEPT

In the real world

Reconstruction-error anomaly detection is deployed wherever anomalies are too rare or too varied to label. Banks and card networks score transactions against a model of a customer's normal behavior to catch fraud in milliseconds. In **predictive maintenance**, autoencoders trained on healthy vibration and temperature signals flag failing bearings and motors before they break. NASA's telemetry datasets (SMAP/MSL) and the Numenta Anomaly Benchmark are standard testbeds. In medicine, autoencoders trained on *normal* heartbeats detect arrhythmias on the ECG5000 benchmark (the worked example in autoencoders, Chapter 19), because a rhythm the model has never seen reconstructs badly. The same idea guards computer networks against novel intrusions and inspects manufactured parts for defects. In every case the discipline is the one this study emphasizes: fit on clean data, take the threshold from the normal error distribution, and remember that a contaminated "normal" set silently blinds the detector.

PART VII

Appendices

Self-contained reference material: refreshers on the linear algebra, calculus and probability the book uses, hands-on primers on Python/NumPy and PyTorch, and a guide to datasets and further reading.

35 Appendix A · Linear algebra refresher

36 Appendix B · Calculus & the chain rule

37 Appendix C · Probability & statistics

38 Appendix D · Information theory

39 Appendix E · Python & NumPy primer

40 Appendix F · PyTorch primer

41 Appendix G · Autograd from scratch

42 Appendix H · Datasets & further reading

43 Appendix I · Formulas & symbols quick reference

APPENDIX

Appendix A · Linear algebra refresher

> *Neural networks are, at their core, almost nothing but linear algebra. A forward pass is a chain of matrix multiplications with a nonlinearity between them, and a backward pass is the same multiplications transposed. This appendix collects the small vocabulary you really need — scalars, vectors, matrices and tensors, shapes, the dot product and its geometry, matrix multiplication as composition, transpose and inverse, rank, eigenvalues, the singular value decomposition, norms, the Hadamard product and broadcasting — with runnable NumPy so every idea is shown with real numbers. It is the reference behind the neuron (chapter 1), the layer-as-matmul view (chapter 3) and the transposes and elementwise products of backpropagation (chapter 8).*

Scalars, vectors, matrices, tensors

Everything a network computes with is one of four shapes of number. They differ only in how many indices you need to pick out an element.

scalar a single number, no index — a learning rate η, a loss value L. NumPy calls this a 0-D array.

vector an ordered list, one index — an input $\mathbf{x} = [x_1, \ldots, x_n]$, a neuron's weights $\mathbf{w}$. Shape (n,).

matrix a rectangular grid, two indices — a weight matrix W with entry W_{ij} in row i, column j. Shape (m, n).

tensor the general case, any number of indices. A batch of RGB images is a 4-D tensor (batch, height, width, channel). Here the word "tensor" just means "n-dimensional array".

The **shape** is the tuple of sizes along each axis. In deep learning, most bugs are shape bugs. For example, a (32, 784) batch meeting a (10, 128) weight matrix will crash. Check shapes constantly. Throughout this book we use the convention that a batch of m examples is stored as the *rows* of a matrix X of shape (m, n) — one example per row, one feature per column — and we explain at the end why rows and not columns.

Shapes and the dot product

The **dot product** (or inner product) of two equal-length vectors multiplies them component-wise and sums:

$$\mathbf{a}\cdot\mathbf{b} = \Sigma_{i=1..n}\, a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

This single operation *is* a neuron's weighted sum. The pre-activation $z = \mathbf{w}\cdot\mathbf{x} + b$ of chapter 1 is one dot product plus a bias. Geometrically $\mathbf{a}\cdot\mathbf{b} = \|a\|\|b\|\cos\theta$, so the dot product measures alignment. It is large and positive when two vectors point the same way, zero when they are perpendicular, and negative when they point in opposite directions. A neuron therefore fires most strongly for inputs aligned with its weight vector — the weight vector is the pattern the neuron is "looking for".

The geometry of the dot product

Behind that algebraic sum is a geometric picture. For any two vectors the dot product equals the product of their lengths times the cosine of the angle θ between them, so $\cos\theta = (\mathbf{a}\cdot\mathbf{b})/(\|\mathbf{a}\|\|\mathbf{b}\|)$. Three consequences matter most in deep learning. **Orthogonality**: nonzero vectors are perpendicular exactly when their dot product is zero. **Projection**: the amount of $\mathbf{a}$ lying along a unit vector $\hat{\mathbf{u}}$ is just $\mathbf{a}\cdot\hat{\mathbf{u}}$ — this is what a neuron's $\mathbf{w}\cdot\mathbf{x}$ measures. **Cosine similarity**: dividing out both lengths leaves $\cos\theta \in [-1, 1]$, a measure of alignment that does not depend on length. It is the standard way to compare the learned embeddings of chapter 5. Two vectors are similar when they point the same way ($\cos\theta$ near 1), unrelated when orthogonal (near 0) and opposite when anti-aligned (near −1), no matter how long they are.

Worked example — cosine similarity of two embeddings

Take $\mathbf{a} = [2, 2, 1]$ and $\mathbf{b} = [0, 3, 4]$. Then $\mathbf{a}\cdot\mathbf{b} = 2\cdot0 + 2\cdot3 + 1\cdot4 = 10$, with $\|\mathbf{a}\| = \sqrt{4 + 4 + 1} = 3$ and $\|\mathbf{b}\| = \sqrt{9 + 16} = 5$, so $\cos\theta = 10/(3\cdot5) = \mathbf{0.667}$ and $\theta = \arccos 0.667 \approx \mathbf{48.2°}$. Rescaling $\mathbf{a}$ to [20, 20, 10] multiplies the dot product tenfold to 100, but the cosine

stays exactly 0.667, because it now divides by a length that is ten times larger too. This size-independence is why similarity search over embeddings uses the cosine rather than the raw dot product.

Matrix multiplication — and why it is a neural layer

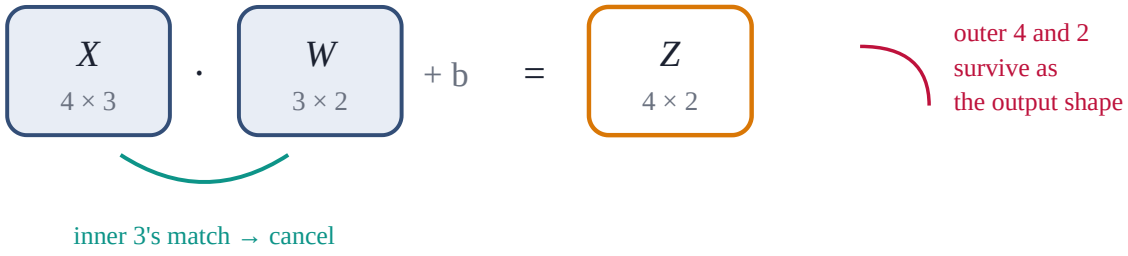
To multiply matrices A (shape m×n) and B (shape n×p), the **inner dimensions must match**. The result C has shape m×p, and each entry is a dot product of a row of A with a column of B:

$$(m \times n)(n \times p) \to (m \times p), \quad C_{ij} = \Sigma_{k=1..n} A_{ik} B_{kj}$$

The inner n's cancel, and the outer m and p survive. That one rule controls almost every shape check you will ever do. Now the key realization of chapter 3: **a fully-connected layer is a single matrix multiplication.** Stack the weight vectors of k neurons as the *columns* of a matrix W of shape (n, k). Feed one input row **x** of shape (1, n). Then

$$\mathbf{z} = \mathbf{x}W + \mathbf{b}$$

computes all k weighted sums at once — column j of W is neuron j's weights, so entry $z_j = \mathbf{x} \cdot W_{:,j} + b_j$ is exactly that neuron's dot product. Push a whole batch X of shape (m, n) through the same W and you get every neuron's answer for every example in one operation, Z = XW + **b**, of shape (m, k). The panel-of-judges picture of chapter 3 is literally this product. Matrix multiplication is not a metaphor for a layer. It is the layer.



One layer, one multiply. The inner dimensions (both 3) must agree and disappear. The outer dimensions — batch size 4 and neuron count 2 — become the output shape. This is the shape arithmetic you trace in the Playground's Math Log.

Matrix multiplication is **associative** ((AB)C = A(BC)) and **distributive** over addition, but *not* commutative. AB ≠ BA in general, and often only one order even has matching shapes. Stacking layers just chains the products, $\hat{y} = g(g(XW^{(1)}+b^{(1)})W^{(2)}+b^{(2)})\ldots$, and associativity is exactly why the whole forward pass can be read left-to-right as "multiply, add, squash, repeat".

Python · NumPy ▶ runnable in the app

```python
import numpy as np
A = np.array([[1, 2, 3],
              [4, 5, 6]])          # shape (2, 3)
B = np.array([[1, 0],
              [0, 1],
              [1, 1]])             # shape (3, 2)
print("A.shape =", A.shape, "  B.shape =", B.shape)
C = A @ B                          # inner dims 3 and 3 match -> (2, 2)
print("A @ B =\n", C)
print("result shape =", C.shape)
# a single dot product is one neuron's weighted sum
print("dot([1,2,3],[4,5,6]) =", np.dot([1, 2, 3], [4, 5, 6]))
```

Matrices as linear maps, and multiplication as composition

A matrix has a second meaning, in addition to "a stack of neuron weights": it *is a function*. The product $\mathbf{y} = A\mathbf{x}$ sends every input vector to an output, and the map is **linear** — it fixes the origin and respects scaling and addition, $A(\alpha\mathbf{x} + \beta\mathbf{z}) = \alpha A\mathbf{x} + \beta A\mathbf{z}$. Geometrically A rotates, reflects, scales and shears space. Its columns are where it sends the standard basis vectors, so the unit square is mapped to the parallelogram they span, with area scaled by $|\det A|$. This says what matrix multiplication *is*: **function composition**. The product AB is the single matrix that applies B first and then A, since $(AB)\mathbf{x} = A(B\mathbf{x})$. That is why order matters (AB ≠ BA in general), and why a deep network — a composition of linear maps with nonlinearities placed between them to stop them collapsing into one matrix (chapter 3) — is precisely a chain of such products.

Worked example — a 2×2 product is a composition

Let $A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ (a shear) and $B = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$ (a 90° rotation). Multiplying row-by-column, $AB = \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}$ while $BA = \begin{bmatrix} 0 & -1 \\ 1 & 2 \end{bmatrix}$ — different matrices, because composition order matters. Check it on $\mathbf{x} = [1, 0]$: B rotates it to [0, 1], then A shears that to [2, 1], and $(AB)\mathbf{x} = [2, 1]$ in a single step.

Transpose, identity, inverse

The **transpose** A^T flips a matrix across its diagonal: rows become columns, so a shape (m, n) becomes (n, m) and $(A^T)_{ij} = A_{ji}$. Transpose is how a matrix reverses direction, and it reverses inside a product: $(AB)^T = B^T A^T$. This is not just a minor detail — it is the whole reason a backward pass looks like a forward pass "run in reverse". In chapter 8 the layer that computes $Z = XW$ on the way forward sends gradient back with $\delta_{in} = \delta_{out} W^T$, and accumulates its own weight gradient as $X^T \delta$. Every transpose in backprop is a forward matmul turned around by this rule.

The **identity matrix** I is the matrix version of the number 1: a square matrix with 1's on the diagonal and 0's elsewhere, satisfying $AI = IA = A$. The **inverse** A^{-1} is the matrix version of a reciprocal — the matrix with $A\,A^{-1} = A^{-1}A = I$ — and it exists only for square matrices whose columns are linearly independent (nonzero determinant). Inverses matter in theory (the normal equations of linear regression solve $(X^T X)^{-1} X^T y$ in closed form), but neural networks almost never compute them. Inverting a large matrix costs on the order of n^3 operations and is numerically unstable, so we minimize the loss step by step with gradient descent (chapter 7) instead of solving it exactly.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
A = np.array([[1., 2., 3.],
              [4., 5., 6.]])        # (2, 3)
print("A.T =\n", A.T, "\nshape", A.shape, "->", A.T.shape)
# transpose reverses a product: (A @ B).T == B.T @ A.T
B = np.array([[1., 0.], [0., 1.], [1., 1.]])
print("(A@B).T == B.T@A.T :", np.allclose((A @ B).T, B.T @ A.T))
# inverse undoes a square matrix: M @ inv(M) = I
M = np.array([[1., 2.], [3., 4.]])
print("M @ inv(M) =\n", np.round(M @ np.linalg.inv(M), 6))
```

Linear independence, span, and rank

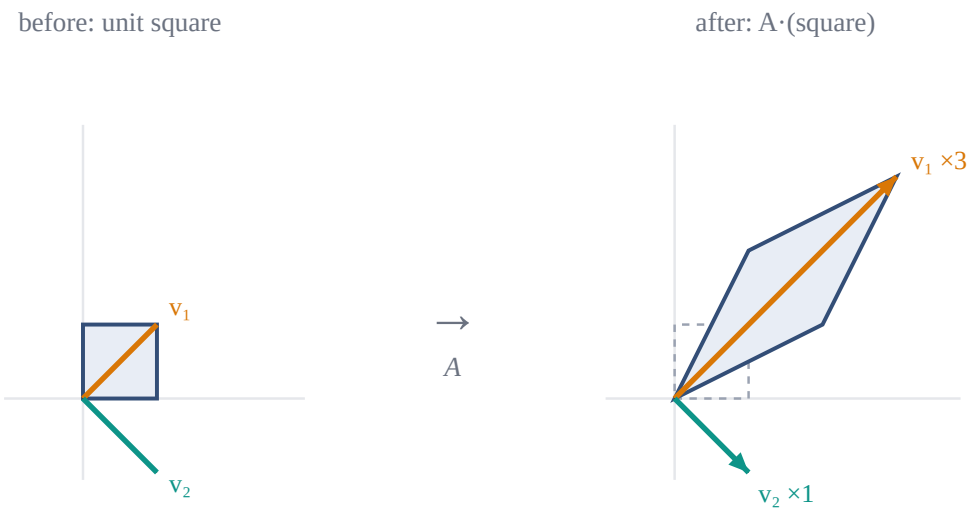
A set of vectors is **linearly independent** when none of them is a weighted sum of the others — none is redundant. Their **span** is the set of all such combinations, the subspace they together reach. The **rank** of a matrix is the number of independent columns (always equal to the number of independent rows): the dimension of the space its columns span. A shape-(m, n) matrix has rank at most min(m, n). At that maximum it is **full rank**. Below that, it is **rank-deficient**, collapsing some input directions to zero. Rank is the true measure of what a matrix does. A square matrix is invertible exactly when it is full rank (nonzero determinant, independent columns), which ties this section back to the last one. And a rank-r weight matrix, no matter how large, can only produce outputs in an r-dimensional subspace, so deliberately reducing rank (made precise by the SVD below) is behind both PCA (chapter 19) and LoRA (chapter 22).

Eigenvalues and eigenvectors

When a matrix acts on a vector it usually changes both its length and its direction. A few special directions change only in length — the map stretches or flips them without ever rotating them off their own line. These are the **eigenvectors v**, and their stretch factors are the **eigenvalues** λ:

$$A\mathbf{v} = \lambda\mathbf{v} \quad (\mathbf{v} \neq \mathbf{0})$$

You find them by solving $\det(A - \lambda I) = 0$ for the eigenvalues, then $(A - \lambda I)\mathbf{v} = \mathbf{0}$ for each direction. An eigenvalue is the factor by which the matrix stretches along its natural axis, and that factor controls *repeated* multiplication: applying A many times scales each eigen-direction by λ each time, so directions with $|\lambda| > 1$ grow and those with $|\lambda| < 1$ shrink. That is the linear-algebra core of exploding and vanishing gradients (chapter 10). A deep stack multiplies signals by many matrices, and they survive only when the relevant eigenvalues and singular values are near 1. Symmetric matrices, such as the covariance matrices and Hessians we see often, are the easy case: they always have real eigenvalues and a full set of orthogonal eigenvectors.



The matrix $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ maps the unit square (left) to a parallelogram (right). Its eigenvectors are the two directions that only scale: the diagonal $\mathbf{v}_1 = [1, 1]$ is stretched ×3 (its eigenvalue), while the perpendicular $\mathbf{v}_2 = [1, -1]$ is left unchanged (×1). Every other vector is both stretched and rotated.

Worked example — eigenvalues of a 2×2

For the symmetric $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ the characteristic equation is $(2 - \lambda)^2 - 1 = \lambda^2 - 4\lambda + 3 = (\lambda - 1)(\lambda - 3) = 0$, so $\lambda = 3$ and $\lambda = 1$. Solving $(A - 3I)\mathbf{v} = \mathbf{0}$ gives $\mathbf{v} \propto [1, 1]$, the diagonal, stretched ×3. Solving $(A - I)\mathbf{v} = \mathbf{0}$ gives $\mathbf{v} \propto [1, -1]$, left unchanged. Those two orthogonal directions and their stretch factors 3 and 1 completely describe the map — exactly the transformation drawn above.

The singular value decomposition

Eigenvectors require a square matrix and do not always behave nicely. The **singular value decomposition** (SVD) is the general replacement: *every* matrix A of shape (m, n) factors as

$$A = U \Sigma V^T$$

with U (m×m) and V (n×n) **orthogonal** — pure rotations and reflections — and Σ (m×n) diagonal, holding the non-negative **singular values** $\sigma_1 \geq \sigma_2 \geq \ldots \geq 0$. The geometry is the key insight: because U and V only rotate and Σ only scales the axes, *any* linear map is a **rotate → scale → rotate**. Push a unit sphere through A and you get an ellipsoid whose semi-axis lengths are the singular values. The largest, σ_1, is the map's greatest stretch (its operator norm), and the count of nonzero σ's is exactly the rank. For a symmetric positive-definite matrix the SVD and the eigendecomposition are the same, so the singular values of our $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ are 3 and 1.

The SVD also gives the best low-rank approximation of A: keep the r largest singular values and zero the rest, and the **Eckart–Young theorem** proves the result is the closest rank-r matrix to A in both Frobenius and operator norm. That single fact is behind much of deep learning. **PCA** is the SVD of mean-centered data, keeping the top directions of variance — the linear counterpart of the autoencoders of chapter 19. **LoRA** (chapter 22) freezes a huge pretrained weight matrix and learns only a low-rank update — the product of two thin matrices. It is cheap exactly because a good adaptation lives in a low-rank subspace. And zeroing the smallest singular values is a standard way to compress and remove noise, since those σ's usually carry noise, not signal.

The two decompositions — and the matmul and norms that lead to them — are one-liners in NumPy:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.random.seed(0)

# matrix multiplication is composition; order matters
A = np.array([[1., 2.], [0., 1.]])     # a shear
B = np.array([[0., -1.], [1., 0.]])    # a 90-degree rotation
```

```
print("A @ B =\n", (A @ B).astype(int))    # [[2 -1] [1 0]]
print("B @ A =\n", (B @ A).astype(int))    # [[0 -1] [1 2]]  -> A@B != B@A

# vector and matrix norms
v = np.array([2., 2., 1.])
print("L1, L2, Linf =",
      np.linalg.norm(v, 1), np.linalg.norm(v, 2), np.linalg.norm(v, np.inf))
# 5.0 3.0 2.0
M = np.array([[2., 1.], [1., 2.]])
print("Frobenius =", round(float(np.linalg.norm(M, 'fro')), 4))    #
sqrt(10) = 3.1623

# eigen-decomposition: symmetric M has real eigenvalues 3 and 1
vals, vecs = np.linalg.eig(M)
print("eigenvalues =", np.round(vals, 4))
print("eigenvectors (columns) =\n", np.round(vecs, 4))

# singular value decomposition  A = U @ diag(S) @ Vt
U, S, Vt = np.linalg.svd(M)
print("singular values =", np.round(S, 4))                  # [3. 1.] ==
|eigenvalues|
print("reconstructs M:", np.allclose(U @ np.diag(S) @ Vt, M))   # True
```

The Hadamard product and broadcasting

Two operations combine arrays *elementwise*, and confusing them with matrix multiplication is a common mistake. The **Hadamard product** $\odot$ multiplies matching-shape arrays entry by entry:

$$(A \odot B)_{ij} = A_{ij}B_{ij}$$

It is how an activation's derivative is applied: in backprop the elementwise factor $g'(z)$ is included as $\delta = (\delta_{out}W^{T}) \odot g'(z)$ (chapter 8), and dropout (chapter 12) is a Hadamard product with a random 0/1 mask. In NumPy the plain `*` is Hadamard, and `@` is matrix multiplication — they are two different operations.

Broadcasting is NumPy's rule for combining arrays of different but compatible shapes by stretching size-1 axes to fit, without copying memory. Adding a length-k bias row to an (m, k) batch adds it to every one of the m rows — that is broadcasting, and it is why the bias in Z =

XW + **b** "just works" for any batch size. The rule: align shapes on the right. Two axes are compatible if they are equal or one of them is 1. A missing axis counts as 1.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
A = np.array([[1., 2.], [3., 4.]])
B = np.array([[10., 20.], [30., 40.]])
print("Hadamard A*B =\n", A * B)      # elementwise, NOT matrix product
print("matmul   A@B =\n", A @ B)      # for contrast
# broadcasting: one bias per feature added to every row
X = np.array([[1., 2., 3.],
              [4., 5., 6.]])          # (2, 3): two samples in rows
b = np.array([10., 20., 30.])         # (3,): stretched across both rows
print("X + b =\n", X + b)
col = np.array([[100.], [200.]])      # (2, 1): stretched across columns
print("X + col =\n", X + col)
```

A whole layer, and the batch-as-rows convention

We can now build a real layer and follow how the shapes change. A batch X of shape (m, n) meets weights W of shape (n, k) and a length-k bias. The result is (m, k), and an elementwise activation leaves the shape unchanged.

Worked example — one layer by hand

Input row **x** = [1, 2, 3]. Weights W = $\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$ (so neuron 1's weights are the first column [1, 0, 1] and neuron 2's are [0, 1, 1]). Bias **b** = [0.5, −1].

1. Neuron 1: **x**·[1, 0, 1] = 1·1 + 2·0 + 3·1 = 4, then add the bias 0.5 → z_1 = **4.5**.
2. Neuron 2: **x**·[0,1,1] = 0 + 2 + 3 = 5, then + (−1) = **4.0**.
3. So **z** = **x**W + **b** = [4.5, 4.0], shape (1, 2) — one number per neuron. A ReLU activation would leave both positive values unchanged: **a** = [4.5, 4.0].

Every layer in the app is this same three-step arithmetic, only with the batch added as extra rows.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)
X = rng.randn(4, 3)                  # batch of 4 samples, 3 features each
W = rng.randn(3, 2)                  # 3 inputs -> 2 neurons (one column
each)
b = np.array([0.5, -1.0])           # one bias per neuron
Z = X @ W + b                        # (4,3) @ (3,2) + (2,) -> (4,2)
print("Z shape:", Z.shape)
print("Z =\n", np.round(Z, 3))
A = np.maximum(0.0, Z)               # ReLU, elementwise -> same shape
print("after ReLU:\n", np.round(A, 3))
```

Why store the batch as *rows*? Because then the sample axis is axis 0 and passes unchanged through every `X @ W`: each row is one independent example, the bias broadcasts naturally across rows, and summing gradients "over the batch" (chapters 7–8) is a sum over axis 0. It also matches how PyTorch, TensorFlow and JAX lay out data, so the mental model transfers directly to real code. Columns-as-samples works too, but then every equation needs an extra transpose and the convention works against the tools.

Norms: measuring the size of a vector

A **norm** assigns a non-negative length to a vector. Two of them appear very often. The **L2 (Euclidean) norm** is ordinary straight-line length. The **L1 norm** is the sum of absolute values ("taxicab" distance):

$$\|\mathbf{x}\|_2 = \sqrt{\Sigma_i \, x_i^2}\,, \quad \|\mathbf{x}\|_1 = \Sigma_i \, |x_i|$$

Both are special cases of the L_p norm $\|\mathbf{x}\|_p = (\Sigma_i \, |x_i|^p)^{1/p}$. Norms are how we measure the size of things we want small. The L2 norm of an error vector is the Euclidean distance behind mean-squared-error (chapter 6). The L2 norm of the weights drives weight decay, and the L1 norm

drives sparsity-inducing regularization (chapter 12). The norm of the gradient tells you how large a step gradient descent is about to take, and clipping it controls exploding gradients (chapters 10 and 15). The classic 3–4–5 right triangle shows the L2 norm clearly: $\|[3, -4]\|_2 = \sqrt{9 + 16} = 5$, while its L1 norm is 3 + 4 = 7.

Two more complete the toolkit. The **L∞ (maximum) norm** $\|\mathbf{x}\|_\infty = \max_i |x_i|$ is the $p \to \infty$ limit of the family and gives the largest-magnitude entry — what value-clipping controls. For a matrix the usual choice is the **Frobenius norm** $\|A\|_F = \sqrt{\Sigma_{ij} A_{ij}^2}$, the L2 norm of all entries at once, which also equals $\sqrt{\Sigma_k \sigma_k^2}$ — the root-sum-square of the singular values. It is what weight decay shrinks on a weight matrix, and the natural size of a gradient tensor.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
v = np.array([3., -4.])
print("L1 =", np.sum(np.abs(v)))          # 3 + 4 = 7
print("L2 =", np.sqrt(np.sum(v**2)))      # sqrt(9 + 16) = 5
print("np.linalg.norm(v, 1) =", np.linalg.norm(v, 1))
print("np.linalg.norm(v, 2) =", np.linalg.norm(v, 2))
# L2 length of each ROW of a matrix (per-sample sizes)
M = np.array([[3., 4.], [5., 12.]])
print("row L2 norms:", np.linalg.norm(M, axis=1))   # [5, 13]
```

Positive-definiteness and the Hessian

A symmetric matrix A defines a **quadratic form** $\mathbf{x}^T A\mathbf{x}$, one number for every vector. A is **positive-definite** when that number is strictly positive for every nonzero **x**, and **positive-semidefinite** (PSD) when it is only non-negative — in other words, when all of A's eigenvalues are > 0, or all ≥ 0. The form is then a bowl curving up in every direction, or flat along some. This is the language of curvature, and curvature is second derivatives. The **Hessian** H of a scalar loss L is the symmetric matrix of second partial derivatives $H_{ij} = \partial^2 L / \partial\theta_i \partial\theta_j$ (Appendix B). At a minimum the gradient is zero and H is PSD. A negative eigenvalue marks a saddle point that the optimizer can still slide down. The Hessian's eigenvalues are the curvatures along its eigenvectors, and their spread — the condition number $\lambda_{max} / \lambda_{min}$ — sets the difficulty. A

large ratio makes gradient descent zig-zag down a narrow valley. This is the problem that momentum and Adam (chapter 9) exist to cure. The Gram matrices X^TX behind least squares are PSD for the same reason, which is what makes those problems convex.

In the real world

The reason a GPU trains a network so much faster than a CPU is exactly the topic of this appendix: a graphics processor is a machine built to do huge matrix multiplications in parallel, and cuBLAS/cuDNN — the libraries under PyTorch and TensorFlow — spend almost all their time in the $C_{ij} = \Sigma_k A_{ik}B_{kj}$ loop. A modern large language model is billions of parameters arranged as weight matrices, and generating one token is one long sequence of the matrix–vector products defined here. Becoming comfortable with shapes, transpose and broadcasting is not just academic. It is how you read, debug and reason about every model you will build.

Exercises

1. A is 2×3 and B is 3×4. What is the shape of AB? Of B^TA^T? Can you form A + B? MATH

2. Compute the dot product of [2, −1, 3] and [1, 4, 2], then the L1 and L2 norms of [2, −1, 3]. MATH

3. Write NumPy that builds a reproducible random batch of 5 samples with 3 features, runs it through a 3→4 linear layer with a bias, and prints the output shape. CODE

4. In NumPy, `A * B` and `A @ B` can both be legal yet give different answers. Explain the difference and give a case where only one is defined. CONCEPT

5. Why does this book store a batch as rows (m×n) rather than columns (n×m)? Give two concrete consequences. CONCEPT

6. Compute the cosine similarity of **a** = [1, 0, 1] and **b** = [1, 1, 0], and the angle between them. MATH

7. Find the eigenvalues of $A = \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix}$. MATH

8. LoRA (chapter 22) replaces a 1000×1000 weight update with a rank-8 factorization. How many numbers does the full update store, how many does the rank-8 version store, and what fraction is that? CONCEPT

APPENDIX

Appendix B · Calculus & the chain rule

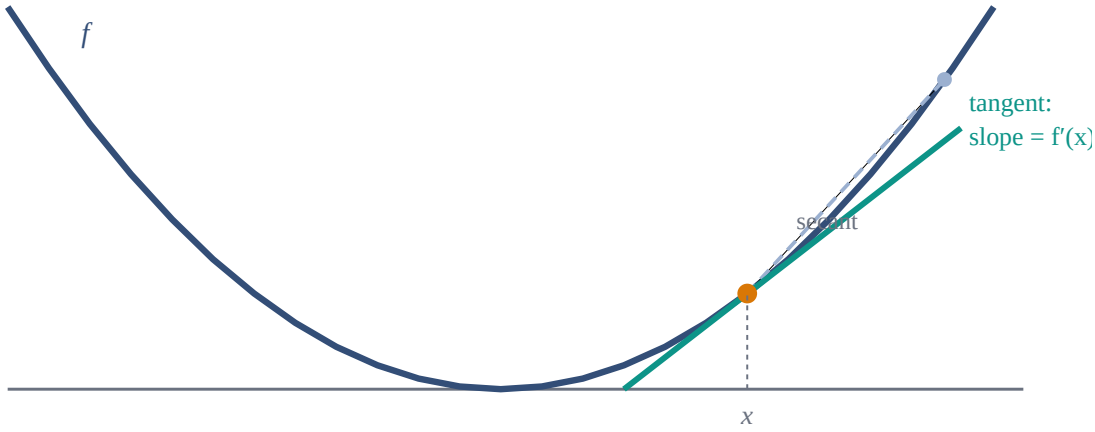
> *Learning, in this book, means changing parameters a little in the direction that reduces the loss. And "the direction that reduces the loss" is a statement about **derivatives**. This appendix builds the calculus a neural network actually uses: the derivative as a slope, the few rules you need, partial derivatives and the gradient, why the gradient points uphill and crosses the level sets, and above all the **chain rule**. We start with the single-variable form, then the multivariable, fan-out form that is the entire mathematical content of backpropagation. From there we assemble the **Jacobian** and **Hessian**, the **Taylor expansion** that turns them into gradient descent and Newton's method, and a short dictionary of the gradients you meet everywhere else. It is the reference behind gradient descent (chapter 7) and backpropagation (chapter 8). It builds on the vectors of Appendix A, and is used in the autograd engine of Appendix G, with runnable NumPy that checks every derivative against the definition.*

The derivative as slope

The **derivative** of a function at a point is the slope of its graph there. It is the rate at which the output changes per unit change of input. Formally it is the limit of the slope of a shrinking secant line:

$$f'(x) = \frac{dy}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

Read it as a sensitivity: if $f'(x) = 3$, then near x a tiny change dx to the input produces about $3 \cdot dx$ of change in the output. If $f'(x) = 0$ the function is momentarily flat: a peak, a valley, or a saddle. This is exactly the condition training drives toward, a point where the loss cannot be reduced by any small step. The sign tells you which way is downhill. The magnitude tells you how steep.



The derivative is the slope of the tangent (teal). It is the limit of secant slopes (gray dashed) as the second point slides into the first. This gives the ratio $\Delta f/\Delta x$ as $\Delta x \to 0$. A finite difference computes exactly such a secant slope with a small but nonzero step.

The rules you need

You rarely take limits by hand. A few rules cover everything in this book. With u, v functions of x and c constant:

constant	$d/dx\,(c) = 0$
power	$d/dx\,(x^n) = n \cdot x^{n-1}$
sum	$(u + v)' = u' + v'$ (and $(c \cdot u)' = c \cdot u'$)
product	$(u \cdot v)' = u'v + uv'$
quotient	$(u/v)' = (u'v - uv')/v^2$
chain	$(f(g(x)))' = f'(g(x)) \cdot g'(x)$ — the most important rule in this appendix

Three special derivatives recur because they are the activations and losses of later chapters: $d/dx\ e^x = e^x$; $d/dx \ln x = 1/x$; and for the sigmoid $\sigma(z) = 1/(1 + e^{-z})$, the simple $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ (chapter 1). The ReLU $\max(0, x)$ has derivative 1 for $x > 0$ and 0 for $x < 0$. It is undefined at the kink, where we simply pick 0 (chapter 4).

Partial derivatives and the gradient

A loss depends on millions of parameters, not one. A **partial derivative** $\partial f/\partial x_i$ is the ordinary derivative with respect to one variable while all others are held fixed. It is the sensitivity of the output to changing that one input alone. Collecting all the partials into a vector gives the **gradient**:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right)$$

The gradient is the multivariable slope: one number per parameter, each saying how the loss responds to that parameter in isolation. For a network, ∇L is the list of "blame" values backpropagation computes, one per weight. Gradient descent (chapter 7) steps every weight against its own partial simultaneously.

The gradient points uphill

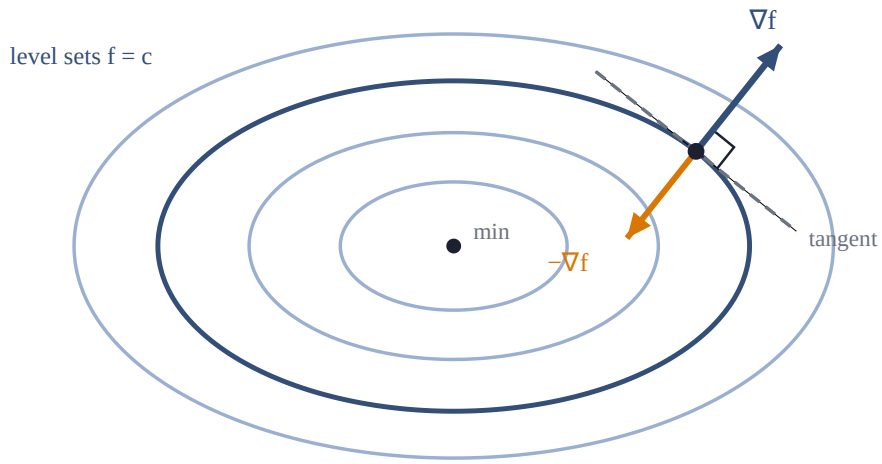
The gradient is not just a bundle of slopes. As a vector it has a direction, and that direction is the one of **steepest ascent**. The rate of change of f as you move a unit step in direction **u** is the directional derivative $\nabla f \cdot \mathbf{u}$, a dot product (Appendix A). By the alignment property of the dot product this is largest exactly when **u** points the same way as ∇f. So:

$$+\nabla f = \text{steepest uphill}, \quad -\nabla f = \text{steepest downhill}$$

This is the entire justification for the update rule $\theta \leftarrow \theta - \eta \nabla L$: to *reduce* a loss, step in the direction opposite its gradient. The learning rate η sets how far. At a minimum $\nabla L = 0$ and the update stops moving. This is the flat-slope condition from the first section, now in many dimensions.

The same dot product reveals a second, geometric fact: the gradient is **perpendicular to the level sets** of f. These are the contours $\{\mathbf{x} : f(\mathbf{x}) = c\}$ on which f is constant. A tiny step **t** along a contour leaves f unchanged, so $\nabla f \cdot \mathbf{t} = 0$. A vector orthogonal to every tangent direction is normal to the surface. So ∇f points straight across the contours, and descent steps cross them at

right angles. This is also why a stretched, ravine-shaped loss makes plain gradient descent zig-zag (chapter 7): the steepest direction aims at the nearest contour, not at the minimum, unless the contours are circular.



Level sets of a quadratic bowl. At any point the gradient ∇f is perpendicular to the contour and points to steepest ascent. The descent step $-\nabla f$ (amber) is its opposite, crossing the contour at the same right angle. Because the contours are stretched, neither arrow aims straight at the minimum. This is the reason plain gradient descent zig-zags (chapter 7).

Numeric derivatives: finite differences

You can approximate any derivative without calculus by using a small but nonzero h. This is a **finite difference**. The **central difference** is far more accurate than the one-sided version, with error shrinking like h^2 instead of h:

$$f'(x) \approx \frac{f(x + h) - f(x - h)}{2h} \quad (\text{error} \propto h^2)$$

This is not just a backup: it is the standard way to **check** a hand-derived or hand-coded gradient. Backprop (chapter 8) is exact but error-prone to implement, so we compare each analytic gradient against a central difference of the loss. Agreement to several decimals is strong evidence the derivation is right. The app includes dozens of such gradient-checking tests for exactly this reason.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
f = lambda x: x**3 - 2*x            # analytic derivative: 3x^2 - 2
def deriv(f, x, h=1e-5):
    return (f(x + h) - f(x - h)) / (2*h)   # central difference
x0 = 1.5
print("numeric  f'(1.5) =", round(deriv(f, x0), 6))
print("analytic 3x^2-2  =", 3*x0**2 - 2)
# the one-sided forward difference is cruder (error ~ h, not h^2)
fwd = (f(x0 + 1e-5) - f(x0)) / 1e-5
print("forward diff     =", round(fwd, 6))
```

The same idea handles many variables: slightly change one coordinate at a time to estimate each partial, and stack them into a numeric gradient.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
def f(v):
    x, y = v
    return x**2 + 3*x*y + y**2       # true gradient: (2x + 3y, 3x + 2y)
def num_grad(f, v, h=1e-5):
    v = np.asarray(v, float)
    g = np.zeros_like(v)
    for i in range(len(v)):
        step = np.zeros_like(v); step[i] = h
        g[i] = (f(v + step) - f(v - step)) / (2*h)   # partial i, rest fixed
    return g
p = np.array([1.0, 2.0])
print("numeric  grad:", np.round(num_grad(f, p), 6))
print("analytic grad:", np.array([2*1 + 3*2, 3*1 + 2*2]))   # (8, 7)
```

The chain rule is the engine of backprop

The single most important rule for neural networks is the **chain rule**: the derivative of a composition is the product of the derivatives of its parts. If y = f(u) and u = g(x), then

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

Sensitivities *multiply* along a chain of dependencies. A neural network *is* a long composition. Inputs feed weighted sums, which feed activations, which feed the next layer, which feeds the loss. So the derivative of the loss with respect to an early weight is a product of local derivatives all the way down the graph. Backpropagation (chapter 8) is nothing more than computing these products efficiently, from the loss backward, reusing shared sub-products so no path is computed twice.

Worked example — a two-link chain, forward and backward

Let $g(x) = 3x + 1$ and $f(u) = u^2$, so $y = f(g(x)) = (3x + 1)^2$. Evaluate the derivative at $x = 2$, like a forward and backward pass.

1. **Forward.** $u = g(2) = 3 \cdot 2 + 1 = \mathbf{7}$. Then $y = f(7) = 7^2 = \mathbf{49}$.
2. **Local derivatives.** $g'(x) = 3$, so $du/dx = \mathbf{3}$. $f'(u) = 2u = 14$, so $dy/du = \mathbf{14}$.
3. **Backward (chain).** $dy/dx = (dy/du) \cdot (du/dx) = 14 \cdot 3 = \mathbf{42}$. A change dx at the input becomes $3 \cdot dx$ at u, then 14× that at the output. The sensitivities compound to 42.
4. **Check.** $y(2.001) = (3 \cdot 2.001 + 1)^2 = 7.003^2 = 49.042009$, so the slope $\approx (49.042009 - 49)/0.001 = \mathbf{42.009}$ ✓ (a central difference gives 42.000).

That "message × local derivative, passed down the chain" is precisely the δ recurrence of backpropagation (chapter 8). The only new thing there is that the messages are vectors and the products become matrix multiplications.

Python · NumPy ▶ runnable in the app

```
import numpy as np
g = lambda x: 3*x + 1              # inner function
f = lambda u: u**2                 # outer function
comp = lambda x: f(g(x))           # y = (3x + 1)^2
x0 = 2.0
# analytic chain rule: dy/dx = f'(g(x)) * g'(x) = 2*(3x+1) * 3
analytic = 2*(3*x0 + 1) * 3
```

```
numeric = (comp(x0 + 1e-6) - comp(x0 - 1e-6)) / (2e-6)
print("analytic chain rule:", analytic)          # 42
print("numeric  derivative:", round(numeric, 4))
```

Now the same rule on a real neuron, tying together chapters 1, 6 and 8. A single-input neuron computes $z = wx + b$, $a = \sigma(z)$, with squared-error loss $L = \tfrac{1}{2}(a - y)^2$. The chain rule passes through three links:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w} = (a - y) \cdot \sigma'(z) \cdot x, \quad \sigma'(z) = a(1 - a)$$

Python · NumPy ▶ runnable in the app

```
import numpy as np
sig = lambda z: 1/(1 + np.exp(-z))
x, w, b, y = 1.0, 0.5, 0.0, 1.0
def loss(w):
    a = sig(w*x + b)
    return 0.5*(a - y)**2
z = w*x + b; a = sig(z)
analytic = (a - y) * a*(1 - a) * x        # chain rule, three links
numeric = (loss(w + 1e-6) - loss(w - 1e-6)) / (2e-6)
print("a =", round(a, 6), "  loss =", round(loss(w), 6))
print("analytic dL/dw =", round(analytic, 6))
print("numeric  dL/dw =", round(numeric, 6))
```

The multivariable chain rule and fan-out

Backpropagation needs one more version of the chain rule. This is where a variable influences the output through *several* paths at once. If f depends on intermediates $u_1, \ldots, u_k$, each a function of t, the sensitivities along every path **add**:

$$\frac{df}{dt} = \Sigma_i \frac{\partial f}{\partial u_i} \cdot \frac{du_i}{dt}$$

Multiply along each path, then sum across paths. In vector form, if $\mathbf{u} = \mathbf{u}(\mathbf{x})$ and f is scalar, $\nabla_{\mathbf{x}} f = J^{T} \nabla_{\mathbf{u}} f$, where $J = \partial \mathbf{u} / \partial \mathbf{x}$ is the Jacobian below. This is the upstream gradient pulled back through a transpose. That transpose-multiply is the **vector–Jacobian product** at the core of

reverse-mode autodiff (Appendix G).

The sum across paths is the **fan-out** rule: a neuron whose activation feeds many units in the next layer gets a gradient back from each, and its total gradient is their sum. This is the Σ_k in backpropagation's error recurrence $\delta^{(l)}_j = \Sigma_k \, \delta^{(l+1)}_k \cdot \partial z^{(l+1)}_k / \partial z^{(l)}_j$ (chapter 8). A quick numeric check: with $a = x^2$, $b = 3x$ and $f = a \cdot b$, the input x reaches f two ways, so $df/dx = (\partial f/\partial a)(da/dx) + (\partial f/\partial b)(db/dx) = b \cdot 2x + a \cdot 3 = 9x^2$. At $x = 2$ that is $6 \cdot 4 + 4 \cdot 3 = \mathbf{36}$, matching $9x^2 = 36$. Follow only one path and you get half the answer. Forgetting to *add* at a fan-out is the most common backward-pass bug, which is why an autograd node accumulates its incoming gradient rather than overwriting it (Appendix G).

The Jacobian (brief)

When a function maps a vector to a vector, $f : \mathbb{R}^n \to \mathbb{R}^m$, its full first derivative is the **Jacobian**. This is an $m \times n$ matrix of every output's partial with respect to every input:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

For a scalar-valued function ($m = 1$) the Jacobian is a single row, and the gradient ∇f is just its transpose. The chain rule generalizes cleanly: the Jacobian of a composition is the *product* of the Jacobians, $J_{f \circ g} = J_f \, J_g$. A deep network is a stack of such maps, so its overall Jacobian is a long chain of matrix products. And because the final loss is scalar, backprop multiplies these Jacobians from the left as cheap vector–matrix products (reverse-mode automatic differentiation, Appendix G) instead of forming the giant matrices explicitly. That efficiency is why we differentiate backward from the loss, not forward from the inputs. The softmax's Jacobian $\partial p_k / \partial z_i = p_k(\delta_{ki} - p_i)$ that collapses cross-entropy's gradient to $p - y$ (chapter 6) is one concrete instance.

The Hessian and curvature

The gradient is the first derivative. The **Hessian** is the second. For a scalar function of n variables, it is the n×n matrix of all second partials, i.e. the Jacobian of the gradient:

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \, \partial x_j}, \quad H = \nabla^2 f$$

With continuous second derivatives the order of differentiation does not matter ($\partial^2 f / \partial x \partial y = \partial^2 f / \partial y \partial x$), so H is **symmetric**, with real eigenvalues and orthogonal eigenvectors (Appendix A). Those eigenvalues are the **curvatures** of f: along the eigenvector with eigenvalue λ the graph bends like a parabola of second derivative λ. Positive λ curves up (a valley wall), negative λ curves down. Their ratio, the **condition number**, says how stretched the bowl is. A large one makes gradient descent zig-zag down a ravine and is exactly what momentum and Adam counter (chapter 9).

Taylor expansion: a local model

Gradient and Hessian are the ingredients of the **Taylor expansion**, which replaces a function near a point by a polynomial matching its value, slope, and curvature. In one variable, stepping by Δ:

$$f(x + \Delta) \approx f(x) + f'(x)\,\Delta + \frac{1}{2} f''(x)\,\Delta^2$$

and in many variables, stepping by a vector **p**:

$$f(\mathbf{x} + \mathbf{p}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^{\top}\mathbf{p} + \frac{1}{2}\,\mathbf{p}^{\top} H \mathbf{p}$$

These two truncations are the two major optimization algorithms. Keep only the **first-order** term and the fastest way to decrease f is to move opposite the gradient. This is **gradient descent**, $\theta \leftarrow \theta - \eta \nabla L$, with η representing how far we trust the linear model (chapter 7). Keep the **second-order** term and setting the quadratic model's gradient $\nabla f + H\mathbf{p}$ to zero gives **Newton's method**,

$$\mathbf{p} = -H^{-1}\nabla f$$

a jump straight to the bottom of the local parabola in one step. Newton needs far fewer steps, but H is n×n, which is unaffordable at millions of parameters. So deep learning keeps first-order descent and lets adaptive optimizers approximate the curvature cheaply (chapter 9).

Convexity and the second-derivative test

Curvature also classifies flat spots. A **stationary point** $\mathbf{x}^*$ has $\nabla f = 0$, the update rule's fixed point, but it may be a minimum, a maximum, or a saddle. The **second-derivative test** reads it from the Hessian there: positive definite (all eigenvalues > 0) ⇒ local **minimum**; negative definite (all < 0) ⇒ local **maximum**; mixed signs ⇒ **saddle**. In one dimension this is the familiar $f''(x^*) > 0 \Rightarrow$ min, $f''(x^*) < 0 \Rightarrow$ max, $f''(x^*) = 0$ inconclusive.

A function is **convex** when it curves upward everywhere: $f'' \geq 0$ in one variable, or a positive-semidefinite Hessian (eigenvalues ≥ 0) in many. Convexity is the ideal case for optimizers: no bad local minima, so any point with $\nabla f = 0$ is a *global* minimum. Squared error in a linear model's weights and softmax cross-entropy in the logits (chapter 6) are convex. A deep network's loss is definitely **not**, full of saddle points that the noise of stochastic gradient descent and the momentum of chapter 9 move us past.

A gradient dictionary

A few gradients cover most of the functions you differentiate in this book (vectors in bold, Appendix A). With $\mathbf{x} \in \mathbb{R}^n$, a constant vector $\mathbf{w}$, and a matrix A:

linear $f = \mathbf{w}^T\mathbf{x}$	$\nabla_{\mathbf{x}} f = \mathbf{w}$. The gradient of a weighted sum is just its weights (a neuron's pre-activation, chapter 1).
quadratic $f = \tfrac{1}{2}\mathbf{x}^T A\mathbf{x}$	$\nabla f = \tfrac{1}{2}(A + A^T)\mathbf{x}$, which is $A\mathbf{x}$ when A is symmetric. The Hessian is A itself.
squared norm $f = \lVert\mathbf{x}\rVert^2 = \mathbf{x}^T\mathbf{x}$	$\nabla f = 2\mathbf{x}$ (the case $A = 2I$). This drives L2 weight decay (chapter 12).
log-sum-exp $f = \ln \Sigma_j e^{z_j}$	$\nabla f = \text{softmax}(\mathbf{z}) = \mathbf{p}$. Its Hessian $\text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^T$ is positive-semidefinite, so f is convex and yields the simple cross-entropy gradient $\mathbf{p} - \mathbf{y}$ (chapter 6).

Worked example — the gradient of a quadratic at a point

Take $f(\mathbf{x}) = \tfrac{1}{2}\,\mathbf{x}^{\mathsf{T}}A\mathbf{x}$ with the symmetric $A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$, evaluated at $\mathbf{x} = (1, -2)$.

1. **Gradient.** $\nabla f = A\mathbf{x} = (2\cdot 1 + 1\cdot(-2),\ \ 1\cdot 1 + 3\cdot(-2)) = (0, -5)$.
2. **Value.** $f = \tfrac{1}{2}\,\mathbf{x}^{\mathsf{T}}(A\mathbf{x}) = \tfrac{1}{2}\,(1\cdot 0 + (-2)(-5)) = \tfrac{1}{2}\cdot 10 = 5$.
3. **What it says.** The steepest *descent* step is $-\nabla f = (0, 5)$: hold x_1, increase x_2. The Hessian is A, eigenvalues ≈ 1.38 and 3.62. Both are positive, so f is a convex bowl whose only stationary point, the origin, is the global minimum.

To trust any of these formulas, **gradient-check** it. Compare the analytic gradient against a central difference, coordinate by coordinate. The cell below checks the quadratic above and log-sum-exp. Both agree to about ten digits.

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)
def grad_check(f, grad, x, h=1e-5):
    x = np.asarray(x, float)
    num = np.zeros_like(x)
    for i in range(x.size):
        e = np.zeros_like(x); e.flat[i] = h
        num.flat[i] = (f(x + e) - f(x - e)) / (2*h)   # central difference
    ana = grad(x)
    return np.max(np.abs(num - ana)) / (np.max(np.abs(ana)) + 1e-12)
# (1) quadratic  q(x) = 1/2 xᵀA x,  A symmetric  ->  grad = A x
A = np.array([[2.0, 1.0], [1.0, 3.0]])
q, q_grad = lambda x: 0.5 * x @ A @ x, lambda x: A @ x
x0 = np.array([1.0, -2.0])
print("quadratic  analytic", q_grad(x0), " rel.err %.1e" % grad_check(q,
q_grad, x0))
# (2) log-sum-exp  ->  grad = softmax
def lse(z):     m = z.max(); return m + np.log(np.exp(z - m).sum())
def softmax(z): e = np.exp(z - z.max()); return e / e.sum()
z0 = np.array([2.0, 1.0, 0.0])
```

```
print("softmax    analytic", np.round(softmax(z0), 4), " rel.err %.1e" %
grad_check(lse, softmax, z0))
print("both gradients confirmed to ~1e-10 relative error")
```

In the real world

Every modern deep-learning framework (PyTorch, TensorFlow, JAX) is essentially an **automatic differentiation** engine that applies exactly the chain rule of this appendix, one operation at a time, over a recorded computation graph (the very engine Appendix G builds from scratch). When you call `loss.backward()`, the library goes through the graph in reverse and multiplies local derivatives just as the worked examples do, accumulating at every fan-out, and gives you $\partial L/\partial \theta$ for millions of parameters at roughly the cost of one extra forward pass. Finite-difference checking, meanwhile, is a real debugging tool: research code that implements a custom gradient routinely gradient-checks it against a central difference before trusting it. This is the same technique the app uses on its own backprop.

Exercises

1. Differentiate $f(x) = x^4 - 3x^2 + 2x$ and evaluate $f'(1)$. MATH

2. Use the chain rule to differentiate $y = (2x^2 + 1)^3$, and evaluate at $x = 1$. MATH

3. Write a central-difference gradient checker and use it to verify the analytic gradient of $f(x, y) = x^2y$ at the point $(2, 3)$. CODE

4. Why does gradient descent step along $-\nabla L$ rather than $+\nabla L$, and why is a central difference preferred over a forward difference when checking gradients? CONCEPT

5. The sigmoid $\sigma(z) = 1/(1 + e^{-z})$ satisfies $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. Compute $\sigma(0)$ and $\sigma'(0)$, and explain what $\sigma'(0)$ says about learning. MATH

6. Use the multivariable chain rule to find dz/dt for $z = x^2 + y^2$ with $x = t^2$ and $y = 3t$, and evaluate at $t = 1$. MATH

7. For the quadratic $f(\mathbf{x}) = \tfrac{1}{2}\,\mathbf{x}^{T}A\mathbf{x}$ with $A = \begin{bmatrix} 4 & 0 \\ 0 & 1 \end{bmatrix}$, write ∇f and the Hessian H, find the stationary point, and classify it. MATH

8. Write the second-order Taylor expansion of $f(x) = x^2 - 4x + 7$ at $x = 0$, then show that one Newton step lands exactly on the minimum. MATH

APPENDIX

Appendix C · Probability & statistics

> *A network's outputs are best read as **probabilities**, and its training objective is best read as a statement about probability: choose the parameters that make the data we actually observed most likely. This appendix brings together the statistics that idea needs: the axioms, conditional probability and Bayes' theorem, random variables and expectation, the mean/variance vocabulary, the few distributions that appear everywhere, sampling and the central limit theorem, and the two principles of maximum likelihood and maximum-a-posteriori estimation. It then shows how one principle produces both mean-squared error and cross-entropy (chapter 6), why softmax is a probability distribution, how a prior becomes regularization (chapter 12), and how entropy and KL divergence underlie autoencoders (chapter 19) and generative models (chapter 21). Runnable NumPy makes each idea concrete. The companion Appendix D develops information theory in full.*

Sample space, events, and the axioms

Probability starts with a **sample space** Ω, which lists every outcome an experiment could produce: {H, T} for a coin, {1, …, 6} for a die, the real line for a noisy measurement. An **event** is any subset $A \subseteq \Omega$, such as "the die shows even" = {2, 4, 6}. A probability P assigns a number to each event under three axioms (Kolmogorov, 1933):

$$\text{(i) } P(A) \geq 0 \quad \text{(ii) } P(\Omega) = 1 \quad \text{(iii) } P(A \cup B) = P(A) + P(B) \text{ when } A \cap B = \varnothing$$

From these we get the complement rule $P(A^c) = 1 - P(A)$, $P(\varnothing) = 0$, and inclusion–exclusion $P(A \cup B) = P(A) + P(B) - P(A \cap B)$ for overlapping events. That is the whole foundation.

Conditional probability and Bayes' theorem

The probability of A *given* that B occurred is the **conditional probability**, and rearranging its definition gives the **product rule** $P(A \cap B) = P(A \mid B)\,P(B)$:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}, \quad P(B) > 0$$

Summing the product rule over a partition of causes $B_1, \ldots, B_n$ gives the **law of total probability** $P(E) = \Sigma_i P(E \mid B_i)\, P(B_i)$. Writing it both ways and dividing gives **Bayes' theorem**, which inverts a conditional. It turns "evidence given cause" into "cause given evidence":

$$P(H \mid E) = \frac{P(E \mid H) \cdot P(H)}{P(E)}, \quad P(E) = \Sigma_H P(E \mid H)\, P(H)$$

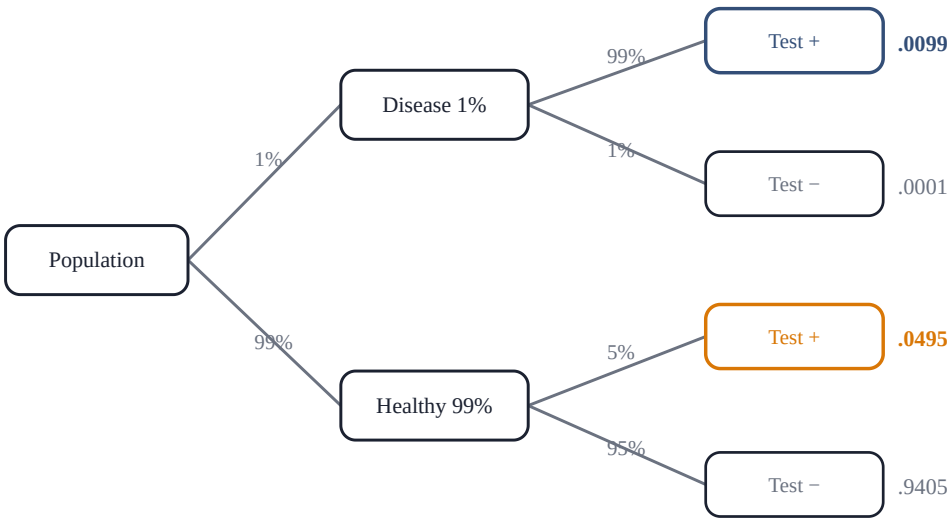
P(H) prior — belief in hypothesis H before the evidence

P(E | H) likelihood — how well H predicts the evidence E

P(E) evidence — the normalizer that makes the posterior a valid distribution

P(H | E) posterior — updated belief in H after seeing E

This is how a likelihood becomes a belief about parameters (MAP, below), and how classifier outputs are read as posteriors over classes. Its clearest lesson is how badly intuition handles rare events.



P(Disease | +) = .0099 ⁄ (.0099 + .0495) ≈ 0.167

Bayes as a method. A positive test's total mass is .0099 + .0495 = .0594. The true positives are only .0099, so P(disease | +) ≈ 0.167. The rare disease's few true positives (blue) are outnumbered by false positives (amber) drawn from the healthy majority.

Worked example — a positive test for a rare disease

A disease affects 1 person in 100. A test detects 99% of true cases (*sensitivity*) and correctly clears 95% of healthy people (*specificity* — a 5% false-positive rate). You test positive.

1. Prior: P(D) = 0.01, so P(healthy) = 0.99.
2. Likelihoods of a positive: P(+ | D) = 0.99 and P(+ | healthy) = 0.05.
3. Evidence: P(+) = 0.99·0.01 + 0.05·0.99 = 0.0099 + 0.0495 = 0.0594.
4. Posterior: P(D | +) = 0.0099 ⁄ 0.0594 ≈ **0.167**.

Even after a positive from a "99% accurate" test, the chance you are ill is only about **17%**: the 5% false positives among the healthy majority (0.0495) outnumber the true positives (0.0099) five to one. Ignoring the base rate — the *base-rate fallacy* — is why calibration and known prevalence matter whenever a model flags something rare (chapter 24).

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)
prev, sens, spec = 0.01, 0.99, 0.95        # prevalence, sensitivity, specificity
N = 200000
disease  = rng.binomial(1, prev, N).astype(bool)         # who is truly ill
pos_ill  = rng.binomial(1, sens, N).astype(bool)         # test + given ill
pos_well = rng.binomial(1, 1 - spec, N).astype(bool)     # test + given healthy
positive = np.where(disease, pos_ill, pos_well)
emp = disease[positive].mean()             # empirical P(disease | positive)
den = sens * prev + (1 - spec) * (1 - prev)   # law of total probability -> P(+)
print("evidence P(+)      =", round(den, 4))            # 0.0594
print("posterior (Bayes)  =", round(sens*prev/den, 4))  # 0.1667
print("posterior (sim)    =", round(float(emp), 4))     # ~0.17
```

Random variables and distributions

A **random variable** X is a quantity whose value is uncertain — the outcome of a coin flip, the noise on a measurement, the class of an image. Its behavior is described by a **distribution**. For a *discrete* variable, this is a probability mass p(x) attached to each possible value. For a *continuous* variable, it is a probability *density* p(x), whose area over an interval gives the probability of landing in it. Every distribution follows two rules — probabilities are non-negative, and they sum (or integrate) to 1:

$$\Sigma_x\, p(x) = 1 \;\text{(discrete)}, \quad \int p(x)\,dx = 1 \;\text{(continuous)}$$

Three named functions make this precise. The **probability mass function** (pmf) p(x) = P(X = x) is for discrete X. The **probability density function** (pdf) p(x) is for continuous X; it is a density, not a probability, so probabilities come from integrating it. The **cumulative distribution function** (cdf) F(x) = P(X ≤ x) is defined for either kind; it rises from 0 to 1, and its slope (or jumps) gives back the density (or masses).

In deep learning, the inputs, the labels, and the network's own predictions are all treated as random variables. Modeling means proposing a distribution and fitting its parameters to data.

Expectation, mean, variance, standard deviation

The **expectation** (or mean) E[X] is the probability-weighted average. It is the value you would get by averaging infinitely many samples. The **variance** measures spread as the expected squared distance from the mean, and the **standard deviation** is its square root, back in the original units:

$$\mu = E[X] = \Sigma_x\, x\,p(x), \quad \sigma^2 = E[(X-\mu)^2] = E[X^2] - \mu^2, \quad \sigma = \sqrt{\sigma^2}$$

Expectation is **linear**: E[aX + b] = aE[X] + b. This is why averaging a loss over a mini-batch is an unbiased substitute for averaging over the whole dataset (chapters 6–7). The identity $\sigma^2 = E[X^2] - \mu^2$ is the useful "mean of the square minus the square of the mean". Mean and variance

are also the language of initialization and normalization: we set initial weights to control the variance of activations (chapter 10), and batch/layer normalization directly rescales activations to a target mean and variance (chapter 11).

Two variables that move together have a nonzero **covariance**. Scaling by both standard deviations gives the unit-free **correlation** $\rho \in [-1, 1]$: positive when they rise together, negative when one rises as the other falls, and zero when they are linearly unrelated. Stacked over a vector, the covariances form the symmetric matrix Σ. PCA and whitening diagonalize this matrix, and a multivariate Gaussian carries it as its shape (chapter 11):

$$\mathrm{Cov}(X, Y) = \mathrm{E}[(X - \mu_X)(Y - \mu_Y)] = \mathrm{E}[XY] - \mu_X\mu_Y, \quad \rho_{XY} = \frac{\mathrm{Cov}(X, Y)}{\sigma_X\,\sigma_Y}$$

Worked example — mean and variance of a small sample

Take the eight numbers 2, 4, 4, 4, 5, 5, 7, 9.

1. Mean: $(2 + 4 + 4 + 4 + 5 + 5 + 7 + 9)/8 = 40/8 = \mathbf{5}$.
2. Squared deviations from 5: 9, 1, 1, 1, 0, 0, 4, 16, summing to 32; variance = $32/8 = \mathbf{4}$; std = $\sqrt{4} = \mathbf{2}$.
3. Cross-check with $\mathrm{E}[X^2] - \mu^2$: mean of squares = $(4 + 16 + 16 + 16 + 25 + 25 + 49 + 81)/8 = 232/8 = 29$, and $29 - 5^2 = 29 - 25 = \mathbf{4}$ ✓.

Python · NumPy ▶ runnable in the app

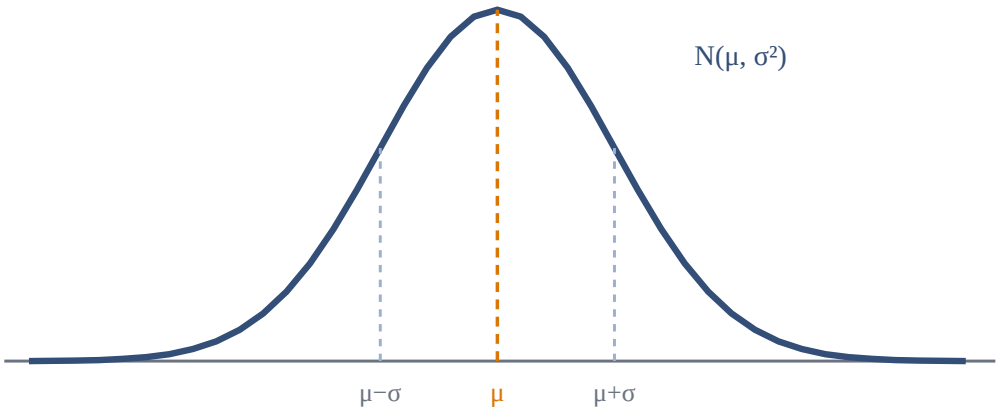
```python
import numpy as np
data = np.array([2., 4., 4., 4., 5., 5., 7., 9.])
mu = data.mean()
var = data.var()                    # population variance (divides by N)
print("mean     =", mu)             # 5.0
print("variance =", var)            # 4.0
print("std      =", np.sqrt(var))   # 2.0
print("E[X^2]-mu^2 =", (data**2).mean() - mu**2)   # same 4.0
# sample estimates converge to the truth as n grows (law of large numbers)
```

```
rng = np.random.RandomState(0)
big = rng.normal(10.0, 3.0, size=100000)           # true mean 10, std 3
print("est mean =", round(big.mean(), 3), " est std =", round(big.std(), 3))
```

Two main distributions

Almost every loss in this book comes from one of two distributions. The **Gaussian** (normal) distribution models continuous quantities and measurement noise. Its bell curve is set by a mean μ and variance σ^2:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$



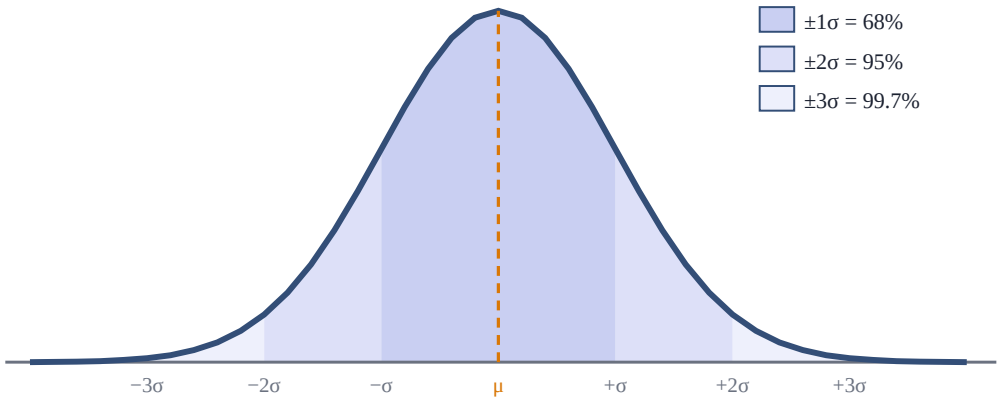
The Gaussian, set entirely by its mean μ (center) and standard deviation σ (width). About 68% of the mass lies within μ ± σ. Squared-error loss is what you get by assuming your targets are a clean signal plus Gaussian noise (chapter 6).

The **Bernoulli** distribution models a single yes/no event: X = 1 with probability p, X = 0 with probability 1 − p, giving mean p and variance p(1 − p). Its multi-class generalization, the **categorical** distribution, assigns probabilities $p_1, \ldots, p_K$ to K mutually exclusive outcomes with $\Sigma_k\, p_k = 1$. A binary classifier's sigmoid output is the p of a Bernoulli. A multi-class classifier's softmax output is a categorical distribution over the labels.

The Gaussian in depth: standardization and the empirical rule

Every Gaussian is a shifted, stretched copy of one curve. **Standardizing** with $z = (X - \mu)/\sigma$ maps any $N(\mu, \sigma^2)$ onto the **standard normal** $N(0, 1)$; a *z-score* counts σ's from the mean. So one table of areas serves every bell. Those areas give the **68–95–99.7 rule**: about 68%, 95%, and 99.7% of the mass lie within ±1σ, ±2σ, and ±3σ. This is why a "3σ event" means "shouldn't happen", and why anomaly detectors threshold a standardized score.

$$Z = \frac{X - \mu}{\sigma} \sim N(0,1), \quad P(|X - \mu| \le k\sigma) \approx 0.68, 0.95, 0.997 \text{ for } k = 1, 2, 3$$



The empirical (68–95–99.7) rule. Standardizing any Gaussian with $z = (x - \mu)/\sigma$ maps it onto this one standard normal, so the same three percentages apply to every bell curve. This is the fact behind z-scores, error bars, and the "σ" language of anomaly detection.

A quick guide to distributions

Beyond those two, a small set covers most of deep learning's needs. Each has a mean and variance worth remembering. Note the Poisson's special feature (mean = variance), and that a Binomial is a sum of n independent Bernoullis:

DISTRIBUTION	PMF / PDF	MEAN	VARIANCE	APPEARS AS…
Bernoulli(p)	$p^x(1-p)^{1-x}$, $x \in \{0,1\}$	p	$p(1-p)$	one sigmoid output; a yes/no label
Binomial(n, p)	$C(n,k)\, p^k(1-p)^{n-k}$	np	$np(1-p)$	successes in n independent trials
Categorical($p_{1\dots K}$)	p_k for class k	p_k (per class)	$p_k(1-p_k)$	a softmax output; a class label
Uniform(a, b)	$1/(b-a)$ on $[a, b]$	$(a+b)/2$	$(b-a)^2/12$	weight-init ranges; random baselines
Gaussian $N(\mu, \sigma^2)$	$e^{-(x-\mu)^2/2\sigma^2}/\sqrt{2\pi\sigma^2}$	μ	σ^2	noise; activations; priors
Exponential(λ)	$\lambda e^{-\lambda x}$, $x \geq 0$	$1/\lambda$	$1/\lambda^2$	waiting times; gaps between events
Poisson(λ)	$\lambda^k e^{-\lambda}/k!$	λ	λ	counts of rare events in a window

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)
n = 200000
p, lam, beta = 0.3, 4.0, 2.0                    # Bernoulli p, Poisson rate, Exp scale
tests = [
    ("Bernoulli(0.3)",   rng.binomial(1, p, n),    p,      p*(1-p)),
    ("Binomial(10,0.3)", rng.binomial(10, p, n),    10*p,   10*p*(1-p)),
    ("Poisson(4)",       rng.poisson(lam, n),       lam,    lam),
    ("Exponential(2)",   rng.exponential(beta, n),  beta,   beta**2),
    ("Uniform(0,1)",     rng.uniform(0, 1, n),      0.5,    1/12),
]
for name, x, m, v in tests:                      # empirical moments match theory
    print(f"{name:16s} mean {x.mean():6.3f} (th {m:5.3f})   var {x.var():7.3f} (th {v:6.3f})")
```

Sampling and the law of large numbers

To **sample** is to draw a specific value according to a distribution's probabilities. Sampling is how we simulate, how we split data, how dropout picks units to drop (chapter 12), and how generative models produce new images or text (chapter 21). The **law of large numbers** guarantees that the average of many independent samples converges to the true expectation. This is the theoretical justification for estimating any expectation — a loss, an accuracy, a gradient — from a finite batch. The cell below draws ten thousand standard-normal samples and prints a histogram as plain counts (no plotting library needed) so you can see the bell shape appear.

Python · NumPy ▶ runnable in the app

```
import numpy as np
rng = np.random.RandomState(0)
x = rng.normal(0.0, 1.0, size=10000)       # 10k standard-normal samples
edges = np.arange(-4, 5)                     # integer bin edges -4..4
counts, _ = np.histogram(x, bins=edges)
for lo, hi, c in zip(edges[:-1], edges[1:], counts):
    print(f"[{int(lo):+d}, {int(hi):+d})  {int(c):5d}  " + "#" * (int(c) //
100))
print("total counted:", int(counts.sum()))
```

More generally, averaging samples estimates any expectation. This is **Monte-Carlo estimation**, $E[f(X)] \approx (1 / N) \Sigma_i f(x^{(i)})$. It is the method a variational autoencoder uses to estimate its objective from a single latent sample (chapter 19), and the way dropout masks or sampled weights become predictive uncertainty (chapter 24). Its error shrinks like $1 / \sqrt{N}$ in any dimension.

Independence and the central limit theorem

Two events are **independent** when $P(A \cap B) = P(A) P(B)$. For random variables, the joint distribution then factorizes, $p(x, y) = p(x) p(y)$, expectations multiply ($E[XY] = E[X] E[Y]$), covariance is zero, and variances add: $\mathrm{Var}(X_1 + \ldots + X_n) = \Sigma_i \mathrm{Var}(X_i)$. Independence is what turned the dataset's likelihood into a product.

This additivity drives the **central limit theorem** (CLT): the average of many independent variables, whatever their individual shape, tends toward a Gaussian. If each X_i has mean μ and variance σ^2, the sample mean has mean μ and variance σ^2/n, so it both *concentrates* (spread $\propto 1/\sqrt{n}$, the law of large numbers with a rate) and turns *bell-shaped*:

$$X_n = \frac{1}{n}\Sigma_i X_i, \quad \frac{X_n - \mu}{\sigma/\sqrt{n}} \longrightarrow N(0,1) \text{ as } n \to \infty$$

So aggregated quantities look Gaussian, mini-batch gradient noise does, and a standard error carries that characteristic $\sqrt{n}$. The cell averages batches of thirty *uniform* draws and shows a Gaussian appearing from a clearly non-Gaussian source:

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(1)
n, trials = 30, 20000
draws = rng.uniform(0, 1, size=(trials, n))   # each row: 30 Uniform(0,1) draws
means = draws.mean(axis=1)                    # sample means -> ~Gaussian by CLT
print("mean of sample-means =", round(float(means.mean()), 4))   # ~0.5
print("std  of sample-means =", round(float(means.std()), 4))    # ~0.053
print("CLT predicts std     =", round(float(np.sqrt((1/12)/n)), 4))
within = np.mean(np.less(np.abs(means - 0.5), means.std()))
print("fraction within 1 sd =", round(float(within), 3))         # ~0.68 (Gaussian)
```

Likelihood: where MSE and cross-entropy come from

Given a model with parameters θ, the **likelihood** of a dataset is the probability the model assigns to exactly the data we observed. Assuming the m examples are independent, that probability is a product. Because products of small numbers are hard to work with, we take its logarithm and change the sign, turning "maximize a product" into "minimize a sum":

$$\theta^\star = \operatorname{argmax}_\theta \Pi_{i=1..m}\, p(x^{(i)};\theta) = \operatorname{argmin}_\theta \left(-\Sigma_{i=1..m} \ln p(x^{(i)};\theta) \right)$$

This is **maximum likelihood estimation** (MLE), and it is the single principle that gives chapter 6's two losses.

Gaussian noise gives MSE. Model a real-valued target as a prediction plus Gaussian noise, $y = f(x; \theta) + \varepsilon$ with $\varepsilon \sim N(0, \sigma^2)$. Each term's negative log-density is $(y - f)^2 / 2\sigma^2$ plus a constant that does not involve θ. Summing and dropping the constants, maximizing likelihood is *exactly* minimizing $\Sigma(y - f)^2$ — the mean squared error.

Categorical labels give cross-entropy. Model a class label with a categorical distribution whose probabilities are the network's softmax outputs p_k, and write the true label one-hot ($y_k = 1$ for the correct class). The probability of one observed label is $\Pi_k \, p_k^{y_k} = p_{true}$, so its negative log-likelihood is $-\Sigma_k \, y_k \ln p_k$ — the **cross-entropy**. MSE and cross-entropy are not arbitrary. They are Gaussian and categorical maximum likelihood in another form (chapter 6).

Python · NumPy ▶ runnable in the app

```python
import numpy as np
rng = np.random.RandomState(0)
flips = rng.binomial(1, 0.7, size=200)          # true p = 0.7, 200 trials
k, n = int(flips.sum()), len(flips)
print("heads:", k, "of", n, "  sample mean:", round(k/n, 3))
# scan the Bernoulli log-likelihood over candidate p values
grid = np.linspace(0.01, 0.99, 99)
loglik = k*np.log(grid) + (n - k)*np.log(1 - grid)
p_hat = grid[np.argmax(loglik)]
print("MLE p (argmax of log-likelihood):", round(float(p_hat), 3))
print("-> the MLE equals the sample mean")
```

From maximum likelihood to MAP

Maximum likelihood treats θ as a fixed unknown. The **Bayesian** view makes it random, with a **prior** $p(\theta)$ — what we believed before the data — and forms the **posterior** $p(\theta \mid \text{data}) \propto p(\text{data} \mid \theta)\, p(\theta)$. Maximizing that posterior is **maximum-a-posteriori** (MAP) estimation:

$$\theta_{MAP} = \text{argmax}_\theta \, p(\theta \mid \text{data}) = \text{argmax}_\theta \, [\, \ln p(\text{data} \mid \theta) + \ln p(\theta) \,]$$

The first term is the log-likelihood — chapter 6's loss — and the second is a penalty set entirely by the prior. A zero-mean Gaussian prior on the weights contributes a squared norm. The algebra *is* weight decay:

$$\theta \sim N(0, \tau^2 I) \implies -\ln p(\theta) = \frac{1}{2\tau^2}\|\theta\|^2 + \text{const} = \lambda\|\theta\|^2 + \text{const}$$

So **L2 regularization is a Gaussian prior** (strength $\lambda = 1/2\tau^2$), and a heavier-tailed Laplace prior gives the sparsity-inducing L1 penalty (chapter 12): regularization is a stated belief that weights are small, its strength the prior's inverse variance. As data grows, the likelihood overwhelms the prior, and MAP moves back toward MLE. Priors matter most when data is limited.

Worked example — an MLE, and its MAP shrinkage

Seven of ten sampled emails are spam; model each as Bernoulli(p).

1. Log-likelihood: $\ell(p) = 7 \ln p + 3 \ln(1 - p)$.
2. Set $\ell'(p) = 7/p - 3/(1 - p) = 0 \implies 7(1 - p) = 3p \implies \hat{p} = 7/10 =$ **0.7** — the MLE is just the observed fraction.
3. Add a Beta(2, 2) prior (a gentle pull toward 0.5); it contributes $\ln p + \ln(1 - p)$, so MAP maximizes $8 \ln p + 4 \ln(1 - p)$, giving $p_{MAP} = (7 + 1)/(10 + 2) = 8/12 \approx$ **0.667**.

The prior acts like one extra "pseudo-spam" and one "pseudo-ham", shrinking 0.70 toward the prior's 0.5 — exactly as L2 shrinks weights toward 0 (chapter 12).

Softmax as a distribution

The **softmax** turns a vector of any real logits z into a categorical distribution:

$$p_k = \frac{e^{z_k}}{\Sigma_j e^{z_j}}, \quad p_k > 0, \quad \Sigma_k p_k = 1$$

Exponentiating makes every entry positive. Dividing by the sum makes them add to 1. So softmax outputs are a genuine probability distribution over the K classes, ranked in the same order as the logits. It is **shift-invariant**: adding the same constant to every logit leaves p unchanged, because the constant factors out of numerator and denominator. This is not just an elegant fact. It is how we compute softmax safely, by subtracting the largest logit first so that e^z never overflows.

Python · NumPy ▶ runnable in the app

```
import numpy as np
def softmax(z):
    z = z - z.max()                 # shift-invariant: subtract max for
safety
    e = np.exp(z)
    return e / e.sum()
z = np.array([2.0, 1.0, 0.1])
p = softmax(z)
print("logits :", z)
print("softmax:", np.round(p, 4))       # positive, sums to 1, ordered like
z
print("sums to:", round(float(p.sum()), 6))
# huge logits would overflow exp() without the max-subtraction; here they
don't
print("big logits ->", np.round(softmax(np.array([1000., 1001., 1002.])),
4))
```

Entropy and KL divergence (brief)

The **entropy** of a distribution measures its average surprise (in nats when using ln; Appendix D gives the fuller story): $H(p) = -\Sigma_k\, p_k \ln p_k$. It is largest for a uniform distribution: maximal uncertainty, with $H = \ln K$ for K equally likely classes. It is zero when one outcome is certain. That ln K is exactly the cross-entropy of random guessing, the baseline a classifier must beat (chapter 6). The **cross-entropy** between a true distribution y and a model p is $H(y, p) = -\Sigma_k\, y_k \ln p_k$, and the **Kullback–Leibler divergence** measures how far p is from y:

$$D_{KL}(y \,\|\, p) = \Sigma_k \, y_k \ln \frac{y_k}{p_k} = H(y, p) - H(y) \geq 0 \,, \;\; = 0 \Leftrightarrow p = y$$

Because the label's own entropy H(y) is fixed, minimizing cross-entropy over the model is the same as minimizing the KL divergence from the prediction to the truth. Training a classifier is pulling its predicted distribution toward the data's. The same divergence reappears across the architectures: a variational autoencoder adds a KL term that pulls its latent distribution toward a standard-normal prior (chapter 19), and generative models in general are trained by reducing a divergence between the model's distribution and the data's (chapter 21). Entropy and KL are the link between "loss function" and "probability", and Appendix D develops both from first principles.

In the real world

When a large language model reports a "temperature", it is rescaling logits before the softmax of this appendix: dividing z by a temperature $T > 1$ flattens the categorical distribution (more random, more creative sampling), while $T < 1$ sharpens it toward the argmax (more deterministic). The model's training objective is the cross-entropy / negative log-likelihood derived above, computed over trillions of tokens. Its "perplexity" — the main quality metric for language models — is simply e raised to that average cross-entropy. Probability is not a side topic here. It is the objective, the output, and the evaluation all at once.

Exercises

1. For a fair six-sided die, compute E[X] and Var(X). MATH

2. A Bernoulli variable has p = 0.3. Give its mean and variance, and say at which p the variance is largest. MATH

3. A spam filter activates on 90% of spam and on 4% of good mail; 20% of all mail is spam. If a message triggers the filter, what is the probability it is really spam? MATH

4. Draw 50000 samples from $N(5, 2^2)$ and compare the estimated mean and standard deviation to the true values. CODE

5. Show that an L2 penalty $\lambda\|\theta\|^2$ is the same as a zero-mean Gaussian prior on the weights, and identify λ. CONCEPT

6. A Poisson($\lambda = 3$) counts events in a window. Give its mean and variance, and say why a histogram of counts has spread comparable to its center. MATH

7. Compute softmax([1, 2, 3]) to three decimals by hand and confirm it sums to 1. MATH

8. Why is softmax paired with cross-entropy, and why do we subtract the maximum logit before exponentiating? CONCEPT

APPENDIX

Appendix D · Information theory

> *Almost every objective in this book is a quantity from information theory in a different form — the loss that trains a classifier, the metric that grades a language model, the regularizer inside a variational autoencoder, the score a self-supervised method maximizes. This appendix builds the vocabulary from one small idea: the information in an event is the surprise of seeing it happen, and surprise is $-\log p$. From surprise come entropy (a source's average surprise), cross-entropy (your data's surprise measured with the wrong model — exactly the classification loss of chapter 6), the Kullback–Leibler divergence (the extra bits that wrong model costs), mutual information (how much two variables reveal about each other), and perplexity (how confused a language model is, chapter 17). The same set of ideas shows why a Gaussian is the noise model that assumes the least, and so why squared error is the natural loss for regression. Runnable NumPy makes each quantity concrete and checks the identities that connect them.*

Self-information: the surprise of an event

Suppose an event of probability p occurs. How much have you learned? A near-certain event ($p \approx 1$) tells you almost nothing. A rare event ($p \approx 0$) is a big surprise. We want a measure of information that is large when p is small, is zero when $p = 1$, and *adds* for independent events, even though their joint probability is a product. The only function turning products into sums is the logarithm, so the **self-information** (or **surprisal**) of an outcome x is

$$I(x) = -\log p(x) = \log \frac{1}{p(x)}$$

The base of the logarithm sets the *unit*. Base 2 measures information in **bits**: one bit is the surprise of a fair coin landing heads, $-\log_2 \tfrac{1}{2} = 1$. Base e measures in **nats**. Since $\ln = \ln 2 \cdot \log_2$, one nat $= 1/\ln 2 \approx 1.4427$ bits and one bit $= \ln 2 \approx 0.6931$ nats. Deep-learning code uses nats (`np.log` and PyTorch's losses are natural-log), while textbooks and language-model reports

often quote bits. The two differ only by the constant ln 2, so remember which you are reading. A one-in-a-thousand event carries $\log_2 1000 \approx 9.97$ bits of surprise, about ten yes/no questions' worth. A coin flip carries exactly one.

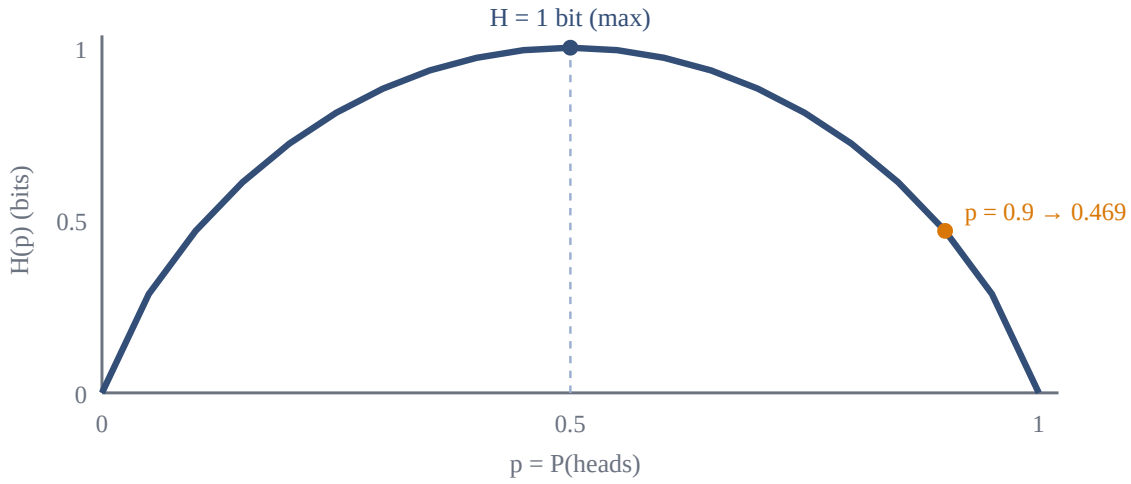
Shannon entropy: average surprise of a source

A single outcome has a surprisal. A whole *distribution* has an average surprisal, weighted by how often each outcome actually occurs. That average is the **Shannon entropy** of the random variable X with distribution p:

$$H(X) = \mathbb{E}_{x\sim p}[\, -\log p(x)\,] = -\,\Sigma_x\, p(x) \log p(x)$$

$p(x)$	probability mass of outcome x; $p(x) \geq 0$, $\Sigma_x\, p(x) = 1$
$H(X)$	entropy — expected surprisal of a draw from p; bits if log = $\log_2$, nats if ln
$-\log p(x)$	self-information of x; rarer outcome, larger value
$\mathbb{E}_{x\sim p}[\cdot]$	expectation over $x \sim p$ (Appendix C) — the probability-weighted average

Entropy is the smallest average number of bits needed to encode one draw from the source under an optimal code (Shannon's source-coding theorem). You can compress it to H bits per symbol, and no fewer. It is **largest** for the uniform distribution, where $H = \log K$ for K equally likely outcomes, and **zero** when one outcome is certain, since a certain outcome carries no information (impossible outcomes contribute nothing, using $0 \cdot \log 0 = 0$). Everything in between falls between these extremes: entropy is a one-number summary of how *uncertain* a distribution is. In an output layer it is literally the predictive uncertainty. A confident softmax has low entropy, an unsure one high. This is why it reappears as a calibration diagnostic (chapter 24).



The binary entropy $H(p) = -p\,\log_2 p - (1-p)\,\log_2(1-p)$ of one yes/no variable. It peaks at exactly 1 bit for a fair coin ($p = ½$) and falls to 0 at the certain corners $p = 0$ and $p = 1$. The amber dot marks the worked example: at $p = 0.9$ the outcome is mostly predictable, so only 0.469 bits remain.

Worked example — entropy of a biased coin (p = 0.9)

A coin lands heads with probability 0.9, tails with 0.1. How uncertain is one flip, in bits?

1. Heads term: $-0.9\,\log_2 0.9$. Since $\log_2 0.9 = \ln 0.9 / \ln 2 = -0.15200$, this is $-0.9 \cdot (-0.15200) =$ **0.13680**.
2. Tails term: $-0.1\,\log_2 0.1$. Since $\log_2 0.1 = -3.32193$, this is $-0.1 \cdot (-3.32193) =$ **0.33219**.
3. Add them: H = 0.13680 + 0.33219 = **0.469 bits** (equivalently $0.469 \cdot \ln 2 =$ 0.325 nats).

What the number means: a biased coin is far more predictable than a fair one, so a flip gives less than half a bit, compared with a full bit for p = ½. Notice the *rare* tail makes up most of the sum (0.332 of 0.469): unlikely events are individually the most informative.

Joint and conditional entropy

With two random variables the same method applies to their joint distribution. The **joint entropy** measures the total uncertainty in the pair (X, Y):

$$H(X, Y) = -\Sigma_x \Sigma_y\, p(x, y) \log p(x, y)$$

The **conditional entropy** H(Y | X) measures how much uncertainty about Y *remains* once X is known. It is the average, over values of X, of the entropy of Y given that value:

$$H(Y \mid X) = -\Sigma_x\, p(x) \Sigma_y\, p(y \mid x) \log p(y \mid x) = H(X, Y) - H(X)$$

That last equality is the **chain rule for entropy**: the uncertainty of the pair is the uncertainty of X plus whatever is left in Y once X is revealed, H(X, Y) = H(X) + H(Y | X). It turns an obvious fact — knowing more never hurts on average — into the inequality H(Y | X) ≤ H(Y), with equality only when X and Y are independent. This conditional entropy is exactly what a sequence model reports: a language model's per-token loss is H(next token | context), the uncertainty about the next word once the preceding text is known.

Cross-entropy: measuring your data with the wrong model

Entropy assumes you know the true distribution p and encode optimally for it. But a model only ever has an *estimate* q. If you build your code for q and then encode data that truly comes from p, your average message length is the **cross-entropy**:

$$H(p, q) = \mathbb{E}_{x \sim p}[\, -\log q(x)\,] = -\Sigma_x\, p(x) \log q(x)$$

Read it carefully: outcomes are weighted by their *true* frequencies p(x), but the surprisal is computed with the *model's* probabilities q(x). You pay for how often each event really happens, at the rate your model is surprised to see it. Cross-entropy is minimized, equaling the plain entropy H(p), exactly when q = p. Any mismatch makes it strictly larger. That is why cross-entropy is **the** classification loss of chapter 6. There the true distribution p is the one-hot label y and the model q is the softmax output, so a one-hot p removes every term except the correct class k*, leaving

$$H(y, q) = -\Sigma_k\, y_k \log q_k = -\log q_{k^*}$$

the negative log-likelihood of the right answer. Minimizing it drives the predicted probability of the true class toward 1, and, as chapter 6 derives, its gradient with respect to the logits simplifies to the clean $q - y$. Cross-entropy is not just a formula attached to softmax. It is the bit-cost of describing the labels with your model, and every bit above the unavoidable floor is the model's error made measurable.

KL divergence: the price of the wrong model

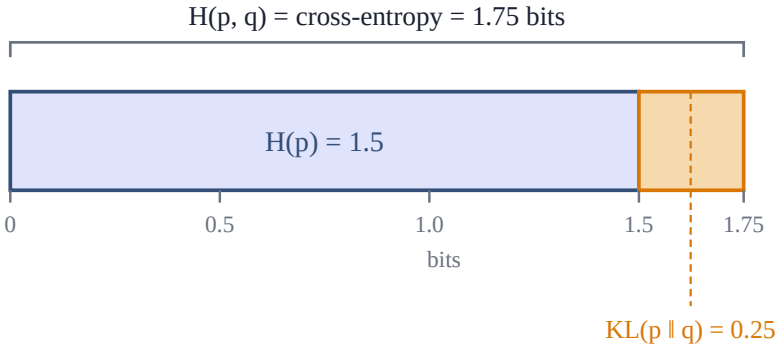
How far is q from the truth p? Subtract the best achievable code length H(p) from what q actually costs, H(p, q). The difference is the **Kullback–Leibler divergence**, the average number of *extra* bits the wrong model wastes:

$$D_{KL}(p \,\|\, q) = \Sigma_x\, p(x) \log \frac{p(x)}{q(x)} = H(p, q) - H(p)$$

Rearranging gives the identity every classifier relies on:

$$H(p, q) = H(p) + D_{KL}(p \,\|\, q)$$

Cross-entropy splits into two parts: H(p), the *unavoidable* uncertainty in the data that no model can remove, and $D_{KL}(p \,\|\, q)$, the *avoidable* extra cost from the model being wrong. Since H(p) is independent of the parameters, minimizing cross-entropy over q is **identical** to minimizing the KL divergence. Training a classifier is nothing more than reducing the divergence between prediction and truth toward zero, with the label's own entropy a constant floor the optimizer cannot change.



Cross-entropy splits into entropy plus divergence, $H(p, q) = H(p) + D_{KL}(p \| q)$. The blue block is the data's entropy $H(p) = 1.5$ bits, an unavoidable floor. The small amber block is the KL divergence 0.25 bits, the avoidable waste from an imperfect model. Training slides the amber block toward zero. The blue never moves. Numbers from the worked example below.

Non-negativity (Gibbs' inequality). The divergence is never negative: $D_{KL}(p \| q) \geq 0$, with equality only when $q = p$. The proof is one line of Jensen's inequality (Appendix B), since log is concave so $\mathbb{E}[\log Z] \leq \log \mathbb{E}[Z]$:

$$-D_{KL}(p \| q) = \Sigma_x\, p(x) \log \frac{q(x)}{p(x)} \leq \log \Sigma_x\, p(x) \cdot \frac{q(x)}{p(x)} = \log \Sigma_x\, q(x) = \log 1 = 0$$

So $D_{KL} \geq 0$, which guarantees $H(p, q) \geq H(p)$: you can never encode data more cheaply with the wrong model than the right one, and the best a classifier can do is drive its loss down to the label entropy.

Worked example — KL divergence of two 3-outcome distributions, and the identity

Let $p = (0.5, 0.25, 0.25)$ be the truth and $q = (0.25, 0.25, 0.5)$ a model of the same three outcomes. Work in bits ($\log_2$).

1. Entropy of p: $H(p) = -[0.5 \log_2 0.5 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25] = -[0.5(-1) + 0.25(-2) + 0.25(-2)] =$ **1.5 bits**.
2. Cross-entropy: $H(p, q) = -[0.5 \log_2 0.25 + 0.25 \log_2 0.25 + 0.25 \log_2 0.5] = -[0.5(-2) + 0.25(-2) + 0.25(-1)] =$ **1.75 bits**.

3. KL directly: $D_{KL}(p \,\|\, q) = 0.5 \log_2 \frac{0.5}{0.25} + 0.25 \log_2 \frac{0.25}{0.25} + 0.25 \log_2 \frac{0.25}{0.5}$

$= 0.5(1) + 0.25(0) + 0.25(-1) =$ **0.25 bits**.

4. Check the identity: $H(p) + D_{KL}(p \,\|\, q) = 1.5 + 0.25 =$ **1.75 bits** $= H(p, q)$ ✓.

The model wastes a quarter-bit per symbol by mis-ranking the outcomes. The next cell reproduces every number and asserts the identity.

Python · NumPy ▶ runnable in the app

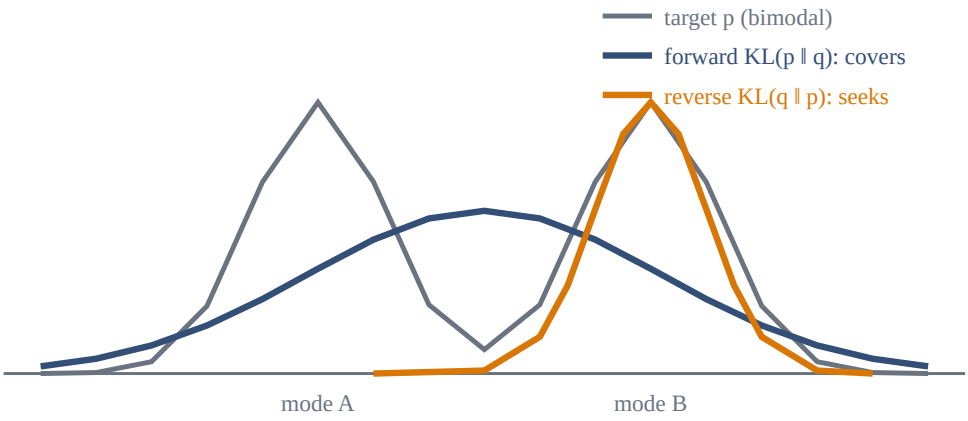
```python
import numpy as np
np.random.seed(0)
EPS = 1e-12                                  # guards log(0); p=0 ->
0*log(EPS)=0

def entropy(p, bits=True):
    p  = np.asarray(p, dtype=float)
    lg = np.log2 if bits else np.log         # log2 -> bits, log -> nats
    return float(-np.sum(p * lg(p + EPS)))

def cross_entropy(p, q, bits=True):
    p, q = np.asarray(p, float), np.asarray(q, float)
    lg = np.log2 if bits else np.log
    return float(-np.sum(p * lg(q + EPS)))

def kl(p, q, bits=True):
    p, q = np.asarray(p, float), np.asarray(q, float)
    lg = np.log2 if bits else np.log
    return float(np.sum(p * lg((p + EPS) / (q + EPS))))

p = np.array([0.5,  0.25, 0.25])             # true distribution over 3
outcomes
q = np.array([0.25, 0.25, 0.5 ])             # a model of the same 3
outcomes

H, Hpq, D = entropy(p), cross_entropy(p, q), kl(p, q)
print("H(p)      =", round(H, 4),   "bits =", round(entropy(p, bits=False),
4), "nats")
print("H(p,q)    =", round(Hpq, 4), "bits")
```

```
print("KL(p||q)  =", round(D, 4),   "bits")
assert np.isclose(Hpq, H + D)                # cross-entropy = entropy + KL
divergence
print("identity  H(p,q) == H(p) + KL(p||q):", bool(np.isclose(Hpq, H + D)))

r = np.array([0.1, 0.2, 0.7])                # asymmetry: KL(p||r) !=
KL(r||p)
print("asymmetry KL(p||r) =", round(kl(p, r), 4), " vs  KL(r||p) =",
round(kl(r, p), 4))
```

Asymmetry: forward versus reverse KL

The divergence is **not** a distance: in general $D_{KL}(p \Vert q) \neq D_{KL}(q \Vert p)$, as the code above shows (0.870 versus 0.743 bits). The two orders show opposite behaviors when a simple q is fit to a complicated p. Forward KL weights each term by the *true* p(x), so it is **zero-avoiding**: wherever p has mass, q must too, or log(p/q) becomes huge. A single q covering every mode is stretched wide, spreading probability across regions p never visits. Reverse KL weights by q(x) and is **zero-forcing**: q is penalized only where it itself puts mass, so it may ignore whole modes and concentrate on one, but must never place mass where p is near zero. These short rules are worth memorizing:

forward $D_{KL}(p \Vert q)$: **mode-covering** (mean-seeking) · reverse $D_{KL}(q \Vert p)$: **mode-seeking** (zero-forcing)



*Fitting one Gaussian q to a two-humped target p. **Forward** KL (blue) spreads q across both modes so it never leaves out any mass of p — mode-covering, at the price of probability in the empty valley. **Reverse** KL (amber) locks onto a single mode and ignores the other — mode-seeking, sharp but incomplete. Same target, opposite failures, set only by which distribution weights the sum.*

This decides how generative models behave. A **variational autoencoder** (chapter 19) minimizes a reverse-KL term inside its evidence lower bound. The zero-forcing behavior is one reason VAE samples can look blurry and mode-collapsed, favoring a safe average over sharp diversity. More broadly, the generative models of chapter 21 are trained by reducing a divergence between the model's distribution and the data's, and the *choice* of divergence — forward KL as in maximum likelihood, reverse KL, or the Jensen–Shannon and Wasserstein alternatives used by GANs — is exactly a choice between covering every mode and committing to a few sharp ones.

Mutual information: how much X tells you about Y

The **mutual information** $I(X;\,Y)$ measures how much knowing one variable reduces uncertainty about the other. It is the entropy of Y minus the entropy of Y that remains after X is revealed:

$$I(X;\,Y) = H(X) - H(X \mid Y) = H(Y) - H(Y \mid X) = H(X) + H(Y) - H(X,\,Y)$$

All three forms are equal and symmetric in X and Y. A fourth is the most revealing: mutual information is the KL divergence between the true joint and the product of the marginals,

$$I(X;Y) = D_{KL}(\,p(x,y) \,\|\, p(x)\,p(y)\,) \geq 0$$

which measures how far X and Y are from independent. By Gibbs' inequality it is non-negative, and **zero exactly when** X and Y are independent, that is, when $p(x, y) = p(x)\,p(y)$ and knowing one tells you nothing about the other. Mutual information is the honest answer to "how related are these two signals?", capturing nonlinear dependence that a correlation coefficient misses. It lies behind feature selection, the information-bottleneck view of what networks discard, and — most concretely here — self-supervised contrastive learning (chapter 20), whose InfoNCE objective maximizes a lower bound on the mutual information between two views of the same data point. The cell below computes I(X; Y) for a 2×2 joint, confirming it is zero when the variables are independent and positive when they are related.

Python · NumPy ▶ runnable in the app

```python
import numpy as np
np.random.seed(0)

def mutual_information(P):
    P  = np.asarray(P, dtype=float)
    P  = P / P.sum()                              # normalise to a valid joint
pmf
    px = P.sum(axis=1, keepdims=True)             # P(X): marginal over Y
    py = P.sum(axis=0, keepdims=True)             # P(Y): marginal over X
    m  = P > 0                                    # 0 log 0 := 0, so skip empty
cells
    return float(np.sum(P[m] * np.log2(P[m] / (px * py)[m])))

# independent: joint is the outer product of the marginals  ->  I(X;Y) = 0
px, py = np.array([0.6, 0.4]), np.array([0.7, 0.3])
indep  = np.outer(px, py)
print("independent joint =", indep.ravel())
print("I(X;Y) independent =", round(abs(mutual_information(indep)), 6),
"bits  (exactly 0)")

# dependent: mass on the diagonal, so knowing X pins down Y  ->  I(X;Y) > 0
dep = np.array([[0.45, 0.05],
                [0.05, 0.45]])
print("I(X;Y) dependent   =", round(mutual_information(dep), 4), "bits  (>
0)")
```

Perplexity: entropy in language-model form

Language-model quality is almost never quoted as a cross-entropy in nats. It is quoted as **perplexity**, simply the exponential of the (per-token) entropy:

$$\text{perplexity} = b^{H} = \exp(\, H(p, q)\,) \quad \text{(base } b \text{ matches the log in } H\text{)}$$

Exponentiating undoes the logarithm inside entropy and returns to the scale of *counts*. A model with cross-entropy H bits per token has perplexity 2^{H}, the **effective branching factor**: as uncertain, on average, as choosing uniformly among 2^{H} equally likely next tokens. Perplexity 1 means perfect prediction. Perplexity equal to the vocabulary size means nothing learned beyond the uniform guess. Because it is a strictly increasing function of cross-entropy, minimizing the training loss and minimizing perplexity are the same optimization on two scales: one linear in bits, one multiplicative in branching (chapter 17).

In the real world

"Perplexity 12 on WikiText" means an average per-token cross-entropy of $\log_2 12 \approx 3.58$ bits, as unsure as picking among twelve equally likely words per step. This figure has fallen from the hundreds for early n-gram models to single digits for modern transformers. The same cross-entropy trains the model and grades it, and the KL decomposition sets the floor: no model can push perplexity below the exponential of the language's own entropy. Information theory is not a side topic in deep learning. It is the objective, the metric, and the diagnostic at once.

Differential entropy and the maximum-entropy principle

For a *continuous* variable with density p(x), the sum becomes an integral, giving the **differential entropy**:

$$h(X) = -\int p(x) \log p(x)\, dx$$

It behaves mostly like the discrete version, with one exception: it can be **negative**, because a density may exceed 1 (a distribution squeezed into a narrow interval has negative differential entropy), and it is not invariant to rescaling the axis. What stays valid is the KL divergence between densities, $D_{KL}(p \,\|\, q) = \int p \log(p/q)\, dx$, which stays non-negative and coordinate-independent. This is one more reason KL, not entropy, is the quantity that appears in continuous-latent models like the VAE.

Differential entropy answers a basic question: among all distributions consistent with what you know, which should you assume? The **maximum-entropy principle** says pick the one with the largest entropy: the distribution that assumes the least while still fitting your constraints. With only a fixed range, the answer is the *uniform*. With a fixed positive mean, it is the *exponential*. With a fixed mean and variance σ^2, it is the **Gaussian**, whose differential entropy is

$$h(\mathcal{N}) = \tfrac{1}{2} \log(\, 2\pi e\, \sigma^2 \,)$$

This is why the Gaussian sits at the center of machine learning (Appendix C): it assumes the *least* beyond a mean and a spread. And it connects back to loss functions. Model regression noise with that least-assuming distribution at fixed variance, and maximum likelihood gives you exactly mean-squared error (chapter 6). Squared error is not a convenient default. It is honestly the maximum-entropy choice, assuming as little about your noise as its variance allows, just as cross-entropy grades a classifier by the bits it wastes. Both losses, and the metric that grades your language model, are one information-theoretic idea seen from different angles.

Exercises

1. A fair eight-sided die is rolled. Compute its entropy in bits and in nats, and give the self-information of any single face. MATH

2. Let the truth be $p = (0.5, 0.5)$ and a model $q = (0.9, 0.1)$. Compute $H(p)$, the cross-entropy $H(p, q)$, and $D_{KL}(p \,\|\, q)$ in bits, and verify the identity. MATH

3. A joint distribution over (X, Y) is $P = \begin{bmatrix} ½ & ¼ \\ 0 & ¼ \end{bmatrix}$. Compute H(X), H(Y), H(X, Y), H(Y | X), and I(X; Y), and confirm I(X; Y) = H(X) + H(Y) − H(X, Y). MATH

4. A language model assigns probabilities (0.5, 0.5, 0.25, 0.5) to the four tokens that actually occurred in a sentence. Compute the per-token cross-entropy in bits and the perplexity, and interpret it. MATH

5. Explain why minimizing cross-entropy over a model's parameters is identical to minimizing the KL divergence from the model to the truth, and what role the entropy H(p) plays. CONCEPT

6. A variational autoencoder minimizes a *reverse*-KL term, $D_{KL}(q \| p)$, between its approximate posterior q and a target p. Would you expect its fitted q to cover all modes of a multimodal p or to concentrate on one, and what visible symptom does this produce? CONCEPT

APPENDIX

Appendix E · Python & NumPy primer

> *Every runnable cell in this book is Python, and almost every one relies on **NumPy**. NumPy is the array library that makes numerical code short, fast, and readable. This appendix is an introduction for complete beginners. It assumes you can open the code cells and press **Run**, but nothing more. We build the language essentials (types, containers, functions, loops) and then the one library you truly need to follow the from-scratch derivations. There are many small cells you can edit and re-run. If you have written Python before, skim to NumPy: the array.*

Why Python, and how to read a code cell

Python became popular in scientific computing for a simple reason: it is easy to read, and it is built on top of fast, compiled numerical libraries. You write a line that looks like mathematics. Underneath, optimized C and Fortran do the hard work. Throughout the book you will see two kinds of code cell. A **runnable** cell (labeled *Python · NumPy*) runs right here in your browser through a bundled Python. Press Run and read the printed output. A **PyTorch** listing is meant to be *read* and copied into Google Colab. It shows how the same idea is written in the modern deep-learning stack (Appendix F). Everything below is runnable. Change a number, run again, and watch what happens. That loop of experiment then observe is the fastest way to learn.

Variables, numbers and text

A variable is a name bound to a value. Python is **dynamically typed**: you never declare a type, the value carries it. The core scalar types are `int` (whole numbers), `float` (decimals), `str` (text), and `bool` (`True`/`False`). Arithmetic is what you expect, with two divisions worth remembering: `/` always gives a float, while `//` is floor (integer) division.

Python · NumPy ▶ runnable in the app

```python
# Python infers the type from the value (dynamic typing)
count = 3               # int
rate  = 0.05            # float
name  = "layer"         # str
ok    = True            # bool

print(type(count).__name__, type(rate).__name__, type(name).__name__)
print("floor division 7 // 2 =", 7 // 2)   # -> 3
print("true  division 7 /  2 =", 7 / 2)    # -> 3.5
print("power        2 ** 10 =", 2 ** 10)   # -> 1024
print(name + "_" + str(count))             # join text with str()
```

Note the last line: you cannot add a string to an integer directly, so `str(count)` converts first. Comments begin with `#` and are ignored by Python. Use them freely.

Lists, tuples and dictionaries

Three containers cover most needs. A **list** `[…]` is an ordered, editable sequence, and it is the one you will use most. A **tuple** `(…)` is an ordered but *immutable* sequence, ideal for fixed things like an array's shape. A **dictionary** `{key: value}` maps keys to values for instant lookup. It is perfect for configuration. Indexing starts at **0**, and negative indices count from the end (`-1` is the last element).

Python · NumPy ▶ runnable in the app

```python
# list: ordered and mutable
layers = [4, 8, 8, 1]
layers.append(1)                # add to the end
print("list:", layers, "len", len(layers), "first", layers[0], "last",
layers[-1])

# tuple: ordered and immutable (good for fixed shapes)
shape = (28, 28)
print("tuple:", shape, "pixels", shape[0] * shape[1])

# dict: key -> value lookup
```

```
cfg = {"lr": 0.01, "epochs": 20}
cfg["batch"] = 32                # add a new key
print("lr:", cfg["lr"], " momentum:", cfg.get("momentum", "default 0.9"))
```

`cfg.get(key, default)` returns a fallback when the key is missing, which avoids errors on optional settings.

Functions

A function packages a computation behind a name so you can reuse it. Define one with `def`; values after `return` come back to the caller. Arguments may have **defaults**, making them optional, and callers may pass them by name (`w=2.0`) for clarity. For a quick one-line function, `lambda` builds an anonymous function.

Python · NumPy ▶ runnable in the app

```
def affine(x, w=1.0, b=0.0):
    """A tiny linear unit w*x + b (the neuron of chapter 1). w, b default to
1, 0."""
    return w * x + b

print(affine(3))                 # defaults -> 1*3 + 0 = 3.0
print(affine(3, w=2.0))          # keyword arg -> 2*3 + 0 = 6.0
print(affine(3, w=2.0, b=1.0))   # -> 2*3 + 1 = 7.0

square = lambda x: x * x         # a one-line anonymous function
print("square(5) =", square(5))
```

That triple-quoted line is a **docstring**: human documentation the tools can read. Passing `w=2.0` by name is a **keyword argument**. It is clearer than relying on position once a function has several settings.

Loops and comprehensions

A `for` loop repeats over any sequence. `range(n)` yields `0..n-1`. When you also need the position, `enumerate` gives `(index, value)` pairs. To go through two sequences together, `zip` pairs them up. A **list comprehension** `[expr for x in seq if cond]` builds a list in one

readable line, and you will see it often.

Python · NumPy ▶ runnable in the app

```python
# a plain loop
squares = []
for i in range(1, 6):
    squares.append(i * i)
print("loop         :", squares)

# the same result as a comprehension
print("comprehension:", [i * i for i in range(1, 6)])

# comprehension with a filter
print("evens 0..9   :", [n for n in range(10) if n % 2 == 0])

# enumerate and zip
for idx, nm in enumerate(["w1", "w2", "w3"]):
    print("   index", idx, "->", nm)
for a, b in zip([1, 2, 3], [10, 20, 30]):
    print("   pair", a, b)
```

The comprehension and the loop above produce identical lists. The comprehension is simply shorter. Indentation (four spaces) is how Python marks a block. There are no braces.

NumPy: the array

Plain Python lists are flexible but slow for numbers, and they have no way to represent a matrix. **NumPy** introduces the `ndarray`: a grid of numbers, all of one type, stored compactly so operations run at C speed. By convention we import it as `np`. You create arrays from lists, or with constructors like `zeros`, `ones`, `arange` (range with a step) and `linspace` (evenly spaced points). Every array has a `.shape` (its dimensions), an `.ndim` (how many axes) and a `.dtype` (its element type, usually `float64` or an integer type).

Python · NumPy ▶ runnable in the app

```python
import numpy as np

a = np.array([1, 2, 3])                  # from a Python list
b = np.zeros((2, 3))                     # a 2x3 block of zeros
d = np.arange(0, 10, 2)                  # start, stop, step -> 0 2 4 6 8
```

```
e = np.linspace(0, 1, 5)              # 5 points evenly from 0 to 1

print("a =", a, " shape", a.shape)
print("b shape:", b.shape, " ndim", b.ndim)
print("arange :", d)
print("linspace:", e)
print("b dtype:", b.dtype)             # float64: zeros/ones are float by
default
```

An array's power is that arithmetic is **elementwise** and *vectorized*: `a * 2` doubles every element with no loop, and `a + a` adds two arrays position by position. This is the style the whole book is written in.

Two ideas will appear on every page. First, the **axes** are the array's dimensions, numbered from the outside in. For a 2-D array, axis 0 runs down the rows and axis 1 runs across the columns. A length-*k* vector has just one axis. Naming the axis is how you tell a reduction which way to collapse (see below). By the convention used in this book, a batch is stored with one example per *row*. So axis 0 is the batch and axis 1 is the features. Second, the **dtype** is fixed for the whole array and sets both memory use and precision. You can request one when you create the array with `np.zeros((2, 3), dtype=np.float32)`, or convert later with `a.astype(np.float64)`. Almost all learning code is floating point. If you use an integer array in float math, NumPy automatically converts it to float. This is usually convenient, but occasionally a surprise (see *Common pitfalls*).

Indexing, slicing and boolean masks

You access an array by position. For a 2-D array, `M[row, col]` selects one element; a colon `:` means "all of this axis", so `M[0]` is the first row and `M[:, 3]` is the fourth column. A **slice** `start:stop` selects a range (stop excluded). Slices are *views*, not copies. Writing into a slice edits the original (see the box below).

Python · NumPy ▶ runnable in the app

```
import numpy as np

M = np.arange(12).reshape(3, 4)         # 3 rows, 4 columns: 0..11
print(M)
```

```
print("element [1,2]:", M[1, 2])        # row 1, col 2 -> 6
print("row 0       :", M[0])            # [0 1 2 3]
print("column 3    :", M[:, 3])         # [3 7 11]
print("top-left 2x2:\n", M[:2, :2])     # rows 0-1, cols 0-1
print("last row    :", M[-1])           # [8 9 10 11]
```

The most expressive tool is the **boolean mask**. A comparison like `x > 0` returns an array of `True`/`False` values. Indexing with it keeps only the `True` positions. Assigning through a mask edits just those entries. That is a whole ReLU in one line.

Python · NumPy ▶ runnable in the app

```
import numpy as np

x = np.array([-2.0, 0.5, -1.0, 3.0, -0.2, 4.0])
mask = x > 0                             # a boolean array
print("mask     :", mask)
print("positives:", x[mask])            # keep where True
print("count > 0:", mask.sum())         # True counts as 1

x[x < 0] = 0.0                          # ReLU: clamp negatives to zero
print("relu(x)  :", x)
```

A third form, **fancy indexing**, uses an array of integers to pick out elements in any order, and repeats are allowed. Its most important use in this book is to *gather* one entry per row. Give a row-index array and a column-index array of equal length, and NumPy returns one element for each pair. This is exactly how cross-entropy picks out the predicted probability of the correct class from a whole batch.

Python · NumPy ▶ runnable in the app

```
import numpy as np

v = np.array([10, 20, 30, 40])
idx = np.array([3, 0, 0, 2])             # reorder and repeat freely
print("reorder :", v[idx])              # [40 10 10 30]

scores = np.array([[2.0, 1.0, 0.1],
                   [0.5, 2.5, 0.3],
                   [1.2, 0.7, 3.1]])
```

```
rows   = np.arange(3)                    # [0 1 2]
labels = np.array([0, 1, 2])            # the "correct" column of each row
print("gather  :", scores[rows, labels])   # one per row -> [2.0 2.5 3.1]
```

Common trap — views vs copies

Two variables can name the *same* underlying array. After `b = a`, writing `b[0] = 99` also changes `a`, because `b` is another name for the same data, not a fresh copy. Basic slicing (`a[1:4]`) likewise returns a **view** that shares memory. When you need an independent duplicate, ask for one explicitly with `a.copy()`. Boolean-mask indexing (`a[a > 0]`) and fancy indexing (`a[[0, 2]]`), by contrast, always return a new array. Forgetting this is the most common source of "why did that value change on its own?" bugs.

Reshaping and axis operations

The same numbers can take different shapes. `reshape` rearranges an array as long as the total element count matches. A `-1` means "infer this dimension", so `X.reshape(-1)` flattens to 1-D. Reductions like `sum`, `mean` and `max` collapse the array, and the **axis** argument chooses *which* direction to collapse. With rows as samples and columns as features, `axis=0` goes down each column (per-feature statistics) and `axis=1` goes across each row (per-sample statistics). Getting the axis right is a big part of using NumPy well.

Python · NumPy ▶ runnable in the app

```
import numpy as np

X = np.arange(6).reshape(2, 3)          # 2 samples, 3 features
print("X:\n", X)
print("sum all     :", X.sum())         # 15
print("sum axis=0  :", X.sum(axis=0))   # down columns -> [3 5 7]
print("sum axis=1  :", X.sum(axis=1))   # across rows  -> [3 12]
print("mean axis=0 :", X.mean(axis=0))  # per-feature mean
print("flatten     :", X.reshape(-1))   # [0 1 2 3 4 5]
```

The operations you'll use everywhere

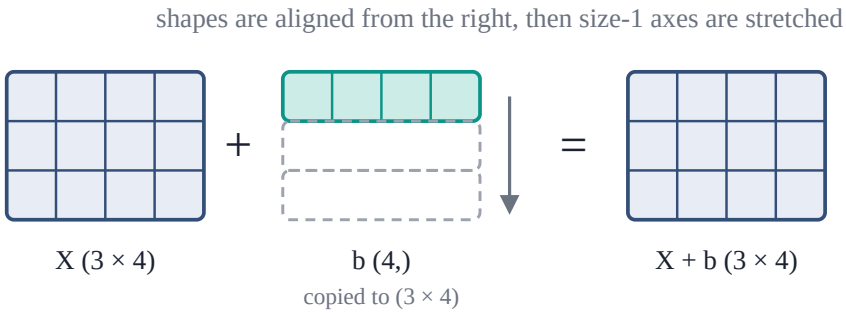
A small handful of functions do most of the work in the book. Matrix multiplication uses the `@` operator (equivalently `np.matmul`): for a batch `X` of shape m×n and a weight matrix `W` of shape n×k, `X @ W` has shape m×k, and the shared inner dimension n is summed over and removed. Do not confuse it with `*`, which is *elementwise*. The reductions `sum`, `mean` and `max` take an `axis` and often a `keepdims=True`. This leaves the collapsed axis in place as size 1, so the result still lines up with the original for a follow-up operation. This is the method behind row-wise normalization. `reshape` and `.T` (transpose) rearrange without copying data. `np.stack` adds a new axis to join arrays into a batch, while `np.concatenate` joins them along an existing one. Finally, `np.where(cond, a, b)` chooses elementwise between two arrays, and `np.clip(x, lo, hi)` bounds every value into a range.

Python · NumPy ▶ runnable in the app

```python
import numpy as np

X = np.arange(6).reshape(2, 3).astype(float)   # (2,3)
W = np.ones((3, 4))                            # (3,4)
print("X @ W shape :", (X @ W).shape)          # (2,4): inner 3 summed away
print("elementwise :", (X * 2).shape)          # (2,3), shape unchanged

s = X.sum(axis=1, keepdims=True)               # (2,1) keeps the collapsed axis
print("keepdims    :", s.shape, "->", s.ravel())
print("row-normalize:\n", (X / s).round(3))    # (2,3)/(2,1) broadcasts per row

print("transpose   :", X.T.shape)              # (3,2)
print("stack       :", np.stack([X, X]).shape) # (2,2,3): a new leading axis
print("concatenate :", np.concatenate([X, X], axis=0).shape)  # (4,3)
print("where>2 :\n", np.where(X > 2, X, 0.0))  # keep big values, else 0
print("clip1..4:\n", np.clip(X, 1, 4))         # bound into [1, 4]
```

Broadcasting

Broadcasting is NumPy's rule for combining arrays of *different* shapes without writing loops or copying data. Compare the shapes from the right. Two dimensions are compatible when they are equal, or when one of them is **1**. In that case, that array is stretched (virtually) to match. This is why you can subtract a length-3 row of column means from a whole (2, 3) matrix: the row is broadcast across every sample. It is the method behind adding a bias vector to a layer's outputs, and normalizing features.



Adding a per-feature bias to a batch. The bias `b` of shape (4,) aligns with the trailing axis of the (3 × 4) matrix. Its missing leading axis is treated as 1 and stretched to 3, so the same four numbers are added to every row. No copy is actually made (the stretch is virtual), but the arithmetic behaves as if it were.

Python · NumPy ▶ runnable in the app

```python
import numpy as np

X = np.array([[1., 2., 3.],
              [4., 5., 6.]])          # shape (2, 3)
col_mean = X.mean(axis=0)             # shape (3,) -> [2.5 3.5 4.5]
Xc = X - col_mean                     # (2,3) - (3,) broadcasts over rows
print("centered:\n", Xc)
print("column means after:", Xc.mean(axis=0))   # ~ [0 0 0]

# a (2,1) column broadcasts across the 3 columns instead
bias = np.array([[10.], [20.]])       # shape (2, 1)
print("plus per-row bias:\n", X + bias)
```

Here `col_mean` has shape `(3,)` and `X` has shape `(2, 3)`. The trailing 3s match, and the missing leading dimension is treated as 1 and stretched to 2. The column vector `bias` of shape `(2, 1)` stretches the other way, across columns.

Here is the rule in two lines, because you will apply it constantly. **(1)** Right-align the two shapes and, if one is shorter, pad it on the left with 1s. **(2)** Along each axis, the sizes must be equal, or one must be 1 (which then stretches). Otherwise NumPy raises an error rather than guessing. This is what makes adding a bias to a batch simple and safe: a per-feature bias of shape `(n,)` lines up with the trailing axis of an `(m, n)` batch, and is added in the same way to every example. The problem case is a *per-example* quantity of shape `(m,)`. Right-aligned, it compares its `m` against the batch's `n`, which usually differ, so the add fails. And even when m happens to equal n, it silently does the wrong thing. The fix is to make your intent clear by giving it shape `(m, 1)`, so it stretches across columns.

Python · NumPy ▶ runnable in the app

```python
import numpy as np

X = np.arange(12).reshape(4, 3).astype(float)   # batch: 4 examples, 3 features
b = np.array([100., 200., 300.])                # per-feature bias, shape (3,)
print("X + b (per feature):\n", X + b)          # (4,3) + (3,) aligns on the 3

# a per-example bias of shape (4,) does NOT align with (4,3):
row_bias = np.array([10., 20., 30., 40.])       # shape (4,)
try:
    X + row_bias                                # trailing 3 vs 4 -> mismatch
except ValueError as err:
    print("X + row_bias fails:", err)
    # -> operands could not be broadcast together with shapes (4,3) (4,)

# fix: reshape to a (4,1) column so it broadcasts across the 3 columns
fixed = X + row_bias[:, None]                    # None inserts a length-1 axis
print("fixed with (4,1):\n", fixed)
```

Worked example — tracing a broadcast by hand

1. Add a bias `b` of shape `(3,)` to a batch `X` of shape `(4, 3)`. Right-align the shapes and pad the shorter: `(4, 3)` versus `(·, 3)` → `(1, 3)`.
2. Compare axis by axis from the right. Trailing axis: 3 vs 3, equal ✓. Leading axis: 4 vs 1, so the 1 stretches to 4 ✓. Result shape `(4, 3)`: row *i* is `X[i] + b` with the *same* three bias numbers each time.
3. Now try a per-example bias of shape `(4,)`. Right-aligned, it is `(·, 4)`. Comparing the trailing axis gives 3 vs 4. Neither is 1 and they differ, so NumPy stops with an error instead of guessing.

What the numbers mean: a quantity you add *per feature* has shape `(n,)`. A quantity you add *per example* must be a column `(m, 1)`. Choosing the shape is choosing the direction of the stretch.

Two vectorized building blocks: ReLU and softmax

Two functions appear on nearly every page, and both are a single vectorized expression over an entire batch, with no loops. The **ReLU** keeps positives and sets negatives to zero. `np.maximum(0.0, z)` does it elementwise and, unlike the masked assignment we saw earlier, returns a fresh array without changing its input. The **softmax** turns each row of scores into a probability distribution, dividing every exponential by the row's sum. Written in the obvious way it overflows, because `exp` of a large logit is enormous. The standard fix is to subtract each row's maximum first. Softmax is *shift-invariant*: subtracting the same constant from a row leaves the output unchanged. So this costs nothing in accuracy and everything in stability. Using `keepdims=True` on the max and the sum keeps them as `(m, 1)` columns that broadcast cleanly back across the row.

Worked example — a stable softmax by hand

1. Take one logit row $\mathbf{z} = (2, 0, -1)$. Row max = 2, so the shifted row is $\mathbf{z} - 2 = (0, -2, -3)$.
2. Take the exponential: $(e^0, e^{-2}, e^{-3}) = (1, 0.135, 0.0498)$, which sum to 1.185.
3. Divide by the sum: (0.844, 0.114, 0.042). These are non-negative and add to 1, so they form a probability distribution.
4. Check shift-invariance: doing it *without* the shift gives $(e^2, e^0, e^{-1}) = (7.389, 1, 0.368)$, sum 8.757, and the ratios are again (0.844, 0.114, 0.042). The result is identical, but the un-shifted exponentials are the ones that overflow for large logits.

What the numbers mean: the largest logit gets the largest probability, the max-subtraction changes nothing about the answer, and doing it row-wise (axis 1, keepdims) applies the same method independently to every example in the batch.

The cell below assembles these pieces into a complete **batched forward pass** of a small two-layer network. This is the same computation the multilayer networks of Chapter 3 perform, and every intermediate shape is printed. Watch how `X @ W + b` broadcasts the bias across the batch at each layer, and how the final probabilities in each row sum to 1.

Python · NumPy ▶ runnable in the app

```
import numpy as np

def relu(z):
    return np.maximum(0.0, z)              # elementwise, no loop, no
mutation

def softmax(z):                            # row-wise and numerically stable
    z = z - z.max(axis=1, keepdims=True)   # shift by row max (safe: shift-
invariant)
    e = np.exp(z)
    return e / e.sum(axis=1, keepdims=True)

rng = np.random.default_rng(0)
```

```
X  = rng.standard_normal((4, 3))           # batch of 4 examples, 3 features
each
W1 = rng.standard_normal((3, 5)) * 0.5     # layer 1: features -> hidden
b1 = np.zeros(5)                           # bias broadcast across the whole
batch
W2 = rng.standard_normal((5, 2)) * 0.5     # layer 2: hidden -> 2 classes
b2 = np.zeros(2)

H = relu(X @ W1 + b1)                       # (4,3)@(3,5) + (5,) -> (4,5)
S = H  @ W2 + b2                            # (4,5)@(5,2) + (2,) -> (4,2)
logits
P = softmax(S)                             # (4,2) probabilities

print("shapes  X:", X.shape, " H:", H.shape, " P:", P.shape)
print("logits row 0:", S[0].round(3))      # -> [-0.007 -0.047]
print("probs  row 0:", P[0].round(3))      # -> [0.51  0.49]
print("row sums    :", P.sum(axis=1).round(6))   # every row -> 1.0
```

Random numbers and reproducibility

Weights are initialized randomly and data is shuffled, so experiments involve randomness. But you want them to be *reproducible*. Seeding a generator fixes the stream of "random" numbers: the same seed always gives the same sequence. This book uses `np.random.RandomState(0)` for exactly this reason, so your output matches the text. (The newer `np.random.default_rng(0)` does the same job and is preferred in new code.)

Python · NumPy ▶ runnable in the app

```
import numpy as np

rng = np.random.RandomState(0)          # a seeded generator
print("uniform [0,1):", rng.rand(3))
print("standard normal:", rng.randn(3))
print("integers 0..9 :", rng.randint(0, 10, 4))

# the same seed reproduces the identical stream
again = np.random.RandomState(0)
print("reproduced    :", again.rand(3))
```

Notice the last line reproduces the first exactly. Seed at the top of an experiment and its results become repeatable. This is essential for debugging and for comparing two ideas fairly.

New code should prefer the modern **Generator** API built on `np.random.default_rng`. You make one generator and draw from it. Passing a seed makes the stream reproducible exactly as before, and the method names are a little cleaner: `random` for uniforms, `standard_normal` for Gaussians, `integers` for whole numbers, `choice` for sampling. It is what the forward-pass cell above used, and what you should use by default.

Python · NumPy ▶ runnable in the app

```python
import numpy as np

rng = np.random.default_rng(0)          # the modern Generator, seeded
print("uniform      :", rng.random(3).round(3))
print("normal       :", rng.standard_normal(3).round(3))
print("integers 0..9:", rng.integers(0, 10, 4))
print("choice       :", rng.choice([10, 20, 30], size=4))

again = np.random.default_rng(0)        # same seed -> identical stream
print("reproduced   :", again.random(3).round(3))
```

Vectorization vs loops (and why it matters)

You *could* write numerical code with Python loops, but it would be slow: every iteration pays interpreter overhead. The NumPy way is to express the whole operation as one array expression, which runs in optimized compiled code over the entire array at once. The speed difference is not small. It is often one to two orders of magnitude. The cell below sums a million squares both ways and times them. The results agree, but the vectorized version is much faster (your exact numbers will vary by machine).

Python · NumPy ▶ runnable in the app

```python
import numpy as np, time

n = 1000000
xs = np.random.RandomState(0).rand(n)

t0 = time.perf_counter()
```

```
total = 0.0
for v in xs:                    # pure-Python loop
    total += v * v
t1 = time.perf_counter()

t2 = time.perf_counter()
total_vec = np.sum(xs * xs)     # vectorized: one compiled call
t3 = time.perf_counter()

print("sum of squares :", round(float(total_vec), 1))
print("results match  :", bool(np.isclose(total, total_vec)))
print("loop time  (s) :", round(t1 - t0, 4))
print("vector time(s) :", round(t3 - t2, 4))
print("speedup    (x) :", round((t1 - t0) / (t3 - t2), 1))
```

This lesson runs through the entire book: **if you find yourself writing a loop over array elements, look for the vectorized form first.** It is usually shorter, always faster, and often clearer once the pattern is familiar.

Common pitfalls

Here is a short list of the surprises that cost beginners the most time. **Copy vs view** is first on the list and has its own box above. `b = a` and a basic slice share memory, so an in-place edit through one name (including `+=`, which changes the array in place rather than rebinding) is visible through the other. Take `a.copy()` when you need independence. **Integer division and integer dtype** cause problems in two ways: `/` always produces floats while `//` floors toward negative infinity, and writing a float into an integer array *truncates* it silently rather than converting up. **Float precision** means you should never test arrays with `==`. `0.1 + 0.2` is not exactly `0.3`, so compare with `np.isclose` or `np.allclose`. Remember that this same limitation is why the naive softmax overflows. **Broadcasting bugs** are the quiet ones. A 1-D array of shape `(m,)` is neither a row nor a column, and transposing it with `.T` does nothing, so an intended outer product needs an explicit `u[:, None] + v[None, :]`. When a computation returns a shape you did not expect, print `.shape` before looking anywhere else.

Python · NumPy ▶ runnable in the app

```python
import numpy as np

# 1) integer arrays truncate on assignment (no automatic upcast in place)
a = np.array([1, 2, 3])                  # dtype int64
a[0] = 3.9                               # 3.9 is TRUNCATED to 3, not rounded
print("int assign  :", a)                # [3 2 3]

# 2) float equality is unreliable; compare with a tolerance
print("== on floats:", 0.1 + 0.2 == 0.3)            # False!
print("np.isclose  :", np.isclose(0.1 + 0.2, 0.3))  # True

# 3) a 1-D vector is neither row nor column: .T is a no-op on it
u = np.array([1., 2., 3.])
print("u.T no-op   :", np.array_equal(u + u, u + u.T))    # True
print("real outer  :", (u[:, None] + u[None, :]).shape)   # (3, 3), as
intended
```

The bridge to PyTorch

Deep-learning frameworks generalize the NumPy array into a **tensor**. A tensor also records the operations performed on it, so gradients can be computed automatically (automatic differentiation, Appendix G), and it can live on a GPU (PyTorch, Appendix F). The array-first habits you build here transfer directly: the same indexing, the same `@` for matrix multiply, the same broadcasting. This listing (read-only, run it in Colab) shows how similar the two are.

PyTorch copy → Colab

```python
import numpy as np, torch

x_np = np.array([[1., 2.], [3., 4.]])
x_t  = torch.from_numpy(x_np)          # a tensor sharing the array's memory
print(x_t, x_t.dtype)

y = x_t @ x_t                          # same @ operator as NumPy
print(y)
print("back to NumPy:", y.numpy())     # and back again
```

Exercises

1. Standardize the columns of a matrix so each has mean 0 and standard deviation 1, using one NumPy expression (subtract the column mean, divide by the column standard deviation). CODE

2. After `a = np.zeros(3); b = a; b[0] = 5`, what is `a`, and why? How would you make `b` independent? CONCEPT

3. Using the broadcasting rule, what shape does `np.ones((3, 1)) + np.ones((1, 4))` produce, and what does it contain? MATH

4. Replace a Python loop that builds cumulative sums with the vectorized `np.cumsum`, and check the two agree. CODE

5. For an integer NumPy array `x`, how do `x / 2` and `x // 2` differ? CONCEPT

6. Implement a numerically stable row-wise softmax and confirm two properties: every row sums to 1, and adding the same constant to a whole row leaves the output unchanged (shift-invariance). CODE

7. You have a batch of class `logits` (shape m×k) and integer `labels` (length m). Using fancy indexing, gather the model's probability for each correct class and compute the mean cross-entropy $-\text{mean}(\log p_{\text{correct}})$. CODE

APPENDIX

Appendix F · PyTorch primer

> *The from-scratch NumPy in this book shows you how a network learns.* ***PyTorch*** *is how you build one for real. It gives you three things NumPy does not: tensors that run on a GPU, automatic gradients (so you never derive backpropagation by hand again), and a library of layers, losses and optimizers. This appendix is a practical tour of tensors, autograd,* `nn.Module`*, the training loop, data loading, and saving. It comes with a clean, reusable training template that the case studies (chapters 30–34) build on. If Appendix E gave you NumPy, treat this as its sequel. Nearly everything you learned there carries over unchanged, and Appendix G then rebuilds PyTorch's autograd from scratch so you can see exactly what it costs. The PyTorch cells here are for reading and copying into Google Colab. The NumPy cells run in place so you can see the ideas work with real numbers.*

Tensors: NumPy arrays that can learn

A PyTorch **tensor** is almost exactly a NumPy array. It has the same shapes, the same indexing, the same `@` for matrix multiply, and the same broadcasting rules (Appendix E). It adds two extra abilities. First, a tensor can be moved to an accelerator with `.to(device)`, so the same code runs on CPU or GPU. Second, if you set `requires_grad=True`, the tensor records the operations applied to it so gradients can be computed automatically. Tensors and NumPy arrays even share memory on the CPU via `torch.from_numpy` and `.numpy()`, so moving between the two is free.

PyTorch copy → Colab

```
import torch
import numpy as np

a = torch.tensor([[1., 2.], [3., 4.]])   # from a Python list (float32 by default)
b = torch.zeros(2, 2)
r = torch.randn(2, 2)                     # standard-normal random
print(a.shape, a.dtype)                   # torch.Size([2, 2]) torch.float32
```

```
# NumPy <-> tensor (shares memory on the CPU)
n = np.array([1., 2., 3.])
t = torch.from_numpy(n)
print("as tensor:", t, " back to numpy:", t.numpy())

# reshape, index and slice just like NumPy
print(a.view(4))          # 1-D view, shares storage (cf. a.reshape(4))
print(a[:, 0])            # first column -> tensor([1., 3.])

# pick a device once, move data/models onto it
device = "cuda" if torch.cuda.is_available() else "cpu"
x = a.to(device)
print("on device:", x.device)

print(a + b)        # elementwise, like NumPy
print(a @ a)        # matrix multiply
```

A common early error is a device or dtype mismatch. A model on the GPU cannot multiply a tensor still on the CPU, and most losses need `float32` inputs with `int64` class labels. Decide on a `device` once at the top and send both model and data there. One hidden trap sits in that first line. NumPy defaults to `float64` while PyTorch defaults to `float32`, so a tensor built from a NumPy array can carry a dtype your model cannot multiply.

Because the two libraries are so similar, most of the NumPy from Appendix E translates line for line. Keep this small reference table handy. The last two rows are the operations that have no NumPy column at all. These are the features that make a tensor more than an array.

NUMPY	PYTORCH	NOTE
np.array(x)	torch.tensor(x)	copies the data into a new tensor
np.zeros((2,3)), np.ones	torch.zeros(2,3), torch.ones	shape passed as args, not a tuple
np.random.randn(2,3)	torch.randn(2,3)	seed with torch.manual_seed(0)
a.shape, a.ndim	a.shape, a.ndim	identical
default dtype float64	default dtype float32	the classic dtype-mismatch trap
a.reshape(2,3)	a.reshape(2,3) / a.view(2,3)	view shares storage, no copy

NUMPY	PYTORCH	NOTE
a @ b	a @ b	or torch.matmul(a, b)
a.sum(axis=0)	a.sum(dim=0)	axis → dim everywhere
np.exp(a), np.maximum(0,a)	torch.exp(a), torch.clamp(a, min=0)	most ufuncs have a torch twin
a.astype(np.float32)	a.float() / a.to(torch.float32)	cast dtype
— (no equivalent)	torch.from_numpy(n), t.numpy()	bridge; shares memory on CPU
— (no equivalent)	requires_grad=True, .backward(), .grad	automatic differentiation
— (CPU only)	a.to("cuda")	run on a GPU

Autograd: gradients for free

The single feature that makes deep learning practical is **automatic differentiation**. As you compute with tensors that require gradients, PyTorch quietly builds a graph of every operation. Calling `.backward()` on a scalar result walks that graph in reverse. This *is* backpropagation (Chapter 8). It stores the derivative with respect to each input in its `.grad` attribute. You never write the chain rule by hand.

PyTorch — copy → Colab

```
import torch

# a leaf tensor we want gradients for
x = torch.tensor(2.0, requires_grad=True)
y = x**2 + 3*x                # PyTorch records this computation
y.backward()                  # compute dy/dx, store it in x.grad
print("y     :", y.item())     # 2^2 + 3*2 = 10
print("dy/dx :", x.grad.item())  # 2x + 3 = 7 at x = 2

# gradients ACCUMULATE across .backward() calls, so reset between steps
x.grad.zero_()
z = 5 * x
z.backward()
print("dz/dx :", x.grad.item())  # 5
```

To see that autograd is not magic, here is the same derivative estimated with a **finite difference** in plain NumPy. Move the input by a tiny `h` and measure the change. Autograd computes exactly this slope, but analytically and for millions of parameters at once.

Python · NumPy ▶ runnable in the app

```python
import numpy as np

def f(x):
    return x**2 + 3*x            # f'(x) = 2x + 3

x0, h = 2.0, 1e-5
fd = (f(x0 + h) - f(x0 - h)) / (2 * h)   # central difference
print("finite-diff f'(2):", round(fd, 5))
print("exact       f'(2):", 2*x0 + 3)     # 7
```

Two tools let you stop recording when you do not need a gradient. `torch.no_grad()` is a context manager. Nothing computed inside the block is recorded, which is what you want at prediction time. There you never call `.backward()`, so tracking history would only waste memory. `.detach()` works on one tensor rather than a whole region. It returns a tensor with the same numbers but cut free from the graph, so you can log it, plot it, or hand it to NumPy without carrying the whole computation with it. Both answer the same question: "do I need to differentiate through this?" At inference the answer is no.

PyTorch copy → Colab

```python
import torch

x = torch.randn(3, requires_grad=True)

# no_grad: a whole block where nothing is recorded (inference / evaluation)
with torch.no_grad():
    y = x * 2
print("inside no_grad, tracked? ", y.requires_grad)   # False

# detach: pull one tensor off the tape, leave the rest tracking
z = (x * 2).detach()
print("detached, tracked?      ", z.requires_grad)    # False
print("safe hand-off to numpy  :", z.numpy())         # no graph attached
```

The graph is **dynamic**. PyTorch builds it as your Python runs (define-by-run), which is why ordinary `if` and `for` control flow inside a model just works. The tape simply records whatever path ran. That tape, and the reverse walk over it, are not built-in magic. Appendix G rebuilds this exact mechanism from scratch in about forty lines of NumPy: a small `Value` class that remembers each operation and a reverse-topological `.backward()`. Read it once and `.grad` stops feeling mysterious.

Building models: nn.Module and nn.Linear

Real models are assembled from layers. `nn.Linear(d_in, d_out)` is the fully-connected layer of chapter 3. It holds a weight matrix and a bias, and computes $\mathbf{x}\mathbf{W}^{T} + \mathbf{b}$. You combine layers by subclassing `nn.Module`: register the layers in `__init__`, then describe the forward pass in `forward`. PyTorch tracks every parameter for you, so `model.parameters()` gives all of them to an optimizer.

PyTorch copy → Colab

```
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, d_in, d_hidden, d_out):
        super().__init__()
        self.fc1 = nn.Linear(d_in, d_hidden)   # weight (d_hidden, d_in) +
bias
        self.fc2 = nn.Linear(d_hidden, d_out)

    def forward(self, x):
        x = torch.relu(self.fc1(x))            # hidden layer + ReLU
        return self.fc2(x)                     # raw output "logits"

model = MLP(4, 16, 3)
print(model)
out = model(torch.randn(8, 4))                 # a batch of 8 -> shape (8,
3)
print("output shape:", out.shape)
print("param count :", sum(p.numel() for p in model.parameters()))  # 131
```

Notice the model returns raw **logits**, with no softmax at the end. That is deliberate. The classification loss below includes the softmax for numerical stability, so you leave the final activation off during training. For quick stacks of layers, `nn.Sequential(nn.Linear(4,16), nn.ReLU(), nn.Linear(16,3))` does the same thing without a class. Because every layer you assign to `self` registers itself, printing the model shows the whole tree, and `model.parameters()` (or `model.named_parameters()`, which also gives you each tensor's name) shows exactly the tensors an optimizer will update.

Activations and losses

Activations (chapter 4) are in `torch.nn.functional` (imported as `F`), or as module versions like `nn.ReLU()`. The **loss** measures how wrong a prediction is (chapter 6). Match the loss to the task. Use `MSELoss` for regression. Use `CrossEntropyLoss` for multi-class classification (it expects raw logits and integer class labels, and applies log-softmax internally). Use `BCEWithLogitsLoss` for binary or multi-label problems (raw logits plus a sigmoid, combined for stability).

PyTorch — copy → Colab

```
import torch
import torch.nn.functional as F

z = torch.tensor([-1.0, 0.0, 2.0])
print("relu   :", F.relu(z))
print("sigmoid:", torch.sigmoid(z))
print("softmax:", F.softmax(z, dim=0))    # sums to 1

# regression: mean squared error
pred = torch.tensor([2.5, 0.0, 2.0])
tgt  = torch.tensor([3.0, -0.5, 2.0])
print("MSE:", F.mse_loss(pred, tgt).item())

# classification: raw logits + integer labels (NOT one-hot)
logits = torch.tensor([[2.0, 0.5, 0.1]])   # one sample, 3 classes
label  = torch.tensor([0])                 # the true class index
print("cross-entropy:", F.cross_entropy(logits, label).item())
```

Optimizers

An **optimizer** holds the update rule that turns gradients into weight changes (chapters 7 and 9). You give it the model's parameters and a learning rate. It reads each parameter's `.grad` and steps. `SGD` (optionally with momentum) is the classic one. `Adam` adapts a per-parameter step size (Chapter 9) and is the usual first choice. `weight_decay` adds L2 regularization (chapter 12). You can even give different parts of the model different learning rates through **parameter groups**.

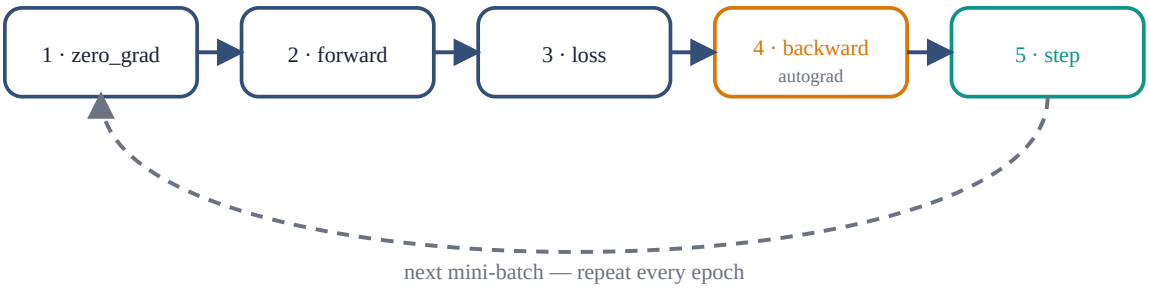
PyTorch copy → Colab

```
import torch
import torch.nn as nn

model = nn.Linear(10, 1)

# SGD with momentum and L2 weight decay
opt = torch.optim.SGD(model.parameters(), lr=0.01,
                      momentum=0.9, weight_decay=1e-4)

# Adam: the common default; adaptive per-parameter step sizes
opt = torch.optim.Adam(model.parameters(), lr=1e-3)

# parameter groups: a different lr for different tensors
opt = torch.optim.Adam([
    {"params": model.weight, "lr": 1e-3},
    {"params": model.bias,   "lr": 1e-2},
])
print(opt)
```

The training loop

Everything comes together in one loop that you will write, in some form, for every project. At its core are five steps per batch, in this order: **zero the gradients**, **forward** to predictions, compute the **loss**, **backward** to fill `.grad`, and **step** the optimizer. Forgetting `zero_grad()` is the classic bug. Otherwise, gradients accumulate across batches and corrupt the update. This template is deliberately general and is the one the case studies reuse. Swap the loss for your task

and it is ready. The loop itself is only the skeleton. Many decisions sit around it: how many epochs to run, which learning-rate schedule to use, when to stop, and how to validate. These are the complete training recipe of Chapter 25.



The training loop's five steps, in order, for one mini-batch. Steps 1–3 and 5 are ordinary tensor code. Step 4, `backward`, is autograd running backpropagation (Chapter 8). The dashed arrow returns control for the next batch, and the whole loop repeats every epoch. The scheduling around it is the complete recipe of Chapter 25. Miss step 1 and gradients from earlier batches build up and corrupt the update.

PyTorch copy → Colab

```
import torch
import torch.nn as nn

def train(model, loader, epochs=10, lr=1e-3, device="cpu"):
    model.to(device)
    loss_fn = nn.CrossEntropyLoss()                # swap for your task
    opt = torch.optim.Adam(model.parameters(), lr=lr)

    for epoch in range(epochs):
        model.train()                              # training mode
        running = 0.0
        for xb, yb in loader:                      # one batch at a time
            xb, yb = xb.to(device), yb.to(device)
            opt.zero_grad()                        # 1. clear old gradients
            pred = model(xb)                       # 2. forward pass
            loss = loss_fn(pred, yb)               # 3. compute loss
            loss.backward()                        # 4. backprop (autograd)
            opt.step()                             # 5. update the weights
            running += loss.item() * xb.size(0)
```

```
        avg = running / len(loader.dataset)
        print(f"epoch {epoch+1:2d}  loss {avg:.4f}")
    return model
```

To prove the loop is doing ordinary calculus, here is the same forward → loss → backward → step cycle written by hand in NumPy for a one-parameter problem, minimizing $L(w) = (w - 3)^2$. The analytic gradient is $2(w - 3)$. Each step moves `w` downhill and the loss falls toward zero. This is exactly what `loss.backward(); opt.step()` automate.

Python · NumPy ▶ runnable in the app

```python
import numpy as np

# minimize L(w) = (w - 3)**2 ;  dL/dw = 2*(w - 3)
w, lr = 0.0, 0.1
for step in range(1, 21):
    grad = 2 * (w - 3)          # "backward"
    w = w - lr * grad           # "step"
    if step % 5 == 0:
        print(f"step {step:2d}: w = {w:.4f}  loss = {(w - 3)**2:.5f}")
print("converged near w =", round(w, 4), "(target 3)")
```

Datasets and DataLoader

Feeding data in batches is the job of two small classes. A **Dataset** answers "how many samples?" (`__len__`) and "give me sample i" (`__getitem__`). If your data is already in tensors, `TensorDataset` wraps them for you. A **DataLoader** wraps a Dataset to produce shuffled mini-batches, optionally across several worker processes. The training loop above simply iterates the loader.

PyTorch copy → Colab

```python
import torch
from torch.utils.data import Dataset, DataLoader, TensorDataset

# quickest path: wrap tensors you already have
X = torch.randn(500, 4)
y = torch.randint(0, 3, (500,))
train_ds = TensorDataset(X, y)
```

```
# or define your own for custom / lazy loading
class MyData(Dataset):
    def __init__(self, X, y):
        self.X, self.y = X, y
    def __len__(self):
        return len(self.X)
    def __getitem__(self, i):
        return self.X[i], self.y[i]

loader = DataLoader(train_ds, batch_size=32, shuffle=True)
xb, yb = next(iter(loader))
print(xb.shape, yb.shape)     # torch.Size([32, 4]) torch.Size([32])
```

The `DataLoader` arguments set the batching policy: `batch_size` sets how many samples make up one optimizer step, `shuffle=True` reorders the data every epoch so batches differ each pass, `num_workers` spreads loading across processes to keep a GPU busy, and `drop_last=True` discards an incomplete final batch when a fixed batch size matters.

Train vs eval mode, and saving models

Some layers behave differently while training than at prediction time. **Dropout** (chapter 12) is active in training but off at test, and **batch normalization** uses batch statistics while training and running averages afterward. `model.train()` and `model.eval()` switch between these two modes. At inference you also wrap the forward pass in `torch.no_grad()`, which skips gradient tracking to save memory and time. To store a model, save its **state dict** (the dictionary of learned tensors) and load it back into a matching architecture.

PyTorch — copy → Colab

```
import torch
import torch.nn as nn

model = nn.Linear(4, 3)

# ---- inference: eval mode + no gradient tracking ----
model.eval()
with torch.no_grad():
    preds = model(torch.randn(16, 4)).argmax(dim=1)
print("predicted classes:", preds[:5])
```

```
# ---- save and load the learned weights ----
torch.save(model.state_dict(), "model.pt")

fresh = nn.Linear(4, 3)                      # same architecture
fresh.load_state_dict(torch.load("model.pt"))
fresh.eval()
print("reloaded OK")
```

Save the `state_dict`, not the whole model object. The dictionary of tensors is portable across code versions, while pickling the object ties the file to your exact class layout. Rebuild the architecture in fresh code, then load the weights back in.

Putting it together: a complete run

Here is everything above assembled into one screen: make data, wrap it, build a model, train with the five steps, and measure. Nothing in it is new. Read it top to bottom and you will recognize every line from the sections above. This is the minimal end-to-end program, and it is exactly the shape the case studies (chapters 30–34) grow into. Swap `nn.Sequential` for your architecture and `CrossEntropyLoss` for your task, and the same forty lines train almost anything.

PyTorch — copy → Colab

```
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader

torch.manual_seed(0)
device = "cuda" if torch.cuda.is_available() else "cpu"

# 1. toy data: two Gaussian blobs -> a 2-class problem
n = 400
X = torch.randn(n, 2)
X[:n // 2] += 2.0                        # push class 0 up and to the right
y = torch.zeros(n, dtype=torch.long)     # int64 labels for CrossEntropyLoss
y[n // 2:] = 1
loader = DataLoader(TensorDataset(X, y), batch_size=32, shuffle=True)

# 2. model, loss, optimizer
```

```
model = nn.Sequential(
    nn.Linear(2, 16), nn.ReLU(), nn.Linear(16, 2)
).to(device)
loss_fn = nn.CrossEntropyLoss()         # expects raw logits + int64 labels
opt = torch.optim.Adam(model.parameters(), lr=1e-2)

# 3. train: the five steps, every batch, every epoch
for epoch in range(20):
    model.train()
    for xb, yb in loader:
        xb, yb = xb.to(device), yb.to(device)
        opt.zero_grad()                 # 1. clear
        logits = model(xb)              # 2. forward
        loss = loss_fn(logits, yb)      # 3. loss
        loss.backward()                 # 4. backward (autograd)
        opt.step()                      # 5. update

# 4. evaluate: eval mode + no gradient tracking
model.eval()
with torch.no_grad():
    pred = model(X.to(device)).argmax(dim=1).cpu()
acc = (pred == y).float().mean().item()
print(f"train accuracy: {acc:.3f}")     # ~0.99 on these well-separated
blobs
```

Common PyTorch errors, and their fixes

Forgot `zero_grad()`: the loss stops improving or explodes. Gradients from old batches are building up. **Device mismatch** ("expected all tensors on the same device"): send both model and batch to the same `device`. **Wrong label dtype** for `CrossEntropyLoss`: pass integer class indices (`int64`), not one-hot floats, and give it raw logits, not softmax outputs. **Metrics that need a plain number**: call `.item()` on a scalar loss, or `.detach()` before converting to NumPy, so you do not carry the whole computation graph along.

In the real world

This exact stack (`nn.Module` models, autograd, an `Adam` optimizer and a `DataLoader` loop) is what trains production systems, from medical-image classifiers to the language models behind chat assistants. It is only scaled up with more layers, more data and many GPUs. Research papers release their code as PyTorch `nn.Module` classes. Frameworks such as PyTorch Lightning and the Hugging Face `Trainer` are thin wrappers that automate this same loop above. Learn this template well and the rest is mostly about choosing the right layers and data.

Exercises

1. In the training loop, what fails if you delete the `opt.zero_grad()` line? CONCEPT

2. Verify the derivative of the sigmoid, $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, at $z = 0.5$ using a finite difference in NumPy. CODE

3. Why do we call `model.eval()` and wrap inference in `torch.no_grad()` at prediction time? CONCEPT

4. How many learnable parameters does `nn.Linear(20, 5)` hold? MATH

5. Implement a single gradient-descent step for linear regression in NumPy: compute the mean-squared-error gradient with respect to the weights and update them, mirroring `loss.backward(); opt.step()`. CODE

6. A colleague's classifier trains but the loss barely moves. Their model ends the forward pass with `return torch.softmax(self.fc2(x), dim=1)` and the loss is `nn.CrossEntropyLoss()`. What is the bug, and what is the fix? CODE

7. Both `torch.no_grad()` and `.detach()` stop gradients from flowing. How do they differ, and when would you reach for each? CONCEPT

8. Translate this NumPy to PyTorch using the mapping table, and name the one thing you must watch: `a = np.zeros((4, 4)); s = a.sum(axis=0).` CODE

APPENDIX

Appendix G · Autograd from scratch

> *In Appendix F you called* `loss.backward()` *and got a full gradient — one line, no calculus. That is not magic. It is a small machine you can understand, and in this appendix we build it. In about fifty lines of pure Python we will write a scalar* ***autograd engine****: a* `Value` *object that records every arithmetic step as you compute, then walks that record backward to give you every derivative at once. It is a small version of PyTorch's autograd, and it is the reverse-mode automatic differentiation of Chapter 8 made concrete — the same δ signals, the same fan-out sum rule, the same one-forward-one-backward cost. When it runs, we will train a tiny network with it and check its gradients against finite differences.*

Fundamental concept

The frameworks of Appendix F feel like they differentiate your formulas symbolically, the way you would with pencil and paper. They do not. They differentiate a *trace*: as your program runs, each primitive operation quietly adds a node to a graph, recording what it did and to which inputs. This is a **dynamic computational graph**. It is dynamic because it is built by running ordinary code, not declared ahead of time, so a Python `if` or `for` loop shapes the graph as the program runs. By the time your forward pass reaches the loss, the entire history of the computation sits in memory as a directed acyclic graph of tiny, differentiable steps.

Think of it as a factory that keeps every receipt. Raw materials (the inputs and parameters) pass through stations (add, multiply, tanh), and each station attaches a note to its output: *"I combined these particular parts this particular way."* When the finished product is graded (a single scalar loss), the complaint is walked back through the receipts in reverse. Each station reads the complaint arriving from downstream, checks its own note to see how sensitive its output was to each input, and sends a properly scaled complaint upstream. Nobody re-derives the whole

factory. Each station only does its one local calculation. That backward walk, accumulating a derivative at every node, is exactly backpropagation (Chapter 8). Here it is generalized from a layered network to *any* expression you can type.

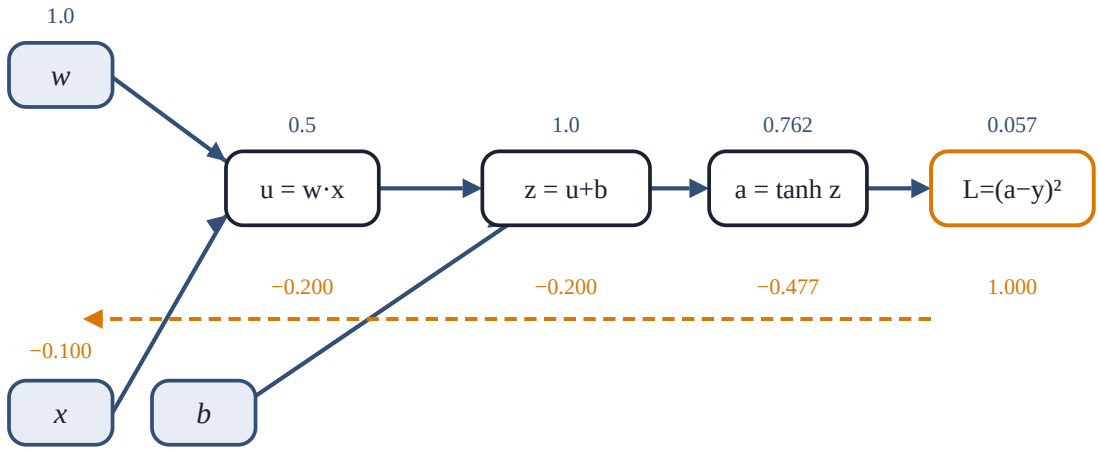
The engine is based on one idea from calculus: the chain rule, $\partial L/\partial u = (\partial L/\partial v)\cdot(\partial v/\partial u)$, applied locally at every node (Appendix B). If a node knows the derivative of its own output with respect to its own inputs (its **local gradient**), and it is told $\partial L/\partial$output by whatever used it, then it can produce $\partial L/\partial$input by a single multiply. Chain those multiplies from the loss back to the leaves and every parameter learns its share of the blame. Everything below is just a careful method around that one multiply.

Anatomy of a Value

The whole engine is built on one small object. A `Value` wraps a single number and remembers where it came from. It carries four things. `.data` is the forward number itself. `.grad` is the derivative of the final loss with respect to this value, $\partial L/\partial$(this). It starts at zero and is filled in during the backward pass. It is precisely the **adjoint** $\bar{v}$ of Chapter 8, and for a layer's pre-activation it is exactly that layer's error signal δ. `_prev` is the set of **parent** `Value`s this one was built from, the incoming edges of the graph. And `_backward` is a little function, a **closure**, that knows this node's local rule: when called, it takes the node's own `.grad` and pushes the correctly scaled contribution into each parent's `.grad`.

Every operation is **overloaded** to do two jobs at once: compute the forward number, and set up the backward closure. When you write `c = a * b`, the `__mul__` method builds a new `Value` whose `.data` is `a.data * b.data`, whose parents are `{a, b}`, and whose `_backward` encodes the local derivatives $\partial c/\partial a = b$ and $\partial c/\partial b = a$. Nothing is differentiated yet. The closure just *captures* the rule, to be run later when `c.grad` is known. Addition captures $\partial(a+b)/\partial a = \partial(a+b)/\partial b = 1$. The power `a**k` captures $k\cdot a^{k-1}$. A nonlinearity such as tanh captures its slope $1 - \tanh^2(z)$ (Chapter 4). Each op is a self-contained pair of a forward formula and its local gradient. Add a new op by writing exactly that pair.

One rule is the key to the whole scheme, and it is the same subtle point noted in Chapter 8: when a value **fans out** (is used by more than one downstream operation), its adjoint is the **sum** of the contributions returning along every branch. That is why `_backward` always writes `self.grad += …`, accumulating, never `self.grad = …`. A parameter reused in several places, a bias applied across a batch, a residual connection (Chapter 14) all collect gradient from all the paths they influence. The simple `+=` is what implements the multivariate chain rule. Forget it, overwrite instead of add, and every fan-out silently loses all but one of its gradient paths.



backward: seed ∂L/∂L = 1, then push grad = ∂L/∂· right → left

The graph a single expression builds. Running `L = (tanh(w·x + b) − y)**2` leaves this trace: white boxes are op nodes, blue boxes are input `Value`s, and every arrow is a parent → child edge stored in `_prev`. Blue numbers (above) are the forward `.data` for w=1, x=0.5, b=0.5, y=1. Amber numbers (below) are the `.grad` filled in by the backward sweep. The output is seeded with ∂L/∂L = 1 and each node's local rule multiplies it downward.

Warm-up: gradients by hand-built closures

Before the class, do it once by hand so nothing stays hidden. Take the tiny expression f = a·b + a. Note that a is *used twice* (once inside the product and once added on), so it fans out, and its gradient must accumulate. We build the two backward steps as explicit lines and check the answer against a finite difference. This is the entire engine in its simplest form. The `Value` class just automates writing these lines for you.

Python · NumPy ▶ runnable in the app

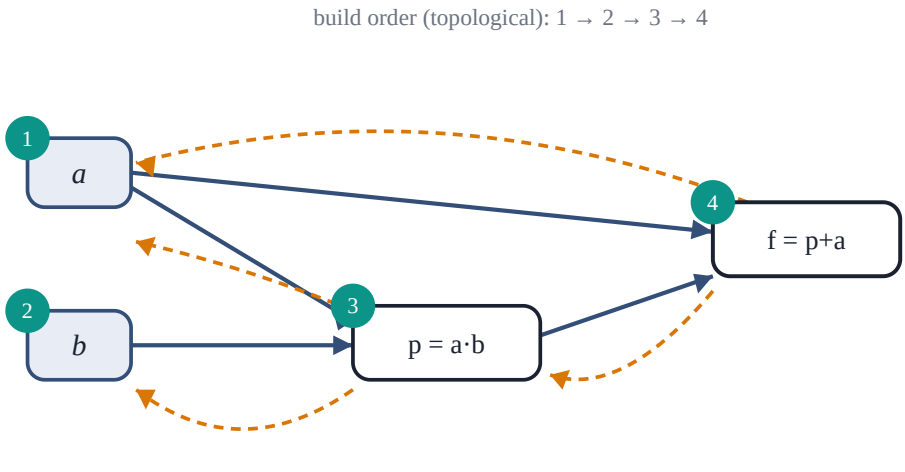
```python
import numpy as np

# f = a*b + a, differentiated by hand-built "backward" steps.
a, b = 3.0, 4.0
p = a * b                  # p = a*b
f = p + a                  # f = p + a   (a fans out: used in p AND here)

# adjoints: seed df/df = 1, then push gradient back through each node
f_bar = 1.0
p_bar = f_bar * 1.0                # node f = p + a:  df/dp = 1
a_bar = f_bar * 1.0                # node f = p + a:  df/da = 1  (first path)
a_bar += p_bar * b                 # node p = a*b:    a FANS OUT, so accumulate (+=)
b_bar  = p_bar * a                 # node p = a*b:    dp/db = a

print("df/da =", a_bar, " (expect b+1 =", b + 1, ")")   # 5.0
print("df/db =", b_bar, " (expect a   =", a, ")")       # 3.0

# finite-difference sanity check on df/da (central difference)
h = 1e-6
fd = ((a + h) * b + (a + h) - ((a - h) * b + (a - h))) / (2 * h)
print("finite-diff df/da =", round(fd, 5))              # 5.0
```

The two contributions to `a_bar` (the direct 1 from the sum and the b from the product) add to b + 1, the correct ∂f/∂a. Written by hand this is slow and easy to get wrong for anything larger. The point of the engine is to generate exactly these lines automatically from the ops you actually used, in the right order, with the accumulation handled for you.

Topological order and the backward sweep

Automating the warm-up needs one more idea: *order*. A node may only run its `_backward` once its own `.grad` is complete. That is, only after **every** node that uses it has already pushed its contribution back. In the warm-up, `a`'s gradient is not finished until both `f` and `p` have contributed. So the backward pass cannot visit nodes in any order. It must visit each node only after all of its children, which is the reverse of a **topological order** of the graph.

A topological order lists every node after all of its parents (a valid build/forward order). Sweeping it in reverse therefore lists every node after all of its children (a valid backward order). We get one with a depth-first traversal: to place a node, first recurse into its parents, then append the node itself. Because the graph is a directed acyclic graph (operations only ever point from older values to newer ones, never in a cycle), this always succeeds. With the list ready, `backward()` does three things: build the topological order from the output, seed the output's gradient with $\partial L/\partial L = 1$, then call each node's `_backward` in reverse. That single reverse sweep is the whole backward pass. It touches each node once and each edge once, so its cost is proportional to the size of the graph. One backward pass costs about the same as one forward pass, no matter how many parameters feed in. This is exactly why reverse mode, not forward mode or finite differences, is the right tool when there is one scalar loss and many inputs (Chapter 8).



The backward sweep follows reverse topological order. Nodes are numbered in a valid forward/build order (teal badges). `backward()` seeds node 4 with grad 1 and runs `_backward` in the reverse order 4, 3, 2, 1. Because `a` fans out to both `p` (node 3) and `f` (node 4), it is visited last and receives gradient from both amber branches. This is the `+=` accumulation of the fan-out sum rule.

The engine, in code

Here is the whole thing. The `Value` class is under fifty lines. The rest builds a 2-input → 2-hidden (tanh) → 1-output network entirely out of `Value` objects, runs one forward and one backward pass, prints two parameter gradients, **finite-difference-checks** one of them, and then

runs a short gradient-descent loop that actually trains the net. Press **Run**: the analytic and numeric gradients agree to six decimals, and the loss falls to almost zero.

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(1)

class Value:
    """A scalar node in a dynamic autodiff graph."""
    def __init__(self, data, _children=()):
        self.data = float(data)          # the forward value
        self.grad = 0.0                  # dL/d(self), filled by backward()
        self._backward = lambda: None    # local rule: push grad to parents
        self._prev = set(_children)      # the Values this one was built
from

    def __add__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data + other.data, (self, other))
        def _backward():
            self.grad  += out.grad       # d(a+b)/da = 1
            other.grad += out.grad       # fan-out: ACCUMULATE, never
overwrite
        out._backward = _backward
        return out

    def __mul__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data * other.data, (self, other))
        def _backward():
            self.grad  += other.data * out.grad     # d(ab)/da = b
            other.grad += self.data  * out.grad     # d(ab)/db = a
        out._backward = _backward
        return out

    def __pow__(self, k):                           # k is a plain number
        out = Value(self.data ** k, (self,))
        def _backward():
            self.grad += k * self.data ** (k - 1) * out.grad
        out._backward = _backward
        return out
```

```
    def tanh(self):
        t = np.tanh(self.data)
        out = Value(t, (self,))
        def _backward():
            self.grad += (1 - t * t) * out.grad     # tanh'(z) = 1 -
tanh(z)^2
        out._backward = _backward
        return out

    def __neg__(self):         return self * -1
    def __sub__(self, other):  return self + (-other)
    def __radd__(self, other): return self + other
    def __rmul__(self, other): return self * other

    def backward(self):
        topo, seen = [], set()
        def build(v):                       # depth-first topological sort
            if v not in seen:
                seen.add(v)
                for parent in v._prev:
                    build(parent)
                topo.append(v)              # node appended AFTER its parents
        build(self)
        self.grad = 1.0                     # seed dL/dL = 1
        for v in reversed(topo):            # sweep in reverse topological
order
            v._backward()

# ---- a tiny 2-input -> 2-hidden(tanh) -> 1-output network of Values ----
rv = lambda: Value(np.random.randn() * 0.5)    # one seeded random parameter
W1 = [[rv(), rv()], [rv(), rv()]]    # W1[j][i]: hidden unit j, input i
b1 = [rv(), rv()]
W2 = [rv(), rv()]                     # output weights
b2 = rv()
params = [w for row in W1 for w in row] + b1 + W2 + [b2]

x = [Value(0.5), Value(-1.0)]         # one input example
y = Value(1.0)                        # its target

def forward(x):
    h = []
    for j in range(2):                # hidden layer, tanh units
        s = b1[j]
```

```
        for i in range(2):
            s = s + W1[j][i] * x[i]
        h.append(s.tanh())
    o = b2                              # linear output unit
    for j in range(2):
        o = o + W2[j] * h[j]
    return o

pred = forward(x)
loss = (pred - y) ** 2                  # squared error on one example
loss.backward()                         # ONE reverse sweep fills every .grad
print(f"pred = {pred.data:.4f}   loss = {loss.data:.4f}")
print(f"dL/dW2[0] = {W2[0].grad:.6f}   dL/db2 = {b2.grad:.6f}")

# ---- finite-difference check on one parameter (central difference) ----
def loss_value():                       # rebuild the graph at the current
params
    return ((forward(x) - y) ** 2).data
p = W2[0]; g = p.grad; o = p.data; eps = 1e-6
p.data = o + eps; Lp = loss_value()
p.data = o - eps; Lm = loss_value()
p.data = o                                                  # restore
print(f"grad check W2[0]: analytic {g:.6f}   numeric {(Lp - Lm) / (2 *
eps):.6f}")

# ---- gradient-descent loop: the engine can now TRAIN the network ----
for step in range(200):
    loss = (forward(x) - y) ** 2
    for q in params: q.grad = 0.0    # zero_grad -- grads accumulate, so
reset!
    loss.backward()                  # backprop
    for q in params: q.data -= 0.1 * q.grad    # SGD step:  theta <- theta -
eta*grad
print(f"after 200 steps: loss = {loss.data:.2e}   pred =
{forward(x).data:.4f}  (target 1.0)")
```

Read what was printed. The single `loss.backward()` filled `.grad` on every one of the eleven parameters, and the finite-difference check (change one weight by ±ε, remeasure the loss, divide by 2ε) reproduces the engine's `dL/dW2[0]` to the last printed digit, agreeing to better than one part in a billion. That agreement is the standard proof that a gradient is correct, the very check Chapter 8 runs on its hand-derived δ's and the one the app's engine includes forty-four of. Then

the training loop brings it all together: forward to a loss, `zero_grad`, `backward`, and a gradient-descent step $\theta \leftarrow \theta - \eta \nabla L$ (Chapter 7), two hundred times, driving the prediction onto its target. Notice the `zero_grad` is not optional. Because every `_backward` accumulates with `+=`, gradients from the previous step would otherwise add onto this one, exactly the bug Appendix F warns about when you forget `opt.zero_grad()`.

From δ to .grad: the same algorithm

Nothing in this engine is specific to neural networks, and that is the point. Chapter 8 presents backpropagation as four matrix equations for a layered network. This appendix presents it as a graph walk over scalar primitives. They are the same algorithm at two levels of detail. The chapter's error signal $\delta^{(l)} = \partial L / \partial z^{(l)}$ is just the `.grad` of the pre-activation node (the adjoint), and its recurrence $\delta^{(l)} = (\delta^{(l+1)} W^{(l+1)\mathrm{T}}) \odot g'(z^{(l)})$ is what our `_backward` closures compute one scalar at a time. The W^{T} multiply is the sum over a fan-out, and the $\odot\, g'$ factor is the local derivative a nonlinearity's closure applies. Where the chapter hand-derives those four equations once and hard-codes them, the engine rediscovers them automatically for whatever graph you happen to build. This is why you never re-derive backprop when you invent a new architecture.

Scale this up and you have PyTorch. Its tensors are our `Value` with an n-dimensional array in `.data` instead of one float. `requires_grad=True` marks the leaves whose `.grad` you want. Each operation registers a `grad_fn` that is our `_backward`. And `loss.backward()` runs the identical seed-then-reverse-topological-sweep you just wrote (Appendix F). The real versions add three things this toy leaves out: **tensors**, so one node carries a whole matrix and one local rule is a matmul rather than a scalar multiply; a wider library of differentiable primitives, each still just a forward formula paired with its **vector-Jacobian product**; and engineering for speed and memory, such as in-place-safe buffers, GPU kernels, and the activation-storage-versus-recompute tradeoff of gradient checkpointing (Chapter 8). The concepts do not change. If you understand why `self.grad += out.grad` has a plus, you understand the core of every deep-learning framework.

Two honest limitations. First, a scalar engine like this is perfect for teaching but hopeless for real computation: representing a weight matrix as thousands of separate `Value` nodes, each with its own Python closure, is orders of magnitude slower than one NumPy matmul. Real engines

vectorize exactly to avoid this. Second, autodiff differentiates the exact operations you ran, so it inherits their discontinuities. At a non-differentiable kink such as ReLU's elbow or an `abs` at zero, it returns a convenient subgradient. A truly discontinuous choice (an `argmax`, a hard threshold) blocks gradient flow entirely. Autodiff is not symbolic simplification and not numerical approximation. It is the exact chain rule evaluated along one execution trace, which is all training ever needs.

Exercises

1. In every `_backward` the parent's gradient is updated with `+=`, not `=`. Explain, using the warm-up f = a·b + a, exactly what goes wrong if you overwrite instead of accumulate. CONCEPT

2. Why must `backward()` call the nodes in reverse topological order? What could go wrong if it ran `_backward` in the order the nodes happened to be created? CONCEPT

3. Add a `relu` method to the `Value` class. State its forward value and the local gradient its `_backward` must push, and write the method. CODE

4. The engine finite-difference-checks an output weight. Show the same central-difference method works for a **bias**: with the `Value` class, build the single tanh neuron of the opening figure (w=1, x=0.5, b=0.5, y=1) and check the gradient of its bias b against the figure's −0.200. CODE

5. Reverse mode computes the gradient of one scalar loss with respect to all N inputs in a single backward pass. Estimating the same gradient by central finite differences costs how many forward evaluations of the loss, and why does that make finite differences unusable for training a real network? MATH

6. The engine defines `__sub__` as `self + (-other)` and `__neg__` as `self * -1`. Show that these reuse existing closures rather than needing new ones, and give the resulting local gradient of `d = a - b` with respect to `a` and `b`. MATH

APPENDIX

Appendix H · Datasets & further reading

> *Every project starts with data. A few well-known datasets have become the shared benchmarks used to compare methods and to give learners their first practice. This appendix is a guided tour. It covers what the classic datasets are, what each one teaches, where to find them, and how to load them. It then describes the synthetic sets built into this app's Playground and gives a further-reading list with notes. None of this needs heavy computation. The goal is a map of the area you will practice on. A reference catalog then groups the standard benchmarks by domain. A section on data hygiene — licenses, bias, canonical splits and leakage — ties the mechanics back to data-centric deep learning (chapter 26), ethics and fairness (chapter 29), and the case studies (chapters 30–34) where these datasets are used.*

Image datasets: MNIST, Fashion-MNIST, CIFAR-10

MNIST is the "hello world" of deep learning: 70,000 grayscale 28×28 images of handwritten digits 0–9 (60k train, 10k test), from LeCun and Cortes. It is small, clean and nearly solved. A plain MLP reaches ~98%. This makes it perfect for getting a pipeline working from start to finish (the MNIST case study, chapter 30). Its harder drop-in replacement, **Fashion-MNIST** (Zalando), keeps the exact 28×28, 10-class format but uses photos of clothing. So code written for one runs on the other unchanged, while needing a real model. **CIFAR-10** is harder again: 60,000 color 32×32 images across ten object classes (airplane, cat, ship, …). The three color channels and natural clutter make convolutional networks (chapter 13) truly necessary. All three come with `torchvision` and download on first use. Beyond these teaching sets is **ImageNet**: over a million high-resolution photos across 1,000 categories. It is the benchmark whose 2012 leaderboard (AlexNet) started the deep-learning era. It is too large to train casually, but it is the source of the pretrained backbones you will fine-tune in practice (modern vision architectures, chapter 14; transfer learning, chapter 22).

PyTorch copy → Colab

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

tfm = transforms.Compose([
    transforms.ToTensor(),                        # uint8 [0,255] -> float
[0,1]
    transforms.Normalize((0.1307,), (0.3081,)),  # MNIST mean / std
])

train = datasets.MNIST(root="data", train=True,  download=True,
transform=tfm)
test  = datasets.MNIST(root="data", train=False, download=True,
transform=tfm)

loader = DataLoader(train, batch_size=64, shuffle=True)
xb, yb = next(iter(loader))
print(xb.shape, yb.shape)   # torch.Size([64, 1, 28, 28]) torch.Size([64])
# swap datasets.FashionMNIST or datasets.CIFAR10 in for the other two
```

Tabular datasets: Iris and the UCI repository

Much of the world's data is a table of rows and columns. Here the classic datasets are small. **Iris** (Fisher, 1936) has 150 flowers, four measurements each, three species. You can reason about its four features by hand, which makes it ideal for a first classifier. The **UCI Machine Learning Repository** provides hundreds more: Wine, Breast Cancer Wisconsin, Adult (income), Bank Marketing, and the customer-**churn** tables used in the tabular case study (chapter 31). Tabular data usually comes as a CSV. The workflow is to load it (commonly with `pandas`), separate the target column, standardize the numeric features, and convert to tensors. Categorical columns must be encoded as numbers first: use one-hot for a few categories and learned embeddings for many. Fit any scaling on the *training* split alone, and reuse those statistics on validation and test. If you do not, you leak information about the held-out data. Scikit-learn also includes the smallest classic datasets for one-line access.

PyTorch copy → Colab

```
import pandas as pd
import torch

df = pd.read_csv("churn.csv")
y = torch.tensor(df.pop("churn").values, dtype=torch.long)     # target
column
X = torch.tensor(df.values, dtype=torch.float32)               # the
features
X = (X - X.mean(0)) / (X.std(0) + 1e-8)                        # standardize
print(X.shape, y.shape)

# scikit-learn ships the small classics directly:
from sklearn.datasets import load_iris
iris = load_iris()
print(iris.data.shape, iris.target[:5])   # (150, 4)  [0 0 0 0 0]
```

Text datasets: IMDB and friends

For a first language task, **IMDB** (Maas et al., 2011) is the standard: 50,000 movie reviews labeled positive or negative, a balanced binary sentiment problem (the text case study, chapter 32). Related datasets include **AG News** (topic classification), **SST** (fine-grained sentiment) and **Yelp** reviews. Raw text must be turned into integer token ids before a model sees it. The modern way is the Hugging Face `datasets` and `transformers` libraries, which fetch the data and provide matching tokenizers. The older `torchtext` pipelines still work but are mostly deprecated.

PyTorch copy → Colab

```
# modern route: the Hugging Face 'datasets' + 'transformers' libraries
from datasets import load_dataset
from transformers import AutoTokenizer

imdb = load_dataset("imdb")                  # train / test splits of reviews
print(imdb["train"][0]["label"], imdb["train"][0]["text"][:60])

tok = AutoTokenizer.from_pretrained("bert-base-uncased")
```

```
batch = tok(imdb["train"][:4]["text"], padding=True,
            truncation=True, return_tensors="pt")
print(batch["input_ids"].shape)             # (4, sequence_length)
```

Time-series datasets

Sequence problems (recurrent networks, chapter 15; the forecasting case study, chapter 33) come from sensors, markets and logs. Small teaching sets include the classic **airline passengers** series, **daily minimum temperatures**, and **sunspots**. Larger benchmarks include the UCI **electricity** and **household power** load sets, hourly **traffic** and **weather**, and the **M4** forecasting competition series. Unlike images, a raw series is not yet a supervised dataset. You must cut it into overlapping windows, where each past window predicts the next value. Importantly, the train/test split must be *chronological*: never shuffle, or the future leaks into training. Building the windows is a few lines of NumPy, and this one actually runs here.

Python · NumPy ▶ runnable in the app

```
import numpy as np

series = np.arange(1, 13, dtype=float)      # a toy monthly series 1..12
window = 3

X, y = [], []
for i in range(len(series) - window):
    X.append(series[i:i + window])          # the past 'window' values
    y.append(series[i + window])            # the next value to predict
X, y = np.array(X), np.array(y)

print("X shape:", X.shape, " y shape:", y.shape)
print("first 3 windows:\n", X[:3])
print("their targets  :", y[:3])
```

A catalog by domain

The text above covers the datasets you meet first. The tables below give a quick reference across the whole field. Sizes are rounded to orders of magnitude, not exact counts. Every set loads through one of the tools in the next section. Pick by *task* and *scale*: small enough to work

with quickly, hard enough that a real improvement shows.

Vision. Pixels in; a label, box or mask out. This is where convolutional networks prove themselves (chapters 13–14).

DATASET	SIZE	TASK	NOTE
MNIST	70k	28×28 gray · 10 digits	the "hello world"; ~99% is easy
Fashion-MNIST	70k	28×28 gray · 10 clothes	drop-in, genuinely harder MNIST
CIFAR-10	60k	32×32 color · 10 classes	needs a real CNN
CIFAR-100	60k	32×32 color · 100 classes	fine-grained; 20 superclasses
SVHN	~600k	32×32 color · house numbers	real-world digits
ImageNet (ILSVRC)	1.28M	~224px · 1000 classes	the pretraining benchmark
COCO	330k	detection · segmentation · captions	boxes, masks and captions

Text / NLP. Tokens in; a class, span or next word out. See attention and language models (chapters 16–17).

DATASET	SIZE	TASK	NOTE
IMDB	50k	binary sentiment	balanced movie reviews
SST-2	68k	binary sentiment	phrase-level; part of GLUE
AG News	128k	4-way topic	news headlines + bodies
GLUE	9 tasks	language-understanding suite	one leaderboard, many tasks
SQuAD	100k+	extractive question answering	v2 adds unanswerable questions
WikiText	2M / 103M tok	language modeling	clean Wikipedia prose
C4 / Common Crawl	~750GB / PB	LM pretraining corpus	web-scraped; needs filtering

Tabular. Rows and columns. This is the format most real-world data actually comes in (the churn case study, chapter 31).

DATASET	SIZE	TASK	NOTE
UCI repository	600+ sets	classification / regression	the classic collection point
Kaggle Datasets	100k+ sets	everything	community; check the license
Adult / Census Income	48.8k	predict income > $50k	sensitive columns; fairness testbed
Titanic	1,309	survival classification	the Kaggle tutorial set

Audio. Waveforms and spectrograms, usually turned into a sequence problem.

DATASET	SIZE	TASK	NOTE
LibriSpeech	1,000 h	speech recognition	read audiobooks; clean/other splits
UrbanSound8K	8,732 clips	10-way sound tagging	street sounds; 10 fixed folds

Time-series. One value after another. Split by time, never at random (recurrent nets, chapter 15; the forecasting case study, chapter 33).

DATASET	SIZE	TASK	NOTE
M4	100k series	forecasting competition	mixed frequencies and domains
Electricity (UCI)	370 clients	load forecasting	hourly consumption
Traffic (PeMS)	862 sensors	occupancy forecasting	hourly freeway-lane readings

Graph. Nodes joined by edges. Features spread along the links (graph neural networks, chapter 18).

DATASET	SIZE	TASK	NOTE
Cora	2,708 nodes	node classification	citation network; 7 classes
Citeseer	3,327 nodes	node classification	citation network; 6 classes
OGB	many scales	node · link · graph tasks	large, with standardized splits

Loading datasets: one line each

Three ecosystems cover almost everything. `torchvision.datasets` provides the vision classic datasets. It caches each one on first use, behind a `train`/`test` flag for the canonical split. The Hugging Face `datasets` library covers the most: thousands of text, audio, image and tabular sets behind one `load_dataset` call. It has a `streaming=True` mode for datasets too large for disk (C4, Common Crawl). And scikit-learn puts the small classic datasets one function away, plus `fetch_openml` for a few hundred more. All three return the *official* splits. Use them, so your numbers compare with everyone else's.

PyTorch copy → Colab

```
# 1) torchvision — vision sets, downloaded and cached on first use
from torchvision import datasets
mnist = datasets.MNIST("data", train=True,  download=True)
cifar = datasets.CIFAR10("data", train=True, download=True)   # also
CIFAR100, SVHN, ...

# 2) Hugging Face 'datasets' — text, audio, tabular, images (and streamable)
from datasets import load_dataset
imdb = load_dataset("imdb")                          # train / test
sst2 = load_dataset("glue", "sst2")                  # a named GLUE sub-task
c4   = load_dataset("c4", "en", streaming=True)   # too big for disk ->
stream it

# 3) scikit-learn — the small classics, straight into memory
from sklearn.datasets import load_iris, fetch_openml
iris  = load_iris()                                  # (150, 4) arrays
adult = fetch_openml("adult", version=2, as_frame=True)   # hundreds more
via OpenML
```

When a dataset comes with no standard split (as with much tabular data), you make one yourself in a reproducible way: seed the generator, shuffle the row *indices* once, and slice. The same seed gives the same split on every machine. This is the one piece you can run here.

Python · NumPy ▶ runnable in the app

```
import numpy as np
np.random.seed(0)                          # determinism -> reproducible
split
```

```
n = 200
X = np.random.normal(size=(n, 2))          # a synthetic 2-feature dataset
y = (X[:, 0] + X[:, 1] > 0).astype(int)    # a simple linear labelling rule

idx = np.random.permutation(n)             # shuffle the ROW INDICES, once
tr, va = int(0.70 * n), int(0.85 * n)      # 70 / 15 / 15 boundaries
train, val, test = idx[:tr], idx[tr:va], idx[va:]

print("sizes:", train.size, val.size, test.size)      # expected: 140 30 30
print("train balance:", round(float(y[train].mean()), 3))
print("splits disjoint:", len(set(train) & set(test)) == 0)   # expected:
True
```

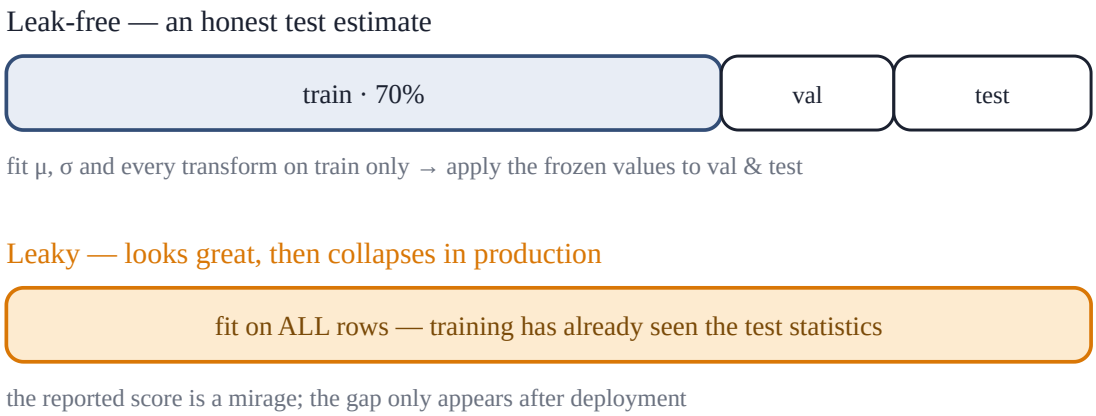
Data hygiene: licenses, bias, splits and leakage

A dataset is not automatically yours to use. Licenses vary widely. Some are permissive: MNIST, CIFAR and most UCI sets are effectively free for research. Others are share-alike Creative Commons licenses. Others are research-only terms that forbid commercial use. For example, ImageNet's images are licensed from their owners for non-commercial research. Kaggle competition data carries contest-specific rules. And web-scraped datasets like C4 and Common Crawl inherit the copyright and consent status of billions of pages that nobody cleared. Read the license before you train, and certainly before you release your work. The ethics chapter (chapter 29) treats consent, copyright and the rights of the people *in* the data as central concerns, not just paperwork.

Every dataset is a sample, and every sample is skewed. ImageNet's photos over-represent certain countries, so a model recognizes a Western wedding more reliably than others. The Adult/Census set builds the unfairness of 1994 census data into its sensitive columns. Face and medical datasets often under-represent darker skin tones. These are not bugs to remove, but properties to measure and report. Data-centric deep learning (chapter 26) is the practice of checking, cleaning and rebalancing data on purpose. The ethics chapter (chapter 29) connects those skews to real harms once a model is deployed.

Most benchmarks provide a fixed train/test split for a reason: it lets a number in your paper be compared with a number in someone else's. If you invent your own, you can no longer claim to beat the published result. You have measured a different problem. Use the canonical split, take a

validation slice from the *training* portion for tuning, and use the test set only to report a single final number, once. The case studies (chapters 30–34) follow exactly this protocol.



Fit every data-dependent transform (scaling, imputation, encoders, feature selection, resampling) on the training split alone. Then apply the fixed statistics to validation and test. Fitting on the full dataset lets held-out information leak into training and makes the score you report too high.

Leakage is any path by which information about the test set gets into training. It is the most common reason a model that looked great in a notebook fails in the real world. The less obvious forms are worth remembering. One is fitting a scaler, imputer or encoder on the whole dataset before splitting (fix: fit on train alone, as above). Another is shuffling a time series so the future trains the past (fix: a chronological split). A third is near-duplicate rows, the same image or sentence in both splits, which quietly turn memorization into a high score. The modern version is **benchmark contamination**: because language models pre-train on large web datasets like Common Crawl, a public test set's questions may already be in the training data. So a strong score measures recall, not reasoning. Remove duplicates across splits, keep the test set truly separate, and be suspicious of any result that looks too good.

The Playground datasets

The interactive Playground in this app generates its own simple 2-D datasets so you can watch a network learn a boundary in real time. Each one focuses on a single lesson. **XOR** (four clusters in a checker pattern) is the classic proof that a single linear boundary fails and depth is required (chapter 2). **Spirals** need a highly curved boundary, and they reveal weak capacity or poor initialization. **Moons** (two overlapping crescents) and **circles** (one class surrounding another)

are gentle non-linear problems, perfect for comparing activations and widths. **Checkerboard** repeats the XOR difficulty at a larger scale, and rewards real depth. **Gaussians** (two or more blobs) are near-linearly separable, a basic check where almost anything works. And the **regression curves** (sine, noisy polynomials) switch the task from classification to fitting a continuous function, so you can see under- and over-fitting (chapter 12) directly. Because they are low-dimensional and very fast, they let you build intuition for capacity, learning rate and regularization before you use a large dataset.

Further reading

A short, selective reading list. The books range from beginner level to reference level. The papers are the primary sources cited across the chapters. The courses and living resources are the best places to keep learning after the book ends.

- **Goodfellow, Bengio & Courville — *Deep Learning*** (MIT Press, 2016; free at deeplearningbook.org). The comprehensive reference for the theory in this book: representation, optimization, and every major architecture.
- **Bishop — *Pattern Recognition and Machine Learning*** (Springer, 2006). The rigorous probabilistic foundation (maximum likelihood, Bayesian inference, the bias–variance view) behind chapters 6, 12 and 24.
- **Bishop & Bishop — *Deep Learning: Foundations and Concepts*** (Springer, 2024; free online). The modern follow-up to PRML, rewritten around deep networks and transformers with the same probabilistic clarity.
- **Prince — *Understanding Deep Learning*** (MIT Press, 2023; free at udlbook.com). Very clear diagrams and notebooks. The best single book for geometric intuition about what each layer actually does.
- **Nielsen — *Neural Networks and Deep Learning*** (free online). The easiest intuition-first introduction to neurons, gradient descent and backpropagation. Excellent to read alongside chapters 1, 7 and 8.
- **Zhang, Lipton, Li & Smola — *Dive into Deep Learning*** (d2l.ai). A modern, fully runnable book with PyTorch code for every idea. It is the natural next step after these

appendices.

- **Murphy — *Probabilistic Machine Learning*** (MIT Press, 2022–23, two volumes). A very broad, math-heavy reference covering classical ML and deep learning. The place to turn for the full derivation.
- **Key papers:** Rumelhart, Hinton & Williams (1986) on backpropagation; LeCun et al. (1998) on convolutional networks; Krizhevsky et al. (2012), AlexNet; Kingma & Ba (2014), Adam; Vaswani et al. (2017), *Attention Is All You Need*; Goodfellow et al. (2014), GANs; Ho et al. (2020), diffusion models.
- **PyTorch documentation** (pytorch.org/docs) and its tutorials: the official, searchable reference for every API in Appendix F.

Well-known courses, with lecture notes and videos, all free:

- **Stanford CS231n — *Deep Learning for Computer Vision***. The standard convolutional-network course. Its notes on backpropagation and CNNs are essential reading (chapters 8, 13–14).
- **Stanford CS224n — *NLP with Deep Learning***. From word vectors through attention to large language models, with excellent lectures (chapters 16–17).
- **fast.ai — *Practical Deep Learning for Coders***. Top-down and code-first: you train a working model in lesson one, then look at the details underneath it.

Living resources, where the field keeps updating itself:

- **Papers with Code** (paperswithcode.com). Benchmarks, leaderboards, and, importantly, a link from each result to a runnable implementation. The fastest way to find the current best method for a task and its code.
- **arXiv** (arxiv.org, sections cs.LG, cs.CV, cs.CL). Where nearly every result appears first. Skim the daily listings, and follow citations with tools like Semantic Scholar or alphaXiv.
- **distill.pub and the community explainers**. Distill's interactive articles set the standard for visual explanation. Along with them, Chris Olah's blog, Jay Alammar's *Illustrated Transformer*, and Lilian Weng's *Lil'Log* explain hard topics one diagram at a time.

However you use these, the habit matters more than the source: read one paper a week from a leaderboard you care about, reproduce its main result on the canonical split, and keep a list of ideas to try. The field moves fast, but the fundamentals in this book are the stable core beneath all the change. Almost everything new combines them in new ways.

Exercises

1. Modify the windowing code to predict two steps ahead (a horizon of 2) instead of the very next value. CODE

2. Why must a time-series train/test split be chronological rather than random? CONCEPT

3. You have 500 labeled images and want a reliable accuracy estimate. Why prefer Fashion-MNIST over MNIST to show a new architecture? CONCEPT

4. A teammate reports 96% test accuracy on a churn model. Their pipeline: load the CSV, standardize every numeric column using the mean and standard deviation of the *entire* file, then split 80/20 into train and test and fit the model. Find the leakage and give the fix. CONCEPT

5. You want to show that a new data-augmentation trick really helps a convolutional network, running many controlled trials on a single laptop GPU in an afternoon. Which benchmark do you choose, and why not MNIST or ImageNet? CONCEPT

APPENDIX

Appendix I · Formulas & symbols quick reference

> *The whole book on one sheet. This appendix collects the formulas, derivatives and update rules used everywhere in the text. They are grouped by topic, and each one points to the chapter that derives it. It is meant for scanning, not reading. Every entry is a formula plus a one-line reminder of what it means and where it comes from. Notation follows the book throughout (Chapter 8): bold symbols are vectors or matrices, scalars are plain, and a batch of m examples is stored as the rows of a matrix.*

Notation & symbols

x, X	one input vector; a batch X of shape m×n (one example per row, one feature per column)
$W^{(l)}$, $b^{(l)}$	weight matrix (shape $n_{l-1}\times n_l$) and bias of layer l — the learned parameters
$z^{(l)}$, $a^{(l)}$	pre-activation $z^{(l)} = a^{(l-1)}W^{(l)} + b^{(l)}$ and activation $a^{(l)} = g(z^{(l)})$; $a^{(0)} = x$
g, σ	activation function; σ is specifically the logistic sigmoid; g' its derivative
ŷ, y	network prediction and target (ground-truth label)
L, $\mathscr{L}$	the loss (a single scalar to be minimized); J the regularized objective
m	number of examples averaged over (a batch or the full set)
η	learning rate (step size), a small positive scalar
∇	gradient operator; $\nabla_\theta L$ is the slope of L in the parameters θ
$\delta^{(l)}$	error signal $\partial L/\partial z^{(l)}$ at layer l — the quantity backprop propagates
⊙, ᵀ, ·	elementwise (Hadamard) product, transpose, dot/inner product
λ, p	regularization strength (L1/L2) and dropout probability
γ, β	a normalization layer's learnable scale and shift (β here is unrelated to momentum)
n_{in}, n_{out}	fan-in and fan-out: inputs feeding a neuron, and neurons it feeds
$\mathbb{E}$, $N(\mu,\sigma^2)$	expectation; the normal (Gaussian) distribution with mean μ, variance σ^2
⌊·⌋, Σ, Π	floor, summation, product

Activation functions & their derivatives — Ch.4

The derivative g′ is the factor backprop multiplies by, so it controls learning. Every entry below is exact.

FUNCTION	F(Z)	F′(Z)
Sigmoid σ	$1/(1+e^{-z})$	$\sigma(z)(1-\sigma(z)) \le 0.25$
Tanh	$(e^{z}-e^{-z})/(e^{z}+e^{-z})$	$1-\tanh^2(z)$
ReLU	$\max(0, z)$	1 if $z > 0$, else 0
Leaky ReLU	$\max(\alpha z, z)$	1 if $z > 0$, else α
ELU	z if $z > 0$, else $\alpha(e^{z}-1)$	1 if $z > 0$, else αe^{z}
Softplus	$\ln(1+e^{z})$	$\sigma(z)$
GELU	$z \cdot \Phi(z)$	$\Phi(z) + z \cdot \varphi(z)$
Swish / SiLU	$z \cdot \sigma(z)$	$\sigma(z) + z\,\sigma(z)(1-\sigma(z))$

Φ is the standard-normal CDF and φ its PDF. α is the negative-branch slope (0.01 for Leaky, 1 for ELU). A useful identity and the softmax with its Jacobian:

$$\tanh(z) = 2\sigma(2z) - 1$$

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\Sigma_j\, e^{z_j}} \qquad \frac{\partial p_i}{\partial z_j} = p_i(\delta_{ij} - p_j) \quad \text{— Jacobian} = \text{diag}(p) - p\,p^{T}$$

The finite-difference check below confirms each derivative in the table (import numpy only; press **Run**):

Python · NumPy ▶ runnable in the app

```
import numpy as np
h = 1e-6
z = np.array([-2.0, -0.5, 0.5, 2.0])          # test points, away from kinks
at 0
s = 1/(1+np.exp(-z))                           # sigmoid values
```

```
acts = {                                          # name: (function, analytic
f'(z))
    "sigmoid":  (lambda t: 1/(1+np.exp(-t)),              s*(1-s)),
    "tanh":     (np.tanh,                                 1-np.tanh(z)**2),
    "softplus": (lambda t: np.log1p(np.exp(t)),           1/(1+np.exp(-z))),
    "swish":    (lambda t: t/(1+np.exp(-t)),              s + z*s*(1-s)),
    "elu(a=1)": (lambda t: np.where(t>0, t, np.exp(t)-1), np.where(z>0, 1.0,
np.exp(z))),
}
print("activation   max|analytic - central difference|")
for name, (f, d_ana) in acts.items():
    d_num = (f(z+h) - f(z-h)) / (2*h)          # central difference
    print(f"{name:9}   {np.max(np.abs(d_ana - d_num)):.2e}")   # all ~1e-8:
derivatives verified
```

Loss functions — Ch.6

r = ŷ − y is the residual. p is a predicted probability. Softmax + cross-entropy and sigmoid + binary cross-entropy both give the simple output error $\delta^{(L)} = p - y$.

LOSS	FORMULA	USE / GRADIENT
MSE	$(1/2m)\ \Sigma\ \|\hat{y} - y\|^2$	regression; $\partial/\partial\hat{y} = (\hat{y} - y)/m$; Gaussian MLE
MAE	$(1/m)\ \Sigma\ \|r\|$	robust; grad sign(r); fits the median
Huber$_\delta$	$\frac{1}{2}r^2$ if $\|r\| \le \delta$, else $\delta(\|r\| - \frac{1}{2}\delta)$	quadratic core, linear tails
Binary CE	$-[y \ln p + (1 - y) \ln(1 - p)]$	$p = \sigma(z)$; $\partial L/\partial z = p - y$
Categorical CE	$-(1/m)\ \Sigma_i\ \Sigma_k\ y_k^{(i)} \ln p_k^{(i)}$	$p = \text{softmax}(z)$; $\partial L/\partial z = p - y$
Hinge	$\max(0, 1 - y\cdot\hat{y})$, $y \in \{-1, +1\}$	SVM / max-margin
Focal	$-(1 - p_t)^\gamma \ln p_t$	down-weights easy cases; $p_t = p$ if $y=1$ else $1-p$
KL	$\Sigma_k\ y_k \ln(y_k/p_k) = H(y,p) - H(y)$	distribution distance ≥ 0, $= 0 \Leftrightarrow p = y$

The four backpropagation equations — Ch.8

Forward: $z^{(l)} = a^{(l-1)}W^{(l)} + b^{(l)}$, $a^{(l)} = g(z^{(l)})$. Backward, with $\delta^{(l)} = \partial L/\partial z^{(l)}$:

$$\delta^{(L)} = \nabla_a L \odot g'(z^{(L)}) \quad \text{— output error}$$

$$\delta^{(l)} = (\delta^{(l+1)}\, W^{(l+1)T}) \odot g'(z^{(l)}) \quad \text{— recurrence}$$

$$\partial L \,/\, \partial W^{(l)} = a^{(l-1)T}\, \delta^{(l)} \quad \text{— weight gradient}$$

$$\partial L \,/\, \partial b^{(l)} = \Sigma_{\text{batch}}\, \delta^{(l)} \quad \text{— bias gradient}$$

With softmax + cross-entropy (or sigmoid + BCE) the output error simplifies to $\delta^{(L)} = p - y$, with no g' factor. Central-difference gradient check: $(L(\theta+\varepsilon) - L(\theta-\varepsilon))/2\varepsilon$, error $\propto \varepsilon^2$.

Optimizer update rules — Ch.9

$g_t = \nabla_\theta L$ is the gradient at step t. Defaults $\beta = 0.9$, $\rho = 0.9$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$.

$$\theta_{t+1} = \theta_t - \eta\, g_t \quad \text{— SGD}$$

$$v_t = \beta\, v_{t-1} + g_t\,, \quad \theta_{t+1} = \theta_t - \eta\, v_t \quad \text{— momentum}$$

$$v_t = \beta\, v_{t-1} + \nabla_\theta L(\theta_t - \eta\beta\, v_{t-1})\,, \quad \theta_{t+1} = \theta_t - \eta\, v_t \quad \text{— Nesterov (look-ahead)}$$

$$G_t = G_{t-1} + g_t^{\,2}\,, \quad \theta_{t+1} = \theta_t - \eta\, \frac{g_t}{\sqrt{G_t} + \varepsilon} \quad \text{— AdaGrad (step only shrinks)}$$

$$s_t = \rho\, s_{t-1} + (1-\rho)\, g_t^{\,2}\,, \quad \theta_{t+1} = \theta_t - \eta\, \frac{g_t}{\sqrt{s_t} + \varepsilon} \quad \text{— RMSProp}$$

$$m_t = \beta_1\, m_{t-1} + (1-\beta_1)\, g_t\,, \quad v_t = \beta_2\, v_{t-1} + (1-\beta_2)\, g_t^{\,2} \quad \text{— Adam moments}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^{\,t}}\,, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^{\,t}}\,, \quad \theta_{t+1} = \theta_t - \eta\, \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} \quad \text{— Adam (bias-corrected)}$$

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} - \eta \lambda \theta_t \quad \text{— AdamW (decoupled weight decay)}$$

Along a steady gradient, momentum's step approaches $\eta \cdot g/(1 - \beta)$ (a 10× amplification at $\beta = 0.9$). A cosine schedule lowers the rate over T steps:

$$v_\infty = \frac{g}{1 - \beta} \qquad \eta_t = \tfrac{1}{2}\,\eta_0 \left(1 + \cos\left(\frac{\pi\, t}{T}\right)\right) \quad \text{— warmup-then-cosine anneal}$$

Initialization & normalization — Ch.10–11

To keep the signal variance constant, $\mathrm{Var}(z) = n_{in} \cdot \mathrm{Var}(w) \cdot \mathrm{Var}(x)$ is pushed toward $\mathrm{Var}(z) = \mathrm{Var}(x)$. Sample zero-mean weights with:

SCHEME	VAR(W)	FOR
LeCun	$1/n_{in}$	linear / SELU
Xavier / Glorot	$2/(n_{in} + n_{out})$	tanh / sigmoid (fan-average)
He / Kaiming	$2/n_{in}$	ReLU & variants

$$U(-r, r): \quad \mathrm{Var}(w) = \frac{r^2}{3} \;\Rightarrow\; r = \sqrt{3 \cdot \mathrm{Var}(w)}$$

Normalization standardizes, then rescales. $\varepsilon \approx 10^{-5}$ keeps the division safe.

$$\text{BatchNorm:} \quad \hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma^2{}_B + \varepsilon}}, \quad y_i = \gamma\, \hat{x}_i + \beta \quad \text{— } \mu_B, \sigma^2{}_B \text{ per feature, over the batch}$$

$$\text{LayerNorm:} \quad \hat{x}_k = \frac{x_k - \mu}{\sqrt{\sigma^2 + \varepsilon}}, \quad y_k = \gamma_k\, \hat{x}_k + \beta_k \quad \text{— } \mu, \sigma^2 \text{ per example, over the D features}$$

$$\text{RMSNorm:} \quad y_k = \gamma_k \frac{x_k}{\mathrm{RMS}(x)}, \quad \mathrm{RMS}(x) = \sqrt{\frac{1}{D} \Sigma_k\, x_k{}^2 + \varepsilon} \quad \text{— no mean subtraction}$$

BatchNorm inference: $\mu_{run} \leftarrow (1-\alpha)\,\mu_{run} + \alpha\,\mu_B$, $\sigma^2_{run} \leftarrow (1-\alpha)\,\sigma^2_{run} + \alpha\,\sigma^2_B$ — frozen running stats at test time

Regularization — Ch.12

λ is the penalty strength and p the dropout rate. Biases are usually not penalized.

$$\text{L2 / weight decay: } J = L + \frac{\lambda}{2m}\,\Sigma_i\, w_i^2 \;\Rightarrow\; w \leftarrow \left(1 - \frac{\eta\lambda}{m}\right) w - \eta\,\frac{\partial L}{\partial w}$$

$$\text{L1 / sparsity: } J = L + \frac{\lambda}{m}\,\Sigma_i\,|w_i| \;\Rightarrow\; \text{subgradient } \frac{\lambda}{m}\,\text{sign}(w)$$

$$\text{Inverted dropout: } \tilde{a}_i = \frac{\text{mask}_i}{1-p}\cdot a_i\,,\quad \mathbb{E}[\tilde{a}_i] = a_i \text{ — off at test time}$$

$$\text{Label smoothing: } y_k^{LS} = (1-\varepsilon)\,y_k + \frac{\varepsilon}{K} \text{ — softens one-hot targets over K classes}$$

$$\text{Bias–variance: } \mathbb{E}[(y-\hat{y})^2] = \text{bias}^2 + \text{variance} + \sigma^2_{noise}$$

Early stopping: track the validation error, keep the best-scoring weights, and stop after a patience of no improvement. For a quadratic loss it is equivalent to an L2 penalty.

Convolution: sizing, parameters & receptive field — Ch.13–14

Input side n, kernel k, padding p, stride s, dilation d, C input channels, F filters. Single-channel response $z[i,j] = \Sigma_{m,n}\, I[i+m, j+n]\cdot K[m,n] + b$.

$$o = \left\lfloor \frac{n-k+2p}{s} \right\rfloor + 1 \quad \text{(with dilation)} \quad o = \left\lfloor \frac{n+2p-d(k-1)-1}{s} \right\rfloor + 1$$

$$\text{"same": } p = \frac{k-1}{2}\,,\quad s = 1 \Rightarrow o = n$$

$$\text{params} = k{\cdot}k{\cdot}C{\cdot}F + F \quad \text{MACs} \approx o_H{\cdot}o_W{\cdot}(k{\cdot}k{\cdot}C){\cdot}F \quad (\text{FLOPs} \approx 2\times)$$

$$\text{receptive field:}\quad r_\ell = r_{\ell-1} + (k_\ell - 1)\cdot \Pi_{i<\ell}\, s_i \quad \text{— } 3{\times}3 \text{ stride-1 stack} \Rightarrow r = 2\ell + 1$$

$$\text{pooling:}\quad y_{max} = \max_{(m,n)\in W} a[i+m, j+n]\,, \quad y_{avg} = \frac{1}{|W|} \Sigma_{(m,n)\in W}\, a[i+m, j+n]$$

Attention & the transformer block — Ch.16

Rows of Q, K, V are per-token query, key and value vectors. d_k is the key dimension.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q\,K^T}{\sqrt{d_k}}\right) V$$

$$s_{ij} = \frac{q_i \cdot k_j}{\sqrt{d_k}}\,, \quad \alpha_{ij} = \frac{e^{s_{ij}}}{\Sigma_l\, e^{s_{il}}}\,, \quad o_i = \Sigma_j\, \alpha_{ij}\, v_j$$

$$Q = X\,W_Q\,, K = X\,W_K\,, V = X\,W_V\,; \quad \text{Var}(q{\cdot}k) = d_k \Rightarrow \text{divide by} \sqrt{d_k}$$

$$\text{head}_r = \text{Attention}(X\,W_Q^r, X\,W_K^r, X\,W_V^r)\,, \quad \text{MultiHead} = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\,W_O$$

$$\text{causal mask:}\quad s_{ij} = -\infty \text{ for } j > i \quad \text{— a token attends only to itself and its past}$$

$$\text{block:}\quad z = \text{LayerNorm}(x + \text{MultiHead}(x))\,, \quad \text{out} = \text{LayerNorm}(z + \text{FFN}(z))$$

$$\text{FFN}(z) = \max(0, z\,W_1 + b_1)\,W_2 + b_2\,; \quad PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad PE(pos, 2i{+}1) = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

Metrics, probability & information theory — Ch.14, 17, 24–25 & App. C–D

From the confusion matrix (TP, FP, FN, TN):

METRIC	FORMULA	MEANING
Accuracy	$(TP + TN)/(TP + TN + FP + FN)$	overall correct fraction (can mislead under imbalance)
Precision	$TP/(TP + FP)$	of flagged positives, fraction truly positive
Recall / TPR	$TP/(TP + FN)$	of true positives, fraction caught
Specificity / TNR	$TN/(TN + FP)$	of true negatives, fraction caught (FPR = 1 − TNR)
F1	$2\cdot P\cdot R/(P + R) = 2TP/(2TP + FP + FN)$	harmonic mean of precision & recall
Balanced acc	½ (recall + specificity)	accuracy corrected for imbalance
ROC-AUC	area under TPR-vs-FPR curve = $P(f(x^+) > f(x^-))$	rank quality, threshold-free (0.5 = chance)
IoU	$\lvert A \cap B\rvert/\lvert A \cup B\rvert$	box/mask overlap (detection, segmentation)
ECE	$\Sigma_m\ (\lvert B_m\rvert/n)\cdot \lvert acc(B_m) - conf(B_m)\rvert$	calibration gap over M confidence bins
Perplexity	$\exp(\mathscr{L}) = \left(\Pi_t\ P_\theta(x_t \mid x_{<t})\right)^{-1/T}$	exp of mean cross-entropy; effective #choices

Probability & information-theory quick facts:

$$\text{Bayes:}\quad p(\theta \mid D) = \frac{p(D \mid \theta)\, p(\theta)}{p(D)} \quad \text{— posterior} \propto \text{likelihood} \times \text{prior}$$

$$\text{Gaussian:}\quad p(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\ \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

$$\text{Entropy:}\quad H(p) = -\Sigma_k\, p_k \ln p_k \quad (\max \ln K, \text{uniform}) \qquad \text{Cross-entropy:}\quad H(y, p) = -\Sigma_k\, y_k \ln p_k$$

$$\text{KL:}\quad D_{KL}(y \,\|\, p) = \Sigma_k\, y_k \ln \frac{y_k}{p_k} = H(y, p) - H(y) \geq 0$$

$$\text{Softmax} \leftrightarrow \text{logits:}\quad p_k = \frac{e^{z_k}}{\Sigma_j\, e^{z_j}}\ ,\quad \ln p_k = z_k - \ln \Sigma_j\, e^{z_j} \quad \text{— shift-invariant: subtract } \max_j z_j$$

Exercises

1. State the derivative of Swish/SiLU, $f(z) = z \cdot \sigma(z)$, and evaluate it at $z = 0$. MATH

2. A 224×224 map passes through a 3×3 convolution with padding 1, stride 2, dilation 1. Give the output side, and the padding that would instead make it a "same" convolution. MATH

3. Write Adam's bias-corrected moments and parameter update, and state the two default decay rates. MATH

4. From TP = 8, FP = 2, FN = 4, TN = 86, state precision, recall and F1. MATH

References

The sources for each chapter are collected here, grouped by chapter in the order they appear in the book.

1. From biological to artificial neurons

McCulloch & Pitts (1943), "A logical calculus of the ideas immanent in nervous activity".

Hebb (1949), *The Organization of Behavior*.

Rosenblatt (1958), "The perceptron: a probabilistic model for information storage and organization in the brain".

Minsky & Papert (1969), *Perceptrons*.

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 6.

Nielsen, *Neural Networks and Deep Learning*, ch. 1.

2. The perceptron & linear separability

Rosenblatt (1958), *The Perceptron*.

Novikoff (1962), convergence proof.

Minsky & Papert (1969), *Perceptrons*.

Gallant (1990), pocket algorithm.

Collins (2002), structured perceptron.

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 6.

Bishop, *Pattern Recognition and Machine Learning*, ch. 4.

3. Multilayer networks & the forward pass

Cybenko (1989).

Hornik, Stinchcombe & White (1989).

Hornik (1991).

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 6.

Nielsen, *Neural Networks and Deep Learning*, ch. 2.

4. Activation functions

Nair & Hinton (2010), ReLU.

Clevert et al. (2016), ELU.

Hendrycks & Gimpel (2016), GELU.

Ramachandran et al. (2017), Swish.

Hinton, Vinyals & Dean (2015), distillation & temperature.

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 6.

5. Features, embeddings & what a network learns

Bengio, Courville & Vincent (2013), "Representation Learning: A Review and New Perspectives", *IEEE TPAMI*.

Mikolov et al. (2013), "Efficient Estimation of Word Representations in Vector Space" (word2vec).

Mikolov, Yih & Zweig (2013), linguistic regularities & analogies.

Zeiler & Fergus (2014), "Visualizing and Understanding Convolutional Networks".

Bolukbasi et al. (2016), "Man is to Computer Programmer as Woman is to Homemaker?".

Hinton (1986), distributed representations.

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 15.

Prince, *Understanding Deep Learning* (2023), ch. 18.

6. Loss functions & maximum likelihood

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 5 (maximum likelihood) and §6.2 (output units & cost).

Bishop, *Pattern Recognition and Machine Learning*, ch. 4–5.

Nielsen, *Neural Networks and Deep Learning*, ch. 3 (the cross-entropy cost).

Huber (1964), robust estimation.

Szegedy et al. (2016), label smoothing.

Lin et al. (2017), focal loss.

Gneiting & Raftery (2007), strictly proper scoring rules.

Guo et al. (2017), on calibration of modern neural networks.

7. Gradient descent & its variants

Cauchy (1847), original method.

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 4 & 8.

Bottou, Curtis & Nocedal (2018), "Optimization Methods for Large-Scale Machine Learning".

Ruder (2016), "An overview of gradient descent optimization algorithms".

Goyal et al. (2017), "Accurate, Large Minibatch SGD".

Loshchilov & Hutter (2017), "SGDR: Stochastic Gradient Descent with Warm Restarts" (cosine annealing).

Smith (2017), "Cyclical Learning Rates for Training Neural Networks".

Dauphin et al. (2014), "Identifying and attacking the saddle point problem".

8. Backpropagation

Rumelhart, Hinton & Williams (1986), "Learning representations by back-propagating errors", *Nature*.

Werbos (1974).

Linnainmaa (1970), reverse-mode automatic differentiation.

Goodfellow, Bengio & Courville, *Deep Learning*, §6.5.

Nielsen, *Neural Networks and Deep Learning*, ch. 2.

9. Momentum, RMSProp & Adam

Polyak (1964), heavy-ball momentum.

Nesterov (1983).

Duchi, Hazan & Singer (2011), *Adaptive Subgradient Methods* (AdaGrad).

Tieleman & Hinton (2012), RMSProp.

Kingma & Ba (2015), *Adam*.

Dozat (2016), *Incorporating Nesterov Momentum into Adam* (Nadam).

Reddi, Kale & Kumar (2018), *On the Convergence of Adam and Beyond* (AMSGrad).

Loshchilov & Hutter (2019), *Decoupled Weight Decay Regularization* (AdamW).

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 8.

10. Initialization & the vanishing gradient

Glorot & Bengio (2010), *Xavier init*.

He, Zhang, Ren & Sun (2015), *He init / PReLU*.

LeCun et al. (1998), *Efficient BackProp*.

Ioffe & Szegedy (2015), *Batch Normalization*.

He et al. (2016), *ResNets*.

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 8.

11. Normalization: batch, layer & beyond

Ioffe & Szegedy (2015), *Batch Normalization*.

Santurkar, Tsipras, Ilyas & Madry (2018), *How Does Batch Normalization Help Optimization?*.

Ba, Kiros & Hinton (2016), *Layer Normalization*.

Wu & He (2018), *Group Normalization*.

Ulyanov, Vedaldi & Lempitsky (2016), *Instance Normalization*.

Zhang & Sennrich (2019), *Root Mean Square Layer Normalization*.

Salimans & Kingma (2016), *Weight Normalization*.

Goodfellow, Bengio & Courville, *Deep Learning*, §8.7.

12. Regularization & generalization

Tikhonov (1943), ridge/L2.

Tibshirani (1996), *Lasso*/L1.

Srivastava, Hinton et al. (2014), *Dropout*.

Loshchilov & Hutter (2019), *AdamW*.

Bishop, *Pattern Recognition and Machine Learning*, ch. 3 & 5.

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 7.

MacKay (1992), Bayesian weight priors.

Szegedy et al. (2016), label smoothing.

Belkin, Hsu, Ma & Mandal (2019) and Nakkiran et al. (2021), *double descent*.

13. Convolutional neural networks

LeCun et al. (1998), *Gradient-based learning applied to document recognition*.

Krizhevsky, Sutskever & Hinton (2012), AlexNet.

Simonyan & Zisserman (2014), VGG.

He et al. (2015), *Deep Residual Learning*.

Zeiler & Fergus (2014).

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 9.

14. Modern CNN architectures & computer vision

LeCun et al. (1998), *Gradient-based learning applied to document recognition*.

Krizhevsky, Sutskever & Hinton (2012), AlexNet.

Simonyan & Zisserman (2014), VGG.

Szegedy et al. (2015), *Going deeper with convolutions* (GoogLeNet/Inception).

He, Zhang, Ren & Sun (2015/2016), *Deep Residual Learning* and *Identity Mappings in Deep Residual Networks*.

Huang et al. (2017), DenseNet.

Tan & Le (2019), EfficientNet.

Ren et al. (2015), Faster R-CNN.

Redmon et al. (2016), YOLO.

Liu et al. (2016), SSD.

Ronneberger, Fischer & Brox (2015), U-Net.

Dosovitskiy et al. (2021), *An Image is Worth 16×16 Words* (ViT).

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 9.

15. Recurrent networks & sequences

Elman (1990), *Finding structure in time*.

Werbos (1990), backpropagation through time.

Hochreiter & Schmidhuber (1997), LSTM & the constant error carousel.

Cho et al. (2014), GRU.

Sutskever, Vinyals & Le (2014), sequence-to-sequence learning.

Pascanu, Mikolov & Bengio (2013), gradient clipping.

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 10.

16. Attention & the transformer

Bahdanau, Cho & Bengio (2015), *Neural Machine Translation by Jointly Learning to Align and Translate*.

Vaswani et al. (2017), *Attention Is All You Need*.

Devlin et al. (2019), BERT.

Radford et al. (2018–2019), GPT.

Ba, Kiros & Hinton (2016), *Layer Normalization*.

Dosovitskiy et al. (2021), *An Image is Worth 16×16 Words* (ViT).

Alammar, *The Illustrated Transformer*.

17. Language models & the transformer era

Vaswani et al. (2017), *Attention Is All You Need*.

Radford et al. (2018–2019), GPT and GPT-2.

Brown et al. (2020), *Language Models are Few-Shot Learners* (GPT-3).

Sennrich, Haddow & Birch (2016), BPE.

Kaplan et al. (2020), *Scaling Laws for Neural Language Models*.

Hoffmann et al. (2022), *Training Compute-Optimal Large Language Models* (Chinchilla).

Su et al. (2021), RoPE.

Press, Smith & Lewis (2022), ALiBi.

Ouyang et al. (2022), InstructGPT/RLHF.

Rafailov et al. (2023), DPO.

Wei et al. (2022) and Schaeffer et al. (2023) on emergent abilities.

18. Graph neural networks

Kipf & Welling (2017), *Semi-Supervised Classification with Graph Convolutional Networks*.

Hamilton, Ying & Leskovec (2017), *Inductive Representation Learning on Large Graphs* (GraphSAGE).

Veličković et al. (2018), *Graph Attention Networks*.

Gilmer et al. (2017), *Neural Message Passing for Quantum Chemistry*.

Xu et al. (2019), *How Powerful Are Graph Neural Networks?* (GIN).

Bronstein, Bruna, Cohen & Veličković (2021), *Geometric Deep Learning*.

19. Autoencoders & representation learning

Hinton & Salakhutdinov, “Reducing the Dimensionality of Data with Neural Networks,” *Science* (2006).

Vincent et al., “Extracting and Composing Robust Features with Denoising Autoencoders” (2008).

Rifai et al., “Contractive Auto-Encoders” (2011).

Kingma & Welling, “Auto-Encoding Variational Bayes” (2014).

Higgins et al., “β-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework” (2017).

Bricken et al., “Towards Monosemanticity: Decomposing Language Models with Sparse Autoencoders” (2023).

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 14.

Bishop, *Pattern Recognition and Machine Learning*, ch. 12.

20. Self-supervised & contrastive learning

Oord, Li & Vinyals (2018), "Representation Learning with Contrastive Predictive Coding" (InfoNCE).

Chen et al. (2020), "A Simple Framework for Contrastive Learning of Visual Representations" (SimCLR).

He et al. (2020), "Momentum Contrast for Unsupervised Visual Representation Learning" (MoCo).

Grill et al. (2020), BYOL.

Chen & He (2021), SimSiam.

Zbontar et al. (2021), Barlow Twins.

Bardes, Ponce & LeCun (2022), VICReg.

Caron et al. (2021), DINO.

Devlin et al. (2019), BERT.

He et al. (2022), "Masked Autoencoders Are Scalable Vision Learners".

Radford et al. (2021), CLIP.

21. Generative models: GANs to diffusion

Goodfellow et al., “Generative Adversarial Nets” (2014).

Radford, Metz & Chintala, “Unsupervised Representation Learning with Deep Convolutional GANs” (DCGAN, 2016).

Gulrajani et al., “Improved Training of Wasserstein GANs” (WGAN-GP, 2017).

van den Oord et al., “Pixel Recurrent Neural Networks” (PixelCNN, 2016).

Rezende & Mohamed, “Variational Inference with Normalizing Flows” (2015).

Ho, Jain & Abbeel, “Denoising Diffusion Probabilistic Models” (2020).

Song, Meng & Ermon, “Denoising Diffusion Implicit Models” (DDIM, 2021).

Song & Ermon, score-based generative models (2019).

Ho & Salimans, “Classifier-Free Diffusion Guidance” (2022).

Heusel et al., “GANs Trained by a Two Time-Scale Update Rule” (FID, 2017).

Radford et al., CLIP (2021).

Rombach et al., “High-Resolution Image Synthesis with Latent Diffusion Models” (2022).

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 20.

22. Transfer learning & fine-tuning

Yosinski, Clune, Bengio & Lipson (2014), "How transferable are features in deep neural networks?", *NeurIPS*.

Donahue et al. (2014), DeCAF.

Razavian et al. (2014), "CNN Features off-the-shelf".

Howard & Ruder (2018), ULMFiT.

Houlsby et al. (2019), "Parameter-Efficient Transfer Learning" (adapters).

Hu et al. (2021), "LoRA: Low-Rank Adaptation of Large Language Models".

Li & Liang (2021), prefix tuning.

Lester, Al-Rfou & Constant (2021), prompt tuning.

Ben-Zaken, Goldberg & Ravfogel (2021), BitFit.

Hinton, Vinyals & Dean (2015), "Distilling the Knowledge in a Neural Network".

Kirkpatrick et al. (2017), elastic weight consolidation.

Bommasani et al. (2021), "On the Opportunities and Risks of Foundation Models".

23. Deep reinforcement learning

Sutton & Barto, *Reinforcement Learning: An Introduction* (2nd ed., 2018).

Bellman (1957), *Dynamic Programming*.

Watkins & Dayan (1992), *Q-learning*.

Williams (1992), REINFORCE.

Mnih et al. (2015), *Human-level control through deep reinforcement learning* (DQN).

van Hasselt, Guez & Silver (2016), Double DQN.

Wang et al. (2016), Dueling networks.

Mnih et al. (2016), A3C/A2C.

Schulman et al. (2017), *Proximal Policy Optimization*.

Silver et al. (2017), *AlphaZero*.

Christiano et al. (2017) and Ouyang et al. (2022), RLHF/InstructGPT.

24. Uncertainty, calibration & Bayesian nets

Guo, Pleiss, Sun & Weinberger (2017), "On Calibration of Modern Neural Networks".

Gal & Ghahramani (2016), "Dropout as a Bayesian Approximation".

Lakshminarayanan, Pritzel & Blundell (2017), "Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles".

Blundell et al. (2015), "Weight Uncertainty in Neural Networks" (Bayes by Backprop).

MacKay (1992), the Laplace approximation for neural networks.

Kendall & Gal (2017), aleatoric vs. epistemic uncertainty.

Angelopoulos & Bates (2023), "A Gentle Introduction to Conformal Prediction".

Murphy, *Probabilistic Machine Learning*, chs. on Bayesian deep learning.

25. Training in practice: the complete recipe

Ng, *Machine Learning Yearning* (2018).

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 11 (“Practical Methodology”).

Karpathy, “A Recipe for Training Neural Networks” (2019).

Géron, *Hands-On Machine Learning*, 2nd ed., ch. 2–3.

26. Data-centric deep learning

Ng, "A Chat with Andrew on MLOps: From Model-centric to Data-centric AI" (2021).

Lin, Goyal, Girshick, He & Dollár (2017), "Focal Loss for Dense Object Detection".

Zhang, Cissé, Dauphin & Lopez-Paz (2018), "mixup".

Yun et al. (2019), "CutMix".

Cubuk et al. (2020), "RandAugment".

Cohen (1960), κ statistic.

Gebru et al. (2021), "Datasheets for Datasets".

Mitchell et al. (2019), "Model Cards".

Northcutt, Athalye & Mueller (2021), "Pervasive Label Errors".

Settles (2009), "Active Learning Literature Survey".

Bengio et al. (2009), "Curriculum Learning".

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 7 & 11.

27. Efficiency & deployment

Jacob et al. (2018), "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference".

Han, Mao & Dally (2016), "Deep Compression".

Frankle & Carbin (2019), "The Lottery Ticket Hypothesis".

Hinton, Vinyals & Dean (2015), "Distilling the Knowledge in a Neural Network".

Howard et al. (2017), "MobileNets".

Micikevicius et al. (2018), "Mixed Precision Training".

Williams, Waterman & Patterson (2009), "Roofline".

Prince, *Understanding Deep Learning* (2023), ch. 20.

28. Interpretability & explainability

Simonyan, Vedaldi & Zisserman (2013), saliency maps.

Zeiler & Fergus (2014), occlusion & feature visualization.

Ribeiro, Singh & Guestrin (2016), LIME.

Lundberg & Lee (2017), SHAP.

Shapley (1953), the value of an n-person game.

Sundararajan, Taly & Yan (2017), Integrated Gradients.

Selvaraju et al. (2017), Grad-CAM.

Smilkov et al. (2017), SmoothGrad.

Kim et al. (2018), TCAV.

Adebayo et al. (2018), sanity checks for saliency.

Jain & Wallace (2019) and Wiegreffe & Pinter (2019), attention (not) explanation.

Hooker et al. (2019), ROAR.

Olah et al. (2020), circuits.

Elhage et al. (2022), toy models of superposition.

Molnar, *Interpretable Machine Learning* (2022).

29. Ethics, fairness & safety

Dwork, Hardt, Pitassi, Reingold & Zemel (2012), fairness through awareness.

Hardt, Price & Srebro (2016), equality of opportunity.

Kleinberg, Mullainathan & Raghavan (2016), inherent trade-offs.

Chouldechova (2017), fair prediction with disparate impact.

Buolamwini & Gebru (2018), Gender Shades.

Zhang, Lemoine & Mitchell (2018), adversarial debiasing.

Dwork & Roth (2014), *The Algorithmic Foundations of Differential Privacy*.

Abadi et al. (2016), deep learning with differential privacy (DP-SGD).

Shokri et al. (2017), membership inference.

McMahan et al. (2017), federated learning.

Szegedy et al. (2013) and Goodfellow, Shlens & Szegedy (2015), adversarial examples & FGSM.

Madry et al. (2018), robust optimization.

Mitchell et al. (2019), model cards.

Gebru et al. (2021), datasheets.

Strubell, Ganesh & McCallum (2019), energy & policy.

Christiano et al. (2017) and Ouyang et al. (2022), RLHF.

Barocas, Hardt & Narayanan, *Fairness and Machine Learning* (2019).

30. Case study: handwritten-digit recognition (MNIST)

LeCun, Bottou, Bengio & Haffner, "Gradient-based learning applied to document recognition" (1998).

LeCun, Cortes & Burges, the MNIST database.

Simard, Steinkraus & Platt, "Best practices for convolutional neural networks applied to visual document analysis" (2003, elastic distortions).

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 6 & 9.

PyTorch `torchvision.datasets.MNIST` documentation. Cross-references: multilayer networks (chapter 3), activation functions (chapter 4), loss & cross-entropy (chapter 6), backpropagation (chapter 8), optimizers (chapter 9), normalization (chapter 11), regularization & generalization (chapter 12), convolutional networks (chapter 13), modern CNNs & computer vision (chapter 14), data-centric deep learning (chapter 26), the training recipe (chapter 25).

31. Case study: tabular classification & churn

the IBM Telco Customer Churn dataset (a common teaching benchmark).

Hastie, Tibshirani & Friedman, *The Elements of Statistical Learning*, ch. 4 (logistic regression).

Guo & Berkhahn, "Entity Embeddings of Categorical Variables" (2016).

Lin et al., "Focal Loss for Dense Object Detection" (2017).

Chen & Guestrin, "XGBoost" (2016), on tree methods for tabular data.

Grinsztajn et al., "Why do tree-based models still outperform deep learning on tabular data?" (2022). Cross-references: the neuron as logistic regression (Chapter 1), feature representations & entity embeddings (Chapter 5), cross-entropy from maximum likelihood (Chapter 6), gradient descent (Chapter 7), normalization (Chapter 11), regularization & early stopping (Chapter 12), uncertainty & calibration (Chapter 24), data-centric practice & leakage (Chapter 26), and interpretability & feature importance (Chapter 28).

32. Case study: text sentiment analysis

Maas et al., "Learning Word Vectors for Sentiment Analysis" (2011), the IMDB dataset.

Mikolov et al., "word2vec" (2013), on learned embeddings.

Devlin et al., "BERT" (2019), on pretrained transformer fine-tuning.

PyTorch `nn.Embedding` and `nn.LSTM` documentation. Cross-references: the neuron as logistic regression (chapter 1), features & embeddings (chapter 5), loss functions & cross-entropy (chapter 6), recurrent networks (chapter 15), attention & the transformer (chapter 16), language models (chapter 17), and transfer learning & fine-tuning (chapter 22).

33. Case study: time-series forecasting

Box & Jenkins, *Time Series Analysis: Forecasting and Control* (ARIMA).

Hyndman & Athanasopoulos, *Forecasting: Principles and Practice* (3rd ed.), on train/test in time and walk-forward validation.

Salinas, Flunkert & Gasthaus, "DeepAR" (*Int. J. Forecasting*, 2020).

Makridakis, Spiliotis & Assimakopoulos, the M4/M5 forecasting competitions.

Lim & Zohren, "Time-series forecasting with deep learning: a survey" (2021). See Chapter 15 for RNNs, LSTMs and GRUs, Chapter 16 for attention and transformers, Chapter 24 for calibrated uncertainty, and Chapter 26 for the data-centric view of leakage and splitting.

34. Case study: anomaly detection with autoencoders

Sakurada & Yairi, "Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction" (2014).

An & Cho, "Variational Autoencoder based Anomaly Detection using Reconstruction Probability" (2015).

Chandola, Banerjee & Kumar, "Anomaly Detection: A Survey" (*ACM Computing Surveys*, 2009).

Malhotra et al., LSTM-based encoder–decoder for multi-sensor anomaly detection (2016).

Liu, Ting & Zhou, "Isolation Forest" (2008).

Schölkopf et al., "Estimating the Support of a High-Dimensional Distribution" (one-class SVM, 2001).

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 14 (autoencoders). Cross-references: features & representations (Chapter 5), autoencoders and the variational extension (Chapter 19), generative models (Chapter 21), uncertainty & calibration (Chapter 24), data-centric deep learning (Chapter 26), and the metrics panel of the tabular case study (Chapter 31).

35. Appendix A · Linear algebra refresher

Further reading: Strang, *Introduction to Linear Algebra* and *Linear Algebra and Learning from Data* (the SVD-first modern take).

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 2 (linear algebra for ML).

Trefethen & Bau, *Numerical Linear Algebra* (the SVD and conditioning).

Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models" (2021).

the NumPy documentation on broadcasting, `numpy.linalg.eig` and `numpy.linalg.svd`.

3Blue1Brown's *Essence of Linear Algebra* for geometric intuition. Related chapters: 1 (the neuron as a dot product), 3 (the layer as a matmul), 5 (embeddings & cosine similarity), 8 (transpose and Hadamard in backpropagation), 10 (eigenvalues, spectral radius & initialization), 12 (norms in weight decay & regularization), 19 (SVD, PCA & autoencoders) and 22 (low-rank adaptation / LoRA).

and Appendix B (calculus & the Hessian).

36. Appendix B · Calculus & the chain rule

Further reading: Spivak, *Calculus* (rigorous single-variable).

any multivariable text for the gradient, Jacobian and Hessian.

Nocedal & Wright, *Numerical Optimization* (2006) and Boyd & Vandenberghe, *Convex Optimization* (2004) for Taylor models, Newton's method and convexity.

Goodfellow, Bengio & Courville, *Deep Learning*, §4.3 and §6.5 (gradient-based optimization and back-propagation).

Baydin et al. (2018), "Automatic Differentiation in Machine Learning: a Survey". Related chapters: 7 (gradient descent steps along $-\nabla L$), 8 (backpropagation is the chain rule on a graph), 9 (optimizers approximate curvature).

Appendix A (the linear algebra behind the gradient, Jacobian and Hessian) and Appendix G (autograd builds the chain rule into a graph).

37. Appendix C · Probability & statistics

Further reading: Wasserman, *All of Statistics* (a fast, complete reference).

Bishop, *Pattern Recognition and Machine Learning*, ch. 1–2 (probability and distributions for ML).

Murphy, *Probabilistic Machine Learning: An Introduction* (2022), ch. 2–4 (the Bayesian view of MLE and MAP).

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 3 (probability and information theory) and §5.5 (maximum likelihood).

Cover & Thomas, *Elements of Information Theory* (entropy and KL). Related chapters: 6 (maximum likelihood → MSE and cross-entropy), 12 (priors as regularization / weight decay), 19 (the KL term in variational autoencoders), 21 (divergences in generative models), 24 (calibration and base rates).

and Appendix D (entropy, cross-entropy and KL in depth).

38. Appendix D · Information theory

Further reading: Shannon, *A Mathematical Theory of Communication* (1948) — the founding paper.

Cover & Thomas, *Elements of Information Theory* (2006), ch. 2 (entropy, relative entropy, mutual information) and ch. 8 (differential entropy).

MacKay, *Information Theory, Inference, and Learning Algorithms* (2003).

Goodfellow, Bengio & Courville, *Deep Learning* (2016), §3.13 (information theory) and ch. 5 (maximum likelihood).

Murphy, *Probabilistic Machine Learning* (2022), ch. 6. Related chapters: 6 (cross-entropy as a loss, MSE from Gaussian noise), 17 (perplexity and language models), 19 (the KL term in variational autoencoders), 20 (mutual information in self-supervised learning), 21 (divergences in generative models), 24 (entropy as predictive uncertainty). See also Appendix C (probability & statistics).

39. Appendix E · Python & NumPy primer

Further reading: the official *NumPy: the absolute basics for beginners* and *NumPy Quickstart* guides (numpy.org).

the *Python Tutorial* (docs.python.org) for the language itself.

Harris et al., "Array programming with NumPy," *Nature* 585 (2020).

Wes McKinney, *Python for Data Analysis* (3rd ed.), ch. 4 on NumPy.

VanderPlas, *Python Data Science Handbook*, ch. 2. Appendix F continues this on-ramp into PyTorch and Appendix G into automatic differentiation.

Appendix A gives the linear-algebra background for the array operations used here, and the batched forward pass reappears in the multilayer networks of Chapter 3.

40. Appendix F · PyTorch primer

Further reading: the official *PyTorch tutorials* and *Deep Learning with PyTorch: a 60 Minute Blitz* (pytorch.org).

Stevens, Antiga & Viehmann, *Deep Learning with PyTorch* (Manning, 2020).

Zhang et al., *Dive into Deep Learning* (d2l.ai) for runnable chapters in PyTorch.

Goodfellow, Bengio & Courville, *Deep Learning*, ch. 6–8 for the theory these APIs implement. Appendix E covers the Python and NumPy this builds on, Appendix G rebuilds PyTorch's autograd from scratch so you can see what `.backward()` really does, and Appendix H lists datasets to practise on.

41. Appendix G · Autograd from scratch

Karpathy, *micrograd* (github.com/karpathy/micrograd, 2020), the scalar autograd engine this appendix follows, and his “The spelled-out intro to neural networks and backpropagation: building micrograd” lecture.

Baydin, Pearlmutter, Radul & Siskind (2018), “Automatic Differentiation in Machine Learning: a Survey”, *JMLR*.

Griewank & Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (2nd ed., SIAM, 2008).

Linnainmaa (1970) on reverse-mode accumulation.

Paszke et al. (2019), “PyTorch: An Imperative Style, High-Performance Deep Learning Library”, *NeurIPS*. See Chapter 8 for backpropagation as reverse-mode autodiff and Appendix F for `loss.backward()` in PyTorch.

42. Appendix H · Datasets & further reading

LeCun & Cortes, *The MNIST database*.

Xiao, Rasul & Vollgraf (2017), *Fashion-MNIST*.

Krizhevsky (2009), *Learning Multiple Layers of Features from Tiny Images* (CIFAR-10/100).

Netzer et al. (2011), *SVHN*.

Deng et al. (2009), *ImageNet*.

Lin et al. (2014), *Microsoft COCO*.

Maas et al. (2011), IMDB.

Socher et al. (2013), SST.

Wang et al. (2018), *GLUE*.

Rajpurkar et al. (2016), *SQuAD*.

Merity et al. (2016), *WikiText*.

Raffel et al. (2020), *C4 / T5*.

Dua & Graff, *UCI Machine Learning Repository*.

Panayotov et al. (2015), *LibriSpeech*.

Salamon et al. (2014), *UrbanSound8K*.

Makridakis et al. (2020), *M4*.

Yang et al. (2016), *Planetoid* (Cora/Citeseer).

Hu et al. (2020), *Open Graph Benchmark*.

plus the `torchvision.datasets`, Hugging Face `datasets`, and scikit-learn documentation, and the Stanford CS231n and CS224n course notes. See Appendices E and F for the NumPy and PyTorch used to load and batch these sets.

43. Appendix I · Formulas & symbols quick reference

Goodfellow, Bengio & Courville, *Deep Learning* (2016).

Bishop, *Pattern Recognition and Machine Learning* (2006).

Prince, *Understanding Deep Learning* (2023).

Zhang, Lipton, Li & Smola, *Dive into Deep Learning*. Primary formulas: Rumelhart et al. (1986).

Glorot & Bengio (2010).

He et al. (2015).

Ioffe & Szegedy (2015).

Ba et al. (2016).

Kingma & Ba (2015).

Loshchilov & Hutter (2019).

Vaswani et al. (2017).

Lin et al. (2017).

Guo et al. (2017).

Glossary

Activation function.

The nonlinear function g applied to a neuron's weighted sum z. Without it, stacked layers collapse into a single linear map — it is what lets a network bend and fit curves. Its derivative also controls how the gradient flows backward (chapter 4).

Actor-critic.

A reinforcement-learning method that pairs an actor (the policy) with a critic (a value estimate). The critic gives a learning signal with lower variance than raw returns. Most modern deep-RL algorithms, including PPO, follow this pattern (chapter 23).

Adam.

The default optimizer here. It combines momentum (a running mean of gradients), RMSProp (a running mean of squared gradients), and bias correction. It sets a separate step size for each parameter, so weights with tiny gradients still move (chapter 9).

Adjacency matrix.

The square matrix A whose entry A[i,j] records whether nodes i and j are connected (and sometimes the edge weight). It describes the graph's structure and is used in the neighbor-aggregation step of a GNN (chapter 18).

Adversarial example.

An input changed by a tiny amount, often too small to see, and designed on purpose to make a model classify it wrongly. These examples show how fragile otherwise high-accuracy networks can be (chapter 29).

Aleatoric uncertainty.

Uncertainty that cannot be removed, coming from noise that is built into the data itself, so the same input can give different outcomes. Collecting more data cannot remove it. A model can only try to measure it (chapter 24).

Attention.

A soft lookup where each position forms a query, every position offers a key and a value, and the output is an average of the values weighted by similarity. It is the core of transformers (chapter 16).

Autoencoder.

A network trained to rebuild its own input through a narrow bottleneck, which forces it to find a compact code. Used for compression, denoising, and anomaly detection (chapter 19).

Autograd / automatic differentiation.

The technique frameworks use to compute exact gradients. It records every basic operation on a computation graph, then applies the chain rule backward through it. It is backpropagation extended to any code (appendix G).

Autoregressive model.

A model that generates a sequence one element at a time, where each new element depends on all the ones already produced. It splits the joint probability into a product of next-step conditional probabilities (chapter 17).

Backpropagation.

The chain rule applied backward through the network to get every weight's gradient in a single pass, which costs about twice as much as a forward pass. It is the algorithm that makes training practical (chapter 8).

Backpropagation through time (BPTT).

Applying ordinary backpropagation to a recurrent network by unrolling it across the whole sequence and passing gradients back through every step. Long sequences make it expensive and likely to cause vanishing or exploding gradients (chapter 15).

Batch / mini-batch.

The group of samples used for one gradient step. Full-batch is exact but slow. A small mini-batch is noisier but faster, and the noise even helps generalization (chapter 7).

Batch normalization.

A layer that rescales activations to zero mean and unit variance (with a learnable scale γ and shift β). This stabilizes training and speeds it up (chapter 10).

Bayesian neural network.

A network that puts a probability distribution over each weight instead of one fixed value, so its predictions come with well-grounded uncertainty. Exact inference is too hard to compute, so it is approximated with variational or sampling methods (chapter 24).

Bellman equation.

The recursive rule stating that a state's value equals the immediate reward plus the discounted value of the next state. It is the fixed-point equation that nearly every value-based RL method solves (chapter 23).

Bias (parameter).

The learned offset b added inside a neuron: $z = w \cdot x + b$. It shifts how much evidence is needed before the neuron responds — a weight whose input is always 1 (chapter 1).

Bias–variance tradeoff.

Too little capacity underfits (high bias); too much capacity fits noise (high variance). Good generalization is the balance between them, tuned with capacity and regularization (chapter 12).

Byte-pair encoding (BPE).

A tokenizer that starts from single characters and repeatedly merges the most frequent neighboring pair, building up a subword vocabulary. It keeps common words whole, while still being able to spell out rare or unseen strings (chapter 17).

Calibration.

How well predicted probabilities match observed frequencies. Among events a calibrated model calls 70% likely, about 70% should actually happen. Deep networks are often overconfident and need recalibrating (chapter 24).

Causal mask.

A triangular mask that stops each position from attending to future tokens, so a decoder can learn to predict the next token without looking ahead. It is what makes autoregressive transformers possible (chapter 16).

Class imbalance.

When some classes are much rarer than others, so a model can reach high accuracy by simply ignoring the rare class. You can fight it with resampling, class weights, or losses such as focal loss (chapter 26).

Contrastive learning.

Training representations by pulling matching pairs (two views of the same item) together in embedding space while pushing unrelated pairs apart. It needs no human labels, only a rule for what counts as the same thing (chapter 20).

Convolution.

Sliding a small shared kernel over an input to build a feature map. Sharing the weights gives translation equivariance and greatly reduces the number of parameters compared with a dense layer (chapter 13).

Cross-entropy.

The loss for classification: −log of the probability the model assigned to the true class. It is the maximum-likelihood loss for categorical outputs and pairs cleanly with softmax (chapter 6).

Data augmentation.

Growing a dataset by applying label-preserving transformations — flips, crops, color shifts, added noise — so the model sees more variety and overfits less. It is a cheap, powerful regularizer, especially for images (chapter 26).

Data leakage.

When information that would not be available at prediction time gets into training — a feature derived from the target, or test data touching the pipeline — which inflates scores that then collapse in deployment. It is a leading cause of models that look great but fail in production (chapter 26).

Data-centric.

An approach that treats improving the data — its labels, coverage and cleanliness — as the main way to get a better model, while keeping the architecture fixed. It is often a bigger gain than another change to the model (chapter 26).

DDPM.

Denoising Diffusion Probabilistic Model: a generator trained to reverse a fixed, gradual noising process by predicting the noise added at each step. Sampling starts from pure noise and denoises step by step into data (chapter 21).

Decision boundary.

The surface where a classifier switches from one predicted class to another. One neuron gives a straight boundary; hidden layers bend it — visible as the tinted background in the Playground (chapter 3).

Deep ensemble.

Training several networks from different random seeds and averaging their predictions. This improves accuracy and also estimates epistemic uncertainty from how much they disagree. It is simple and hard to beat (chapter 24).

Demographic parity.

A fairness criterion that requires a model's positive-prediction rate to be the same across protected groups, no matter the true label. It is simple to state, but it can conflict with accuracy and with other fairness goals (chapter 29).

Depthwise-separable convolution.

Splitting an ordinary convolution into a per-channel spatial filter followed by a 1×1 convolution that mixes channels. This cuts the compute several times over at similar accuracy. It is the efficiency trick at the heart of MobileNet (chapter 27).

Differential privacy.

A formal guarantee that a model's outputs barely change whether or not any single person is in the training data, which limits what can be learned about anyone. It is usually achieved by adding carefully tuned noise (chapter 29).

Diffusion model.

A generator that learns to reverse a gradual noising process. You train a network to predict the noise added, then turn pure noise into a sample by denoising it step by step (chapter 21).

Discriminator.

The classifier half of a GAN, trained to tell real samples apart from the generator's fakes. Its feedback is the training signal that pushes the generator to improve (chapter 21).

Double descent.

The surprising curve where test error first rises as a model nears the interpolation threshold, then falls again as the model grows even larger — going beyond the classic U-shaped bias-variance picture. It changed how we think about overparameterized networks (chapter 12).

Dropout.

Randomly turning off a fraction of hidden units at each training step (with inverted $1/(1-p)$ scaling). This acts like averaging many thinned networks and reduces overfitting (chapter 12).

Early stopping.

Stopping training when the validation loss stops improving, which catches the model just before it starts to memorize noise (chapter 12).

ELBO.

The evidence lower bound: the objective a VAE can actually compute and maximize. It balances reconstruction accuracy against a KL term that pulls the latent code toward its prior. Maximizing it puts a lower bound on the true data log-likelihood (chapter 19).

Embedding.

A learned dense vector that represents a discrete item — a word, a user, a category — arranged so that similar items land near each other. Embeddings turn plain symbols into geometry that a network can compute with (chapter 5).

Entropy.

The average number of bits needed to encode outcomes drawn from a distribution — a measure of its uncertainty. It is largest for a uniform distribution and zero when the outcome is certain (appendix D).

Epistemic uncertainty.

Uncertainty that can be reduced, coming from the model's own lack of knowledge — too little data or wrong assumptions. It shrinks as more data arrives and is largest far from the training distribution (chapter 24).

Epoch.

One full pass over the training set. Networks train for many epochs, ideally reshuffling the data each time (chapter 7).

Fairness.

The study and practice of making sure models do not systematically put groups at a disadvantage based on protected attributes such as race or sex. It forces you to choose a clear criterion, because the competing definitions cannot all hold at the same time (chapter 29).

Feature map.

The output image produced by sliding one kernel over the input — a map of where that learned pattern was found (chapter 13).

FID (Fréchet Inception Distance).

A metric that compares the statistics of generated and real images in a pretrained feature space. A lower value means the generated distribution sits closer to the real one. It is the standard score for image generation (chapter 21).

Fine-tuning.

Continuing to train a pretrained model on your own task, updating some or all of its weights with a small learning rate. It adapts general features to your specific task while keeping most of what was already learned (chapter 22).

Focal loss.

A modified cross-entropy that gives less weight to easy, already well-classified examples, so training focuses on the hard and rare ones. It was designed for the extreme class imbalance of dense object detection (chapter 26).

Forward pass.

Computing the network's output by passing inputs forward layer by layer: $z = aW + b$, $a = g(z)$. Every prediction, loss, and the Math Log start here (chapter 3).

GAN.

Generative adversarial network: a generator that creates fake samples and a discriminator that tries to spot the fakes, trained as a two-player minimax game (chapter 21).

GELU.

A smooth activation that scales each input by its own probability of mattering, $\Phi(x) \cdot x$. It is the default in transformers (chapter 4).

Generalization.

How well a model performs on unseen data, not just its training set — the real goal. Measured on a held-out test set (chapter 12).

Grad-CAM.

A visualization that weights a CNN's final convolutional feature maps by the gradient of a target class, giving a rough heatmap of the image regions behind that decision. It is class-specific and widely used (chapter 28).

Gradient.

The vector of partial derivatives of the loss with respect to every parameter; it points uphill, so training steps against it. Backprop computes it (chapter 8).

Gradient clipping.

Capping the global gradient norm before a step, preventing rare huge gradients from blowing up training. Available in the Playground (chapter 10).

Gradient descent.

The core training loop: repeatedly move every parameter a small step against its gradient, $\theta \leftarrow \theta - \eta \nabla L$ (chapter 7).

Graph convolution (GCN).

A GNN layer that updates each node as a normalized average of its neighbors' features, followed by a learned linear map and a nonlinearity. It is convolution extended from grids to graphs (chapter 18).

Graph neural network (GNN).

A network that works directly on graphs, learning each node's representation by repeatedly gathering information from its neighbors. It extends convolution to irregular, non-grid data like molecules, meshes, and social networks (chapter 18).

GRU.

Gated Recurrent Unit: a simplified LSTM with just an update gate and a reset gate, and no separate cell state. It often trains as well as an LSTM while using fewer parameters (chapter 15).

He initialization.

Random weights scaled by $\sqrt{(2/n_in)}$, tuned for ReLU networks (which zero out half of their inputs) to keep the signal variance stable through depth (chapter 10).

Hidden layer.

Any layer between the input and the output. Hidden layers build intermediate features and are what let a network represent XOR and other non-linear patterns (chapter 3).

Hyperparameter.

A setting you choose rather than learn — learning rate, batch size, layer count, λ. Tuned by experiment, ideally one at a time (chapter 25).

InfoNCE / NT-Xent.

The contrastive loss that treats one positive pair against many negatives as a softmax classification over similarities, controlled by a scale parameter. It is the objective behind SimCLR and many self-supervised methods (chapter 20).

Initialization.

How the starting weights are chosen. All zeros fail because of symmetry; too large saturates the units, too small starves them of signal — hence He/Xavier/LeCun scaling (chapter 10).

Integrated gradients.

An attribution method that adds up gradients along a straight path from a baseline input to the actual input, giving feature contributions that satisfy clean mathematical rules. It is more reliable than reading a raw gradient at a single point (chapter 28).

Internal covariate shift.

The changing distribution of a layer's inputs as the layers below it keep updating during training — the original reason given for batch normalization. Whether this is the true reason norm layers help is debated, but the stabilizing effect is real (chapter 11).

Kernel (filter).

The small grid of shared weights a convolution slides over the input. Each kernel detects one pattern, such as a vertical edge (chapter 13).

KL divergence.

A measure of how much one probability distribution differs from a reference, $D(p\|q) \geq 0$, equal to zero only when the two match. It is asymmetric, and it lies behind cross-entropy and the VAE objective (appendix D).

Knowledge distillation.

Training a small student network to copy the soft output probabilities of a large teacher, moving its behavior into a cheaper model. The teacher's full distribution carries more information than hard labels alone (chapter 27).

Language model.

A model that assigns probabilities to sequences of tokens, usually by predicting each next token from the ones before it. Scaling this simple objective on huge text collections gives today's large language models (chapter 17).

Latent space.

The compressed code an autoencoder or VAE learns in its bottleneck. Nearby points decode to similar outputs; sampling it can generate new data (chapter 19).

Layer normalization.

Like batch norm, but it normalizes across the features within each sample. It is standard inside transformers, where batch statistics are awkward (chapter 16).

Leaky ReLU.

ReLU with a small negative slope (0.01x for x<0) so units never fully die and always pass some gradient (chapter 4).

Learning rate (η).

The size of each gradient step — the single most important knob. Too small and training crawls; too large and it overshoots or diverges (chapter 7).

Learning-rate schedule.

A plan for changing the learning rate as training goes on — warmup, step decay, or cosine annealing — instead of holding it fixed. A good schedule speeds up convergence and improves the final model (chapter 9).

Linear separability.

Whether two classes can be split by a single straight boundary. If they cannot (for example, XOR), no single neuron can classify them (chapter 2).

Logit.

A raw output score before softmax — any real number. Softmax turns a vector of logits into probabilities (chapter 6).

LoRA.

Low-Rank Adaptation: freeze the pretrained weights and learn a small pair of low-rank matrices whose product is added to each weight matrix. It fine-tunes enormous models while training only a tiny fraction of parameters (chapter 22).

Loss function.

The single number training minimizes, measuring how wrong predictions are. MSE for regression, cross-entropy for classification (chapter 6).

LSTM.

A recurrent cell with a protected cell-state pathway and forget, input, and output gates. It is designed to carry information across long sequences without it vanishing (chapter 15).

Markov decision process (MDP).

The formal setting of reinforcement learning — states, actions, transition probabilities, rewards and a discount factor — in which the next state depends only on the current state and action. Almost every RL algorithm assumes this structure (chapter 23).

Masked autoencoder (MAE).

A self-supervised method that hides a large fraction of the input — patches of an image or tokens of text — and trains the model to rebuild the missing parts. The forced guessing gives strong, transferable features (chapter 20).

Maximum likelihood.

The principle that the best parameters make the observed data most probable. MSE and cross-entropy are both maximum-likelihood losses under different noise models (chapter 6).

MC dropout.

Estimating uncertainty by leaving dropout switched on at test time and running several random forward passes, then measuring the spread of the predictions. It is a cheap approximation to Bayesian inference (chapter 24).

Mechanistic interpretability.

Reverse-engineering the actual internal circuits and features a network computes, aiming to explain its behavior as an understandable algorithm rather than an input heatmap. It is a bottom-up route to understanding models (chapter 28).

Message passing.

The core GNN step: every node gathers messages from its neighbors, combines them with a permutation-invariant function, and updates its own state. Stacking rounds lets information spread farther across the graph (chapter 18).

Mini-batch.

See Batch. A handful of samples (often 32–256) per gradient step, balancing speed and gradient quality (chapter 7).

Mixed precision.

Training with 16-bit floats for most operations while keeping a 32-bit master copy of the weights. This roughly halves memory use and speeds up math on modern accelerators. Loss scaling protects the tiny 16-bit gradients from underflow (chapter 27).

Mode collapse.

A GAN failure in which the generator produces only a few kinds of output, ignoring the variety in the real data, because a single convincing sample is already enough to fool the discriminator. It is a central difficulty of adversarial training (chapter 21).

Model card.

A short standardized document reporting a model's intended use, training data, performance across groups, and known limitations. It supports transparent and responsible deployment (chapter 29).

Momentum.

Building up a running velocity from past gradients, so the optimizer coasts through small bumps and damps out oscillations, like a heavy ball rolling downhill (chapter 9).

MSE (mean squared error).

The regression loss: the average squared gap between the prediction and the target. It is the maximum-likelihood loss under Gaussian noise (chapter 6).

Multi-head attention.

Running several attention operations in parallel on different learned projections of the input, then concatenating their outputs. Each head can specialize in a different kind of relationship between tokens (chapter 16).

Multilayer perceptron (MLP).

A fully-connected feedforward network: stacked layers of neurons with nonlinear activations. The Playground's model (chapter 3).

Mutual information.

How many bits knowing one variable tells you about another — the drop in uncertainty about one when you know the other. It is zero exactly when the two variables are independent (appendix D).

Neuron.

The basic unit: it weights its inputs, adds a bias, and passes the sum through an activation. A network is just many of these repeated (chapter 1).

Normalization (inputs).

Rescaling features to comparable ranges (for example, zero mean and unit variance) so no single feature dominates the weighted sum — computed on training data only (chapter 25).

One-hot encoding.

Representing a class label as a vector of zeros with a single 1. It is the target format for softmax + cross-entropy (chapter 6).

Optimizer.

The rule that turns gradients into parameter updates: SGD, Momentum, RMSProp or Adam. Compare them in the Gradient-descent lab (chapter 9).

Overfitting.

When a model fits training noise and fails on new data — train loss keeps falling while test loss rises. The Overfitting lab shows it live (chapter 12).

Oversmoothing.

The failure where stacking too many GNN layers makes every node's representation drift toward the same vector, washing out useful differences. It limits how deep plain message-passing networks can usefully go (chapter 18).

Parameter.

A learned number — a weight or a bias. Training adjusts them; a network's size is often quoted as its parameter count (chapter 3).

Parameter-efficient fine-tuning (PEFT).

A family of methods — LoRA, adapters, prompt tuning — that adapt a large pretrained model by training only a small set of extra or selected parameters. It greatly reduces the memory and storage cost of full fine-tuning (chapter 22).

Perceptron.

The 1958 single-neuron classifier with a step activation and its own learning rule. Powerful but limited to linearly separable problems (chapter 2).

Perplexity.

The exponential of the average per-token cross-entropy — roughly how many equally likely choices the model faces at each step. Lower is better, and it is the standard intrinsic score for language models (chapter 17).

Policy.

The agent's strategy: a mapping from states to actions, or to a distribution over actions. Learning good behavior means improving the policy so it collects more reward (chapter 23).

Policy gradient.

Optimizing a parameterized policy directly by climbing the gradient of expected reward, raising the probability of actions that led to high return. It handles continuous actions and stochastic policies naturally (chapter 23).

Pooling.

Downsampling a feature map (usually taking the max of each small window) to shrink it and add tolerance to small shifts (chapter 13).

Positional encoding.

Information about token order that is added to the embeddings, since attention on its own is permutation-invariant and blind to order. It can be fixed sinusoids or learned vectors (chapter 16).

Precision / recall / F1.

Classification metrics that hold up under class imbalance: precision = of the predicted positives, how many were right; recall = of the actual positives, how many were caught; F1 is their harmonic mean (chapter 25).

Pruning.

Removing weights, channels or whole structures that contribute little, leaving a smaller and sparser network. It is usually followed by fine-tuning to recover any accuracy lost (chapter 27).

Q-learning.

An off-policy algorithm that learns the action-value function Q(s,a) by updating toward reward plus the best available next Q, converging to an optimal greedy policy. It learns action values without any model of the environment (chapter 23).

Quantization.

Storing and computing weights and activations in lower precision — say 8-bit integers instead of 32-bit floats — to shrink a model and speed up inference. Modern schemes keep accuracy almost unchanged (chapter 27).

Query, key, value.

The three learned projections of each token in attention: the query asks, the keys advertise, and the score query·key decides how much of each token's value is mixed into the output (chapter 16).

Receptive field.

The region of the input that can influence a particular activation. It grows with depth, stride and pooling, so deep units see broad context while early units see only local detail (chapter 13).

Regularization.

Any technique that fights overfitting — L2 weight decay, dropout, early stopping, data augmentation (chapter 12).

Reinforcement learning.

Learning to act by trial and error: an agent takes actions in an environment and adjusts its behavior to maximize total reward. There are no labeled examples, only delayed, evaluative feedback (chapter 23).

ReLU.

The rectified linear unit max(0,x): cheap, non-saturating on the positive side, and the default hidden activation for deep networks (chapter 4).

Reparameterization trick.

Writing a random sample as $z = \mu + \sigma \odot \varepsilon$ with ε fixed noise, so gradients can flow through the sampling step in a VAE (chapter 19).

Residual connection.

A shortcut $y = F(x) + x$ that lets gradients skip layers, which enables networks hundreds of layers deep (ResNet). It is also the backbone of transformer blocks (chapter 10).

ResNet.

A deep convolutional architecture built from residual blocks ($y = F(x) + x$). Its shortcuts let gradients flow, so networks 50 to 150+ layers deep become trainable. It launched the modern era of very deep vision models (chapter 14).

RMSNorm.

A lighter layer normalization that rescales activations by only their root-mean-square, dropping the mean-centering and the bias term. It is cheaper to compute and now common in large language models such as LLaMA (chapter 11).

RMSProp.

An optimizer that divides each parameter's step by the square root of a running average of its squared gradients, which evens out progress across the different axes (chapter 9).

RNN.

Recurrent neural network: a cell reused at every time step, carrying a hidden state forward — weight sharing across time for sequences (chapter 15).

Saddle point.

A spot where the loss slopes up in some directions and down in others. These, more than local minima, are what slow high-dimensional optimization (chapter 7).

Saliency map.

A heatmap over the input marking which pixels or features most affected a prediction, often taken from the gradient of the output with respect to the input. It is a quick first look at where a model is paying attention (chapter 28).

Saturation.

When a neuron's activation sits in a flat region of its function (for example, sigmoid near 0 or 1), so its derivative ≈ 0 and it stops learning (chapter 4).

Scaled dot-product attention.

The core attention formula $\text{softmax}(QK^T/\sqrt{d})\cdot V$, where dividing by $\sqrt{d}$ keeps the dot products from growing with dimension and saturating the softmax. It is the operation every transformer repeats (chapter 16).

Self-attention.

Attention where the queries, keys, and values all come from the same sequence, letting every token look at every other one in a single step (chapter 16).

Self-supervised learning.

Learning useful representations from unlabeled data by making up a task whose labels come free from the data itself — predicting a masked word, a rotation, or a hidden patch. It is the engine behind modern large-scale pretraining (chapter 20).

Seq2seq.

An encoder-decoder recurrent architecture that reads a whole input sequence into a state and then generates an output sequence from it — the original setup for machine translation. Attention was invented to relieve its single-vector bottleneck (chapter 15).

SGD (stochastic gradient descent).

Gradient descent that uses a mini-batch estimate of the gradient at each step. It is the foundation every other optimizer builds on (chapter 7).

Shapley value / SHAP.

A game-theoretic attribution that fairly divides a prediction among its input features by averaging each feature's marginal contribution over all orderings. SHAP makes this idea practical for machine-learning models (chapter 28).

Sigmoid.

The S-shaped activation $\sigma(z) = 1/(1+e^{-z})$, which squashes any input into (0,1). It was the historic default and is now used mainly at binary output heads (chapter 4).

Softmax.

Turns a vector of logits into a probability distribution (all positive, summing to 1). It is the output activation for multi-class classification (chapter 6).

Tanh.

The zero-centered S-curve in (−1,1). It often trains faster than sigmoid and is common in small nets and RNNs (chapter 4).

Temperature (sampling).

A knob T that divides the logits before the softmax when sampling text: T below 1 sharpens toward the likeliest tokens, T above 1 flattens for more variety, and $T \to 0$ becomes greedy decoding (chapter 17).

Temperature scaling.

A simple after-the-fact calibration that divides the logits by a single learned scalar T, fit on validation data. It softens overconfident probabilities without changing which class wins. It is the standard fix for a miscalibrated classifier (chapter 24).

Test set.

Data held out and used only once, to estimate true generalization. Tuning on it makes the estimate invalid (chapter 25).

Tokenization.

Splitting raw text into the discrete units, or tokens, that a model actually reads — usually subwords rather than whole words or characters. The choice of tokenizer fixes the vocabulary and the effective sequence length (chapter 17).

Transfer learning.

Reusing what a model learned on one large task to jump-start a related task that has far less data. In practice you take a pretrained network and adapt it rather than training from scratch (chapter 22).

Transformer.

The attention-based architecture (self-attention + feedforward + residual blocks) behind modern language and vision models (chapter 16).

U-Net.

An encoder-decoder convolutional network with skip connections between matching resolutions, so fine spatial detail lost during downsampling is restored on the way back up. It is the workhorse of image segmentation and the backbone of many diffusion models (chapter 14).

Universal approximation.

The theorem that a network with one hidden layer can approximate any continuous function — an existence result only; it says nothing about learnability or the size needed (chapter 3).

VAE.

Variational autoencoder: an autoencoder whose bottleneck is a probability distribution, regularized toward a standard Gaussian so the latent space can be sampled to generate data (chapter 19).

Validation set.

Data held out to tune hyperparameters and decide when to stop, separate from the final test set (chapter 25).

Value function.

The expected total (discounted) future reward from a state, or from a state-action pair, under a given policy. It tells the agent which situations are worth being in and guides better decisions (chapter 23).

Vanishing gradient.

When gradients shrink toward zero as they pass back through many layers (or time steps), which stalls learning in the early layers. It is caused by saturating activations and poor init (chapter 10).

Vision Transformer (ViT).

A model that cuts an image into fixed patches, embeds each patch as a token, and processes the sequence with a plain transformer — no convolutions. Given enough training data, it matches or beats CNNs (chapter 14).

Weight.

A learned multiplier on a connection. Its sign sets excite versus inhibit, and its magnitude sets the strength — the indigo/amber edges in the diagram (chapter 1).

Weight decay (L2).

Adding $\lambda\|w\|^2$ to the loss, which shrinks weights toward zero each step and favors simpler, smoother functions (chapter 12).

word2vec.

An early method that learns word embeddings by predicting a word from its neighbors (or the reverse), producing vectors whose directions capture analogies. It made the idea of meaning as geometry popular (chapter 5).

Xavier / Glorot init.

Weight scale $\sqrt{(2/(n_{in}+n_{out}))}$, balancing signal variance forward and backward for tanh/sigmoid networks (chapter 10).

XOR problem.

The exclusive-or pattern no single line can separate — the classic proof that perceptrons need hidden layers (chapter 2).