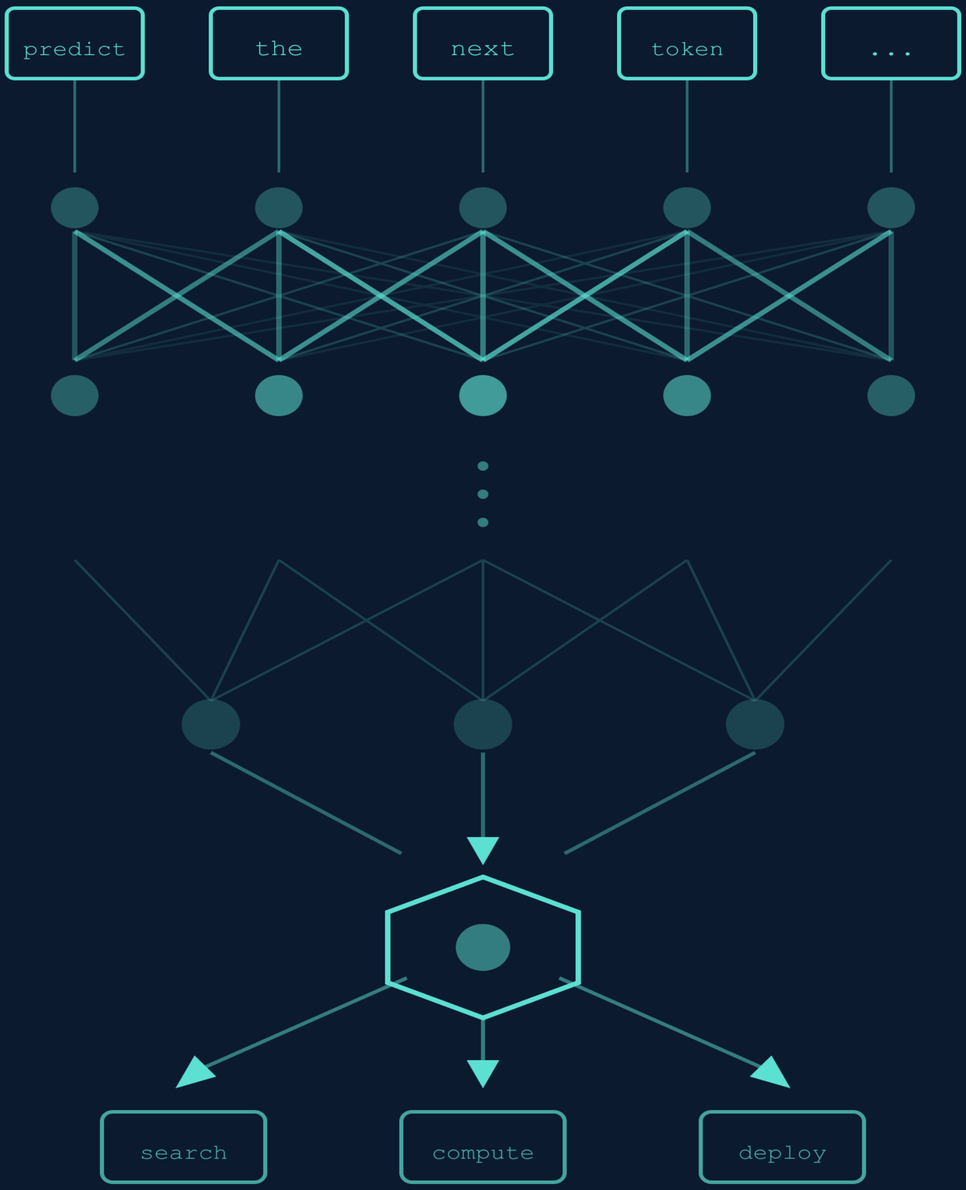
Practical Language Models ▌

From Intuition to Agents in Production
with Python and FastAPI



Igor Benav

Practical Language Models

From Intuition to Agents in Production

Igor Benav

Contents

Preface

To be written.

About This Book

Most machine learning resources fall into two camps: API tutorials that have you calling `.fit()` without understanding the mechanics, or textbooks dense with notation that are targeted on researchers. This book tries to find a middle ground.

I'm not a researcher. I'm a developer who maintains open-source libraries and builds production AI pipelines for enterprise clients. In my experience, the biggest bottleneck in AI engineering isn't the model's complexity; it's building a system that is reliable, efficient, and maintainable. I wrote this book to give you the mental model I use to ship these systems.

We start with the mathematical definition of learning before touching any library. This isn't for academic rigor; it's for survival. When your model fails in production, you need to know if the problem is your data, your architecture, or your loss function. If you skip the math, you'll still know what's happening. If you engage with it, you'll be able to reason about new problems on your own.

We build neural networks from scratch in NumPy before using any framework. We use Python throughout, uv for package management, FastAPI for serving models, and PydanticAI for building LLM applications. We're not surveying every algorithm or teaching you the "best" approach. We're teaching you one coherent path from mathematical foundations to deployed applications, so you have solid ground to stand on when you explore other directions later.

The math is important, but it's not the gatekeeper. Every concept is introduced with intuition first: what we're trying to do and why. The equations formalize what you already understand partially; if you skip the math, you should still get a grasp on what's happening and why, if you engage with it, you'll know how and be able to reason about new problems on your own.

What You'll Build

The book follows a single thread: by the end, you'll have a good understanding of how language models work from the ground up and have built systems that people can actually use.

We start with the learning problem itself: what it means for a machine to learn, what can go wrong, and how you know it's working. Then we build neural networks from scratch in NumPy: perceptrons, forward passes, backpropagation, gradient descent. You'll train a network by hand before any framework does it for you.

Then we tackle the core challenge of this book: how to represent language as numbers. We'll cover word embeddings, attention mechanisms, and the transformer architecture that powers modern language models. You'll understand why these models work, where they break, and what the math is actually doing.

Finally, we build. You'll create LLM-powered agents with tools, structured output, and retrieval-augmented generation (RAG). You'll design multi-step pipelines that combine what LLMs are good at with what code is good at. And you'll deploy the result with FastAPI so it's not just running on your machine.

How to Use This Book

This book assumes you know Python. You should be comfortable with classes, decorators, and asyncio, we'll also use NumPy. Some background in linear algebra and calculus is also necessary; specifically dot products and the chain rule. If you haven't seen these in a while, the book explains what you need when you need it. You don't need a math degree.

The chapters are meant to be read in order. Each one builds on the previous.

Each chapter should have these:

- **Intuition first**: What are we trying to do and why?
- **The math**: Formalizing the intuition with equations and concrete numbers
- **Implementation**: How to actually do it in code
- **Hands-on project**: A practical exercise that uses what you just learned
- **Exercises**: More practice on your own

Type the code yourself. Don't copy and paste. The errors you make and fix along the way are part of learning. Change things. Break things. See what happens.

When you get stuck, resist asking AI for the solution (yes, even though this is a book about AI). Ask for a hint if you need to, but do the thinking yourself. Struggling is normal. It's also how you actually learn, rather than just feeling like you're learning.

The complete code for each chapter's hands-on project is available at github.com/Applied-Computing-League/applied-ai. Consider it a last resort. If you look at the solution before genuinely struggling with the problem, you're only cheating yourself out of the learning.

Prerequisites

You'll need:

- Python 3.11 or newer installed

- A code editor (VS Code works well, I like Zed)
- A terminal you're comfortable using
- Comfortable Python knowledge (variables, functions, loops, lists, dictionaries, classes, decorators, asyncio)
- Basic linear algebra (vectors, dot products, matrix multiplication)
- Basic single and multivariable calculus (derivatives, chain rule, partial derivatives)
- Willingness to try to understand equations and read error messages instead of panicking

You don't need prior machine learning experience. You don't need to know NumPy, PyTorch, or any ML framework. We'll cover what you need as we go.

Book Structure

The book is organized in three parts.

Part I: Foundations covers the math and intuition you need before touching an API. What learning means, how neural networks work, how to represent text as numbers, how attention and transformers work, and how text generation actually happens. By the end of Part I, you'll understand what's going on inside a language model.

Part II: Building with LLMs takes you from understanding to building. You'll work with LLM APIs, create agents with tools, get structured data back from models, build retrieval-augmented generation systems, and design multi-step pipelines. By the end of Part II, you'll have built real AI applications.

Part III: Deploying to Production gets your applications off your laptop. You'll serve models with FastAPI, handle the reliability problems that come with depending on external AI services, and deploy to a real server.

Contributors

Name	Role
Igor Benav	Author

1 The Learning Problem

When we talk about learning, we usually mean getting better at something through study or experience. A child learns to recognize dogs by seeing many dogs. A piano player gets better by playing thousands of songs. In both cases, the learning comes from data: examples, outcomes, patterns observed over time.

We'd love machines to do these kinds of tasks too: identify dogs, read handwriting, translate languages. The obvious approach is programming explicit rules: "if the image has pointy ears and a tail, it's probably a dog." But you'll see that this isn't a feasible approach. Ears come in all shapes. Some dogs don't have tails. Chihuahuas and rottweilers look nothing alike. The rules multiply, the exceptions multiply faster, and you never cover everything. The alternative is to skip the rules entirely and make machines go through the same process we go through: show the machine enough examples and let it figure out the patterns itself. That would be machine learning. The question is how to make it work.

This chapter sets up the problem. We'll define what it means for a machine to learn, look at the main types of learning, and understand what "good enough" means in mathematical terms. Everything else in this book (neural networks, transformers, language models) is a specific method for solving the problem we define here.

1.1 What Does It Mean to Learn?

Think about reading someone's handwriting. You see a weird digit and instantly know it's a 7. You've been doing this your whole life without thinking about it. But try writing down the rules for how you do it. "A 7 has a horizontal line at the top and a diagonal line going down." That works until someone crosses their 7, writes a 1 that looks like a 7, or writes a 7 that looks like a 1. The variation between people's handwriting is enormous, and no set of rules you write down will cover all of it.

We can imagine there's some platonic function that takes an image of a handwritten digit and correctly identifies it as 0 through 9. We as humans can sort of do it, but we can't explain exactly how our brain does it. We've just seen enough handwriting in your life that we learned the pattern. We want machines to do the same thing: look at enough examples and figure it out.

We have data: thousands of handwritten digits that humans have already labeled. We know this scribble is a 7, that one is a 1, this one is a 9. The hope is that a machine can look at these labeled examples and figure out a function that does a reasonable job of classifying new, unseen handwriting.

Let's make this precise. There exists some unknown function $f : X \to Y$ that we want to learn. X is the set of all possible inputs (all possible images of handwritten digits, for example). Y is the set of outputs (the digits 0 through 9 in this case).

We can't access f directly, we don't know it. What we have is a dataset $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$: a collection of input-output pairs where each $y_i = f(x_i)$; these are examples of this function's behavior. D is a finite subset of all possible input-output pairs, a sample from the vast space of everything f could be asked to classify. The hope is that the patterns in this sample are representative of the whole: that what we learn from 60,000 digits generalizes to the millions we'll never see.

Our job is to use D to find a function $g : X \to Y$ such that $g \approx f$. Not just on the examples we've seen, but on new inputs the machine has never encountered. That last part is what separates learning from memorization (Abu-Mostafa, Magdon-Ismail, and Lin 2012).

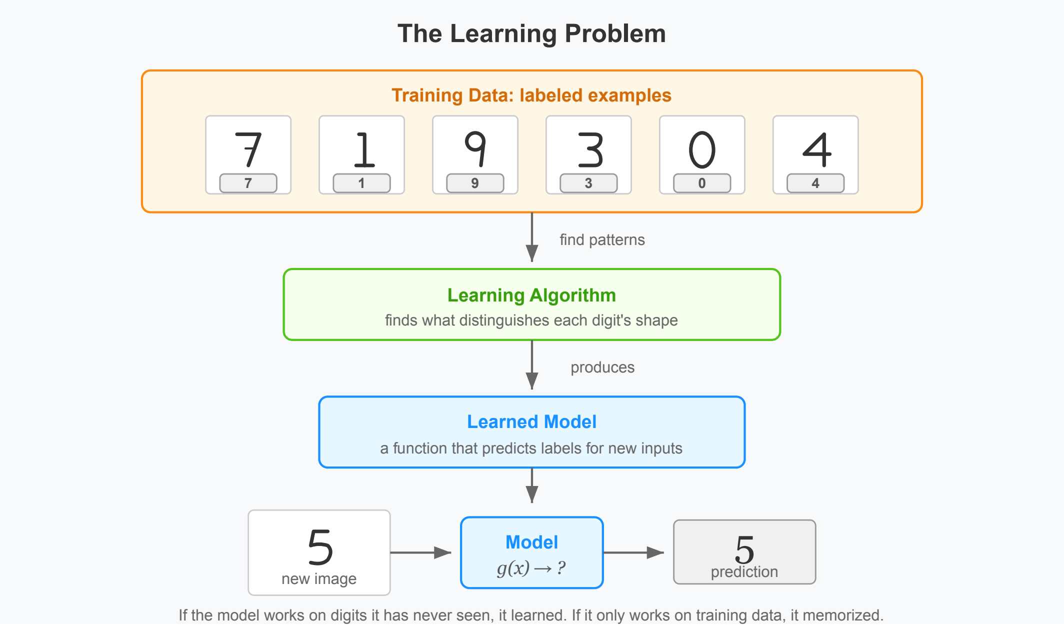

The Learning Problem
Training Data: labeled examples
7
1
9
3
0
4
find patterns
Learning Algorithm
finds what distinguishes each digit's shape
produces
Learned Model
a function that predicts labels for new inputs
5
new image
Model
g(x) → ?
5
prediction
If the model works on digits it has never seen, it learned. If it only works on training data, it memorized.

Figure 1.1: The Learning Problem

In plain language: f is the perfect classifier (that always gets it right). D is our training set (thousands of labeled digit images). g is what the algorithm produces after studying D. The algorithm's goal is to find a g that classifies new handwriting (digits not in D) correctly to a certain margin.

That's the abstract setup: an unknown function, a dataset, and a search for an approximation. But not all datasets look the same, and the kind of data you have determines what kind of learning is possible.

1.2 Supervised and Unsupervised Learning

The setup above assumes something specific about the data: every example has a label. Every digit image is tagged with the correct number. This is **supervised learning**, and it's the most common setting.

In supervised learning, the dataset looks like this:

$$D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

Each example is a pair: an input x_i and its correct output y_i. The algorithm's goal is learning the relationship between inputs and outputs by studying these pairs. It's "supervised" because the labels act like a teacher telling the algorithm the right answer for each example.

But labels are expensive. Someone had to look at each of those thousands of images and write down the correct digit. For many problems, labeled data is scarce. What if you have millions of images but no labels?

Unsupervised learning works with unlabeled data:

$$D = \{x_1, x_2, \dots, x_N\}$$

No labels. The algorithm's task is to find structure on its own. It might discover that some images share similar shapes and group them together, but it doesn't know these groups are "0" and "7" (nobody told it); it simply found a meaningful separation in the data.

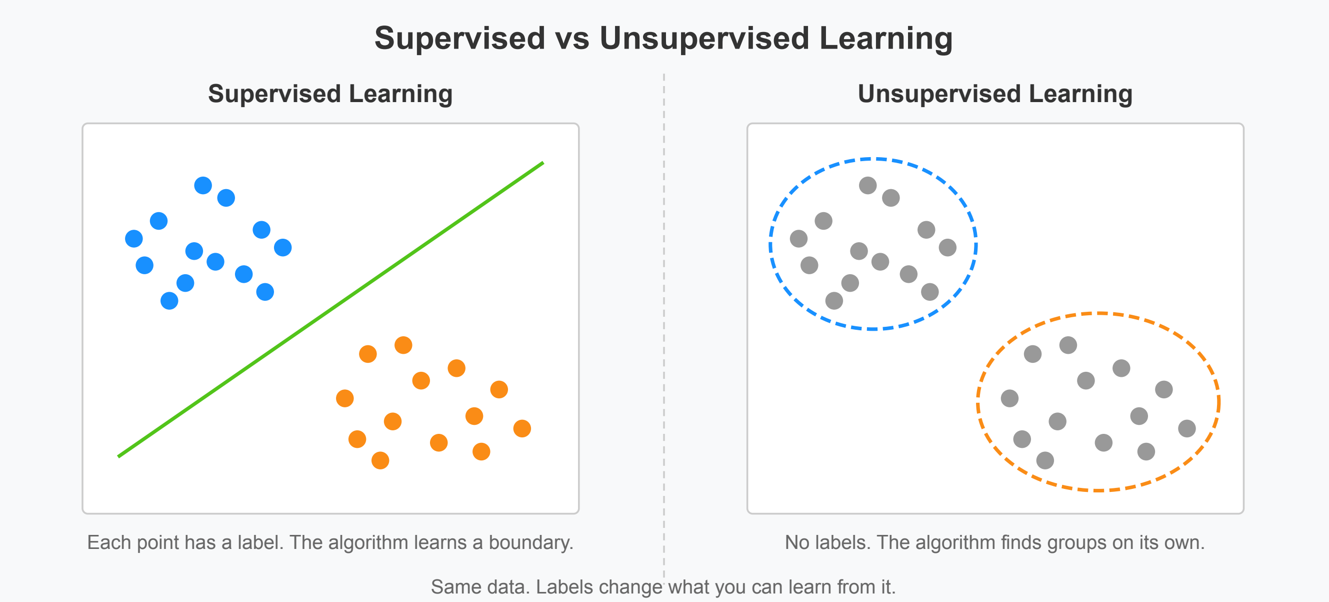

Supervised vs Unsupervised Learning
Supervised Learning
Unsupervised Learning
Each point has a label. The algorithm learns a boundary.
No labels. The algorithm finds groups on its own.
Same data. Labels change what you can learn from it.

Figure 1.2: Supervised vs Unsupervised Learning

The digit recognizer we drafted lead us to supervised learning: the algorithm can't figure out that a scribble means "7" unless someone tells it. But imagine you have a million handwriting samples with no labels. An unsupervised algorithm could still group similar-looking characters together, separating the round shapes from the angular ones, the short ones from the tall ones. It wouldn't know the groups are "0" and "7" (nobody told it), but it found meaningful structure. Unsupervised learning shows up in places where labeling is expensive or impossible: grouping customers by purchasing behavior, detecting unusual network traffic, finding topics in a collection of documents.

There's a third setting worth mentioning briefly. **Reinforcement learning** is learning from interaction: an agent takes actions in an environment, receives rewards or penalties, and learns to maximize reward over time. This is how AlphaGo learned to play Go: not from labeled examples of good moves, but from playing millions of games and learning what leads to winning. We won't cover reinforcement learning in this book, but you should know it exists.

We'll focus on supervised learning for the rest of this chapter and most of the book. The neural networks we build in Chapter 2 are supervised learners. The language models we study later were trained in a way that's technically self-supervised (the labels come from the text itself, as we'll see in Chapter 4), but the mechanics are closer to supervised learning than unsupervised.

Now we know we're doing supervised learning: we have inputs with labels, and we want to learn the relationship between them. But the nature of those labels is important. Predicting a category and predicting a number are fundamentally different problems.

1.3 Two Kinds of Problems

Within supervised learning, there are two types of problems based on what Y looks like.

Classification is when Y is a set of categories. The digit 0 through 9. Spam or not spam. Positive, negative, or neutral sentiment. The output is a label from a finite set. Our digit recognizer is a classification problem with $Y = \{0, 1, 2, \dots, 9\}$.

Regression is when Y is a number on a continuous scale: predicting the price of a house given its square footage; predicting tomorrow's temperature. The output is a value like $425,000 or 23.5°C. There's no finite set of labels to choose from.

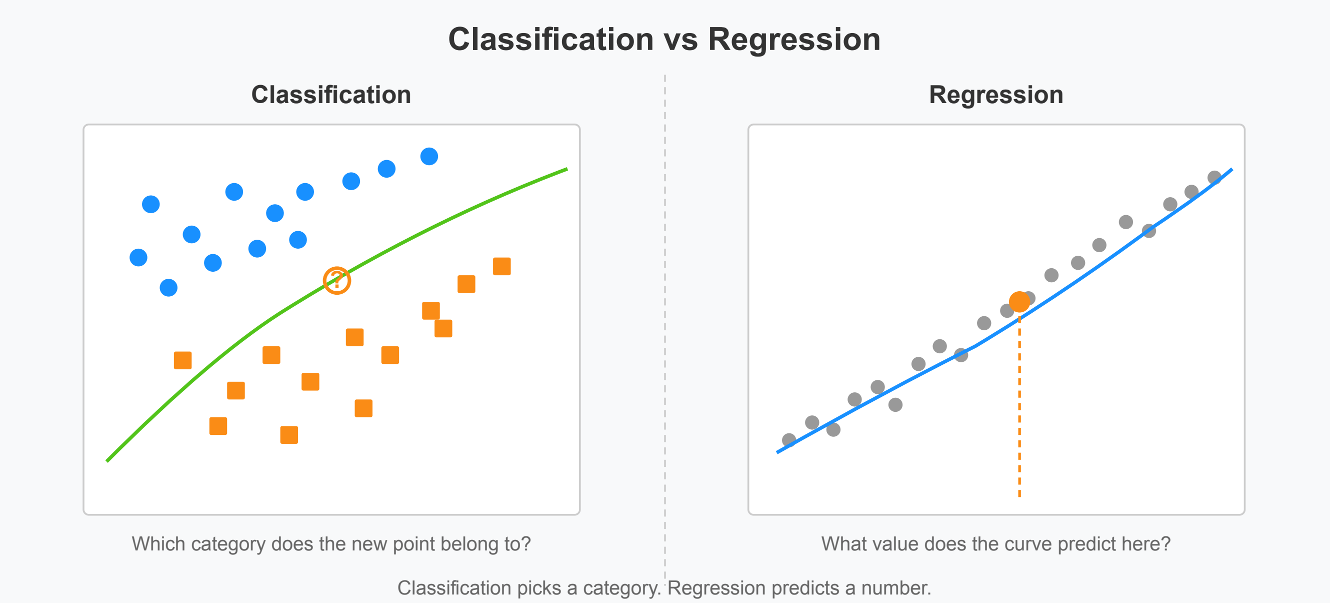


Figure 1.3: Classification vs Regression

The distinction is fundamental because it changes what "getting it wrong" means. In classification, you're either right or wrong: the digit is a 7, and you said 7. In regression, you're almost always a little off: the house sold for $425,000 and you predicted $419,000. How you measure that error is important, and we'll get to it soon.

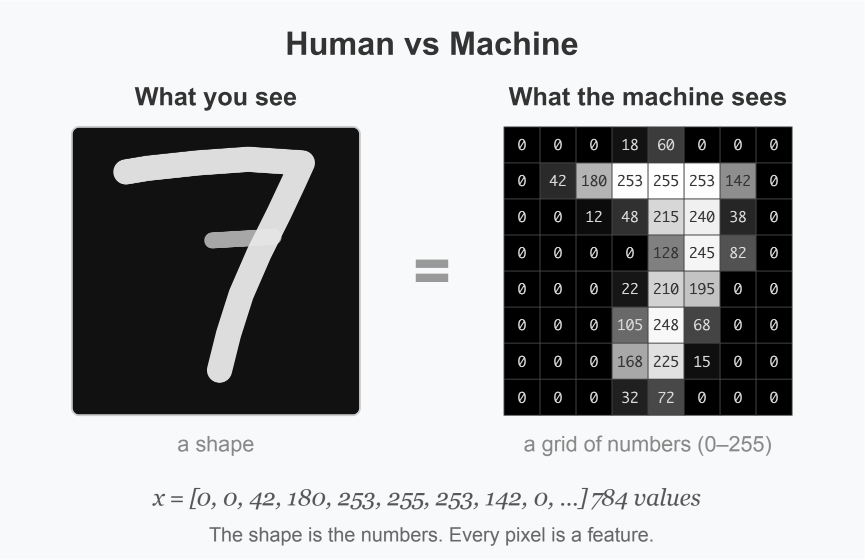
Most of what we build in this book involves classification. When a language model predicts the next word, it's choosing from a vocabulary of possible words: that's classification over a very large set. When a sentiment analyzer reads a review and outputs "positive," that's classification too.

We know the problem (find $g \approx f$), the data setting (supervised), and the type of output (classification). But we've been talking about all of this abstractly. What does x_i actually look like? What does the machine see when it looks at a handwritten digit? And how does it go from that raw input to a prediction?

1.4 From Pixels to Predictions

We've defined the problem abstractly: find g such that $g \approx f$. But what does g actually look like? How does a machine take an image of a handwritten digit and produce the number 7? Before we can measure how good a model is, we need to understand how it even makes a prediction at all.

Start with what the machine actually sees. You look at a handwritten digit and see a shape. The machine sees a grid (matrix) of numbers. Each pixel in a 28×28 image has a brightness value between 0 (black) and 255 (white). Flatten that grid into a row, and you have 784 numbers. That row of numbers (the one dimension vector we get by flattening the matrix) is x_i in our notation. That's what the input looks like to the machine.


Human vs Machine
What you see
What the machine sees
=
a shape
a grid of numbers (0–255)
x = [0, 0, 42, 180, 253, 255, 253, 142, 0, ...] 784 values
The shape is the numbers. Every pixel is a feature.

0	0	0	18	60	0	0	0
0	42	180	253	255	253	142	0
0	0	12	48	215	240	38	0
0	0	0	0	128	245	82	0
0	0	0	22	210	195	0	0
0	0	0	105	248	68	0	0
0	0	0	168	225	15	0	0
0	0	0	32	72	0	0	0

Figure 1.4: What the Machine Sees

$$x_i = [0, 0, 0, 12, 180, 255, 253, 142, 0, ...] \quad \text{(784 values)}$$

The machine has no concept of "stroke" or "curve" or "loop." It sees a list of numbers. Everything it learns, it learns from patterns in those numbers. This numerical representation is called the **features** of the input, and it's one of the most important ideas in machine learning: what the model can learn is limited by what the features capture. Good features make learning easy; bad features make it impossible. We'll see this concretely in the hands-on section, and it becomes the central question of Chapter 3 when we tackle text.

So we have inputs as lists of numbers. How do we get from there to a prediction? The simplest idea is **similarity**: if two images have similar pixel values, they're probably the same digit. A new image that looks like the 7s we've seen before is probably a 7.

But "similar" needs a precise definition. We have two lists of 784 numbers. How different are they? One approach we could try: compare them position by position. If pixel 47 is bright in both images, the difference is small. If it's bright in one and dark in the other, the difference is large. Square each difference (so negatives don't cancel positives), add them all up, and take the square root. This is the **Euclidean distance**:

$$d(a, b) = \sqrt{\sum_{j=1}^{784} (a_j - b_j)^2}$$

Applying it:

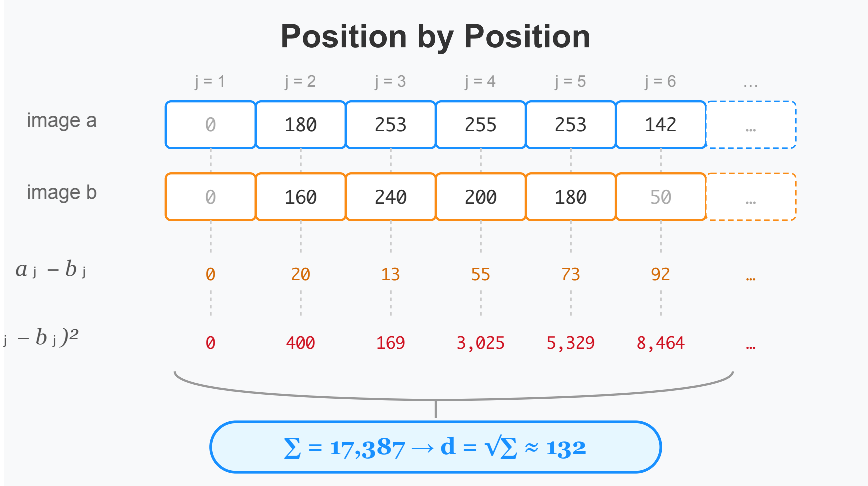


Figure 1.5: Position by Position

If two images are identical, every difference is 0 and the distance is 0. If they're very different, the distance is large. Notice that we're comparing each pixel to the pixel *at the same position* in the other image; this assumes the digit sits in roughly the same spot in both images. We'll come back to that.

This is the same formula as the distance between two points in space, just extended to 784 dimensions. Each image is a point in a 784-dimensional space, and we're measuring how far apart two points are.

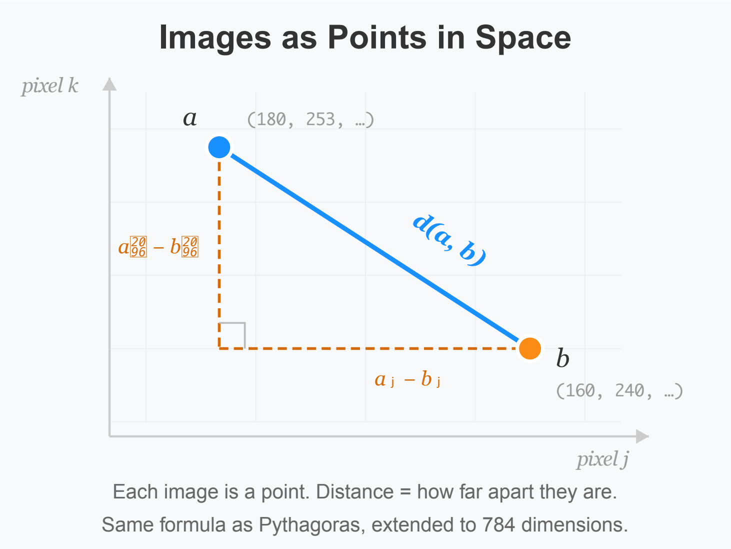

Images as Points in Space
pixel k
a
(180, 253, …)
d(a, b)
$a_k - b_k$
b
$a_j - b_j$
(160, 240, …)
pixel j
Each image is a point. Distance = how far apart they are.
Same formula as Pythagoras, extended to 784 dimensions.

Figure 1.6: Images as Points in Space

Now we have a way to compare images. The simplest possible way to classify follows immediately: given a new image, compare it to every image in the training set. Find the most similar one. Use its label as the prediction. If the closest training image is a 7, predict 7. This is what we call **nearest neighbor** (Cover and Hart 1967).

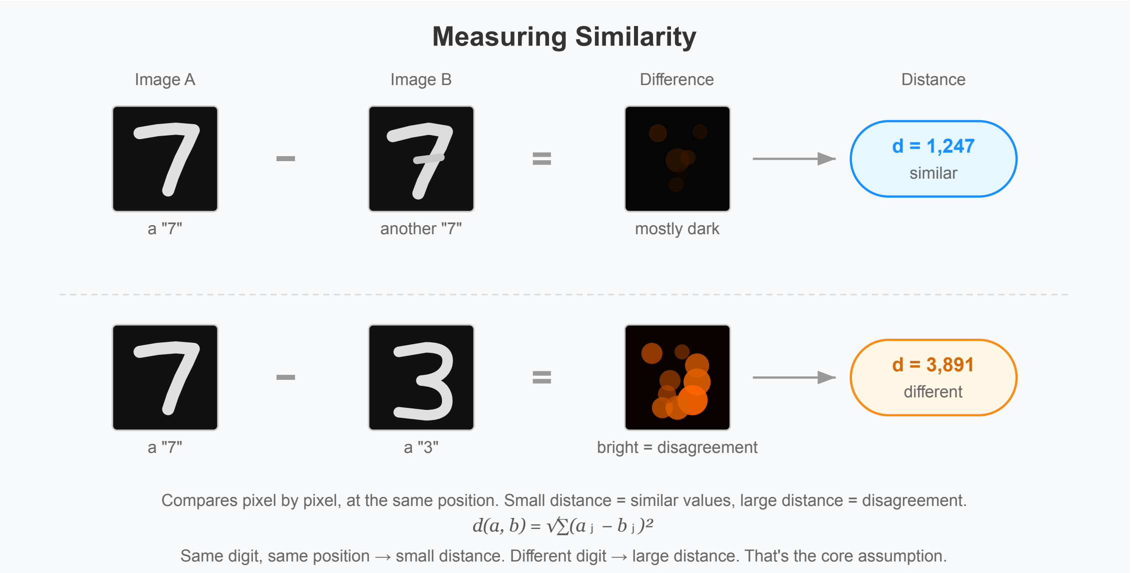

Measuring Similarity
Image A
Image B
Difference
Distance
a "7"
another "7"
mostly dark
d = 1,247
similar
a "7"
a "3"
bright = disagreement
d = 3,891
different
Compares pixel by pixel, at the same position. Small distance = similar values, large distance = disagreement.
d(a, b) = √∑(a ᵢ − b ᵢ)²
Same digit, same position → small distance. Different digit → large distance. That's the core assumption.

Figure 1.7: Measuring Similarity

This kind of works given all the assumptions hold. But think about the cost. To classify one new image, you compare it against all 56,000 training images. To classify 14,000 test images, that's 56,000 × 14,000 = 784 million distance calculations. The model isn't a compact function; it's the entire training set. This is pure memorization: the "model" is the data itself.

Can we do better? What if instead of keeping every training image, we summarized each digit class as a single **template**? Average all the 7s to get a single blurry image of what a typical 7 looks like. Average all the 1s. Do this for each digit, and you have ten templates. To classify a new image, measure its distance to each of the ten templates and pick the closest.

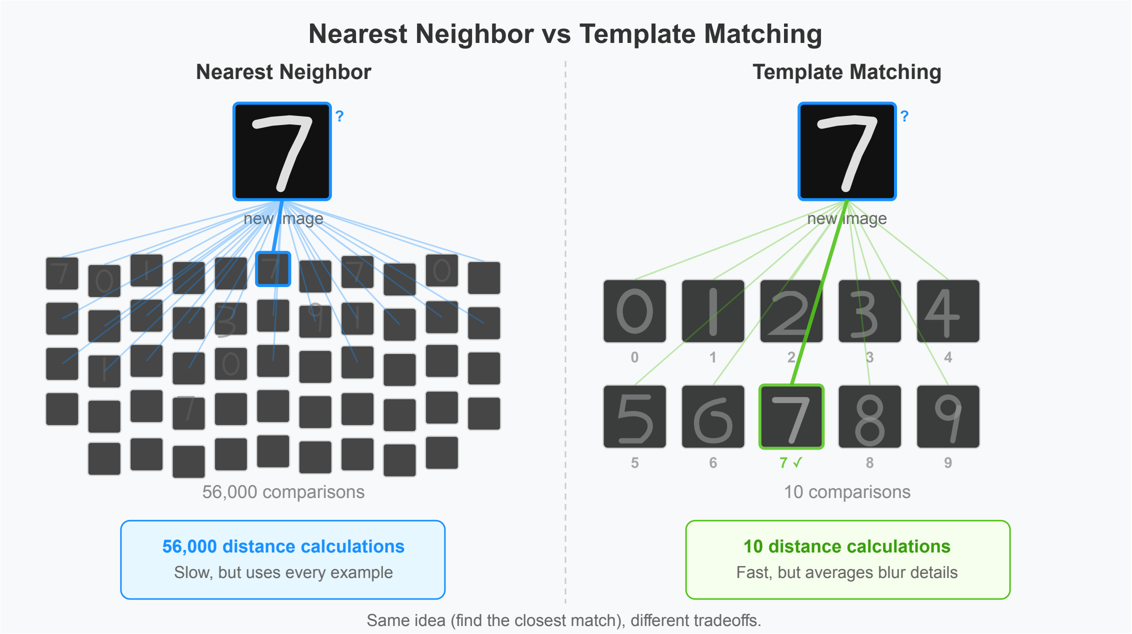

Nearest Neighbor vs Template Matching
Nearest Neighbor
Template Matching
?
new image
?
new image
0
1
2
3
4
5
6
7 ✓
8
9
56,000 comparisons
10 comparisons
56,000 distance calculations
Slow, but uses every example
10 distance calculations
Fast, but averages blur details
Same idea (find the closest match), different tradeoffs.

Figure 1.8: Nearest Neighbor vs Template Matching

This is called **template matching**, and it's our first real g: a function that takes an image and produces a digit label. It's faster (10 distance calculations instead of 56,000) and interpretable (you can literally look at the templates and see what the model "thinks" each digit looks like). The average 0 is a fuzzy oval. The average 1 is a narrow smear.

It's also obviously limited. Averaging thousands of different handwriting styles produces a blurry mess. When two digits have similar average shapes (4 and 9 both have a vertical stroke on the right), the templates can't really tell them apart. And some digits have multiple distinct styles: some people write 1 as a simple vertical stroke, others add serifs or a diagonal lean. One template can't capture both.

You could fix that last problem by keeping multiple templates per digit: instead of one average 7, keep three different "styles" of 7. This is the beginning of what is called **clustering**: automatically discovering distinct groups within a class. But even with more templates, there's a deeper problem we haven't dealt with.

Remember the assumption we flagged earlier: Euclidean distance compares each pixel to the pixel at the same position. Take a 7 and shift it three pixels to the right. Same digit, same handwriting, barely perceptible to a human. But the Euclidean distance to the original is enormous; on real MNIST digits, it can be larger than the distance to a completely different digit. Pixel-wise, a slightly displaced 7 looks less like itself than a 3 does. Why? Everywhere the digit has a sharp edge, shifting creates a large disagreement at two positions: where the edge was, and where it moved to. The distance doesn't measure how different two shapes are. It measures how much their edges have moved.

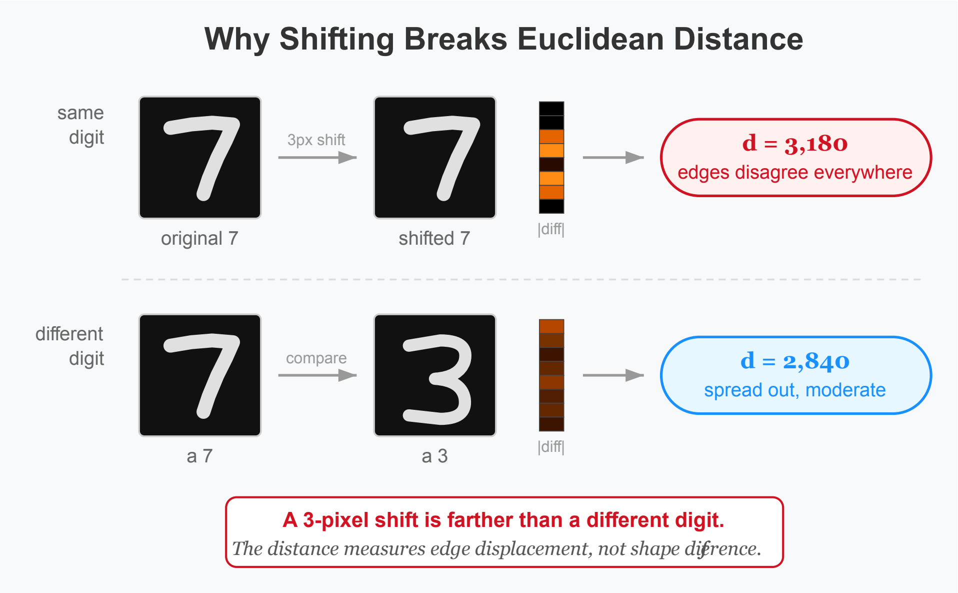

Why Shifting Breaks Euclidean Distance
same digit
original 7
3px shift
shifted 7
|diff|
d = 3,180
edges disagree everywhere
different digit
a 7
compare
a 3
|diff|
d = 2,840
spread out, moderate
A 3-pixel shift is farther than a different digit.
The distance measures edge displacement, not shape difference.

Figure 1.9: Why Shifting Breaks Euclidean Distance

No amount of templates or clusters will fix this. The features (raw pixels) don't capture position-invariance, and Euclidean distance is the wrong way to compare them.

This is why we need more powerful models. The template matcher can't learn that "a 7 shifted left three pixels is still a 7." We need to find something that can.

But the template matcher does something important: it gives us a concrete g that makes predictions we can evaluate. We need that, because the next question is: how do we measure whether g is any good?

1.5 How Good Is Good Enough?

We've said the goal is to find g such that $g \approx f$. But "approximately equal" is vague. We need a number that says how wrong g is, so we can make it less wrong.

Start with classification. Your model looks at a handwritten digit and predicts "7." The true label is "7." Good. Next one: true label is "3," prediction is "8." Bad. Over a set of N examples, you count:

$$\text{Accuracy} = \frac{\text{number correct}}{N}$$

If your model correctly classifies 9,200 out of 10,000 test digits, its accuracy is 92%. That's straightforward, but it hides an assumption: that the classes are roughly balanced. MNIST has about 7,000 images per digit, so accuracy is a fair measure, but if 90% of your emails are not spam, a model that always predicts "not spam" gets 90% accuracy without learning any relevant pattern. For MNIST this isn't a problem, but for other datasets, you'd want to look at per-class performance, which we'll do in the hands-on with the confusion matrix.

Regression is trickier. You're predicting numbers, so you'll essentially never get the exact value. You need to measure how far off you were. The simplest idea: take the difference between your prediction and the true value, and make it positive.

$$|g(x_i) - y_i|$$

If the house sold for $425,000 and you predicted $419,000, your error is $6,000. Average this over all examples and you get the **Mean Absolute Error** (MAE):

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |g(x_i) - y_i|$$

That works, but it treats all errors equally. Being off by $1,000 ten times is the same total error as being off by $10,000 once. In practice, big errors are usually much worse than small ones. A house price model that's usually close but occasionally wildly wrong is dangerous. One that's consistently off by a small amount is useful.

We need to penalize larger errors more than small errors, so instead of taking the absolute value, we can square the error:

$$(g(x_i) - y_i)^2$$

Small errors stay small when squared (5^2 = 25). Big errors explode (100^2 = 10,000). This punishes large mistakes disproportionately. Average these and you get the **Mean Squared Error** (MSE):

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (g(x_i) - y_i)^2$$

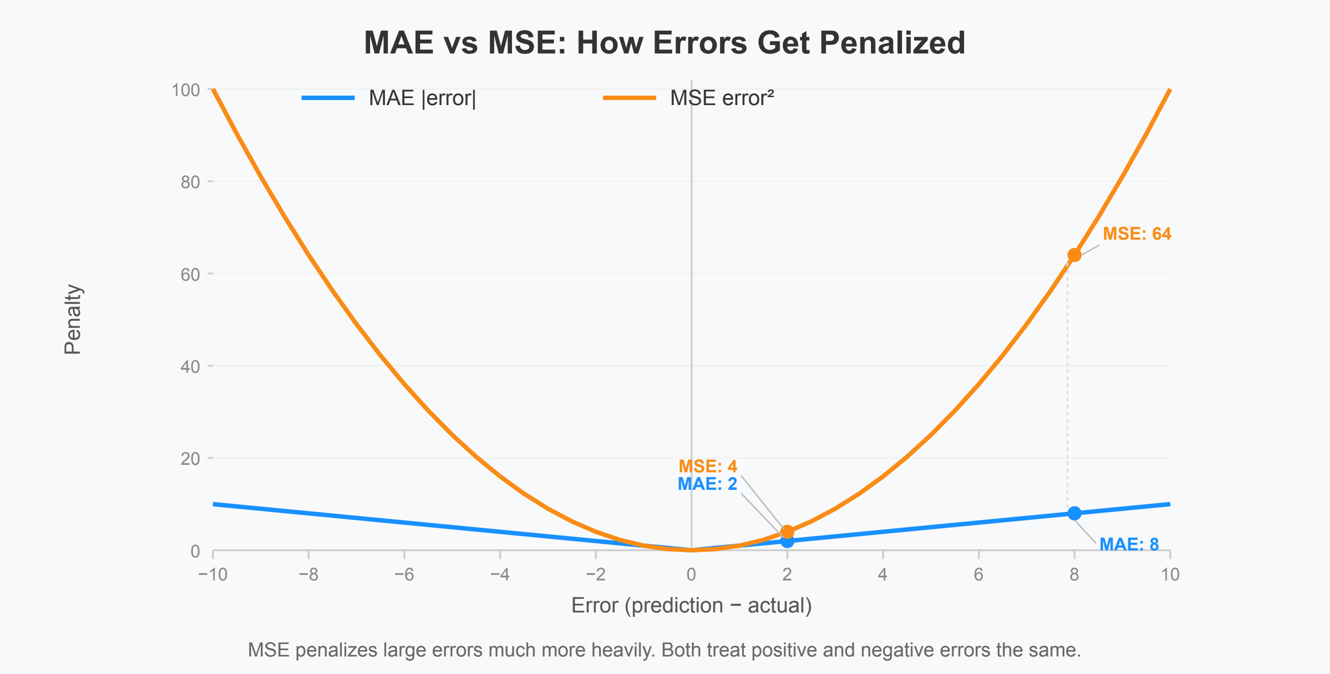


Figure 1.10: MAE vs MSE: How Errors Get Penalized

The diagram makes this concrete. For small errors, MAE and MSE are close. For large errors, MSE grows much faster. This is why MSE is the default for most regression problems: it strongly discourages the kind of catastrophic predictions that make a model useless in practice.

Notice anything familiar? MSE has the same structure as the Euclidean distance from earlier in the chapter. Your predictions form one list of numbers, the true labels form another, and MSE measures how far apart they are; position by position, squared, summed.

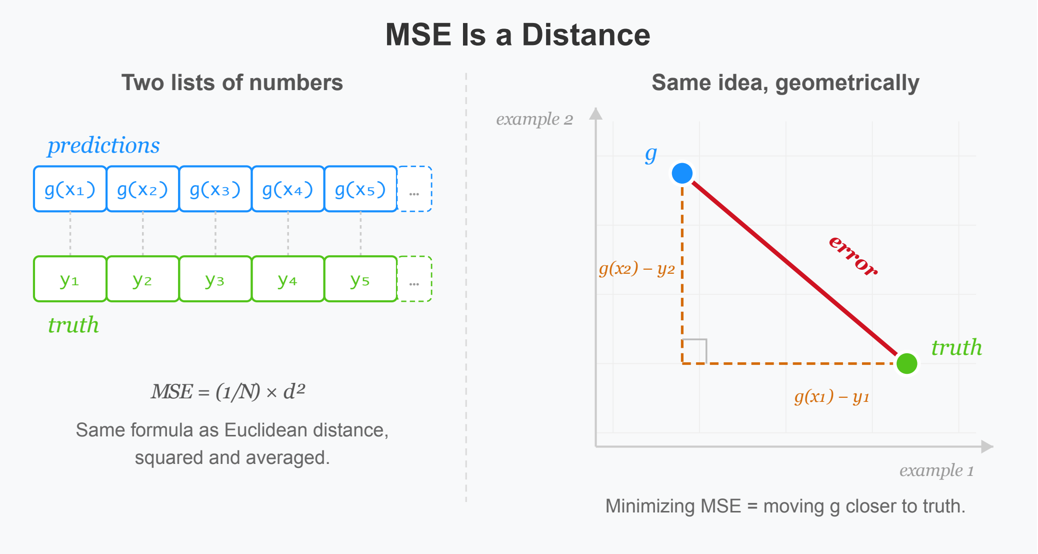


Figure 1.11: MSE Is a Distance

Minimizing MSE means moving your predictions as close as possible to the truth, in exactly the geometric sense we saw before.

These measures of error have a name: **loss functions** (sometimes called cost functions). A loss function takes your model's predictions and the true answers, and returns a single number that says how wrong you are. The entire learning process is about finding the function g that makes this number as small as possible, the closest one to the platonic truth we can get with our data.

The choice of loss function shapes what the model learns. MSE pushes the model to avoid big errors at all costs. MAE pushes the model to be right on average, tolerating occasional larger mistakes. The geometry shows why. MSE measures the straight-line distance between predictions and truth; and a single large error in any direction stretches that line. MAE walks the grid, one axis at a time; ten small errors add up the same as one error ten times larger.

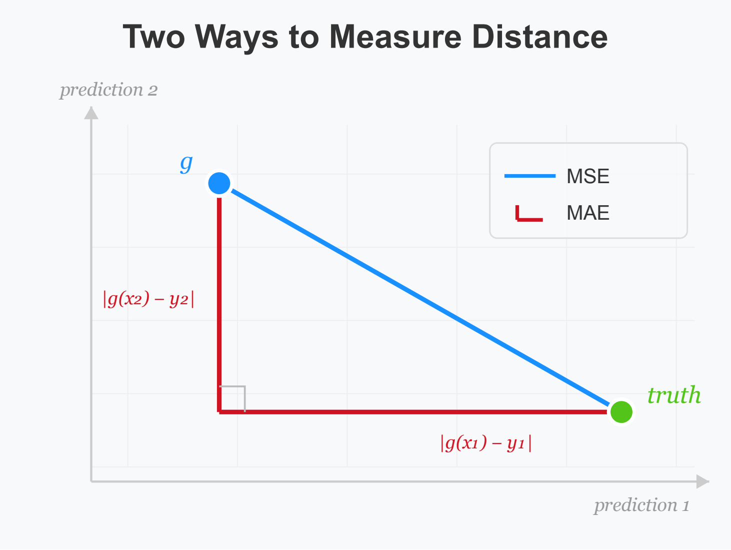


Figure 1.12: Two Ways to Measure Distance

A model that's off by $1,000 on ten houses and perfect on the rest walks the same total distance as one that nails every house but misses one by $10,000. MSE would strongly prefer the first model; MAE treats them equally.

This also tells you when to use which: MSE is the right default when big errors are genuinely rare and mean something went wrong, but if your data has heavy outliers (house prices with a few mansions, sensor readings with occasional spikes), MSE overreacts to them and distorts the whole model. MAE doesn't let a few extreme values dominate the loss. When you pick a loss function, you're making a bet on what your error distribution looks like.

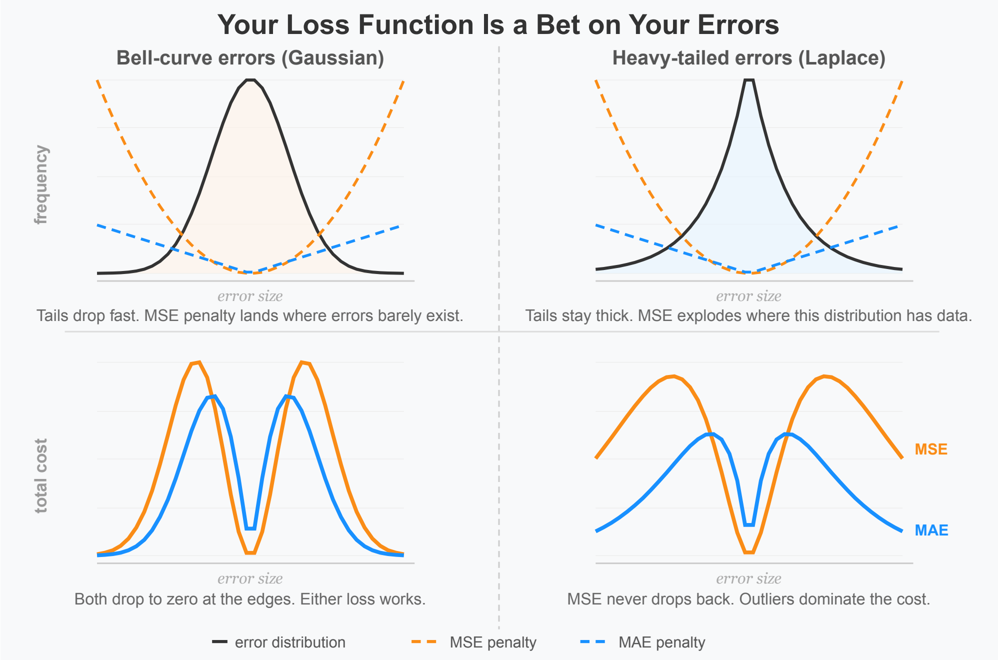


Figure 1.13: Distributions vs Errors

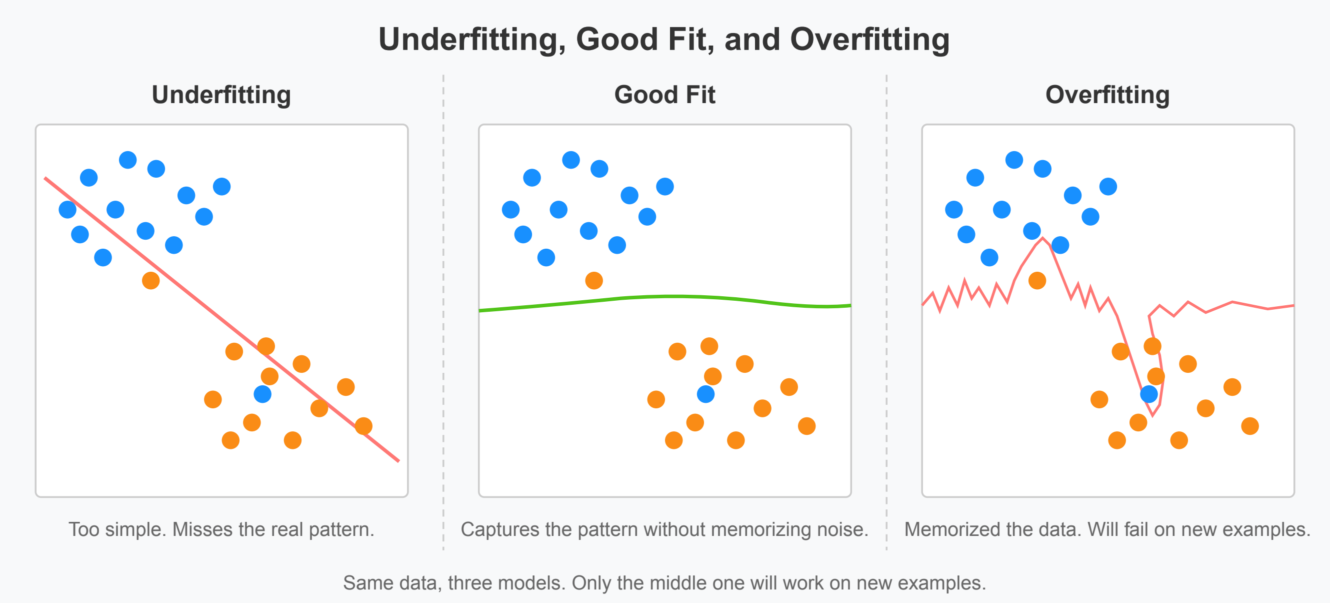
So now we can measure how wrong a model is. But there's a trap. A model can score perfectly on the data it's seen and still be useless. Everything depends on what happens when the model encounters something new.

1.6 Generalization

A model that memorizes its training data can get 100% accuracy on that data. Every digit classified perfectly. MSE of exactly zero. By the loss function's measure, it's a perfect model.

But show it a new digit it hasn't seen, and it might fail completely. It didn't learn the pattern ("round shapes with a closed loop tend to be 0s"). It learned the specific examples ("pixel values matching image #4,721 map to the label 9"). This is **overfitting**: the model fits the training data too well and fails on anything new.

The opposite problem exists too. A model that's too simple might not capture the real patterns in the data. If you try to tell apart 0s and 1s with a single rule ("if the image is dark in the center, it's a 0"), you'll get some right but miss the complexity of real handwriting. This is **underfitting**: the model isn't powerful enough to learn what's actually there.



Underfitting, Good Fit, and Overfitting

Underfitting

Too simple. Misses the real pattern.

Good Fit

Captures the pattern without memorizing noise.

Overfitting

Memorized the data. Will fail on new examples.

Same data, three models. Only the middle one will work on new examples.

Figure 1.14: Underfitting, Good Fit, and Overfitting

Think about it in terms of studying for an exam. You have practice questions and answers from the past five years. You could memorize every answer word for word. On a test that reuses those exact questions, you'd score 100%. On a test with new questions covering the same material, you'd fail. You never learned the subject. You learned the specific answers.

A good student extracts the underlying principles and applies them to new questions. That's generalization. We want our models to do the same.

This is why we split data into **training data** and **test data**. Train the model on one set, evaluate it on another. The test data simulates "new, unseen examples." If the model does well on training data but poorly on test data, it's overfitting. If it does poorly on both, it's underfitting. If it does well on both, it learned something real.

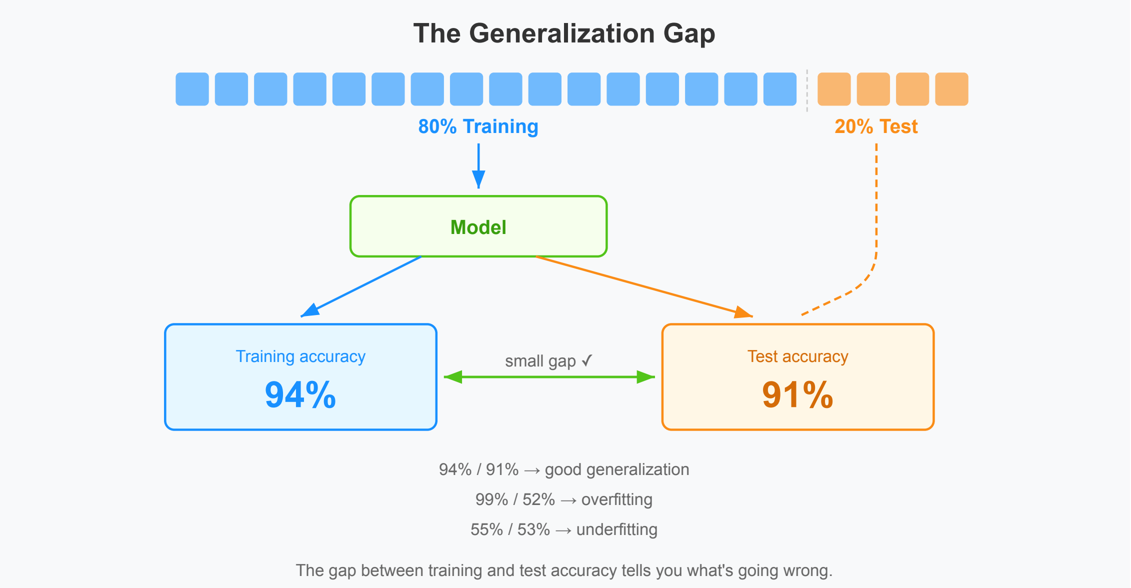


Figure 1.15: The Generalization Gap

A common split is 80% training, 20% testing. The exact ratio isn't sacred, but the principle is: never evaluate a model on data it trained on; that would be like grading students on the homework they copied. You'll see this in practice with companies claiming their language model can solve some benchmark, without mentioning that the benchmark questions were in the training data.

Remember to shuffle the data before splitting if it's necessary; MNIST comes sorted by digit, so a naive split puts all the 0s through 7s in training and all the 8s and 9s in test. The test score would be meaningless: the model never saw those digits during training. Shuffling ensures both sets are representative of the whole dataset.

We talked about template matching and nearest neighbors earlier. Think about where they fall on the overfitting/underfitting spectrum. The nearest-neighbor classifier is the ultimate memorizer: it stores every training example and matches against them. On training data, it gets 100% (every image is its own nearest neighbor). But it might not generalize as well to new data. Template matching is the opposite: ten blurry averages can't memorize anything, so it's unlikely to overfit, but it might underfit by being too simple to capture real patterns. The right model is somewhere in between. That's enough theory. Let's build something.

1.7 Hands-On: Recognizing Handwritten Digits

We've been talking about handwritten digits the entire chapter. Now let's build the template matcher we described in the theory and see how it actually performs. We'll use MNIST (LeCun et al. 1998): 70,000 handwritten digits, each a 28×28 pixel grayscale image labeled 0 through 9. We'll use scikit-learn to download the data, but everything else is pure NumPy. No magic function calls. We build every piece ourselves.

Set up your project:

```
mkdir chapter1
cd chapter1
uv init
uv add numpy matplotlib scikit-learn pandas
```

Create a file called `digits.py`. Let's start by loading the data and seeing what we're working with:

```
# digits.py
from sklearn.datasets import fetch_openml
import numpy as np
import matplotlib.pyplot as plt

mnist = fetch_openml("mnist_784", version=1, as_frame=False, parser="auto")
X = mnist.data
y = mnist.target.astype(int)

print(f"Number of images: {len(X)}")
print(f"Each image: {X.shape[1]} numbers (28 x 28 pixels)")
print(f"Labels: {np.unique(y)}")
```

Run it with `uv run python digits.py`. A few things to notice here. `X` isn't a regular Python list; it's a NumPy array, a grid of numbers that you can manipulate all at once. `X.shape` tells you its dimensions: 70,000 rows (one per image) by 784 columns (one per pixel). `np.unique(y)` returns the distinct values in the labels: the digits 0 through 9. You could do the same with `set(y)`, but NumPy's version returns them sorted.

Let's look at a few images. Each row of `X` is a flat list of 784 numbers. To display it as a picture, we need to fold it back into a 28×28 grid; that's what `.reshape(28, 28)` does. It doesn't change the data, just how it's arranged:

```
# digits.py
# ...existing code above

fig, axes = plt.subplots(2, 8, figsize=(12, 3))
for i, ax in enumerate(axes.flat):
    ax.imshow(X[i].reshape(28, 28), cmap="gray")
    ax.set_title(str(y[i]), fontsize=12)
    ax.axis("off")
plt.tight_layout()
plt.savefig("samples.png", dpi=150)
plt.show()
```

Some are clean. Some are messy. That variation is the whole reason rules don't work and learning does.

Now let's split the data. We need to shuffle first; the dataset comes sorted by digit, and we don't want the test set to be all 9s. NumPy's `rng.permutation` gives us a shuffled list of indices, and then `X[indices]` picks rows in that new order. This is called *fancy indexing*: instead of grabbing one row with `X[5]`, you hand NumPy a whole list of positions and get back all those rows at once:

```
# digits.py
# ...existing code above

rng = np.random.default_rng(42)
indices = rng.permutation(len(X))
X, y = X[indices], y[indices]

split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

print(f"Training: {len(X_train)}, Test: {len(X_test)}")
```

56,000 to train on, 14,000 to test with. The model never sees the test set during training.

Now the template matcher. We described the idea earlier: compute the average image for each digit, then classify by nearest template. Let's wrap it in a class so the logic stays organized. First, the constructor and the fitting method. For each digit d, the template is the mean of all training images with that label:

$$\text{template}_d = \frac{1}{N_d} \sum_{i:y_i=d} x_i$$

where N_d is the number of training images labeled d and each x_i is a 784-dimensional vector:

```
# digits.py
# ...existing code above

class TemplateMatcher:
    def __init__(self):
        self.templates = None

    def fit(self, X, y):
        self.templates = np.zeros((10, X.shape[1]))
        for digit in range(10):
            mask = (y == digit)
            self.templates[digit] = X[mask].mean(axis=0)
```

`fit` builds the templates. The expression `y == digit` compares every label to `digit` at once; it returns an array of `True` and `False` values, one per image. When you use that array to index `X`, NumPy keeps only the rows where the value is `True`. This is called *boolean masking*, and it's the NumPy equivalent of a list comprehension like `[X[i] for i in range(len(X)) if y[i] == digit]`, just faster. Then `.mean(axis=0)` averages those rows column by column. The `axis=0` means "collapse the rows, keep the columns": you go from, say, 5,000 images of the digit 7 down to one row of 784 numbers, the average 7.

Now the prediction method. To classify a new image x, we compute its Euclidean distance to each template and pick the closest one:

$$d(x, \text{template}_k) = \sqrt{\sum_{j=1}^{784} (x_j - \text{template}_{k,j})^2}$$

$$\text{prediction} = \arg \min_{k \in \{0,\ldots,9\}} d(x, \text{template}_k)$$

```
    # digits.py, inside the TemplateMatcher class

    def predict(self, X):
        predictions = np.zeros(len(X), dtype=int)
        for i, image in enumerate(X):
            distances = np.sqrt(((self.templates - image) ** 2).sum(axis=1))
            predictions[i] = distances.argmin()
        return predictions
```

This computes the Euclidean distance from one image to all ten templates at once. `self.templates - image` subtracts the image from every row of the templates (the $(x_j - \text{template}_{k,j})$ part). NumPy automatically lines up the shapes, an operation called *broadcasting*. We square each difference, sum across columns with `.sum(axis=1)` (axis 1 = across each row, the $\sum$ in the formula), and take the square root. The result is ten distances, one per template. `.argmin()` returns the *index* of the smallest one (the arg min), that's our prediction.

Let's fit the model:

```
# digits.py
# ...existing code above

model = TemplateMatcher()
model.fit(X_train, y_train)
```

That's our entire model. Ten rows of 784 numbers each. Let's see what these templates look like:

```
# digits.py
# ...existing code above

fig, axes = plt.subplots(1, 10, figsize=(15, 2))
for digit, ax in enumerate(axes):
    ax.imshow(model.templates[digit].reshape(28, 28), cmap="gray")
    ax.set_title(str(digit), fontsize=12)
    ax.axis("off")
plt.suptitle("Average image per digit", fontsize=14)
plt.tight_layout()
plt.savefig("templates.png", dpi=150)
plt.show()
```

There they are: the blurry ghosts we predicted. The 1 is a narrow smear down the center. The 0 is a fuzzy oval. The 8 looks like a blurry snowman. Look at the 4 and the 9; they look eerily similar: both have a vertical stroke on the right side. We can already predict those two will get confused.

Now let's classify the test set and check accuracy. Accuracy is the fraction of images we got right:

$$\text{accuracy} = \frac{\text{number correct}}{\text{total predictions}} = \frac{\sum_{i=1}^{N} \mathbf{1}[\hat{y}_i = y_i]}{N}$$

where $\hat{y}_i$ is our prediction and y_i is the true label. The $\mathbf{1}[\cdot]$ notation means "1 if the condition is true, 0 otherwise":

```
# digits.py
# ...existing code above

predictions = model.predict(X_test)

accuracy = (predictions == y_test).sum() / len(y_test)
print(f"Correct: {(predictions == y_test).sum()} / {len(y_test)}")
print(f"Accuracy: {accuracy:.1%}")
```

Around 81%. Not bad for ten averaged images and a distance calculation. No learning algorithm, no optimization, no gradients. Just averages and subtraction. But 81% means roughly 1 in 5 digits is wrong. Which ones?

Let's build a confusion matrix: a 10×10 grid where entry C_{ij} counts how many times a true digit i was predicted as digit j:

$$C_{ij} = \sum_{n=1}^{N} \mathbf{1}[y_n = i \;\text{ and }\; \hat{y}_n = j]$$

The diagonal entries C_{ii} are correct predictions. Everything off the diagonal is a mistake:

```
# digits.py
# ...existing code above

def confusion_matrix(y_true, y_pred):
    matrix = np.zeros((10, 10), dtype=int)
    for true, pred in zip(y_true, y_pred):
        matrix[true][pred] += 1
    return matrix

confusion = confusion_matrix(y_test, predictions)
```

Let's plot it:

```
# digits.py
# ...existing code above

def plot_confusion(matrix, filename="confusion.png"):
    fig, ax = plt.subplots(figsize=(8, 8))
    im = ax.imshow(matrix, cmap="Blues")
    ax.set_xlabel("Predicted", fontsize=12)
    ax.set_ylabel("Actual", fontsize=12)
    ax.set_title("What gets confused with what", fontsize=14)
    ax.set_xticks(range(10))
    ax.set_yticks(range(10))
    for i in range(10):
        for j in range(10):
            color = "white" if matrix[i, j] > matrix.max() / 2 else "black"
            ax.text(j, i, str(matrix[i, j]),
                    ha="center", va="center", color=color, fontsize=9)
    plt.colorbar(im, ax=ax, shrink=0.8)
    plt.tight_layout()
    plt.savefig(filename, dpi=150)
    plt.show()

plot_confusion(confusion)
```

Look at where the off-diagonal numbers are biggest. 4 gets confused with 9, just as we predicted from the templates. 3 gets confused with 5 and 8 (similar curves). These are exactly the confusions we'd expect from template matching: when two digits have similar average shapes, the model is worse telling them apart.

Let's compute per-digit accuracy to see which digits are easiest and hardest. For each digit d, we look only at the test images that are actually d and check what fraction we got right:

$$\text{accuracy}_d = \frac{\sum_{i:y_i=d} \mathbf{1}[\hat{y}_i = d]}{\sum_i \mathbf{1}[y_i = d]}$$

```
# digits.py
# ...existing code above

def per_digit_accuracy(y_true, y_pred):
    print("\nPer-digit accuracy:")
    for digit in range(10):
        mask = (y_true == digit)
        digit_acc = (y_pred[mask] == digit).sum() / mask.sum()
        print(f"  {digit}: {digit_acc:.1%}")

per_digit_accuracy(y_test, predictions)
```

0 and 1 are easy (their shapes are distinctive). Digits like 5 and 8 are hard (their averages overlap with other digits). This tells us something about the data itself: some categories are inherently easier to separate than others.

One last experiment. We said that features limit what the model can learn. Let's prove it by giving the model less to work with. We'll train two more template matchers: one that only sees the top half of each image, another that only sees the bottom half.

The notation `X_train[:, start:end]` takes a slice of columns from every row; all images, but only certain pixels. The colon before the comma means "all rows"; `start:end` after the comma selects the columns:

```python
# digits.py
# ...existing code above

def test_partial(X_train, y_train, X_test, y_test, start, end, label):
    model = TemplateMatcher()
    model.fit(X_train[:, start:end], y_train)
    preds = model.predict(X_test[:, start:end])
    acc = (preds == y_test).sum() / len(y_test)
    print(f"{label}: {acc:.1%}")
    return acc
```

Now let's run all three:

```python
# digits.py
# ...existing code above

print(f"\nFull image:       {accuracy:.1%}")
test_partial(X_train, y_train, X_test, y_test, 0, 392, "Top half only   ")
test_partial(X_train, y_train, X_test, y_test, 392, 784, "Bottom half only")
```

The full image wins. But the top half does slightly better than the bottom. The top portion of digits carries more distinguishing information than you might expect: the presence or absence of a horizontal bar, the shape of a curve's peak, whether strokes converge or diverge. Same model, same algorithm, different features, different results.

This is the entire learning problem in about 50 lines of NumPy. Our model was the simplest thing imaginable, and it got close to 81%. In Chapter 2, we'll build a neural network that learns its own features rather than relying on raw pixel averages, and you'll see exactly why it does better.

1.8 Chapter Summary

- Machine learning is finding a function $g \approx f$ from examples, where f is unknown
- Supervised learning uses labeled pairs (x_i, y_i); unsupervised learning finds structure without labels
- Classification predicts categories; regression predicts continuous values
- Features are the numbers the model sees—for images, pixel brightness values
- Euclidean distance measures similarity by comparing vectors position by position
- Template matching classifies by nearest average, but averaging destroys variation
- Loss functions (accuracy, MAE, MSE) measure how wrong a model is
- Generalization is what separates learning from memorization—always evaluate on data the model hasn't seen

In the next chapter, we build a neural network that learns its own features instead of relying on raw pixel averages.

1.9 Exercises

1. Our classifier computes Euclidean distance (square root of sum of squared differences). Try **Manhattan distance** instead (sum of absolute differences). Does accuracy change? Why might one distance metric work better than the other for pixel data?

2. Instead of using the average image as each digit's template, try using the **median** image. The median is less sensitive to outliers than the mean. Does it make a difference? Visualize both templates for a digit like 1, where some people write it with serifs and some without. Which template looks sharper?

3. Our model stores 10 templates. What if we stored more? We mentioned that some digits have multiple distinct writing styles. For each digit, try keeping $k = 3$ templates instead of one: pick 3 random images from the class as starting centers, assign each training image to its nearest center, recompute each center as the mean of the images assigned to it, and repeat that assign-and-recompute cycle 10 times. Now you have 30 templates. Classify by finding the nearest of all 30 and taking its digit label. Does accuracy improve? Which digits benefit most from having multiple templates?

4. Build the **nearest-neighbor classifier** we described: to classify a test image, find the single most similar training image and use its label. What accuracy does it get? It should be higher than template matching. But time how long it takes to classify the full test set versus our 10-template model. What's the tradeoff?

5. Implement a **train/test accuracy comparison**. Compute accuracy on the training set and the test set separately for both the template matcher and your nearest-neighbor classifier from exercise 4. For template matching, the gap should be small (the model is too simple to memorize). What does the gap look like for nearest neighbor? Which model overfits more, and why?

6. We showed that the top half of the image carries slightly more information than the bottom half. Go further: write code that evaluates accuracy using only a single row of pixels (28 features) at each of the 28 possible row positions. Plot accuracy vs row position. Which rows carry the most information? The result tells you where in the image the distinguishing features actually live.

2 Learning to Learn

In Chapter 1, we built a template matcher that classified handwritten digits with about 81% accuracy. It worked by averaging all training images of each digit and comparing new images to those averages. We got an approximation of the target function, but there was nothing to improve. The templates were just averages: you compute them once, and that's your model.

The template matcher performs the task, but it isn't learning. It compresses 60,000 training images into ten blurry averages and throws away everything else. There are no adjustable parts, no way to measure a mistake and correct for it. Remember what learning looks like: you try something, find out how wrong you were, adjust, and repeat.

We need a model that does the same thing. That's what this chapter builds. The ideas here (gradient descent, backpropagation, the training loop) are not specific to neural networks. They are the engine behind almost every model trained today, from the simplest logistic regression to the largest language models.

2.1 The Optimization Loop

Chapter 1 ended with two things: a model that makes predictions (g) and a loss function that measures how wrong those predictions are. For classification, we used accuracy. For regression, we introduced MSE:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(g(x_i) - y_i)^2$$

We said the goal of learning is to find the g that makes the loss as small as possible. But we never said *how*. The template matcher didn't minimize anything. It computed averages, and the averages were what they were.

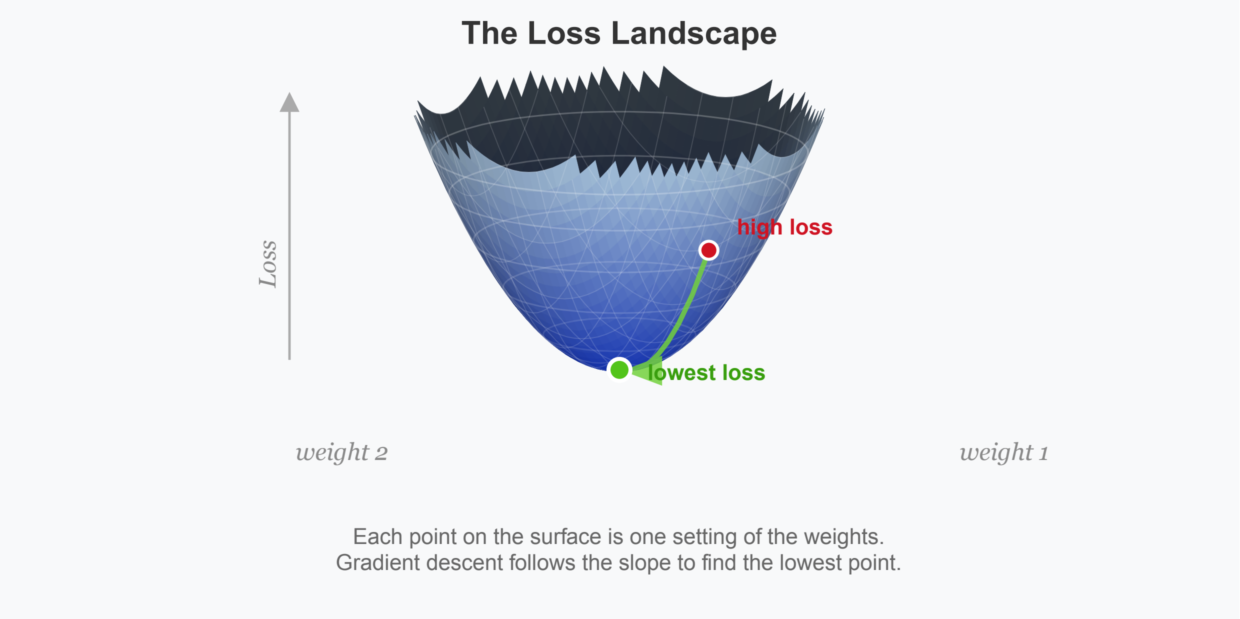
Now suppose our model has adjustable parameters: numbers we can change that affect the model's predictions. Call them **weights**. For example, instead of averaging pixels, we could try a model as simple as a line:

$$g(x) = wx + b$$

Two weights: w (the slope) and b (the intercept). Change w and the line tilts. Change b and it shifts up or down. Different values of w and b produce different predictions, which produce different losses. The learning problem becomes: find the weights that minimize the loss.

Say we have three data points: $(-1, 1)$, $(0, 3)$, $(1, 5)$. If we set $w = 1, b = 0$, our model predicts $-1, 0, 1$, and the MSE is $\frac{1}{3}[(-1-1)^2 + (0-3)^2 + (1-5)^2] = 9.67$. If we try $w = 2, b = 3$, the predictions are $1, 3, 5$, and the MSE is 0. That second guess nailed it.

Now picture trying every possible combination of w and b. Each combination gives one number: the loss. Plot w on one axis, b on the other, and the loss as height. You get a surface, a landscape where high terrain means bad predictions and low valleys mean good ones. Somewhere in that landscape is the point where $w = 2, b = 3$, sitting at the bottom of a valley at height zero.



The Loss Landscape

Each point on the surface is one setting of the weights.
Gradient descent follows the slope to find the lowest point.

Figure 2.1: Actual Loss

This is an **optimization** problem: find the values of the weights that make the loss as small as possible. With two weights, you can visualize the whole surface. In practice, models have millions of weights and you can't see the landscape at all. But you can check the slope where you're standing and figure out which direction is downhill. Take a step that way. Check again. Repeat.

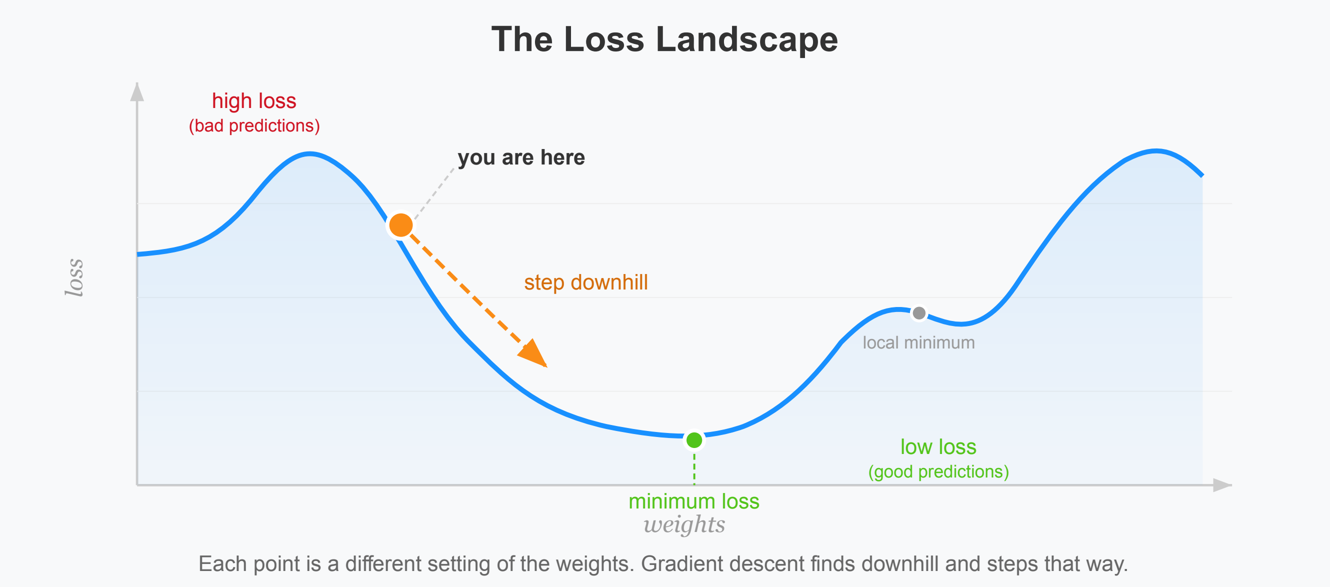


Figure 2.2: The Loss Landscape

The process has three parts that repeat in a loop. First, make predictions with the current weights (the **forward pass**). Second, measure how wrong the predictions are (the **loss**). Third, adjust the weights to reduce the loss (the **update**). Every machine learning model you'll encounter in this book follows this loop. The models differ in architecture and the loss functions differ in what they measure, but the loop is always the same.

Before we can build this loop, we need to solve a specific problem: standing at some point in the loss landscape, how do we figure out which direction is downhill?

2.2 Gradient Descent

Start with the simplest case. You have one weight w, and the loss is $L(w) = (w-3)^2$. This is a parabola with its minimum at $w = 3$. If you could plot it, you'd see the bowl shape immediately and walk to the bottom. But in real models, you have millions of weights and can't visualize the surface. You need a method that works locally: check the slope where you're standing and step downhill.

The slope of a function at a point is its **derivative**. For our parabola, $\frac{dL}{dw} = 2(w-3)$. A positive derivative means the function is increasing (you should move left, toward smaller w). A negative derivative means it's decreasing (move right). Zero means you might be at the bottom.

The update rule follows: move w in the direction opposite to the derivative, by some step size η called the **learning rate**.

$$w_{\text{new}} = w - \eta \frac{dL}{dw}$$

Let's trace this with actual numbers. Start at $w = 0$, with a learning rate $\eta = 0.1$:

$$\left.\frac{dL}{dw}\right|_{w=0} = 2(0-3) = -6 \quad \Rightarrow \quad w = 0 - 0.1 \times (-6) = 0.6$$

The derivative is -6 (steep downhill slope to the right), so we take a big step right to $w = 0.6$. Next step:

$$\left.\frac{dL}{dw}\right|_{w=0.6} = 2(0.6-3) = -4.8 \quad \Rightarrow \quad w = 0.6 - 0.1 \times (-4.8) = 1.08$$

Still moving right, but the step is smaller because the slope is less steep. After a few more steps: $w = 1.46$, then 1.77, then 2.02, each step smaller as we approach the minimum. After 20 steps, $w \approx 2.98$, nearly at the bottom. The loss went from $L(0) = 9$ down to $L(2.98) = 0.0004$.

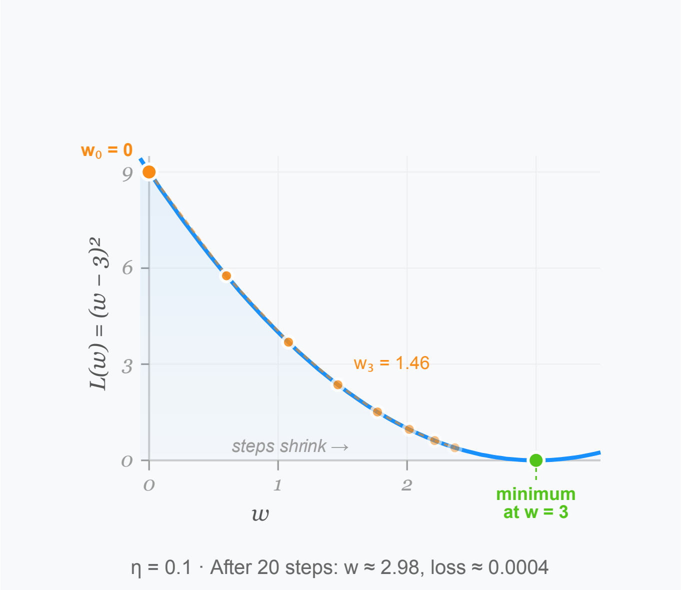


Figure 2.3: Gradient Descent Steps on a Parabola

Notice that the steps were large when the slope was steep (far from the minimum) and small when the slope was gentle (near the minimum). This is automatic and it's actually good for us, we don't want to overshoot and miss the minimum.

The learning rate η controls the overall scale. If we'd used $\eta = 0.01$, each step would be ten times smaller and we'd need 200 steps instead of 20. If we'd used $\eta = 1.0$, the first step would be $w = 0 - 1.0 \times (-6) = 6$, overshooting past the minimum to $w = 6$, and the next step would bounce back to $w = 0$, oscillating forever without converging. Too large and we overshoot. Too small and we crawl. Picking a good learning rate is a real practical challenge of training, and it's often another thing for us to try several values and see what sticks.

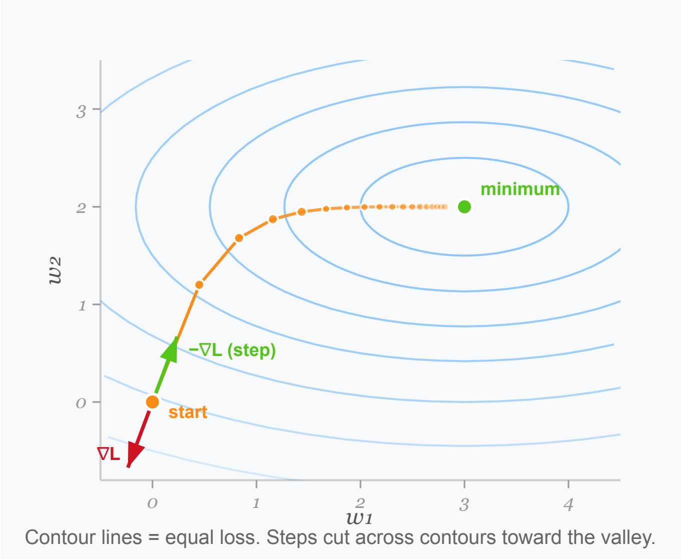
Now extending this to multiple weights: a real model has thousands or millions of weights: $w_1, w_2, \ldots, w_n$. The loss $L(w_1, w_2, \ldots, w_n)$ is a function of all of them. Instead of a single derivative, we compute a **gradient**: a vector of partial derivatives, one per weight.

$$\nabla L = \left[\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_n} \right]$$

Each partial derivative $\frac{\partial L}{\partial w_i}$ answers: "if I nudge just this one weight, holding all others fixed, how does the loss change?" The gradient vector points in the direction of steepest *increase*. We want to go the opposite way, so we subtract it:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L$$

With two weights, the loss surface is a 3D landscape (two weight axes plus the height). Looking down from above, you'd see contour lines, like a topographic map. The gradient at any point is perpendicular to the contour lines, pointing uphill. We step in the opposite direction, cutting across contours toward the valley.



Contour lines = equal loss. Steps cut across contours toward the valley.

Figure 2.4: Gradient Descent on a Contour Map

This is **gradient descent** (Cauchy 1847). Compute the gradient, then take a step opposite to it, and repeat. Each step reduces the loss (if the learning rate isn't too large), and over many steps the weights converge toward values that make the model's predictions good.

There's a practical issue: computing the loss over the entire training set (all 56,000 images, for instance) for every single step is computationally expensive. Instead, we can compute the loss and gradient on a small random subset called a **mini-batch** (typically 32 to 256 examples). The gradient from a mini-batch is noisier than the true gradient, but it's not biased in any direction.

The reason is straightforward: the full gradient is the average of per-example gradients across all 56,000 images; a random mini-batch of 64 is a random sample from those 56,000, and the average of a random sample is an unbiased estimator of the population average:

$$E[\nabla L_{\text{batch}}] = \nabla L_{\text{full}}$$

On any single batch the gradient might be off, but across batches it's exactly right on average. We can take many more steps in the same time, and the noise turns out to help: it prevents the optimizer from settling into sharp, narrow valleys that happen to fit the training data but don't generalize. This variant is called **stochastic gradient descent** (SGD), and it's what virtually every modern model uses (with some variations).

2.2.1 What about getting stuck?

The loss landscape has valleys, ridges, and plateaus. There might be multiple valleys (multiple settings of the weights that give low loss), and there's no guarantee gradient descent finds the deepest one. In one dimension, this is a real worry. A loss curve with two dips can trap you in the shallow one: you walk downhill, reach the bottom, and stop. The gradient is zero, so you have no direction to move. You're stuck in a **local minimum**, even though a deeper minimum exists somewhere else.

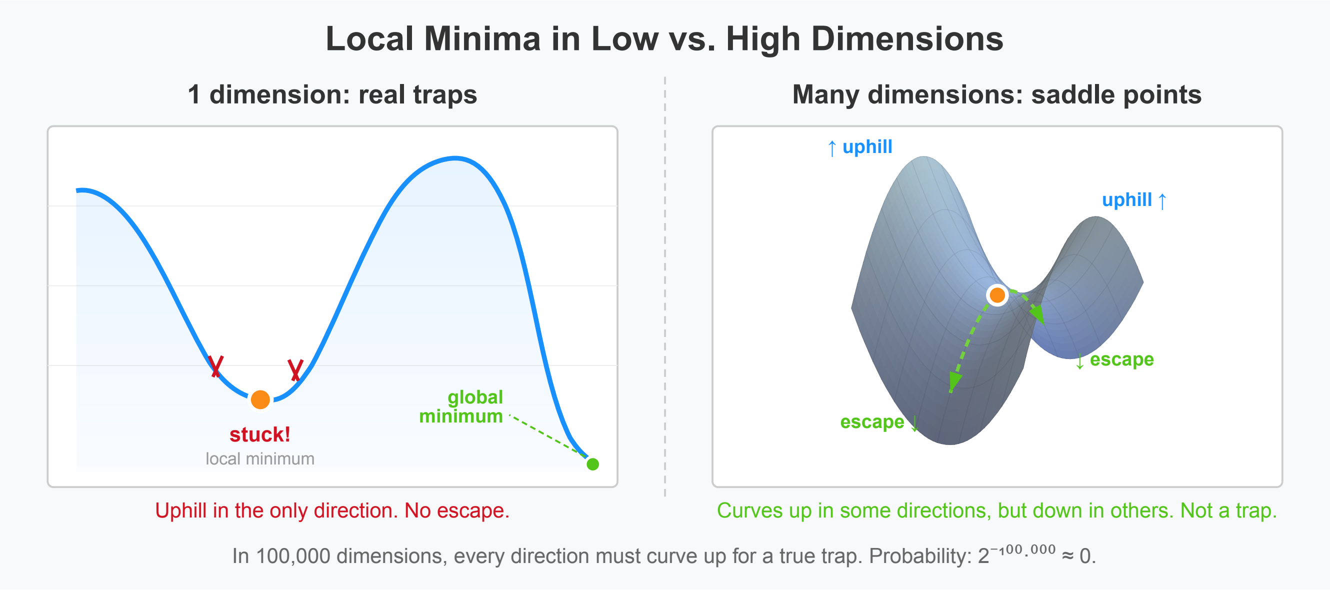

Local Minima in Low vs. High Dimensions
1 dimension: real traps
stuck!
local minimum
global minimum
Uphill in the only direction. No escape.
Many dimensions: saddle points
↑ uphill
uphill ↑
↓ escape
escape ↓
Curves up in some directions, but down in others. Not a trap.
In 100,000 dimensions, every direction must curve up for a true trap. Probability: 2⁻¹⁰⁰·⁰⁰⁰ ≈ 0.

Figure 2.5: Local Minima in Low vs. High Dimensions

In high dimensions, something that helps us happens: local minima are **really** rare.

To understand why, think about what a local minimum requires. At a critical point (where the gradient is zero), the loss surface curves in every direction. A local minimum needs the surface to curve *upward* in every single direction; it's a bowl. A **saddle point** curves up in some directions and down in others; it's shaped like a mountain pass, or a horse saddle.

In one dimension, a critical point has one direction. It either curves up (minimum) or down (maximum). Roughly a coin flip. In two dimensions, both directions must curve up for a minimum: two coin flips, both heads. The probability drops to 25%. In general, for a critical point in n dimensions to be a local minimum, all n directions must curve upward. The probability of this is roughly:

$$P(\text{local minimum}) \approx \frac{1}{2^n}$$

A simple neural network for MNIST has around 100,000 weights: 100,000 directions. The chance that all of them curve upward simultaneously is $2^{-100,000}$, a number so small that picking a specific atom out of all atoms in the universe is trillions of trillions of times more likely. Almost every critical point in the loss landscape is a saddle point, not a local minimum (Dauphin et al. 2014).

Dimensions	Directions that must curve up	Probability of local min
1	1	~50%
2	2	~25%
10	10	~0.1%
100	100	$2^{-100} \approx 10^{-30}$
100,000	100,000	effectively 0

And saddle points aren't traps. They have at least one direction that curves downward: an escape route. Gradient descent follows that escape, and so does the stochastic version (SGD).

What about the astronomically rare local minima that *do* exist? They tend to be nearly as good as the global minimum (Choromanska et al. 2015). The intuition: for a critical point to have both high loss *and* all directions curving upward, the landscape must be unusually pathological. The higher the loss, the more directions are available to curve downward, making saddle points even more likely. Local minima cluster near the bottom of the loss landscape, where they're hardly worse than the global minimum. Getting "stuck" in one of these is less like falling into a pit and more like stopping on one of many nearly identical valley floors.

This is why gradient descent works in practice despite the loss landscape having no obvious structure. The geometry of high-dimensional space works in our favor: the higher the dimension, the fewer traps exist, and the ones that remain are shallow.

We now know how to find the best weights models: compute gradients, step downhill, repeat. But so far our model $g(x) = wx + b$ outputs a raw number; it could be 3.7, or −12, or a million. That's fine for regression, where you're predicting a quantity. For classification, we need something different: an output we can read as "how confident is the model that this is a 7?" We need a way to squeeze that raw number into a probability.

2.3 The Perceptron

In 1958, Frank Rosenblatt introduced the perceptron (Rosenblatt 1958): a single unit that takes a vector of numbers as input and produces a single number as output.

What it computes is: take the input vector $x = [x_1, x_2, \dots, x_d]$ (for MNIST, that's 784 pixel values); multiply each input by a corresponding weight, add them all up, and add a bias term:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b = \mathbf{w} \cdot \mathbf{x} + b$$

This should look familiar: it's a dot product (the same operation from Euclidean distance and template matching in Chapter 1) plus a bias. The weights **w** and bias b are the learnable parameters, the numbers that gradient descent will adjust.

The result z can be any real number. To turn it into a prediction, we pass it through an **activation function** σ:

$$\hat{y} = \sigma(z) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

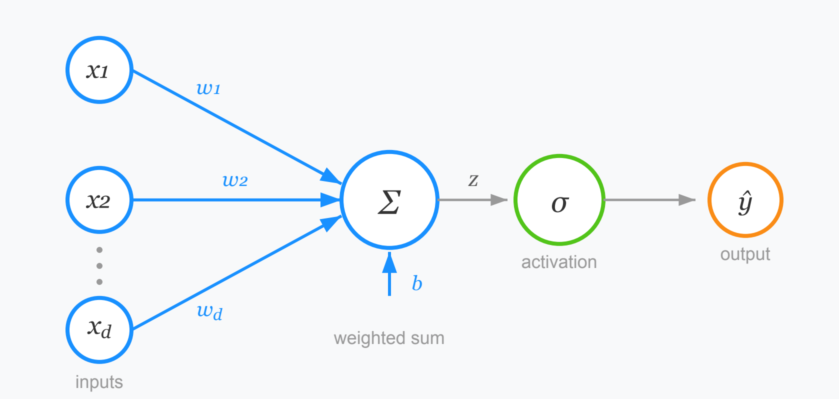


Figure 2.6: The Perceptron

For binary classification (yes or no, dog or cat), the classic choice is the **sigmoid function**:

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

This squashes any real number into the range $(0, 1)$. Large positive values of z get mapped close to 1. Large negative values get mapped close to 0. The output can be interpreted as a probability: "the model is 87% confident this is a dog."

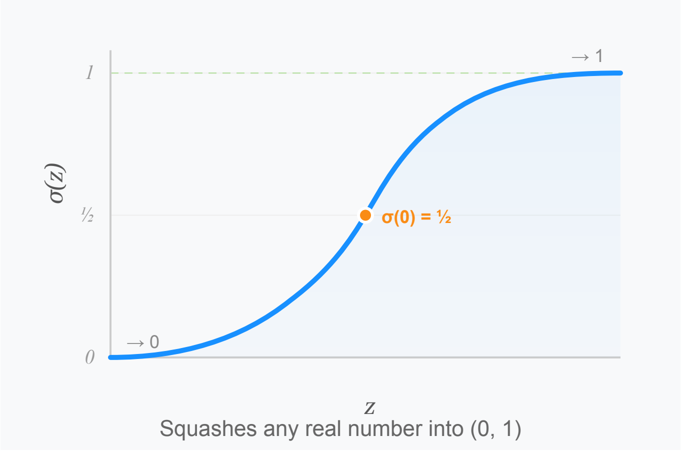


Figure 2.7: The Sigmoid Function

Now think geometrically about what a perceptron does. If the input is two-dimensional, the equation $w_1x_1 + w_2x_2 + b = 0$ defines a line in the plane. Everything on one side of the line (where the weighted sum is positive) gets classified as 1. Everything on the other side gets classified as 0. The weights control the orientation of the line. Changing w_1 and w_2 tilts it; changing b slides it sideways. Gradient descent adjusts these values until the line best separates the two classes.

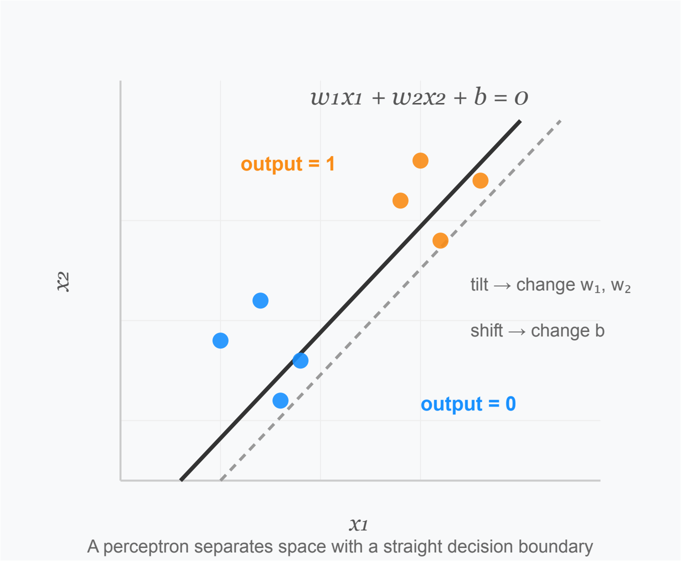


Figure 2.8: The Perceptron as a Line

In higher dimensions, the line becomes a **hyperplane**: a flat boundary that divides the space in half. With 784-dimensional pixel inputs, the perceptron draws a 783-dimensional hyperplane through pixel space. Everything on one side is “this digit,” everything on the other side is “not this digit.”

How well does this work for handwritten digits? If you train 10 separate perceptrons (one per digit class, each deciding “is this the digit x or not”), you get a **linear classifier**. Each perceptron draws its own hyperplane, and we can say that the prediction is whichever perceptron scores highest. Train it with gradient descent on MNIST and you can get about 92% accuracy. That's a meaningful jump from the template matcher's 81%, and it makes sense: the template matcher used fixed averages, while the linear classifier *optimized* its boundaries. The weights aren't just averages of examples; they're the boundaries that gradient descent found to best separate the classes, it captures a lot more nuance.

But 92% is still a ceiling. Since the perceptron computes $z = \mathbf{w} \cdot \mathbf{x} + b$ and classifies based on the sign of z, its decision boundary is the set of points where $\mathbf{w} \cdot \mathbf{x} + b = 0$ — an $(n-1)$-dimensional hyperplane in n-dimensional space. It can only classify data that is **linearly separable**: two sets of points X_0 and X_1 in $\mathbb{R}^n$ are linearly separable if there exist $w_1, w_2, \ldots, w_n$ and k such that every point $\mathbf{x} \in X_0$ satisfies $\sum_{i=1}^{n} w_i x_i > k$ and every point $\mathbf{x} \in X_1$ satisfies $\sum_{i=1}^{n} w_i x_i < k$. If your data can be split by a flat boundary, a perceptron will find it. If it can't, no amount of training will help.

Look at the confusion matrix and you'll see the same problem as Chapter 1: 4s confused with 9s, 3s confused with 5s and 8s; the geometry tells you why it can't capture harder patterns. A 4 and a 9 have a very similar vertical stroke on the right side; in pixel space, these digits overlap: there's no single flat cut through 784-dimensional space that cleanly separates all 4s from all 9s. The hyperplane does the best it can, but the best straight boundary through a curved overlap is still not enough.

Let's think of a simpler problem. The logical disjunction or exclusive or (XOR) takes two binary inputs and returns 1 when exactly one is 1 (given the propositions p and q, p or q assumes the value false if and only if p and q are both false):

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Now the question: can we represent XOR with the perceptron model? Plot these four points: the 1s sit at opposite corners of the square. The representation we need is one that successfully separates `true` (1) and `false` (0).

Since this is a two-dimensional space, the perceptron representation would be a line; we can prove directly that no such line exists. For the perceptron to classify XOR correctly, we need $\sigma(z) > 0.5$ for the 1s and $\sigma(z) < 0.5$ for the 0s, which means $z = w_1 x_1 + w_2 x_2 + b$ must be positive for the 1s and negative for the 0s. Writing out the four constraints:

- $(0,0) \to 0$: $b < 0$
- $(0,1) \to 1$: $w_2 + b > 0$
- $(1,0) \to 1$: $w_1 + b > 0$
- $(1,1) \to 0$: $w_1 + w_2 + b < 0$

Adding the second and third inequalities gives $w_1 + w_2 + 2b > 0$, but the fourth requires $w_1 + w_2 + b < 0$. Subtracting the fourth from the sum: $b > 0$. This contradicts the first constraint. No values of w_1, w_2, b satisfy all four conditions simultaneously.

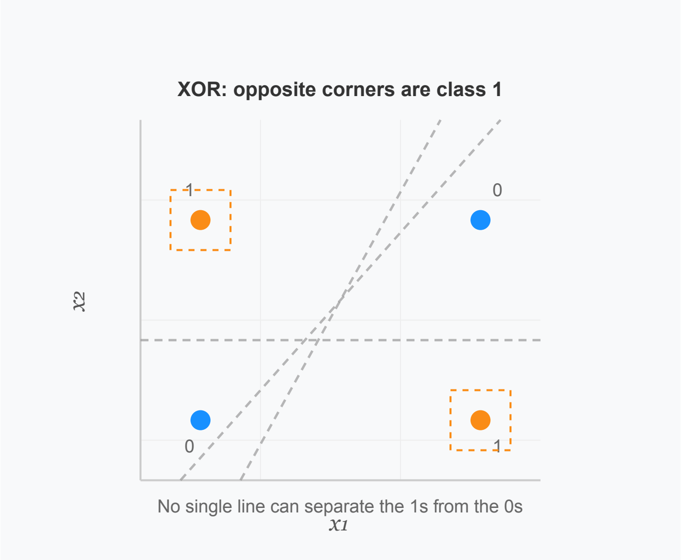


Figure 2.9: The XOR Problem: No Line Can Separate Opposite Corners

You can try to draw a single line that separates the 1 from the 0s as much as you want; it isn't possible. No matter how you tilt or slide the line, at least one point ends up on the wrong side.

The data isn't linearly separable: no flat boundary can sort it correctly. Digits aren't as clean as XOR's four points, but the same principle applies. Some regions of pixel space contain a mix of 4s and 9s that no hyperplane can untangle.

A perceptron can't solve XOR, and it can't cleanly separate 4s from 9s. One straight boundary isn't enough. The natural next question: what if we combine multiple perceptrons, each drawing its own boundary? Could several lines working together separate regions that no single line can?

2.4 From Perceptions to Neural Networks

We can start combining them by considering what happens if we skip the activation function entirely. The perceptron computes $z = \mathbf{w} \cdot \mathbf{x} + b$, a linear function of the input, now let's stack two of these: The first layer produces $z_1 = \mathbf{w}_1 \cdot \mathbf{x} + b_1$. The second layer takes that as input:

$$z_2 = \mathbf{w}_2 \cdot z_1 + b_2 = \mathbf{w}_2 \cdot (\mathbf{w}_1 \cdot \mathbf{x} + b_1) + b_2 = (\mathbf{w}_2 \mathbf{w}_1) \cdot \mathbf{x} + (\mathbf{w}_2 b_1 + b_2)$$

The result is still a linear function of $\mathbf{x}$, the two layers are equivalent to just one. You could stack a hundred linear layers and get the same thing: a single linear transformation. No matter how many perceptrons you combine, a linear function of a linear function is just another linear function. A hundred hyperplanes that all combine linearly just produce one hyperplane; you never escape the limits of a straight boundary.

What if we inserted something nonlinear between the layers: a function that bends the output before passing it to the next layer? Then the composition wouldn't collapse, because a nonlinear function of a linear function isn't linear anymore. Each layer could bend and reshape the space in ways that a linear function can't.

Think about what happens geometrically: without activation functions, each layer can only rotate, scale, and shift the data, operations that preserve straight lines. A square grid of points stays a (possibly rotated) grid after a linear transformation, but add a nonlinear activation between layers, and the layer can fold the space; crease it along a boundary so that what was a straight line becomes bent. Stack enough of these folds, and you can carve the space into a lot more complex regions.

Actually, in 1989, George Cybenko (Cybenko 1989) proved this. A single layer of perceptrons with a nonlinear activation, feeding into one more perceptron, can approximate any continuous function to any desired accuracy, given enough perceptrons in that layer. This is the **universal approximation theorem**: the architecture doesn't limit *what* the model can represent, only how efficiently it does so. In principle, one layer of enough perceptrons can learn anything. In practice, "enough" might mean an absurd number, but this doesn't even take stacking more layers into account.

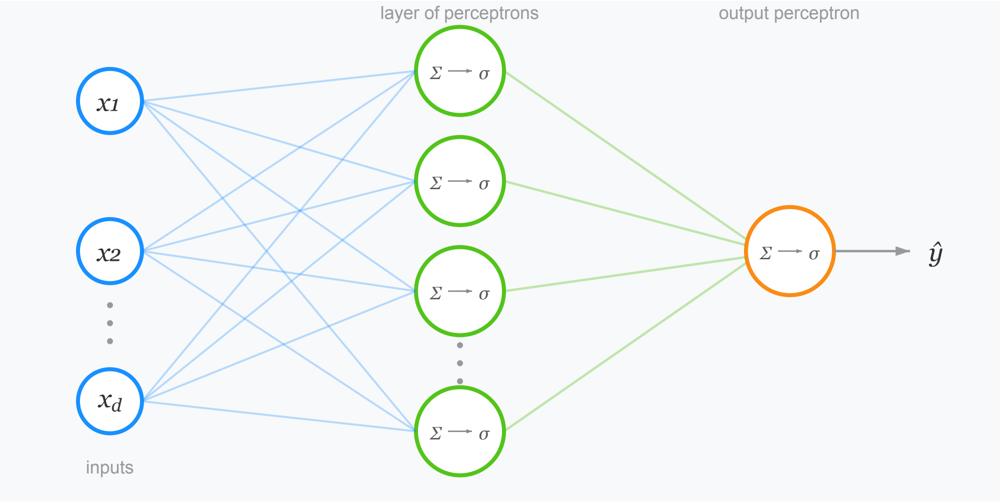


Figure 2.10: Perceptrons Combined

This structure, layers of perceptrons feeding into each other with nonlinear activations between them, is a **neural network**. The name comes from a loose analogy to biology: the brain is made up of billions of interconnected cells called neurons. Each neuron receives electrical signals from other neurons, and if the combined input is strong enough, it fires and sends its own signal onward. The perceptron mirrors this at a very abstract level: it takes weighted inputs, sums them, and "activates" if the result crosses a threshold. The analogy shouldn't be taken too far, but the core idea, many simple units wired together so that each one's output becomes another's input, was enough to inspire the field. The terminology stuck: each perceptron in a network is called a **neuron**, a layer of neurons between the input and the output is a **hidden layer** (hidden because you don't observe its values directly), and the full stack is a **neural network**.

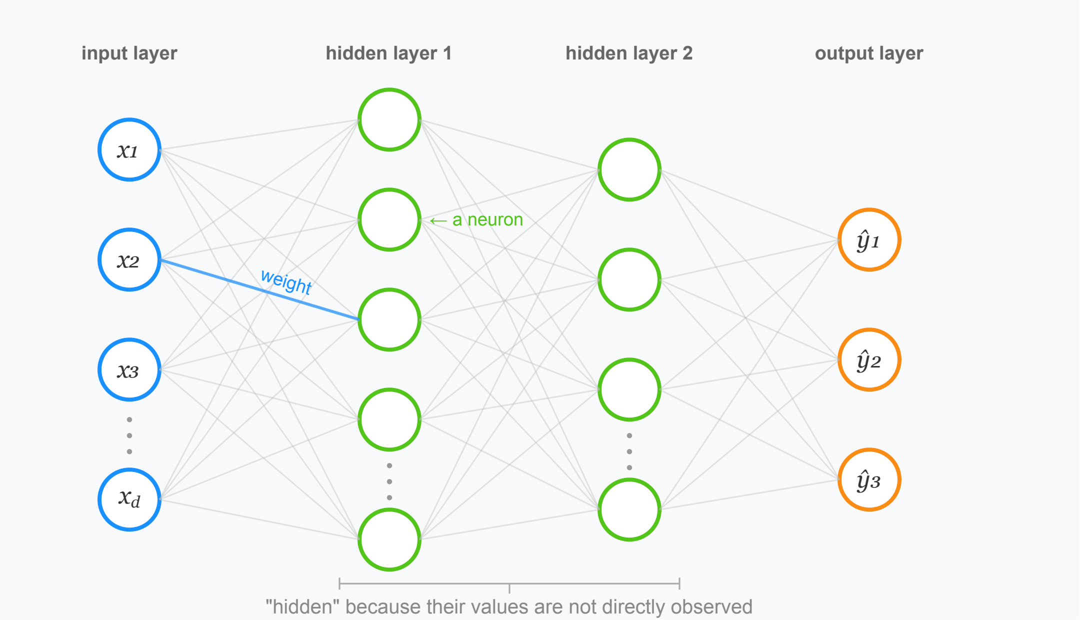


Figure 2.11: Neural Network

The **width** of a layer is the number of neurons in it. The **depth** of a network is how many layers it has. A network with one hidden layer of 128 neurons and an output layer of 10 is a 2-layer network (we don't count the input layer, since it just passes data through). A network with many hidden layers is called a **deep network**, and training such networks is called **deep learning**. The total number of weights is what determines the network's size: a hidden layer connecting 784 inputs to 128 neurons has $784 \times 128 + 128 = 100{,}480$ weights (plus 128 biases, one per neuron). A network with more weights can represent more complex functions (but also needs more data to learn the representation well).

One neuron might not help much, but 128 neurons in a hidden layer, each folding along a different hyperplane, can origami the space into something very different. After enough folds, the XOR points that no single line could separate end up neatly on one side of a new, simple boundary.

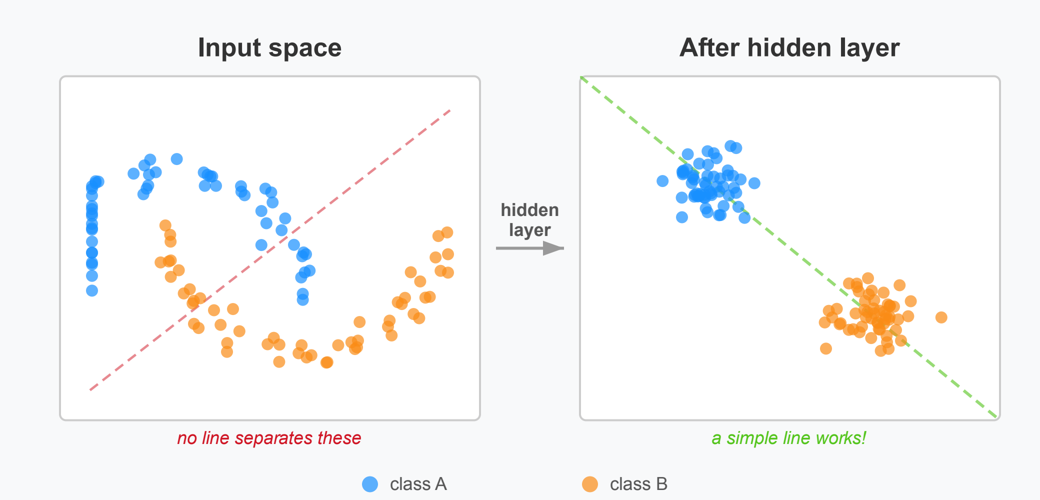


Figure 2.12: Folding the Space: Before and After the Hidden Layer

The **sigmoid** we already saw is one choice of activation:

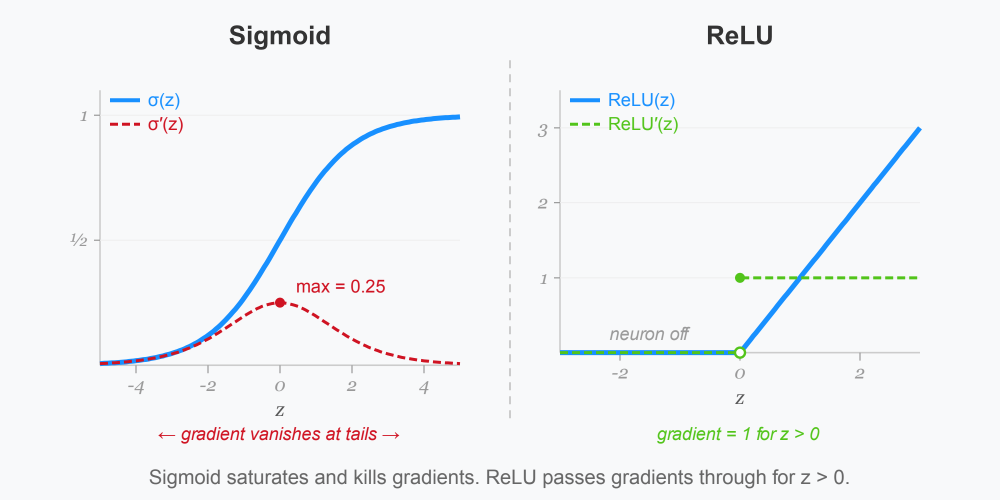
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

It was the default for decades, but it has a problem in deep networks. Look at the sigmoid curve: when z is very large or very small, the curve is nearly flat. Flat means the gradient is close to zero. To update an early layer's weights, the gradient signal has to travel back through every layer between it and the output, and at each layer it gets multiplied by the local gradient. If that local gradient is small (because the sigmoid is in its flat region), the signal shrinks. The derivative of sigmoid is $\sigma(z)(1 - \sigma(z))$, which peaks at just 0.25 and drops rapidly in both directions. Multiply several of these together and the gradient becomes exponentially small by the time it reaches the first layers. This is the **vanishing gradient** problem: early layers stop learning because the gradient that reaches them is too small to make meaningful updates.

The modern default is **ReLU** (Rectified Linear Unit) (Nair and Hinton 2010):

$$\text{ReLU}(z) = \max(0, z)$$

Negative inputs become 0, positive inputs pass through unchanged. The derivative is 1 for positive inputs (the gradient flows through without shrinking) and 0 for negative inputs (the neuron is "off"). ReLU avoids the vanishing gradient problem for positive inputs and is faster to compute than sigmoid. This is the "fold" from our geometric picture: ReLU creases the space along the hyperplane where the neuron's input is zero.



Sigmoid saturates and kills gradients. ReLU passes gradients through for z > 0.

Figure 2.13: Activation Functions: Sigmoid vs ReLU

There are other activation functions (tanh, Leaky ReLU, GELU), each with tradeoffs, but the intuition is the same for all of them: insert nonlinearity between layers so the network can warp the space rather than merely rotating it. For this chapter, we'll use ReLU for hidden layers and a different function (softmax) for the output.

2.5 The Forward Pass

Let's put this together for our digit classifier. The input is a 784-dimensional vector (a flattened 28×28 image). The output should be 10 numbers, one per digit class. We'll put one hidden layer of 128 neurons in between.

Each of the 128 hidden neurons receives all 784 inputs, multiplies them by its own weights, adds a bias, and applies ReLU. We can write all 128 neurons at once using matrix multiplication. Stack the 128 weight vectors into a matrix W_1 of shape 128×784, and the 128 biases into a vector b_1 of length 128:

$$h = \text{ReLU}(W_1 x + b_1)$$

The matrix multiply $W_1 x$ computes all 128 dot products simultaneously, adding b_1 shifts each one. ReLU zeros out the negatives, and the result h is a 128-dimensional vector: the hidden layer's output.

This vector h is a completely new representation of the input. The raw pixels are the same for every task: a photo of a digit or a photo of a face all have the same pixel structure, but h is task-specific. It's been shaped by training to highlight exactly the features that distinguish one digit from another. If the network learned that "there's a loop in the upper half" matters for distinguishing 8 from 1, some neurons in h will fire strongly when a loop is present and stay quiet when it's not. The network isn't working with raw pixels anymore; it's working with a learned vocabulary of visual features.

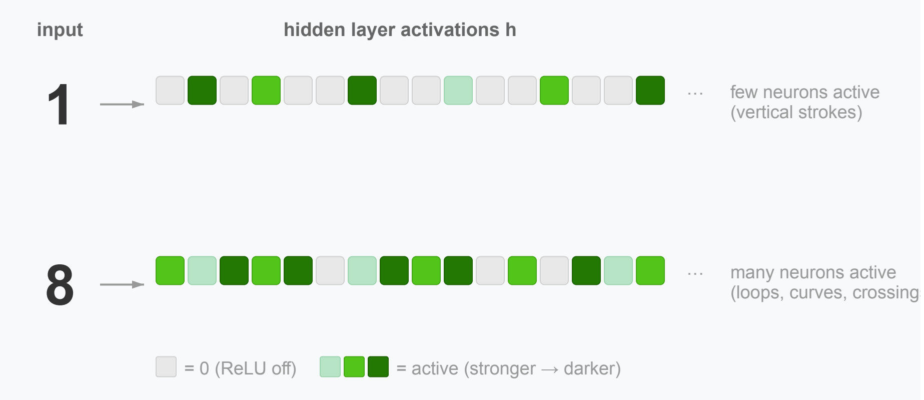


Figure 2.14: Hidden Layer Activations

The linear classifier couldn't separate 4s from 9s because those digits overlap in pixel space, but in the hidden space h, after the nonlinear transformation, the network can *pull them apart*. The 128-dimensional space that h lives in is a space the network designed (through training) specifically so that digits that were tangled in pixel coordinates end up in separate regions. We'll see this directly in the hands-on when we visualize the hidden representations.

The 10 output neurons each receive all 128 hidden values, multiply by their own weights, and add a bias. In matrix form, with W_2 of shape 10×128 and b_2 of length 10:

$$z = W_2 h + b_2$$

The result is 10 raw scores, one per digit class. These aren't probabilities yet (they can be any real number). To turn them into probabilities, we apply the **softmax** activation function:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{10} e^{z_j}}$$

Softmax exponentiates each score (making them all positive) and divides by the total (making them sum to 1). It's the natural choice for turning scores into probabilities: always positive, always sums to 1, and it preserves the ordering of the original scores (the highest score always gets the highest probability). The output is a probability distribution: "32% chance it's a 3, 61% chance it's an 8, …" The predicted class is the one with the highest probability.

One caveat: these numbers look like probabilities, and we'll treat them that way, but they're not necessarily well-calibrated: a model that outputs 0.9 for a digit might only be right 75% of the time when it says that. Neural networks tend to be overconfident; the relative ordering is reliable (higher scores mean the model is more sure), but the absolute numbers shouldn't be trusted as literal frequencies.

The entire computation from input to prediction is the **forward pass**:

$$x \xrightarrow{W_1, b_1} \text{ReLU}(W_1 x + b_1) = h \xrightarrow{W_2, b_2} \text{softmax}(W_2 h + b_2) = \hat{y}$$

The hidden layer transforms the input from a space where digits overlap into a space where they're separable. The output layer draws hyperplanes through the hidden space, which works now because the hidden layer already untangled the data. This is why the neural network beats the linear classifier. The linear classifier draws hyperplanes directly through pixel space, where the data is tangled; the neural network first untangles the data, then draws the hyperplanes.

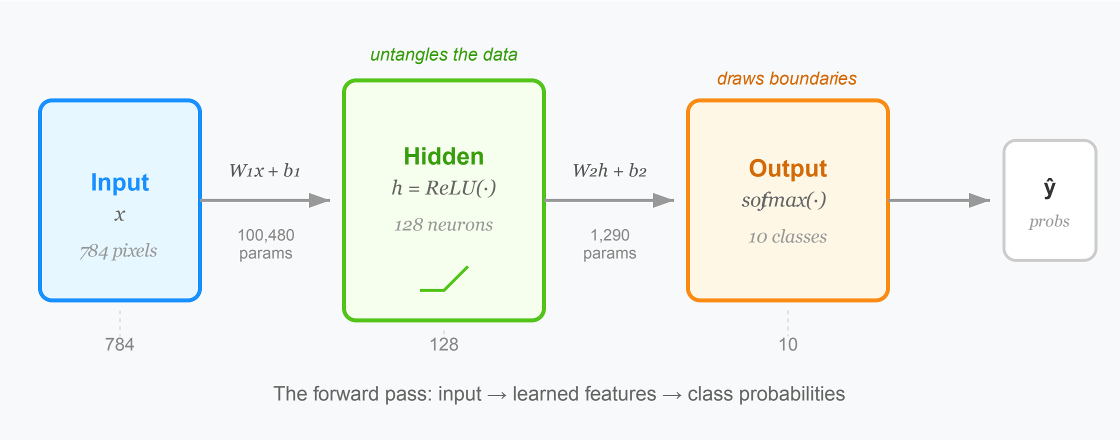


Figure 2.15: The Forward Pass

The forward pass is just matrix multiplies, additions, and element-wise nonlinear functions. What makes neural networks powerful is the ability to *adjust the weights* to make this forward pass produce better predictions. For that, we need to figure out how each weight contributes to the loss.

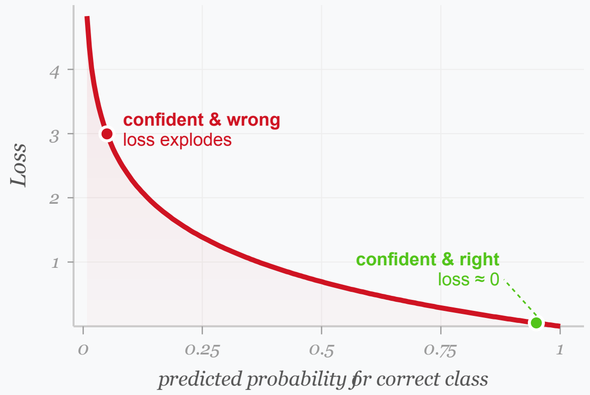
We need a loss function suited for classification into multiple categories. In Chapter 1, we introduced MSE for regression. MSE can technically be used for classification too, but it treats all wrong answers proportionally: a prediction of 0.4 for the correct class is "a little wrong" and gets a gentle nudge. For classification, we want the model to be punished sharply for assigning low probability to the right answer. **Cross-entropy loss** does this.

If the true label is digit k (represented as a one-hot vector y with a 1 at position k and 0s elsewhere), and the model outputs probabilities $\hat{y}$:

$$L = -\sum_{i=0}^{9} y_i \log(\hat{y}_i) = -\log(\hat{y}_k)$$

Since y is one-hot, only the term where $y_i = 1$ survives. The loss is the negative log of the probability assigned to the correct class. If the model is confident and right ($\hat{y}_k$ close to 1), $-\log(1) = 0$: no loss. If the model assigns low probability to the correct answer ($\hat{y}_k$ close to 0), $-\log(\hat{y}_k)$ shoots toward infinity.

There's another way to read this curve: think of $-\log(p)$ as measuring surprise. If the model assigns 95% to the correct class, it's not surprised when that class shows up: low loss. If it assigns 1%, it's very surprised: high loss. Cross-entropy is the model's average surprise across the training set. A perfect model is never surprised, but a bad model is constantly surprised. This is why cross-entropy is the standard for classification: it directly measures how far the model's beliefs are from reality.



$L = -\log(\hat{y}_k)$ · Low confidence in the right answer → severe penalty

Figure 2.16: Cross-Entropy Loss

The forward pass is just matrix multiplies, additions, and element-wise nonlinear functions. What makes neural networks powerful is the ability to adjust the weights to make this forward pass produce better predictions. We already know the recipe: compute the gradient of the loss with respect to every weight, then step downhill. But in a network with multiple layers, the loss doesn't depend on the first layer's weights directly; it depends on them through every layer that comes after. We need a way to trace that chain of dependencies backward.

2.6 Backpropagation

We have a loss L that depends on the output probabilities $\hat{y}$, which depend on W_2 and b_2, which depend on h, which depends on W_1 and b_1, which depend on x. It's clearly a chain: each quantity depends on the one before it.

To update W_1 by gradient descent, we need $\frac{\partial L}{\partial W_1}$: how does the loss change when we tweak the weights in the first layer? The loss doesn't directly involve W_1; it goes through h, then through z, then through $\hat{y}$. The **chain rule** from calculus tells us how to compose these dependencies:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h} \cdot \frac{\partial h}{\partial W_1}$$

Each term is a local derivative: how does one layer's output change with respect to its input? We compute these local derivatives layer by layer, from the output back toward the input. This is **backpropagation** (Rumelhart, Hinton, and Williams 1986): the algorithm that makes training deep networks practical.

The forward pass computed:

$$h = \text{ReLU}(W_1 x + b_1) \qquad z = W_2 h + b_2 \qquad \hat{y} = \text{softmax}(z) \qquad L = -\log(\hat{y}_k)$$

Backpropagation works in reverse.

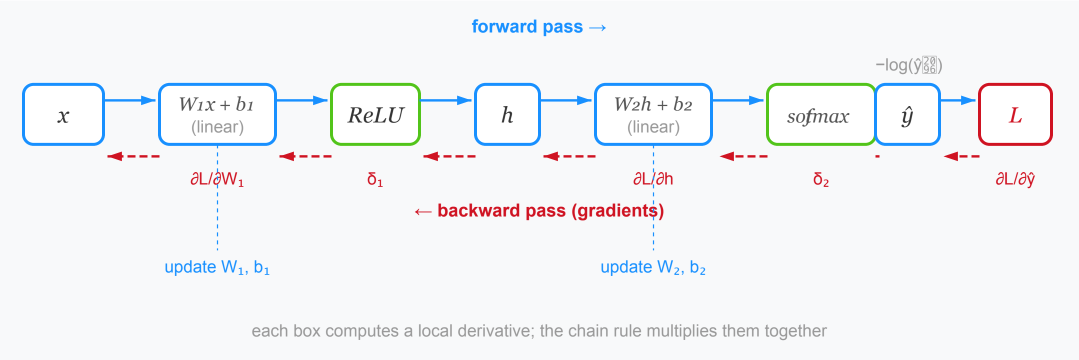


Figure 2.17: Backpropagation

Step 1: Loss -> softmax scores. The gradient of cross-entropy loss combined with softmax has a clean form:

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i$$

For the correct class k, this is $\hat{y}_k - 1$ (the model's probability minus the target 1). For all other classes, it's just $\hat{y}_i$ (the model's probability minus the target 0). The gradient is the difference between what the model predicted and what it should have predicted. That's not a coincidence: softmax and cross-entropy are mathematically designed to pair together, which is why the gradient reduces to something this clean. Call this $\delta_2 = \hat{y} - y$.

Step 2: Output layer -> hidden layer. The output scores were $z = W_2 h + b_2$. The gradients for the second layer's parameters are:

$$\frac{\partial L}{\partial W_2} = \delta_2 \cdot h^T \qquad \frac{\partial L}{\partial b_2} = \delta_2$$

To propagate the error back to the hidden layer:

$$\frac{\partial L}{\partial h} = W_2^T \cdot \delta_2$$

W_2^T transposes the weight matrix, reversing the connections: instead of mapping hidden neurons to output scores, it maps output errors back to hidden neurons. Each hidden neuron receives a weighted sum of how much each output was wrong, weighted by how strongly that neuron was connected to each output.

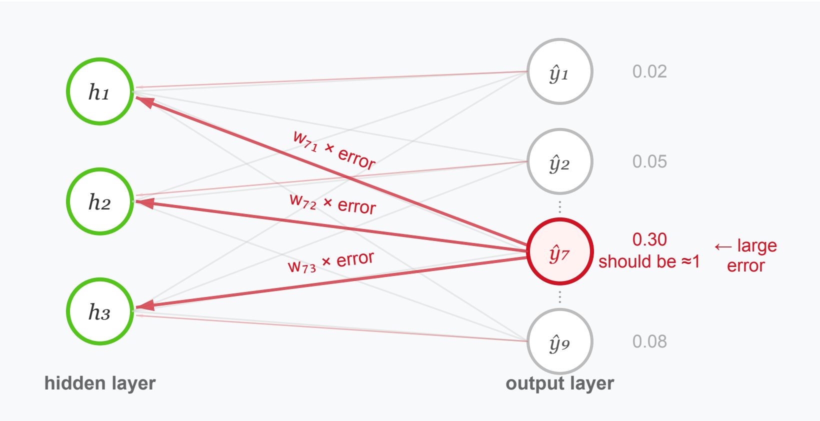


Figure 2.18: Reversing the Connections

Step 3: Hidden layer -> input layer. The hidden layer computed $h = \text{ReLU}(W_1 x + b_1)$. The ReLU derivative is 1 where the input was positive and 0 where it was negative. So the gradient passes unchanged for active neurons and is blocked for inactive ones:

$$\delta_1 = (W_2^T \cdot \delta_2) \odot \text{ReLU}'(W_1 x + b_1)$$

where $\odot$ is element-wise multiplication and ReLU$'$ is 1 for positive values, 0 otherwise. Then:

$$\frac{\partial L}{\partial W_1} = \delta_1 \cdot x^T \qquad \frac{\partial L}{\partial b_1} = \delta_1$$

That's the full backward pass. We now have gradients for every weight and bias in the network. Each gradient tells us: "nudging this parameter in this direction will reduce the loss by this much."

The name "backpropagation" just means applying the chain rule layer by layer from output to input. It isn't mathematically complex (the chain rule is first-year calculus) but with this algorithm you can efficiently reuse computations: each layer's gradient depends on the gradient from the layer above it, which you've already computed. The total computational cost is proportional to one forward pass, regardless of how many layers the network has.

For the intuition, imagine the loss is too high because the model predicted 30% for digit 7 when it should have predicted near 100%; backpropagation traces backward: which output weights made the 7-score too low? Which hidden neurons fed those weights? Which input weights controlled those hidden neurons? Every weight gets a tweak proportional to how much it contributed to the mistake. Weights that had nothing to do with the error (because their neurons were inactive, or they connected to irrelevant outputs) get small or zero gradients. The network doesn't change everywhere at once; it changes most where the mistake originated.

But one correction isn't enough. The first update fixes one batch's mistakes and might introduces new ones; the network needs to see the data again and again, each time refining the weights a little further.

2.7 The Training Loop

One pass through the entire training set is called an **epoch**. Most models need many epochs (10 to 100 or more) before the weights settle into good values. Within each epoch, we iterate over mini-batches, updating the weights after each batch.

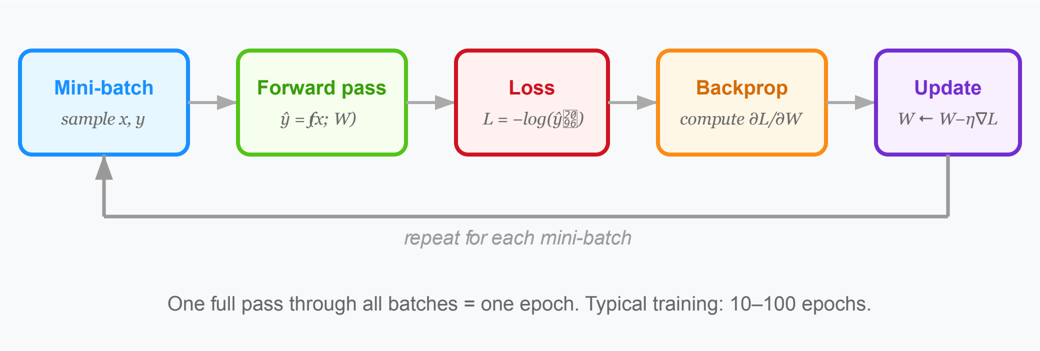

Mini-batch
sample x, y
Forward pass
ŷ = fx; W)
Loss
L = −log(ŷ2096)
Backprop
compute ∂L/∂W
Update
W ← W−η∇L
repeat for each mini-batch
One full pass through all batches = one epoch. Typical training: 10–100 epochs.

Figure 2.19: The Training Loop

The learning rate η is the most important **hyperparameter**: a setting you choose before training, as opposed to the weights, which are learned *during* training. Too large and the updates overshoot, causing the loss to bounce or diverge; too small and training takes forever.

Weight initialization is important too. If all weights start at 0, every neuron computes the same thing, every gradient is identical, and gradient descent will do nothing: all neurons stay identical forever. Instead, we initialize weights to small random values; the specifics of how to sample those values are the subject of careful research (Glorot and Bengio 2010), but the main thing to take away is: start random, not zero.

Each pass through the loop tweaks the decision boundary closer to the right shape:

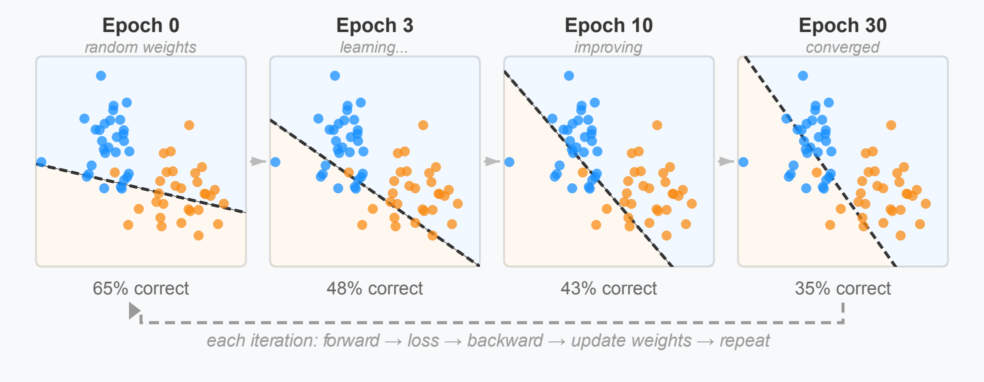


Figure 2.20: Loop Visualized

This loop is the same whether you're training a digit classifier with 100,000 weights or a language model with 100 billion. The architectures and loss functions might change, but the loop doesn't. Every model we build in the rest of this book (word embeddings in Chapter 3, transformers in Chapter 4) follows these exact mechanics. That's enough theory. Let's build one.

2.8 Hands-On: A Neural Network in NumPy

We've spent the chapter building up the theory piece by piece: gradient descent, backpropagation, activation functions, the forward pass. Now let's put all of it into code. We'll build two models from scratch, a linear classifier and a neural network, train them on the same data with the same loop, and see exactly where the hidden layer makes the difference.

Set up your project:

```
mkdir chapter2
cd chapter2
uv init
uv add numpy matplotlib scikit-learn
```

Create a file called `network.py`. We start the same way we did in Chapter 1, loading and splitting MNIST:

```
# network.py
from sklearn.datasets import fetch_openml
import numpy as np
import matplotlib.pyplot as plt

mnist = fetch_openml("mnist_784", version=1, as_frame=False, parser="auto")
X = mnist.data / 255.0
y = mnist.target.astype(int)

rng = np.random.default_rng(42)
```

```
indices = rng.permutation(len(X))
X, y = X[indices], y[indices]

split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

print(f"Training: {len(X_train)}, Test: {len(X_test)}")
```

One difference from Chapter 1: we divide pixel values by 255, scaling them from the range $[0, 255]$ to $[0, 1]$. Neural networks train better when inputs are small numbers. Large input values produce large dot products, which push sigmoid activations into their flat regions (where gradients vanish, as we discussed). Scaling to $[0, 1]$ keeps everything in a reasonable range. This is called **feature scaling**, and it's standard practice.

Next, we need to convert the integer labels (0 through 9) into one-hot vectors for the cross-entropy loss. The label "3" becomes `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]`:

```
# network.py
# ...existing code above

def one_hot(labels, num_classes=10):
    result = np.zeros((len(labels), num_classes))
    for i, label in enumerate(labels):
        result[i, label] = 1.0
    return result

y_train_oh = one_hot(y_train)
y_test_oh = one_hot(y_test)
```

`np.zeros` creates an array of all zeros. For each image, we set a single position to 1.0. The result is a matrix with 56,000 rows and 10 columns: each row is mostly zeros with one 1.

Now the activation functions. ReLU for hidden layers, softmax for the output:

```
# network.py
# ...existing code above

def relu(z):
    return np.maximum(0, z)

def relu_derivative(z):
    return (z > 0).astype(float)
```

`np.maximum(0, z)` compares each element to 0 and keeps the larger one. The derivative function uses a boolean trick: `z > 0` produces `True` where positive and `False` where not, and `.astype(float)` converts those to 1.0 and 0.0. This is the ReLU gate from the backpropagation section: gradient flows through where the neuron was active, and gets blocked where it wasn't.

Softmax needs a numerical trick:

```
# network.py
# ...existing code above

def softmax(z):
    shifted = z - z.max(axis=1, keepdims=True)
    exp_z = np.exp(shifted)
    return exp_z / exp_z.sum(axis=1, keepdims=True)
```

The formula says to exponentiate each score and divide by the total. But e^{z_i} overflows when z_i is large (say, 1000), so we'll subtract the maximum value before exponentiating. This doesn't change the result (the subtracted constant cancels between numerator and denominator) but keeps the numbers from exploding. The `axis=1` means "operate across columns within each row," so each image in a batch gets its own maximum. The `keepdims=True` preserves the shape for broadcasting, the same mechanism Chapter 1 used when we divided arrays element-wise.

And the cross-entropy loss, averaged over a batch:

```
# network.py
# ...existing code above

def cross_entropy_loss(y_hat, y_true):
    y_hat_clipped = np.clip(y_hat, 1e-12, 1.0)
    return -np.mean(np.sum(y_true * np.log(y_hat_clipped), axis=1))
```

`np.clip` ensures no value is exactly 0 before we take the log (since $\log(0)$ is negative infinity). The `y_true * np.log(...)` multiplies element-wise; since `y_true` is one-hot, this zeroes out every term except the correct class. `.sum(axis=1)` gives one loss per image, and `np.mean` averages over the batch.

Now let's first build the model without a hidden layer: input (784) straight to softmax (10). This is **the linear classifier.** from the theory. One weight matrix and one bias vector:

```
# network.py
# ...existing code above

W_lin = rng.normal(0, 0.01, size=(784, 10))
b_lin = np.zeros(10)
```

`rng.normal(0, 0.01, size=(784, 10))` draws 7,840 random numbers from a normal distribution with mean 0 and standard deviation 0.01. Small random values: enough to break symmetry, not so large that they push softmax into weird territory. The biases start at zero.

The training loop follows the same pattern from the theory: forward pass, compute loss, backward pass, update:

```
# network.py
# ...existing code above

epochs = 30
```

```
batch_size = 64
lr = 0.1

for epoch in range(epochs):
    perm = rng.permutation(len(X_train))
    X_shuffled = X_train[perm]
    y_shuffled = y_train_oh[perm]

    for start in range(0, len(X_train), batch_size):
        X_batch = X_shuffled[start:start + batch_size]
        y_batch = y_shuffled[start:start + batch_size]

        # Forward
        z = X_batch @ W_lin + b_lin
        y_hat = softmax(z)

        # Backward
        delta = (y_hat - y_batch) / len(X_batch)
        dW = X_batch.T @ delta
        db = delta.sum(axis=0)

        # Update
        W_lin -= lr * dW
        b_lin -= lr * db
```

Each epoch shuffles the training data (so batches are different every time), then iterates through 64-image mini-batches. The forward pass is one line: `X_batch @ W_lin + b_lin` computes all 10 scores for all 64 images at once, and `softmax` turns them into probabilities. The gradient `y_hat - y_batch` is the elegant cross-entropy + softmax derivative from the backpropagation section. `X_batch.T @ delta` multiplies the transposed inputs by the errors, giving us the gradient for each weight: how much did this pixel, connected to this output, contribute to the mistake?

```
# network.py
# ...existing code above

preds_lin = (X_test @ W_lin + b_lin).argmax(axis=1)
acc_lin = (preds_lin == y_test).sum() / len(y_test)
print(f"Linear classifier accuracy: {acc_lin:.1%}")
```

You should see about 92%. Better than the template matcher's 81%, but far from perfect.

Let's look at what this linear classifier actually learned. Each column of `W_lin` is a 784-dimensional weight vector for one digit class. We can reshape these into 28×28 images:

```
# network.py
# ...existing code above

fig, axes = plt.subplots(1, 10, figsize=(15, 2))
for digit, ax in enumerate(axes):
    ax.imshow(W_lin[:, digit].reshape(28, 28), cmap="RdBu_r", vmin=-0.5, vmax=0.5)
    ax.set_title(str(digit), fontsize=12)
    ax.axis("off")
```

```
plt.suptitle("What the linear classifier looks for", fontsize=14)
plt.tight_layout()
plt.savefig("linear_weights.png", dpi=150)
plt.show()
```

Blue regions are pixels whose brightness makes the model more confident in that digit; red regions are pixels that make it less confident. Compare these to the template matcher's blurry averages from Chapter 1. These are sharper: gradient descent didn't just average, it learned *which pixels discriminate*. But they're still global patterns over the full image. The linear model can't learn "there's a loop" or "two lines meet at an angle," because those are spatial relationships that require combining pixels nonlinearly, not just weighting individual ones.

Now we add a hidden layer and make it a **neural network**. The architecture becomes 784 -> 128 -> 10:

```
# network.py
# ...existing code above

def initialize_weights(rng):
    W1 = rng.normal(0, np.sqrt(2.0 / 784), size=(784, 128))
    b1 = np.zeros(128)
    W2 = rng.normal(0, np.sqrt(2.0 / 128), size=(128, 10))
    b2 = np.zeros(10)
    return W1, b1, W2, b2

W1, b1, W2, b2 = initialize_weights(rng)
print(f"Total parameters: {W1.size + b1.size + W2.size + b2.size}")
```

The weights use Xavier initialization (Glorot and Bengio 2010): random values scaled by $\sqrt{2/n_{\text{in}}}$, where n_{in} is the number of inputs to the layer. Each neuron sums n_{in} weighted inputs, so if the weights aren't scaled down, the output variance grows by a factor of n_{in} at every layer and activations explode before training starts. Setting the weight variance to $2/n_{\text{in}}$ cancels that growth (the 2 accounts for ReLU zeroing out half the outputs). The effect is that activations stay in a reasonable range from the first layer to the last. About 101,770 parameters, thirteen times more than the template matcher's 7,840 values, and every one is adjustable.

The forward pass feeds a batch through both layers:

```
# network.py
# ...existing code above

def forward(X, W1, b1, W2, b2):
    z1 = X @ W1 + b1            # (batch, 784) @ (784, 128) = (batch, 128)
    h = relu(z1)                 # (batch, 128)
    z2 = h @ W2 + b2            # (batch, 128) @ (128, 10) = (batch, 10)
    y_hat = softmax(z2)          # (batch, 10)
    return z1, h, z2, y_hat
```

We write XW_1 rather than W_1X because our images are stored as rows. The shapes in the comments let you trace the dimensions: 784 inputs become 128 hidden features become 10 class probabilities. The function returns intermediate values `z1` and `h` because backpropagation needs them.

The backward pass computes gradients for all four parameter groups. Each line maps to a step from the theory:

```
# network.py
# ...existing code above

def backward(X, z1, h, y_hat, y_true, W2):
    batch_size = X.shape[0]

    # Step 1: output error
    delta2 = (y_hat - y_true) / batch_size

    # Step 2: gradients for W2, b2
    dW2 = h.T @ delta2
    db2 = delta2.sum(axis=0)

    # Step 3: propagate error back through ReLU
    delta1 = (delta2 @ W2.T) * relu_derivative(z1)
    dW1 = X.T @ delta1
    db1 = delta1.sum(axis=0)

    return dW1, db1, dW2, db2
```

`delta2 = y_hat - y_true` is the output error. `h.T @ delta2` gives the gradient for W_2: how much each hidden neuron, connected to each output, contributed to the error. `delta2 @ W2.T` propagates the error backward through the weight matrix, reversing the connections. `* relu_derivative(z1)` masks out the neurons that were inactive during the forward pass; their gradient is zero because they didn't contribute. The `.sum(axis=0)` for bias gradients adds up contributions from every example in the batch.

```
# network.py
# ...existing code above

def update_weights(W1, b1, W2, b2, dW1, db1, dW2, db2, lr):
    W1 -= lr * dW1
    b1 -= lr * db1
    W2 -= lr * dW2
    b2 -= lr * db2
    return W1, b1, W2, b2
```

Now the training loop. Same structure as the linear classifier, but with the hidden layer in between:

```
# network.py
# ...existing code above

epochs = 30
batch_size = 64
lr = 0.1
losses = []

for epoch in range(epochs):
    perm = rng.permutation(len(X_train))
    X_shuffled = X_train[perm]
    y_shuffled = y_train_oh[perm]

    epoch_loss = 0.0
    num_batches = 0

    for start in range(0, len(X_train), batch_size):
        end = start + batch_size
        X_batch = X_shuffled[start:end]
        y_batch = y_shuffled[start:end]

        z1, h, z2, y_hat = forward(X_batch, W1, b1, W2, b2)
        loss = cross_entropy_loss(y_hat, y_batch)
        epoch_loss += loss
        num_batches += 1

        dW1, db1, dW2, db2 = backward(X_batch, z1, h, y_hat, y_batch, W2)
        W1, b1, W2, b2 = update_weights(W1, b1, W2, b2, dW1, db1, dW2, db2, lr)

    avg_loss = epoch_loss / num_batches
    losses.append(avg_loss)

    if (epoch + 1) % 5 == 0 or epoch == 0:
        print(f"Epoch {epoch+1:3d} | Loss: {avg_loss:.4f}")
```

Run this. The loss starts around 2.3, which is $-\log(1/10)$: the loss when the model assigns equal 10% probability to every class, essentially guessing randomly. Watch it drop as the network learns:

```
# network.py
# ...existing code above

plt.figure(figsize=(8, 4))
plt.plot(range(1, epochs + 1), losses)
plt.xlabel("Epoch")
plt.ylabel("Cross-Entropy Loss")
plt.title("Training loss over time")
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig("loss_curve.png", dpi=150)
plt.show()
```

The curve drops steeply in the first few epochs and then levels off: fast initial progress as the network learns the obvious patterns, then diminishing returns as it refines subtle ones.

Now the comparison:

```
# network.py
# ...existing code above

def predict(X, W1, b1, W2, b2):
    _, _, _, y_hat = forward(X, W1, b1, W2, b2)
    return y_hat.argmax(axis=1)
```

```
test_preds = predict(X_test, W1, b1, W2, b2)
accuracy = (test_preds == y_test).sum() / len(y_test)
print(f"\nLinear classifier: {acc_lin:.1%}")
print(f"Neural network:    {accuracy:.1%}")
```

You should see about 97% for the neural network versus 92% for the linear classifier. Same data, same loss function, same gradient descent. The only difference is the hidden layer.

Let's check for overfitting:

```
# network.py
# ...existing code above

train_preds = predict(X_train, W1, b1, W2, b2)
train_accuracy = (train_preds == y_train).sum() / len(y_train)
print(f"\nTraining accuracy: {train_accuracy:.1%}")
print(f"Test accuracy:     {accuracy:.1%}")
print(f"Gap:               {train_accuracy - accuracy:.1%}")
```

Training accuracy should be higher (close to 100% vs about 98% on test). A gap of about 2% means the model generalized well, though it has memorized the training data almost perfectly.

A 2% gap is small here, but on harder problems or smaller datasets it can be much larger. The train/test split tells you overfitting is happening, but it doesn't fix it. The standard fix is **regularization**: adding a penalty to the loss that discourages the model from fitting noise. The most common version adds the sum of squared weights to the loss, which pushes the network toward smaller, smoother boundaries that are less likely to contort around individual training examples. We won't implement it in this chapter, but it's worth knowing that diagnosing overfitting and treating it are different things.

What happens without the nonlinearity? We claimed in the theory that stacking linear layers without activation functions collapses into a single linear transformation. Let's see if that's true. We'll train a network with the same 784 -> 128 -> 10 architecture, but replace ReLU with the identity function (no activation at all):

```
# network.py
# ...existing code above

def forward_no_relu(X, W1, b1, W2, b2):
    z1 = X @ W1 + b1
    h = z1                         # no activation!
    z2 = h @ W2 + b2
    y_hat = softmax(z2)
    return z1, h, z2, y_hat

def backward_no_relu(X, z1, h, y_hat, y_true, W2):
    batch_size = X.shape[0]
    delta2 = (y_hat - y_true) / batch_size
    dW2 = h.T @ delta2
    db2 = delta2.sum(axis=0)
    delta1 = delta2 @ W2.T         # no ReLU gate
    dW1 = X.T @ delta1
    db1 = delta1.sum(axis=0)
    return dW1, db1, dW2, db2

W1_nr, b1_nr, W2_nr, b2_nr = initialize_weights(np.random.default_rng(42))

for epoch in range(30):
    perm = rng.permutation(len(X_train))
    X_shuffled = X_train[perm]
    y_shuffled = y_train_oh[perm]

    for start in range(0, len(X_train), batch_size):
        end = start + batch_size
        X_batch = X_shuffled[start:end]
        y_batch = y_shuffled[start:end]

        z1, h, z2, y_hat = forward_no_relu(X_batch, W1_nr, b1_nr, W2_nr, b2_nr)
        dW1, db1, dW2, db2 = backward_no_relu(X_batch, z1, h, y_hat, y_batch, W2_nr)
        W1_nr, b1_nr, W2_nr, b2_nr = update_weights(
            W1_nr, b1_nr, W2_nr, b2_nr, dW1, db1, dW2, db2, lr)

preds_nr = forward_no_relu(X_test, W1_nr, b1_nr, W2_nr, b2_nr)[3].argmax(axis=1)
acc_nr = (preds_nr == y_test).sum() / len(y_test)
print(f"\nLinear classifier (no hidden layer): {acc_lin:.1%}")
print(f"Two layers, no ReLU:                 {acc_nr:.1%}")
print(f"Neural network (with ReLU):          {accuracy:.1%}")
```

Run it. The two-layer model without ReLU gets about 92%, the same as the linear classifier. 101,770 parameters, and it can't do any better than 7,850, because without the nonlinearity, the two matrices W_1 and W_2 multiply into one: $XW_1W_2 = XW_{\text{combined}}$. The extra layer bought nothing. The hidden layer only helps when ReLU is there to fold the space.

Now let's see **What the hidden layer learned.** We claimed that the hidden layer transforms the data into a space where digits are easier to separate. Let's see it. We'll run the test set through the first layer to get the 128-dimensional hidden representations, then project them to 2D so we can look at them.

Let's also **project to 2D.** The hidden layer produces 128-dimensional vectors. We need to get them down to 2 so we can plot them. **Principal component analysis** (PCA) does this by finding the directions along which the data varies the most, then projecting onto just those directions.

Think about a cloud of points shaped like a cigar. Most of the spread is along the cigar's length. A little spread is along its width. Almost none is along its thickness. If you had to flatten this cloud to 2D, you'd pick the length and the width, the two directions that preserve the most information about where each point sits relative to the others. That's what PCA does: find the axes of greatest variation, discard the rest.

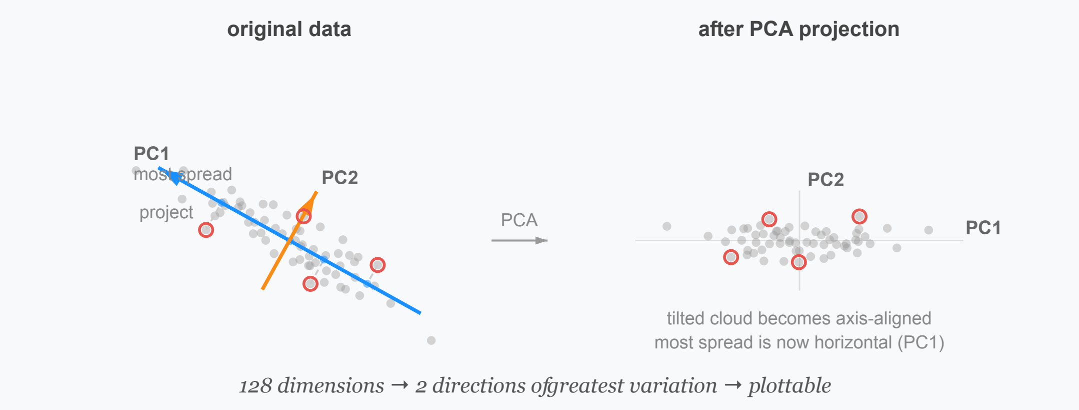

original data
after PCA projection
PC1
most spread
project
PC2
PCA
PC2
PC1
tilted cloud becomes axis-aligned
most spread is now horizontal (PC1)
128 dimensions → 2 directions ofgreatest variation → plottable

Figure 2.21: PCA: Finding the Axes of Greatest Variation

The algorithm works in three steps. First, center the data by subtracting the mean so the cloud sits at the origin. Second, compute the **covariance matrix**: a table where entry (i, j) tells you how much dimensions i and j vary together. If two dimensions rise and fall in sync across the data, their covariance is large; if they're unrelated, it's near zero. Third, find the **eigenvectors** of that matrix. Each eigenvector is a direction in the original space, and its eigenvalue tells you how much variance lies along that direction. The eigenvector with the largest eigenvalue is the cigar's length. The next largest is its width. To project to 2D, multiply each data point by the top two eigenvectors:

```python
# network.py
# ...existing code above

def pca_2d(vecs):
    centered = vecs - vecs.mean(axis=0)
    cov = np.cov(centered, rowvar=False)
    eigenvalues, eigenvectors = np.linalg.eigh(cov)
    top2 = eigenvectors[:, -2:][:, ::-1]
    return centered @ top2
```

`np.cov` computes the covariance matrix. `np.linalg.eigh` returns eigenvalues sorted smallest to largest, so `[:, -2:]` grabs the last two columns (the two largest) and `[:, ::-1]` flips them so the largest comes first. The final `centered @ top2` projects every 128-dimensional point onto these two directions, giving us an $(N, 2)$ array we can plot.

Now we project both the raw pixels and the hidden representations side by side:

```python
# network.py
# ...existing code above

sample_idx = rng.choice(len(X_test), 3000, replace=False)
X_sample = X_test[sample_idx]
y_sample = y_test[sample_idx]

z1_sample = X_sample @ W1 + b1
```

```
h_sample = relu(z1_sample)

pixel_2d = pca_2d(X_sample)
hidden_2d = pca_2d(h_sample)
```

Raw representations:

```
# network.py
# ...existing code above

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))
colors = plt.cm.tab10(np.linspace(0, 1, 10))

for digit in range(10):
    mask = (y_sample == digit)
    ax1.scatter(pixel_2d[mask, 0], pixel_2d[mask, 1],
                c=[colors[digit]], s=5, alpha=0.5, label=str(digit))
    ax2.scatter(hidden_2d[mask, 0], hidden_2d[mask, 1],
                c=[colors[digit]], s=5, alpha=0.5, label=str(digit))

ax1.set_title("Raw pixels (PCA to 2D)", fontsize=14)
ax1.legend(markerscale=3, fontsize=9)
ax1.grid(True, alpha=0.3)

ax2.set_title("After hidden layer (PCA to 2D)", fontsize=14)
ax2.legend(markerscale=3, fontsize=9)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig("representations.png", dpi=150)
plt.show()
```

In the left panel, the raw pixel representations are a tangle: digits overlap, 4s and 9s bleed into each other, 3s and 5s and 8s form one messy region. No straight line can sort this out, which is why the linear classifier tops out at 92%.

In the right panel, after the hidden layer, the same digits are pulled into tighter, more separated clusters. The network learned a transformation that moves 4s away from 9s and pulls 3s away from 8s. The output layer's job (drawing hyperplanes) is now easier because the hidden layer already did the hard work. This is exactly what we predicted: the hidden layer warps the space to make the data linearly separable.

Let's also build the confusion matrix to see which specific confusions improved:

```
# network.py
# ...existing code above

def confusion_matrix(y_true, y_pred):
    matrix = np.zeros((10, 10), dtype=int)
    for true, pred in zip(y_true, y_pred):
        matrix[true][pred] += 1
    return matrix
```

```
confusion = confusion_matrix(y_test, test_preds)

fig, ax = plt.subplots(figsize=(8, 8))
im = ax.imshow(confusion, cmap="Blues")
ax.set_xlabel("Predicted", fontsize=12)
ax.set_ylabel("Actual", fontsize=12)
ax.set_title("Neural network confusion matrix", fontsize=14)
ax.set_xticks(range(10))
ax.set_yticks(range(10))
for i in range(10):
    for j in range(10):
        color = "white" if confusion[i, j] > confusion.max() / 2 else "black"
        ax.text(j, i, str(confusion[i, j]),
                ha="center", va="center", color=color, fontsize=9)
plt.colorbar(im, ax=ax, shrink=0.8)
plt.tight_layout()
plt.savefig("nn_confusion.png", dpi=150)
plt.show()
```

Compare this to the template matcher's confusion matrix from Chapter 1. The same digit pairs that were hard before (4/9, 3/8, 3/5) are still the most confused, but the error counts are much lower. The network learned to distinguish them, though not perfectly.

Finally, let's look at the actual mistakes:

```
# network.py
# ...existing code above

wrong = np.where(test_preds != y_test)[0]
fig, axes = plt.subplots(2, 8, figsize=(12, 3))
for i, ax in enumerate(axes.flat):
    idx = wrong[i]
    ax.imshow(X_test[idx].reshape(28, 28), cmap="gray")
    ax.set_title(f"{test_preds[idx]}(={y_test[idx]})", fontsize=10)
    ax.axis("off")
plt.suptitle("Wrong predictions (predicted = actual)", fontsize=13)
plt.tight_layout()
plt.savefig("errors.png", dpi=150)
plt.show()
```

`np.where(condition)` returns the indices where the condition is true: every position where our prediction doesn't match the label. Look at these images. Many are genuinely ambiguous: a 4 that could be a 9, an 8 that could be a 3. The network's remaining mistakes are often the same ones a human would hesitate over.

That's a complete neural network in about 150 lines of NumPy. We went from a template matcher (81%) to a linear classifier (92%) to a neural network (97%), and we can see exactly why each step improved: the linear classifier optimized its boundaries instead of averaging, and the neural network added a hidden layer that warps the space to untangle the digits before drawing those boundaries. We also proved that the nonlinearity is what matters: remove ReLU, and the extra layer does nothing.

The same training loop (forward, loss, backward, update) will reappear in every chapter from here on.

In Chapter 3, we'll face a different problem: images came with pixel values built in, but how do you turn words into numbers that a neural network can learn from?

2.9 Chapter Summary

- The optimization loop repeats: forward pass (make predictions), compute loss (measure error), backward pass (compute gradients), update weights (reduce error)
- Gradient descent adjusts weights in the direction that reduces the loss, scaled by a learning rate η. Too large and it overshoots, too small and it crawls
- A perceptron computes a weighted sum plus bias, passed through an activation function: $\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$, which geometrically draws a hyperplane through the input space
- A linear classifier (perceptrons without a hidden layer) gets 92% on MNIST. Better than templates, but it can only draw straight boundaries through pixel space
- Activation functions (sigmoid, ReLU) introduce nonlinearity. Without them, stacking layers collapses into a single linear transformation, and you never escape the limits of straight boundaries
- The hidden layer warps the input space so that tangled classes become separable, then the output layer draws simple boundaries in the new space
- Backpropagation uses the chain rule to compute gradients layer by layer from output to input, telling each weight how to change
- The universal approximation theorem guarantees that a wide enough network can represent any continuous function, but representation is not the same as learnability

In the next chapter, we tackle text: how to turn words into numbers that preserve meaning, so that neural networks have something useful to learn from.

2.10 Exercises

1. Our network uses a learning rate of 0.1. Try training with 0.001, 0.01, 0.1, 0.5, and 1.0. Plot the loss curves for all five on the same graph. Which learning rates converge? Which diverge? We walked through the math of overshooting in the gradient descent section; you should see exactly that effect at the larger values. What happens to test accuracy at each learning rate? Is the fastest-converging learning rate also the one that gives the best final accuracy?

2. We used one hidden layer of 128 neurons. Try 32, 64, 128, 256, and 512, keeping everything else fixed. Plot test accuracy versus hidden layer size. Does accuracy keep improving, or does it plateau? Now plot training accuracy alongside test accuracy for each size. The gap between the two tells you how much the model overfits. Does the pattern match what Chapter 1 predicted about underfitting versus overfitting? Which size gives the best test accuracy, and is it the largest?

3. Add a **second hidden layer**. Your architecture becomes 784 -> 128 -> 64 -> 10 (two hidden layers with ReLU, one output layer with softmax). You'll need a third weight matrix and bias vector, and the forward and backward passes each get one more step. Does the deeper network beat the single-hidden-layer version? Run the PCA visualization on the output of each hidden layer separately.

The first layer transforms raw pixels; the second transforms the first layer's output. Do the digit clusters get progressively cleaner from layer to layer?

4. Verify that backpropagation is correct by implementing **numerical gradient checking**. For a single weight w, approximate its gradient with:

$$\frac{\partial L}{\partial w} \approx \frac{L(w+\epsilon) - L(w-\epsilon)}{2\epsilon}$$

   Pick a small batch, set $\epsilon = 10^{-5}$, and compare the numerical gradient to the analytical one from backpropagation for a handful of random weights in W_1 and W_2. They should agree to at least 5 decimal places. This is slow (two forward passes per weight), so only check a few, but it's the standard way to verify that your gradient code is correct. Now deliberately introduce a bug in your backward function (e.g., remove the ReLU derivative mask) and run the check again. How large is the discrepancy?

5. Visualize what individual hidden neurons respond to. For each of the 128 hidden neurons, find the 10 test images that produce the highest activation at that neuron (the value in h after ReLU). Pick 8 neurons with interesting patterns and plot their top-10 images in a grid. Do neurons specialize? Can you find one that fires on loops, one that fires on vertical strokes, one that responds to a specific part of the image? Now look at a neuron that fires on many different-looking digits. What visual feature might those digits share that isn't obvious at first glance?

6. The hands-on proved that removing ReLU collapses two linear layers into one. Try the reverse: what if the hidden layer uses ReLU but you make it very small, say 2 neurons instead of 128? Train a 784 -> 2 -> 10 network. Accuracy will be poor, but now you can skip PCA entirely: the hidden representation *is* 2D. Plot the hidden activations directly, colored by digit. You're looking at the network's entire internal representation of the data. Which digits can it still separate with only 2 dimensions? Which ones overlap? Gradually increase the bottleneck (2, 4, 8, 16, 32, 64, 128) and plot test accuracy versus hidden size. At what width does the network have enough room to untangle the digits?

3 From Words to Numbers

In the first two chapters, we worked with images. A 28×28 pixel grid gave us 784 numbers. The physics of cameras did the whole representation work: brightness values and colors in, vector out. We never had to decide how to turn an image into numbers, this was given to us. We just flattened the grid and started computing distances. We didn't even have to think about it, but the way images are stored in computers (as three matrices of red, green and blue or a brightness black and white matrix) has information about what's in the picture baked in.

For text, we don't get the same benefits. The way the computer represents text data is through a lookup table that turns letters into numbers (like ASCII). When you type "DATA", your computer stores `[68, 65, 84, 65]`, one number per letter. And it actually sees these numbers as binary, so it's really looking at `01000100 01000001 01010100 01000001`.

The difference between this and what we have for images is that it doesn't give us anything to work with. There's no embedded information in the words aside from what letters are in them, no meaning. The solution to this problem is what we'll explore in this chapter: how to turn text into numbers that a model can actually learn from.

3.1 The Text Representation Problem

Our neural network from Chapter 2 expected a vector of numbers as input. For digits, that was 784 pixel values that we know have embedded information. The question now is: what vector do we feed it for a word? What about for a sentence?

Starting with why this is hard, let's consider two sentences: "I bought apples at the store" and "I purchased fruit at the market." A human reads these and immediately knows they mean roughly the same thing. Different words, same meaning. Any useful numerical representation should reflect that: if you computed a distance between their vectors, it should be smaller than the difference between one of them and "I bought aircraft from the shore" even though it has a similar structure.

Now consider "The bank approved my loan" and "We fished along the river bank." Same word, completely different meanings. A useful representation should put these far apart, even though they share the word "bank."

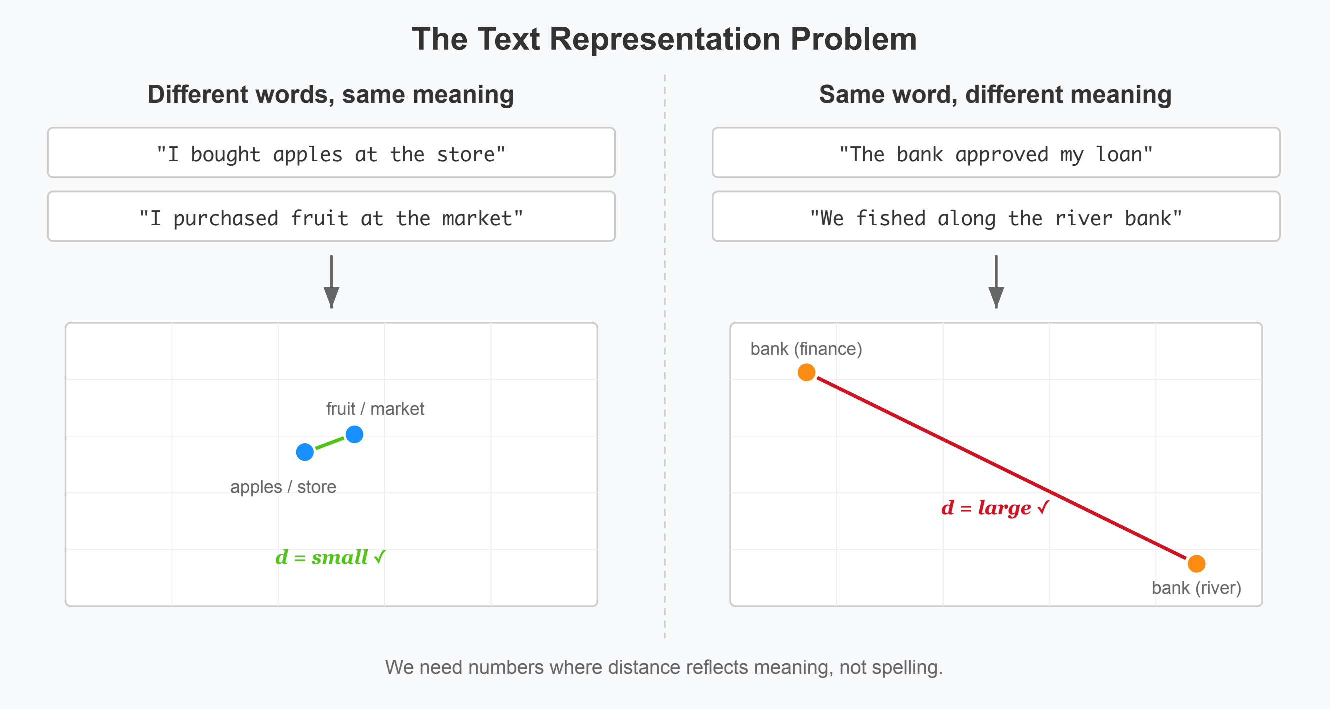

The Text Representation Problem
Different words, same meaning
"I bought apples at the store"
"I purchased fruit at the market"
fruit / market
apples / store
d = small ✓
Same word, different meaning
"The bank approved my loan"
"We fished along the river bank"
bank (finance)
d = large ✓
bank (river)
We need numbers where distance reflects meaning, not spelling.

Figure 3.1: The Text Representation Problem

With images, two similar-looking digits naturally had similar pixel values (ignoring the shifting problem from Chapter 1). Similarity was baked into the raw data. With text, similarity lives entirely in meaning, and meaning has nothing to do with the characters. The letters in “happy” and “joyful” share almost nothing. The letters in “bat” the animal and “bat” for baseball are identical. Surface form and meaning are decoupled in a way that pixels and visual appearance are not.

There are two problems we need to solve. First we need a mapping from words to numbers, and second, we need the *distances* between numbers in this mapping to correspond to the *relationships* between meanings, that is, it should have semantic value. Let’s start with the first problem since it’s easier.

3.2 One-Hot Encoding

Let’s start with the easier problem: getting the mapping from words to number vectors. A simple approach is giving each word a unique ID: its “position” in our vector.

If our vocabulary has five words, each of them is a vector with a single 1 and the rest 0s. “Cat” is `[1, 0, 0, 0, 0]`, “dog” is `[0, 1, 0, 0, 0]`, “car” is `[0, 0, 1, 0, 0]`. This is called **one-hot encoding**: one position is “hot” (set to 1), everything else is zero.

$$\text{cat} = [1, 0, 0, 0, 0], \quad \text{dog} = [0, 1, 0, 0, 0], \quad \text{car} = [0, 0, 1, 0, 0]$$

We've turned words into numbers. Now let's check if the distances say anything about semantics. Compute the Euclidean distance between "cat" and "dog":

$$d(\text{cat}, \text{dog}) = \sqrt{(1-0)^2 + (0-1)^2 + 0 + 0 + 0} = \sqrt{2}$$

Now the distance between "cat" and "car":

$$d(\text{cat}, \text{car}) = \sqrt{(1-0)^2 + 0 + (0-1)^2 + 0 + 0} = \sqrt{2}$$

Identical. And this isn't a coincidence. Every pair of one-hot vectors has the exact same distance from every other one-hot vector. Take any two distinct one-hot vectors of length n. One has a 1 at position i, the other has a 1 at position j, with $i \neq j$. At position i, the difference is $(1-0)^2 = 1$. At position j, the difference is $(0-1)^2 = 1$. At every other position, both values are 0, so the difference is 0. The distance is always:

$$d = \sqrt{1 + 1 + 0 + \cdots + 0} = \sqrt{2}$$

No matter which two words you pick, no matter how large the vocabulary. The math doesn't care which positions are hot.

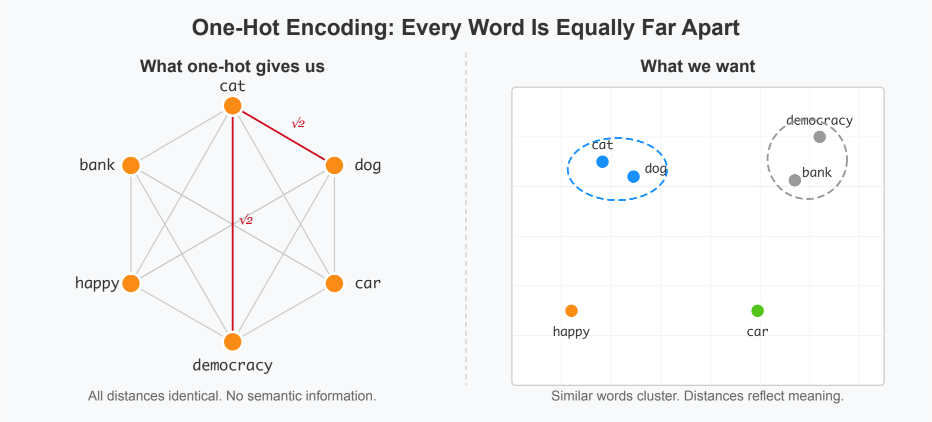

One-Hot Encoding: Every Word Is Equally Far Apart
What one-hot gives us
cat
bank
dog
happy
car
democracy
√2
√2
All distances identical. No semantic information.
What we want
democracy
cat
dog
bank
happy
car
Similar words cluster. Distances reflect meaning.

Figure 3.2: One-Hot Encoding: Every Word Is Equally Far Apart

"Cat" is exactly as far from "dog" as it is from "democracy." The representation encodes identity (is this word X or not?) but says absolutely nothing about meaning. It's the text equivalent of labeling every pixel with a unique ID instead of recording its brightness. We've managed to assign numbers, but those numbers don't capture the semantic value we wanted.

There's also a scale problem. Our toy vocabulary has five words. A real vocabulary has tens of thousands. The MNIST images we worked with in Chapter 1 were 784-dimensional vectors. A one-hot vocabulary of 50,000 words gives you 50,000-dimensional vectors, each with a single 1 and 49,999 zeros. Almost entirely empty.

We call these **sparse vectors**: they are huge, wasteful, and carrying almost no information per dimension. A 50,000-dimensional one-hot vector technically carries only $\log_2(50{,}000) \approx 15.6$ bits of information (enough to say *which* word it is, nothing more). That same amount of information fits in two bytes.

So one-hot encoding isn't our solution. It doesn't capture meaning, and it doesn't scale. We need a representation where the numbers themselves encode semantic relationships. Where "cat" and "dog" are close because they're both animals, and "cat" and "car" are far apart because they have nothing to do with each other even though the words are similar.

But how? Who decides that "cat" and "dog" should be close? We could try assigning coordinates by hand: maybe dimension 1 is "how alive is this thing" and dimension 2 is "how big is it," so "cat" gets `[0.9, 0.3]` and "elephant" gets `[0.9, 0.95]`. For any real vocabulary this would be a herculean task. You'd need to define hundreds of meaningful dimensions and manually score every word on each one. You'd miss relationships you didn't think of, and you'd never agree with anyone else on the scores.

The answer, as with other things so far in this book, is to learn the numbers from data. But learn them from *what*? What signal in raw text tells us that "cat" and "dog" are related?

3.3 The Distributional Hypothesis

Think about how you learned the meaning of words as a kid. Nobody handed you a dictionary definition of "dog" at age two. You heard the word in context: "Look at the dog!" while pointing at a furry thing. "Don't pet that dog, it might bite." "The dog wants to go outside." Thousands of contexts, and from them you built up a sense of what "dog" means. Not from the letters d-o-g, but from what's accompanying the word.

In 1957, linguist J.R. Firth put this intuition into a sentence that became one of the most quoted lines in NLP: "You shall know a word by the company it keeps" (Firth 1957). This is the **distributional hypothesis**: words that appear in similar contexts have similar meanings. It's the idea that makes the rest of this chapter work.

Look at where "cat" and "dog" show up in real text. Both appear after "I fed the **."** ***Both appear in "The*** sat on my lap." Both co-occur with "pet," "fur," "vet," "adopted," and "cute." The overlap in their contexts is enormous. Now look at "democracy." It appears near "vote," "government," "election," "freedom," "constitution." Nearly zero overlap with the contexts of "cat" or "dog."

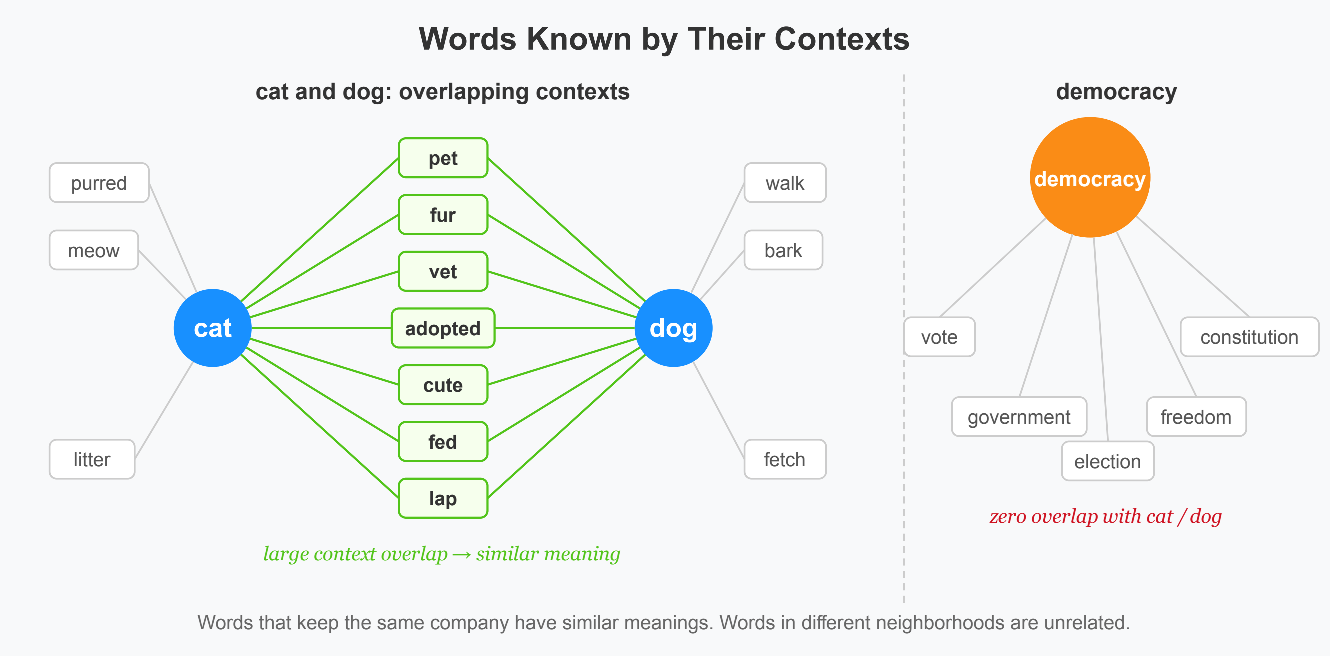

Words Known by Their Contexts
cat and dog: overlapping contexts
purred
meow
litter
cat
pet
fur
vet
adopted
cute
fed
lap
dog
walk
bark
fetch
large context overlap → similar meaning
democracy
democracy
vote
government
election
freedom
constitution
zero overlap with cat / dog
Words that keep the same company have similar meanings. Words in different neighborhoods are unrelated.

Figure 3.3: Words Known by Their Contexts

This works at a finer grain than these examples suggest. "Cat" and "dog" share many contexts, but not all. You say "I took my dog for a walk" but not usually "I took my cat for a walk." You say "The cat purred" but not "The dog purred." The contexts that *differ* encode the differences in meaning. And the contexts that overlap encode the similarities. The distributional hypothesis says this pattern holds across the entire vocabulary: if two words show up in the same linguistic neighborhoods, they mean similar things.

You can see the hypothesis at work when you encounter a word you don't know. Suppose you're reading and you hit: "She stirred the dashi into the broth" and "the dashi gave the soup a rich umami flavor." You've never heard of dashi. But you've seen "stirred the stock into the broth" and "gave the soup a rich savory flavor." The overlapping contexts tell you, without anyone explaining it, that dashi is probably some kind of cooking liquid similar to stock or broth. (It is: it's a Japanese soup base made from kelp and dried fish.) You just did what the distributional hypothesis describes: you inferred meaning from context.

The distributional hypothesis gives us a path. We don't have to hand-assign meaning to dimensions or build a dictionary of synonyms. We need a lot of text, a way to measure context overlap, and an algorithm that turns those patterns into vectors. Words that keep the same company will end up with similar numbers. Words that live in different linguistic neighborhoods will end up far apart.

For decades, the distributional hypothesis remained more of a linguistic observation than a practical tool. Researchers built co-occurrence matrices (counting how often each word appears near each other word), and those matrices did capture some semantic structure, but they were enormous, sparse, and hard to work with. What changed was a 2013 paper that showed you could train a small neural network on a prediction task and get dense, compact vectors that captured meaning far better than any co-occurrence matrix. That was Word2Vec.

3.4 Word Embeddings and Word2Vec

In 2013, Tomas Mikolov and colleagues at Google published Word2Vec (Mikolov, Chen, et al. 2013; Mikolov, Sutskever, et al. 2013), a method for learning word vectors directly from raw text. The idea was not entirely new (researchers had been using distributional methods for years, and neural language models had already shown that word representations could be learned as a byproduct of prediction (Bengio et al. 2003)). But Word2Vec was fast enough to train on billions of words and produced vectors good enough to be immediately useful. It made the concept of **word embeddings** mainstream: compact, learned vectors that represent words as points in a continuous space where distances reflect meaning.

The training works like this. Take a large text corpus (billions of words from news articles, Wikipedia, books), then slide a window across the text. At each position, the center word is the **target** and the surrounding words are the **context**. The model's job is: given a word, predict which words are likely to appear nearby.

Consider the sentence "She poured the cold water into the glass" with a window of size 2 (two words on each side). When the window is centered on "water":

$$\text{context: } [\text{the}, \text{cold}, \underbrace{\text{water}}_{\text{target}}, \text{into}, \text{the}]$$

The model should learn that "water" predicts "cold," "the," and "into" as likely neighbors. This variant is called **skip-gram**: skip the center word, predict the context around it. (There's also CBOW, continuous bag of words, which goes the other direction: given the context, predict the center word. Skip-gram tends to work better for smaller datasets and rare words; in practice, both produce similar results.)

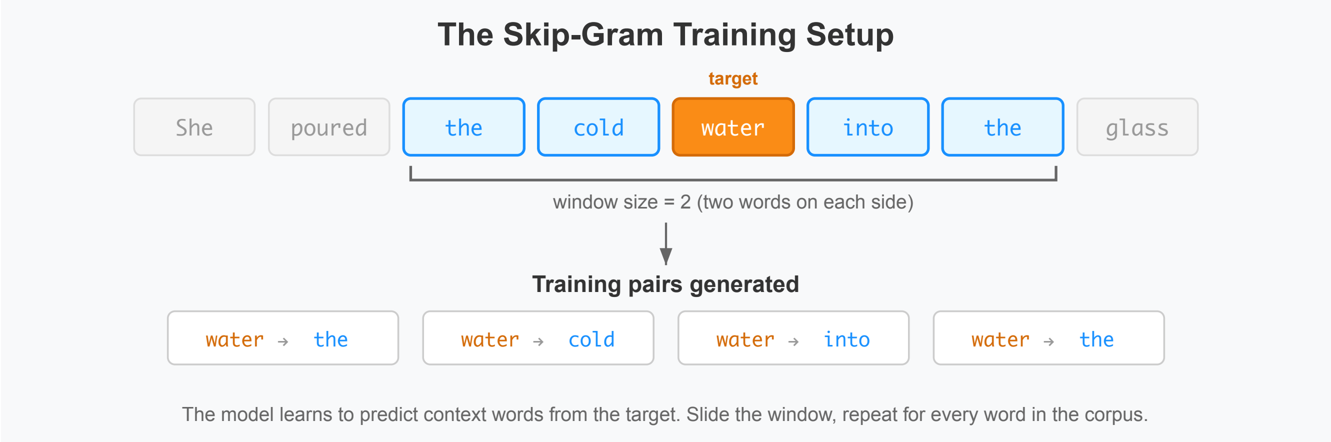

The Skip-Gram Training Setup
target
She
poured
the
cold
water
into
the
glass
window size = 2 (two words on each side)
Training pairs generated
water → the
water → cold
water → into
water → the
The model learns to predict context words from the target. Slide the window, repeat for every word in the corpus.

Figure 3.4: The Skip-Gram Training Setup

Each word in the vocabulary gets two vectors of d real numbers (typically $d = 100$ to 300), initialized randomly. One vector represents the word as a target; the other represents it as context. To predict whether a word c is likely to appear near a target word t, the model computes the dot product of their vectors.

Think about what the dot product measures. Take two vectors and multiply them element by element, then add everything up:

$$w_t \cdot w_c = w_{t,1} \cdot w_{c,1} + w_{t,2} \cdot w_{c,2} + \cdots + w_{t,d} \cdot w_{c,d}$$

If the two vectors point in similar directions (large values in the same positions), the dot product is large. If they point in different directions, it's small or negative. So the model scores word pairs by how aligned their vectors are. A high score means "these words should appear together." A low score means "these words don't belong near each other."

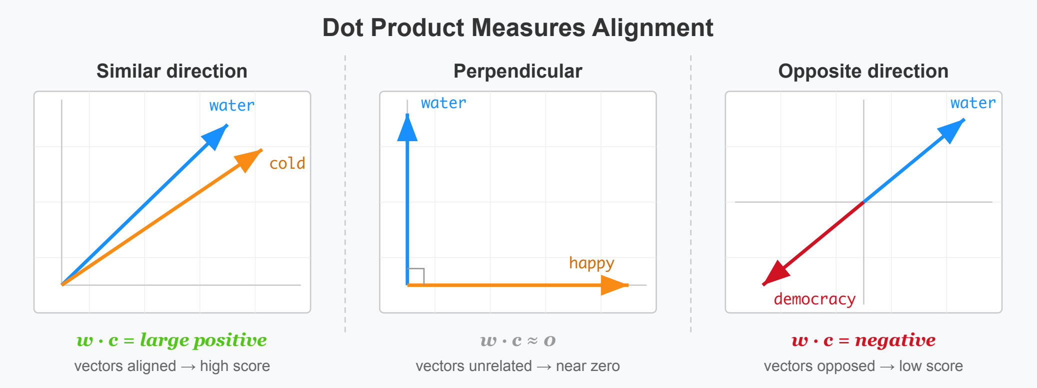


Figure 3.5: Dot Product Measures Alignment

But a raw dot product isn't a probability. It's a number from $-\infty$ to ∞. To turn it into a probability, we push it through the sigmoid function (the same function from Chapter 2's logistic regression):

$$P(\text{c is near t}) = \sigma(w_t \cdot w_c) = \frac{1}{1 + e^{-w_t \cdot w_c}}$$

Training works the way you'd expect: make predictions, compute a loss, adjust the vectors with gradient descent. For each target word, the model creates positive examples (actual context words from the text) and negative examples (random words sampled from the vocabulary that were *not* in the context). It tries to make the dot product high for real context pairs and low for random pairs. After billions of these updates, the vectors settle into positions where words with similar usage patterns have similar vectors.

There's a practical problem with this training process. Without a trick called **negative sampling**, you'd have to compute a probability distribution over the entire vocabulary for every training step (to normalize the scores into probabilities that sum to 1). With 50,000 words, that's 50,000 dot products per step. Negative sampling sidesteps this: instead of computing scores for all words, you score the actual context word (positive) against a handful of random words (negative, typically 5 to 15). This makes training much faster, which is why Word2Vec could scale to billions of words while earlier methods couldn't.

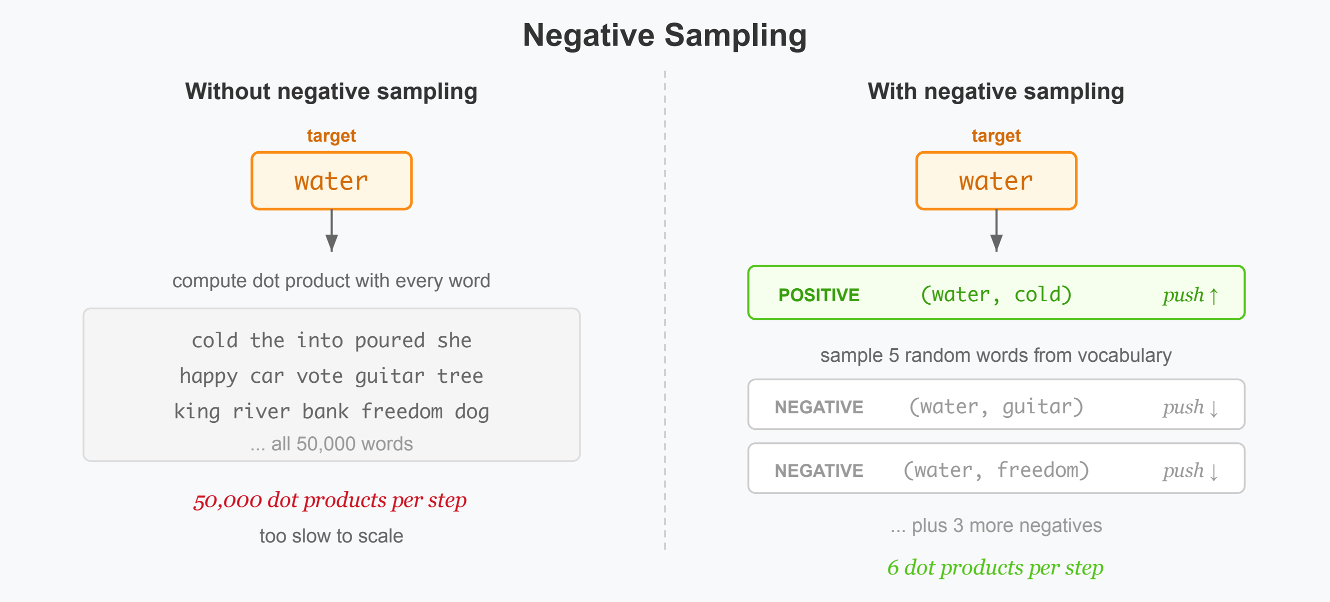

Negative Sampling
Without negative sampling
target
water
compute dot product with every word
cold the into poured she
happy car vote guitar tree
king river bank freedom dog
... all 50,000 words
50,000 dot products per step
too slow to scale
With negative sampling
target
water
POSITIVE (water, cold) push ↑
sample 5 random words from vocabulary
NEGATIVE (water, guitar) push ↓
NEGATIVE (water, freedom) push ↓
... plus 3 more negatives
6 dot products per step

Figure 3.6: Negative Sampling: Score Real Pairs Against Random Ones

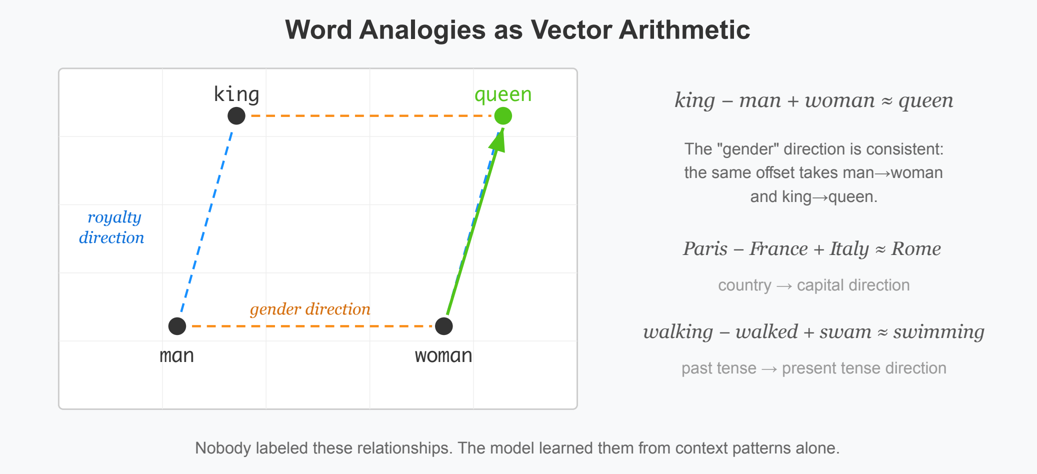
The loss function for negative sampling looks like this. For a target word t, a real context word c, and K random negative words $n_1, \ldots, n_K$:

$$L = -\log \sigma(w_t \cdot w_c) - \sum_{k=1}^{K} \log \sigma(-w_t \cdot w_k)$$

The first term pushes the dot product up for the real pair (make the sigmoid close to 1). Each term in the sum pushes the dot product down for the random pairs (make the sigmoid close to 0). If this looks familiar, it should: it's the same sigmoid and binary cross-entropy from Chapter 2. Each word pair is a binary classification problem: "is this a real context pair or a random one?" Word2Vec isn't a new kind of algorithm. It's logistic regression applied to a clever prediction task, run billions of times until the vectors encode meaning.

The vectors that come out learn more than "these words are similar." They learn structured relationships. Mikolov et al. (2013) showed that you could take the vector for "king," subtract the vector for "man," add the vector for "woman," and the closest word to the result is "queen."

$$\vec{\text{king}} - \vec{\text{man}} + \vec{\text{woman}} \approx \vec{\text{queen}}$$



king − man + woman ≈ queen

The "gender" direction is consistent: the same offset takes man→woman and king→queen.

Paris − France + Italy ≈ Rome

country → capital direction

walking − walked + swam ≈ swimming

past tense → present tense direction

Nobody labeled these relationships. The model learned them from context patterns alone.

Figure 3.7: Word Analogies as Vector Arithmetic

Nobody told the model about gender or royalty. It learned, purely from context patterns in billions of sentences, that there's a direction in the vector space that corresponds to "male to female," and that direction is consistent: it takes you from "king" to "queen," from "man" to "woman," from "brother" to "sister." Similar arithmetic works for country-capital relationships ($\vec{\text{Paris}} - \vec{\text{France}} + \vec{\text{Italy}} \approx \vec{\text{Rome}}$), verb tenses ($\vec{\text{walking}} - \vec{\text{walked}} + \vec{\text{swam}} \approx \vec{\text{swimming}}$), and dozens of other semantic relationships.

To be clear: "gender direction" doesn't mean there's a single dimension labeled "gender." These vectors have 300 dimensions. The direction that takes "man" to "woman" is a pattern spread across many of them. The parallelogram diagram is a 2D projection for intuition, not a literal map of the space.

These vectors are called **embeddings** because each word is embedded (placed) in a continuous space. The word's position *is* its meaning, at least as far as the model is concerned. Unlike the sparse, meaningless one-hot vectors from earlier, these are **dense vectors**: short (100 to 300 dimensions instead of 50,000), with every dimension carrying information. No dimension has a human-interpretable label ("dimension 47 means animacy"), but the overall pattern encodes semantic relationships in a way that supports arithmetic, clustering, and similarity computation.

Word2Vec has real limitations. Each word gets exactly one vector, regardless of context. "Bank" has the same embedding whether you're talking about money or rivers. The vectors are fixed after training: you look them up in a table, and they never adjust to the sentence they appear in. But even with these limitations, Word2Vec was a turning point. Before 2013, the default text representation in NLP was either one-hot vectors or hand-engineered features. After 2013, it was embeddings. Subsequent methods like GloVe (Pennington, Socher, and Manning 2014), which trains on global co-occurrence statistics rather than local windows, and fastText (Bojanowski et al. 2017), which handles unseen words by breaking them into subword pieces, refined the approach. But the core remained the same: learn vectors from context, and distances will reflect meaning.

We now have dense vectors where distances reflect meaning. But which distance? In Chapter 1 we used Euclidean distance, and it worked fine for pixel vectors. For embeddings, there's a better option.

3.5 Cosine Similarity

Euclidean distance measures the straight-line distance between two points. For pixel vectors in Chapter 1, that was natural: two images with similar brightness values are close, two images with different brightness values are far apart. But in embedding space, Euclidean distance has a problem. It conflates two things: how similar two vectors' *directions* are and how different their *magnitudes* are.

Think about what magnitude means for word embeddings. During training, words that appear more frequently get more gradient updates, and their vectors tend to grow longer. The word "the" gets updated billions of times; the word "aardvark" gets updated rarely. Their vector lengths end up very different, for reasons that have nothing to do with meaning. Euclidean distance would say "aardvark" is far from "the" partly because their magnitudes differ. That's not semantic information. That's a training artifact.

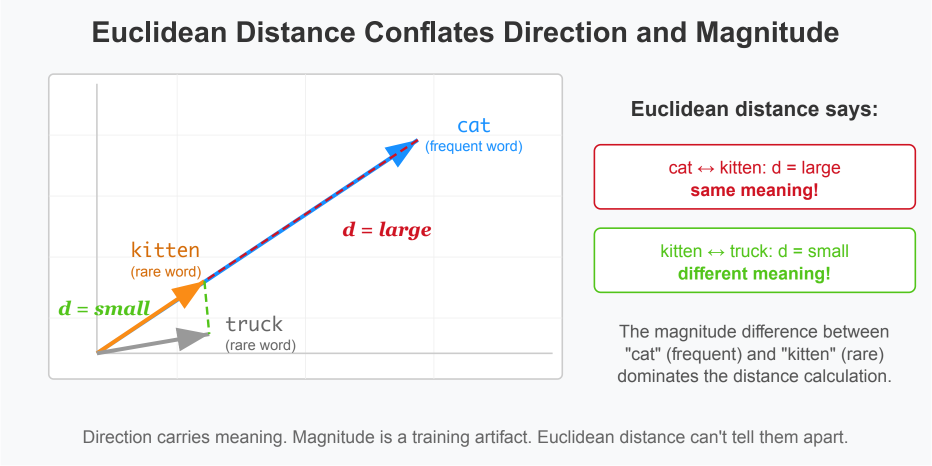

Euclidean Distance Conflates Direction and Magnitude
cat
(frequent word)
d = large
kitten
(rare word)
d = small
truck
(rare word)
Euclidean distance says:
cat ↔ kitten: d = large
same meaning!
kitten ↔ truck: d = small
different meaning!
The magnitude difference between "cat" (frequent) and "kitten" (rare) dominates the distance calculation.
Direction carries meaning. Magnitude is a training artifact. Euclidean distance can't tell them apart.

Figure 3.8: Euclidean Distance Conflates Direction and Magnitude

One fix is to remove magnitude entirely. If we scale every vector to length 1 (dividing each vector by its norm), we get **unit vectors** that live on the surface of a sphere. This is called **L2 normalization**:

$$\hat{a} = \frac{a}{\|a\|}, \quad \text{where } \|a\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_d^2}$$

After normalization, $\|\hat{a}\| = 1$ for every vector. Now magnitude is gone and only direction remains. What happens to Euclidean distance between two unit vectors? Expand the squared distance:

$$\|\hat{a} - \hat{b}\|^2 = \|\hat{a}\|^2 - 2(\hat{a} \cdot \hat{b}) + \|\hat{b}\|^2 = 1 - 2(\hat{a} \cdot \hat{b}) + 1 = 2 - 2(\hat{a} \cdot \hat{b})$$

The dot product of two unit vectors is exactly the cosine of the angle between them. So:

$$\|\hat{a} - \hat{b}\|^2 = 2 - 2\cos\theta = 2(1 - \cos\theta)$$

This tells us something really useful: for unit vectors, squared Euclidean distance and cosine similarity are two sides of the same coin. When cosine similarity is high ($\cos\theta$ close to 1), the distance is small. When cosine similarity is low, the distance is large. Minimizing one is mathematically equivalent to maximizing the other.

So we could L2-normalize all our vectors and keep using Euclidean distance. But in practice, it's simpler to skip the normalization step and compute **cosine similarity** directly:

$$\text{cosine\_similarity}(a, b) = \frac{a \cdot b}{\|a\| \cdot \|b\|}$$

The numerator is the dot product. The denominator is the product of the two lengths. Dividing by the lengths does the normalization inline: it cancels out magnitude and leaves only the angular relationship. The result ranges from -1 (opposite directions) through 0 (perpendicular) to 1 (same direction).

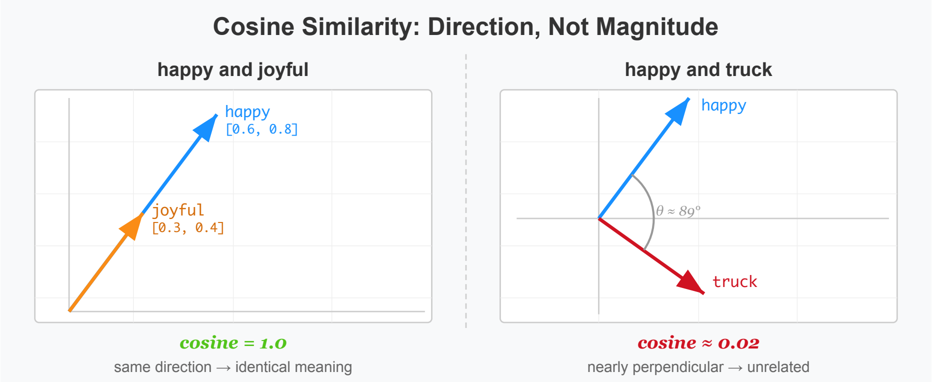

Cosine Similarity: Direction, Not Magnitude
happy and joyful
happy
[0.6, 0.8]
joyful
[0.3, 0.4]
cosine = 1.0
same direction → identical meaning
happy and truck
happy
θ ≈ 89°
truck
cosine ≈ 0.02
nearly perpendicular → unrelated

Figure 3.9: Cosine Similarity: Direction, Not Magnitude

This is what we want for word embeddings. Two words that mean similar things should point in similar directions, regardless of how long their vectors happen to be. "Cat" and "kitten" should point nearly the same way, even if "cat" has a longer vector because it appeared more often in the training data. Cosine similarity captures that; Euclidean distance doesn't.

Let's see this with concrete numbers. Suppose "happy" has the embedding $[0.6, 0.8]$ and "joyful" has $[0.3, 0.4]$. These point in the same direction; one is just a scaled-down version of the other.

$$\text{cosine\_similarity} = \frac{(0.6)(0.3) + (0.8)(0.4)}{\sqrt{0.6^2 + 0.8^2} \cdot \sqrt{0.3^2 + 0.4^2}} = \frac{0.50}{1.0 \times 0.5} = 1.0$$

Perfect similarity. The Euclidean distance between these same vectors is $\sqrt{(0.3)^2 + (0.4)^2} = 0.5$, which is nonzero and might suggest they're somewhat different. Cosine similarity sees through the magnitude difference to the shared direction.

Now compare with a third word. Suppose "truck" has the embedding $[0.7, -0.5]$. This points in a completely different direction from "happy."

$$\text{cosine\_similarity}(\text{happy}, \text{truck}) = \frac{(0.6)(0.7) + (0.8)(-0.5)}{\sqrt{0.6^2 + 0.8^2} \cdot \sqrt{0.7^2 + 0.5^2}} = \frac{0.02}{1.0 \times 0.86} = 0.023$$

Nearly zero. Cosine similarity correctly identifies that "happy" and "joyful" are semantically aligned while "happy" and "truck" are unrelated.

In practice, cosine similarity is the standard way to compare embeddings. When you use a vector database in Chapter 10, you're searching by cosine similarity. When you test word analogies, you're finding the nearest vector by cosine. When you measure whether two sentences mean the same thing, cosine is usually the metric. Euclidean distance asks "how far apart are these points?" Cosine similarity asks "are these pointing the same way?" For text, direction matters more.

That's enough theory. Let's load real embeddings and see what they actually encode.

3.6 Hands-On: Exploring Pre-Trained Embeddings

We've talked about word embeddings as vectors that capture meaning, now let's work with real ones. We'll load pre-trained GloVe embeddings (Pennington, Socher, and Manning 2014) (Global Vectors for Word Representation). GloVe is closely related to Word2Vec but trains on global word co-occurrence statistics rather than local context windows; the resulting vectors exhibit the same properties: analogy arithmetic works, similar words cluster together, and biases carry over from the training text. We're using GloVe here because the pre-trained vectors are freely available and easy to load, but everything we explore in this section applies equally to Word2Vec embeddings. We'll compute similarity ourselves, run the analogy experiments, and investigate where the representation fails.

GloVe embeddings come as a text file: one word per line, followed by its vector values. We're using 50-dimensional vectors trained on 6 billion tokens of text from Wikipedia and news. The file is about 66MB.

Set up your project:

```
mkdir chapter3
cd chapter3
uv init
uv add numpy matplotlib huggingface-hub
```

Create a file called `embeddings.py`. We'll download the vectors from Hugging Face (cached after the first run, so subsequent runs start instantly) and parse them into NumPy arrays:

```
# embeddings.py
import numpy as np
from huggingface_hub import hf_hub_download

def load_glove(path):
    words = []
    vectors = []
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            parts = line.split()
            words.append(parts[0])
            vectors.append([float(x) for x in parts[1:]])
    vectors = np.array(vectors, dtype=np.float32)
    word_to_index = {word: i for i, word in enumerate(words)}
    return words, vectors, word_to_index
```

Each line in the file looks like `cat 0.49 -0.31 0.17 ...`: a word followed by 50 numbers. `split()` breaks the line on whitespace, giving us a list where the first element is the word and the rest are number strings. `parts[1:]` grabs everything after the word, and `float(x) for x in parts[1:]` converts each string to a number. The result is three things: a list of words, a big matrix of vectors (one row per word), and a dictionary mapping each word to its row number.

```
# embeddings.py (continued)

path = hf_hub_download(
    repo_id="igorbenav/glove-6b-50d",
    filename="glove.6B.50d.txt",
    repo_type="dataset",
)
words, vectors, word_to_index = load_glove(path)
print(f"Loaded {len(words)} words, each with {vectors.shape[1]} dimensions")
```

Run it with `uv run python embeddings.py`. You should see 400,000 words, 50 dimensions each. That's 20 million numbers encoding the meaning of 400,000 words, learned entirely from how those words are used in context. Let's look at what a vector actually looks like:

```
# embeddings.py
# ...existing code above

def get_vector(word):
    if word not in word_to_index:
        print(f"'{word}' not in vocabulary")
        return None
    return vectors[word_to_index[word]]

cat = get_vector("cat")
print(f"\n'cat' vector (first 10 dims): {cat[:10].round(4)}")
print(f"Min: {cat.min():.4f}, Max: {cat.max():.4f}")
```

Fifty numbers, most small, some positive, some negative. Nothing about any individual number says "animal" or "small" or "has fur." You can't interpret dimension 23 or dimension 41. The meaning is in the *pattern*: in how these 50 numbers relate to the 50 numbers of every other word.

Now let's implement cosine similarity from scratch. The formula we derived earlier:

$$\text{cosine\_similarity}(a, b) = \frac{a \cdot b}{\|a\| \cdot \|b\|} = \frac{\sum_{i=1}^{d} a_i b_i}{\sqrt{\sum_{i=1}^{d} a_i^2} \cdot \sqrt{\sum_{i=1}^{d} b_i^2}}$$

Three steps: compute the dot product (numerator), compute each vector's length (denominator), divide:

```
# embeddings.py
# ...existing code above

def cosine_similarity(a, b):
    dot = np.dot(a, b)                        # sum of a_i * b_i
    norm_a = np.sqrt(np.sum(a ** 2))          # ||a||
    norm_b = np.sqrt(np.sum(b ** 2))          # ||b||
    if norm_a == 0 or norm_b == 0:
        return 0.0
    return dot / (norm_a * norm_b)
```

Let's test it on word pairs where we have strong intuitions about what the answer should be:

```
# embeddings.py
# ...existing code above

pairs = [
    ("cat", "dog"),
    ("cat", "kitten"),
    ("cat", "car"),
    ("cat", "democracy"),
    ("happy", "joyful"),
    ("happy", "sad"),
    ("king", "queen"),
    ("king", "banana"),
    ("france", "paris"),
    ("germany", "berlin"),
]

print("\nWord pair similarities:")
for w1, w2 in pairs:
    v1, v2 = get_vector(w1), get_vector(w2)
    if v1 is not None and v2 is not None:
        sim = cosine_similarity(v1, v2)
        print(f"  {w1:12s} - {w2:12s}: {sim:.3f}")
```

Run it and study the output. “Cat” and “dog” should be very high (above 0.9), which makes sense: they share enormous context overlap (“pet,” “fed the *,” ”took the* to the vet”). “Cat” and “kitten” is lower than you might expect (around 0.6), because “kitten” appears in a narrower set of contexts (cuteness, youth, smallness) while “cat” is used much more broadly. “Cat” and “car” should be moderate but lower (they share some syntactic contexts but little meaning). “Cat” and “democracy” should be near zero.

Now look at “happy” and “sad.” You might expect them to be far apart (they're opposites), but they're quite similar (around 0.7). This tells you something important about what cosine similarity actually measures on distributional embeddings: **relatedness**, not substitutability. “Happy” and “sad” are antonyms, but they appear in almost identical contexts. You say “I feel happy” and “I feel sad.” You say “She seemed happy” and “She seemed sad.” The distributional hypothesis puts them close because they keep the same company. Cosine similarity captures how much two words have to do with each other, not whether you can swap one for the other without changing the meaning.

Next, let’s find the most similar words to a given word. We need to compute cosine similarity against the entire vocabulary and return the top results. Instead of a Python loop over 400,000 words, we can use a property we proved in the cosine similarity section: for unit vectors, cosine similarity is just the dot product. If we first L2-normalize every vector:

$$\hat{v} = \frac{v}{\|v\|}$$

then finding the most similar word to a query q is just:

$$\text{most similar} = \arg\max_{w} \; \hat{v}_w \cdot \hat{q}$$

And the dot product of one vector against all 400,000 vectors at once is a matrix-vector multiply. We normalize the full matrix once up front so we don’t redo it on every query:

```python
# embeddings.py
# ...existing code above

norms = np.sqrt(np.sum(vectors ** 2, axis=1))
norms[norms == 0] = 1.0
normalized = vectors / norms[:, np.newaxis]
```

`norms[:, np.newaxis]` reshapes the norms from a flat list into a column, so that when we divide vectors / norms[:, np.newaxis], each row gets divided by its own norm. This is the $\hat{v} = v/\|v\|$ step. Once normalized exists, every similarity query is a single matrix-vector multiply:

```python
# embeddings.py
# ...existing code above

def most_similar(word, n=10):
    vec = get_vector(word)
    if vec is None:
        return []

    word_norm = np.sqrt(np.sum(vec ** 2))
    word_normalized = vec / word_norm

    similarities = normalized @ word_normalized

    top_indices = np.argsort(similarities)[::-1][1:n+1]
    return [(words[i], similarities[i]) for i in top_indices]
```

`normalized @ word_normalized` gives us 400,000 cosine similarities in a single matrix multiply. The `@` operator is matrix multiplication, and this is where NumPy is a lot better than doing it with standard Python: one line of code replaces 400,000 loops, it's done in parallel.

The last line chains three operations: `argsort` returns the positions that would sort the array from lowest to highest, `[::-1]` reverses it to highest-first, and `[1:n+1]` skips the first result (the query word itself, which always has similarity 1.0) and takes the next `n`.

Let's see what comes back:

```python
# embeddings.py
# ...existing code above

for query in ["cat", "france", "happy", "python"]:
    print(f"\nMost similar to '{query}':")
    for word, sim in most_similar(query):
        print(f"  {word:15s} {sim:.3f}")
```

The neighbors of "cat" should include "dog," "rabbit," "cats," "pet," and other animals. The neighbors of "france" should be other European countries. Look at "python": you'll likely see a mix of snake-related words and programming-related words, because the single vector has to accommodate both senses. This is the one-vector-per-word limitation we discussed: the embedding for "python" is somewhere between "programming language" and "large snake," which means it's not perfectly representative of either.

Now let's test the analogy arithmetic. We claimed that the vector relationships are consistent: that the offset from "man" to "king" should be roughly the same as the offset from "woman" to "queen." In vector arithmetic:

$$\vec{\text{king}} - \vec{\text{man}} + \vec{\text{woman}} \approx \vec{\text{queen}}$$

To find the answer, we compute the target vector $t = b - a + c$ and search for the word whose vector is most similar to t:

$$\text{answer} = \arg\max_{w} \ \text{cosine\_similarity}(v_w,\ t)$$

```python
# embeddings.py
# ...existing code above

def analogy(a, b, c, n=5):
    """a is to b as c is to ???"""
    va, vb, vc = get_vector(a), get_vector(b), get_vector(c)
    if any(v is None for v in [va, vb, vc]):
        return []

    target = vb - va + vc

    target_norm = np.sqrt(np.sum(target ** 2))
    target_normalized = target / target_norm
    similarities = normalized @ target_normalized

    for w in [a, b, c]:
        if w in word_to_index:
            similarities[word_to_index[w]] = -1

    top_indices = np.argsort(similarities)[::-1][:n]
    return [(words[i], similarities[i]) for i in top_indices]
```

Read the function carefully. "Man is to king as woman is to ???" We compute $\vec{\text{king}} - \vec{\text{man}} + \vec{\text{woman}}$ and find the nearest word to the result. We exclude the three input words from the results (by setting their similarities to -1, so they'll sort to the bottom) because otherwise the closest result is often one of the inputs, which tells us nothing. Let's run it:

```
# embeddings.py
# ...existing code above

print("\n--- Analogies ---")

print("\nman : king :: woman : ???")
for word, sim in analogy("man", "king", "woman"):
    print(f"  {word:15s} {sim:.3f}")

print("\nfrance : paris :: italy : ???")
for word, sim in analogy("france", "paris", "italy"):
    print(f"  {word:15s} {sim:.3f}")

print("\nwalk : walking :: swim : ???")
for word, sim in analogy("walk", "walking", "swim"):
    print(f"  {word:15s} {sim:.3f}")

print("\njapan : sushi :: mexico : ???")
for word, sim in analogy("japan", "sushi", "mexico"):
    print(f"  {word:15s} {sim:.3f}")
```

The first result for king-man+woman should be "queen." The country-capital analogy should return "rome." The verb tense analogy should return "swimming." The food-country analogy is less reliable (cultural associations in the training data are messier than grammatical relationships), but you'll likely get something reasonable.

No labels. No supervised signal about gender, geography, or grammar. Pure context prediction on raw text, and out come vectors that encode relational structure.

But the model learned from human text, and human text contains human biases. Let's look:

```
# embeddings.py
# ...existing code above

print("\n--- Bias in embeddings ---")

print("\nman : doctor :: woman : ???")
```

```
for word, sim in analogy("man", "doctor", "woman"):
    print(f"  {word:15s} {sim:.3f}")

print("\nman : programmer :: woman : ???")
for word, sim in analogy("man", "programmer", "woman"):
    print(f"  {word:15s} {sim:.3f}")

print("\nman : brilliant :: woman : ???")
for word, sim in analogy("man", "brilliant", "woman"):
    print(f"  {word:15s} {sim:.3f}")
```

"Man is to doctor as woman is to ???" should give "doctor" (same profession regardless of gender). But you'll get "nurse." "Man is to programmer as woman is to ???" gives "therapist" instead of "programmer." Bolukbasi et al. (2016) showed that Word2Vec embeddings trained on Google News exhibited exactly these stereotypes, with occupations systematically skewed along a gender axis. Subsequent work by Zhao et al. (2017) showed that the biases don't just reflect the training data: models can actually *amplify* existing stereotypes, making associations stronger in the embedding space than they are in the underlying text statistics.

If you build a search system that uses embeddings to match job candidates to job descriptions (and people do), these biases affect real people's livelihoods. When we build retrieval systems in Chapter 10, remember that the vectors carry the prejudices of their training data.

Let's finish with a visualization. We have 50-dimensional vectors, which we can't plot directly. But we can project them down to 2 dimensions using PCA (principal component analysis), which finds the two directions in the high-dimensional space that capture the most variation. The implementation involves linear algebra we won't cover here; what matters is the effect: we go from 50 dimensions to 2, keeping as much of the original structure as possible. We'll provide `pca_2d` as a helper function:

```
# embeddings.py
# ...existing code above
import matplotlib.pyplot as plt

def pca_2d(vecs):
    """Project vectors to 2 dimensions, preserving as much structure as possible."""
    centered = vecs - vecs.mean(axis=0)
    cov = np.cov(centered, rowvar=False)
    eigenvalues, eigenvectors = np.linalg.eigh(cov)
    top2 = eigenvectors[:, -2:][:, ::-1]
    return centered @ top2
```

Now let's pick some words from three categories and see if the embedding space groups them the way we'd expect:

```
# embeddings.py
# ...existing code above

word_groups = {
    "animals": ["cat", "dog", "fish", "bird", "horse", "cow", "sheep"],
    "countries": ["france", "germany", "italy", "spain", "japan", "china"],
    "emotions": ["happy", "sad", "angry", "afraid", "surprised"],
}

all_words = []
group_labels = []
for group, wds in word_groups.items():
    all_words.extend(wds)
    group_labels.extend([group] * len(wds))

vecs = np.array([get_vector(w) for w in all_words])
coords = pca_2d(vecs)
```

The plotting code puts each word on a 2D scatter plot, colored by category:

```
# embeddings.py
# ...existing code above

colors = {"animals": "steelblue", "countries": "coral", "emotions": "seagreen"}
plt.figure(figsize=(10, 8))
for i, (word, group) in enumerate(zip(all_words, group_labels)):
    plt.scatter(coords[i, 0], coords[i, 1], c=colors[group], s=50)
    plt.annotate(word, (coords[i, 0], coords[i, 1]),
                 fontsize=10, ha="center", va="bottom")
plt.title("Word embeddings projected to 2D (PCA)", fontsize=14)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig("clusters.png", dpi=150)
plt.show()
```

Look at the result. Animals cluster together. Countries cluster together. Emotions cluster together. Three groups of words, separated by meaning, visible in two dimensions despite being compressed from fifty. The structure is real: it exists in the full 50-dimensional space, and PCA just reveals a slice of it.

Let's do one more experiment. We said that "gender" is a consistent direction in the space. If that's true, the vectors connecting male-female word pairs (king□ queen, man□ woman, boy□ girl) should all be roughly parallel. Let's plot them:

```
# embeddings.py
# ...existing code above

gender_pairs = [
    ("king", "queen"), ("man", "woman"), ("boy", "girl"),
    ("he", "she"), ("his", "her"), ("brother", "sister"),
    ("father", "mother"), ("son", "daughter"),
]

male_words = [p[0] for p in gender_pairs]
female_words = [p[1] for p in gender_pairs]
all_gender = male_words + female_words

vecs = np.array([get_vector(w) for w in all_gender])
coords = pca_2d(vecs)
```

```
n = len(gender_pairs)
```

We project all 16 words (8 male, 8 female) to 2D, then draw a gray line connecting each pair:

```
# embeddings.py
# ...existing code above

plt.figure(figsize=(10, 6))
plt.scatter(coords[:n, 0], coords[:n, 1], c="steelblue", s=60, label="male")
plt.scatter(coords[n:, 0], coords[n:, 1], c="coral", s=60, label="female")

for i, (m, f) in enumerate(gender_pairs):
    plt.annotate(m, (coords[i, 0], coords[i, 1]),
                 fontsize=10, ha="center", va="bottom")
    plt.annotate(f, (coords[n+i, 0], coords[n+i, 1]),
                 fontsize=10, ha="center", va="bottom")
    plt.plot([coords[i, 0], coords[n+i, 0]],
             [coords[i, 1], coords[n+i, 1]], "gray", alpha=0.4, linewidth=1)

plt.title("Gender direction in embedding space", fontsize=14)
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig("gender_direction.png", dpi=150)
plt.show()
```

If the space learned a consistent gender direction, the gray lines should be roughly parallel. They won't be perfectly parallel (the space encodes many relationships at once, and 2D projection distorts things), but the trend should be visible. The analogy arithmetic works because the "gender" offset is approximately the same vector across many pairs: king - man + woman ≈ queen holds when that offset is consistent.

Let's quantify this. For each male-female pair, we compute the offset vector and normalize it to unit length:

$$\hat{d}_i = \frac{v_{\text{female}_i} - v_{\text{male}_i}}{\|v_{\text{female}_i} - v_{\text{male}_i}\|}$$

If the gender direction is perfectly consistent, every $\hat{d}_i$ would be the same vector, and the cosine similarity between any two of them would be 1.0. Let's check:

```
# embeddings.py
# ...existing code above

print("\n--- Gender direction consistency ---")
offsets = []
for m, f in gender_pairs:
    offset = get_vector(f) - get_vector(m)
    offsets.append(offset / np.sqrt(np.sum(offset ** 2)))

for i in range(len(offsets)):
    for j in range(i + 1, len(offsets)):
        sim = cosine_similarity(offsets[i], offsets[j])
        print(f"  {gender_pairs[i][0]:8s}→{gender_pairs[i][1]:8s}  vs  "
              f"{gender_pairs[j][0]:8s}→{gender_pairs[j][1]:8s}  : {sim:.3f}")
```

The pronoun pairs (he□ she, his□ her) are the most consistent, often above 0.9, because pronouns are almost pure markers of gender with little other semantic content. Family pairs (father□ mother, son□ daughter) are also strong. The king□ queen pair is less consistent with others, because "king" and "queen" carry additional associations (royalty, power, history) beyond gender. The direction is real but approximate, which is why analogy arithmetic works well on some relationships and poorly on others.

That's 400,000 words, 50 dimensions each, encoding meaning, relationships, biases, and structure, all learned from raw text. But each word is stuck with one vector. In Chapter 4, we'll see how transformers fix this: instead of one fixed vector per word, they compute a new vector for every word in every context. "Bank" will finally get different representations in "river bank" and "bank account."

3.7 Chapter Summary

- One-hot encoding maps words to vectors but makes every word equidistant—it encodes identity, not meaning
- The distributional hypothesis: words that appear in similar contexts have similar meanings
- Word2Vec learns dense embeddings by predicting context words from targets, using dot products, sigmoid, and negative sampling
- Embeddings encode structured relationships: vector arithmetic like $\vec{king} - \vec{man} + \vec{woman} \approx \vec{queen}$ works without supervision
- Cosine similarity compares direction rather than magnitude—the right metric for embeddings
- Embeddings inherit the biases of their training text, and can amplify them
- Antonyms end up close together because they share contexts—distributional similarity is not the same as semantic equivalence
- Each word gets one fixed vector regardless of context, which Chapter 4's transformers will fix

In the next chapter, we see how attention mechanisms compute a fresh representation for every word in every context, giving us the transformer architecture behind modern language models.

3.8 Exercises

1. Implement a **word intrusion** detector. Given a list like `["cat", "dog", "fish", "car", "bird"]`, find the outlier: the word least similar to the rest. For each word, compute its average cosine similarity to every other word in the list. The word with the lowest average is the odd one out. Test it on easy lists first, then try harder cases: `["breakfast", "lunch", "dinner", "snack", "supper"]`. Does the model agree with your intuition about which meal term is least typical? What about `["walk", "run", "sprint", "jog", "crawl"]`?

2. The chapter noted that “happy” and “sad” end up close together despite being opposites. Investigate this systematically. Here are 10 synonym pairs and 10 antonym pairs:

   ```
   synonyms = [
       ("happy", "joyful"), ("big", "large"), ("fast", "quick"),
       ("smart", "clever"), ("strong", "powerful"), ("beautiful", "gorgeous"),
       ("small", "tiny"), ("start", "begin"), ("end", "finish"), ("old", "ancient"),
   ]
   antonyms = [
       ("happy", "sad"), ("big", "small"), ("fast", "slow"),
       ("smart", "stupid"), ("strong", "weak"), ("beautiful", "ugly"),
       ("hot", "cold"), ("love", "hate"), ("old", "young"), ("rich", "poor"),
   ]
   ```

   Compute cosine similarity for each pair. Plot two histograms on the same axes (use `plt.hist` with `alpha=0.5` so they overlap visually). How much do the distributions overlap? If you were building a system that needed to distinguish synonyms from antonyms (say, a contradiction detector), could you do it from cosine similarity alone? What additional signal would you need?

3. Test analogy reliability systematically. Here are 15 country-capital pairs:

   ```
   capitals = [
       ("france", "paris"), ("germany", "berlin"), ("japan", "tokyo"),
       ("italy", "rome"), ("spain", "madrid"), ("russia", "moscow"),
       ("china", "beijing"), ("egypt", "cairo"), ("brazil", "brasilia"),
       ("india", "delhi"), ("canada", "ottawa"), ("australia", "canberra"),
       ("sweden", "stockholm"), ("greece", "athens"), ("turkey", "ankara"),
   ]
   ```

   For each pair, hold it out and use the remaining pairs to establish the pattern: `analogy(held_out_country, "france", "paris")`. Score 1 if the correct capital is the top result, 0 otherwise. What percentage work? Now do the same with these verb tense pairs:

   ```
   tenses = [
       ("walk", "walked"), ("run", "ran"), ("eat", "ate"),
       ("swim", "swam"), ("write", "wrote"), ("speak", "spoke"),
       ("sing", "sang"), ("think", "thought"), ("buy", "bought"),
       ("drive", "drove"), ("give", "gave"), ("take", "took"),
       ("see", "saw"), ("know", "knew"), ("make", "made"),
   ]
   ```

   Which type of relationship is more reliably encoded? Why might regular grammatical patterns (walk□ walked) be more consistent than geographic facts (france□ paris)?

4. Build a **simple sentence similarity function**. Represent each sentence as the average of its word vectors (skip words not in the vocabulary). Given a query, compute its average embedding and find the most similar sentence by cosine similarity. Try a small corpus of 10-20 sentences and see if “How do I fix a flat tire?” matches better with “Repairing a punctured wheel” than with “I ate breakfast this morning.” Then try “The dog bit the man” versus “The man bit the dog.” These sentences have identical words, so their average embeddings are identical. This is a serious flaw:

the representation throws away word order entirely. Think about what information you'd need to preserve to distinguish them.

5. Simulate what Word2Vec training does, without building the full model. Start with random 2D vectors for six words:

   ```
   import numpy as np
   np.random.seed(42)
   words = ["cat", "dog", "pet", "car", "drive", "road"]
   vecs = {w: np.random.randn(2) * 0.1 for w in words}
   ```

   Define three positive pairs (words that appear together: `("cat", "pet")`, `("dog", "pet")`, `("car", "drive")`) and let every other combination be a negative pair. Write a training loop: for each positive pair, nudge their vectors closer (add a small fraction of each vector to the other). For each negative pair, nudge them apart (subtract). Run 200 iterations with a learning rate of 0.01. Plot the vectors before and after training. Do "cat," "dog," and "pet" end up clustered? Do "car," "drive," and "road" form a separate cluster? This is the core of what Word2Vec does: push co-occurring words together, push random pairs apart.

6. The `most_similar` function handles one word at a time. What if you need the neighbors of 100 words? Calling it in a loop works, but there's a faster way: compute all 100 queries at once with a single matrix multiply. `normalized[query_indices]` grabs the rows for your query words, giving you a matrix of shape `(100, 50)`. Multiply it by the full normalized matrix transposed: `(normalized @ normalized[query_indices].T)` gives you a `(400000, 100)` matrix of all pairwise similarities in one operation. Time this batch version against calling `most_similar` in a loop for 100 random words. How much faster is it? This is the core trick behind vector databases, which we'll build with in Chapter 10.

Abu-Mostafa, Yaser S., Malik Magdon-Ismail, and Hsuan-Tien Lin. 2012. *Learning from Data*. AMLBook.

Bengio, Yoshua, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. "A Neural Probabilistic Language Model." *Journal of Machine Learning Research* 3: 1137–55.

Bojanowski, Piotr, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. "Enriching Word Vectors with Subword Information." *Transactions of the Association for Computational Linguistics* 5: 135–46.

Bolukbasi, Tolga, Kai-Wei Chang, James Y. Zou, Venkatesh Sadasivam, and Adam T. Kalai. 2016. "Man Is to Computer Programmer as Woman Is to Homemaker? Debiasing Word Embeddings." In *Advances in Neural Information Processing Systems*. Vol. 29.

Cauchy, Augustin-Louis. 1847. "Méthode générale Pour La résolution Des Systèmes d'équations Simultanées." *Comptes Rendus de l'Académie Des Sciences* 25: 536–38.

Choromanska, Anna, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. 2015. "The Loss Surfaces of Multilayer Networks." In *Proceedings of the 18th International Conference on Artificial Intelligence and Statistics*, 192–204.

Cover, Thomas, and Peter Hart. 1967. "Nearest Neighbor Pattern Classification." *IEEE Transactions on Information Theory* 13 (1): 21–27.

Cybenko, George. 1989. "Approximation by Superpositions of a Sigmoidal Function." *Mathematics of Control, Signals and Systems* 2 (4): 303–14.

Dauphin, Yann N., Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. 2014. "Identifying and Attacking the Saddle Point Problem in High-Dimensional Non-Convex Optimization." In *Advances in Neural Information Processing Systems*. Vol. 27.

Firth, John R. 1957. "A Synopsis of Linguistic Theory, 1930–1955." In *Studies in Linguistic Analysis*, 1–32. Oxford: Blackwell.

Glorot, Xavier, and Yoshua Bengio. 2010. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, 249–56.

Goldberg, Yoav. 2017. *Neural Network Methods for Natural Language Processing*. Morgan & Claypool Publishers.

Hornik, Kurt. 1991. "Approximation Capabilities of Multilayer Feedforward Networks." *Neural Networks* 4 (2): 251–57.

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE* 86 (11): 2278–2324.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Efficient Estimation of Word Representations in Vector Space." *arXiv Preprint arXiv:1301.3781*.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013. "Distributed Representations of Words and Phrases and Their Compositionality." In *Advances in Neural Information Processing Systems*. Vol. 26.

Mikolov, Tomas, Wen-tau Yih, and Geoffrey Zweig. 2013. "Linguistic Regularities in Continuous Space Word Representations." In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 746–51.

Nair, Vinod, and Geoffrey E. Hinton. 2010. "Rectified Linear Units Improve Restricted Boltzmann Machines." In *Proceedings of the 27th International Conference on Machine Learning*, 807–14.

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. 2014. "GloVe: Global Vectors for Word Representation." In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 1532–43.

Rosenblatt, Frank. 1958. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." *Psychological Review* 65 (6): 386–408.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323 (6088): 533–36.

Zhao, Jieyu, Tianlu Wang, Mark Yatskar, Vicente Ordonez, and Kai-Wei Chang. 2017. "Men Also Like Shopping: Reducing Gender Bias Amplification Using Corpus-Level Constraints." In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2979–89.