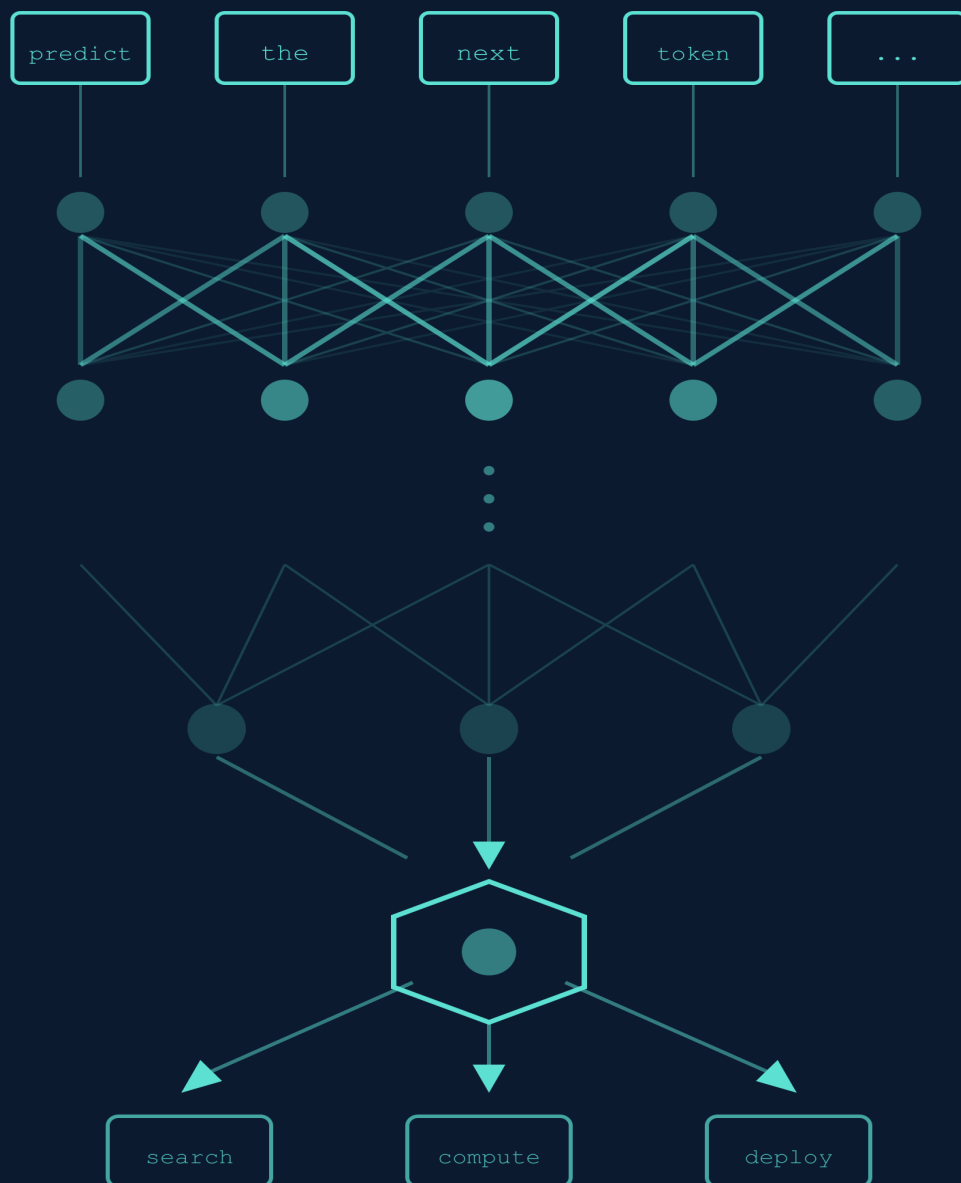


Practical Language Models ■

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	3
About This Book	3
What You'll Build	3
How to Use This Book	4
Prerequisites	4
Book Structure	5
Contributors	5
1 The Learning Problem	6
1.1 What Does It Mean to Learn?	6
1.2 Supervised and Unsupervised Learning	7
1.3 Two Kinds of Problems	9
1.4 From Pixels to Predictions	10
1.5 How Good Is Good Enough?	14
1.6 Generalization	18
1.7 Hands-On: Recognizing Handwritten Digits	20
1.8 Chapter Summary	26
1.9 Exercises	27
2 Learning with Neural Networks	28
3 From Words to Numbers	29
3.1 The Text Representation Problem	29
3.2 One-Hot Encoding	30
3.3 The Distributional Hypothesis	32
3.4 Word Embeddings and Word2Vec	34
3.5 Cosine Similarity	37
3.6 Hands-On: Exploring Pre-Trained Embeddings	40
3.7 Chapter Summary	49
3.8 Exercises	49

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. If you skip the math, you'll still know 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 calculus (derivatives, chain rule)
- Willingness to 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 $f : X \rightarrow Y$ 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.

Our job is to use D to find a function $g : X \rightarrow Y$ 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.

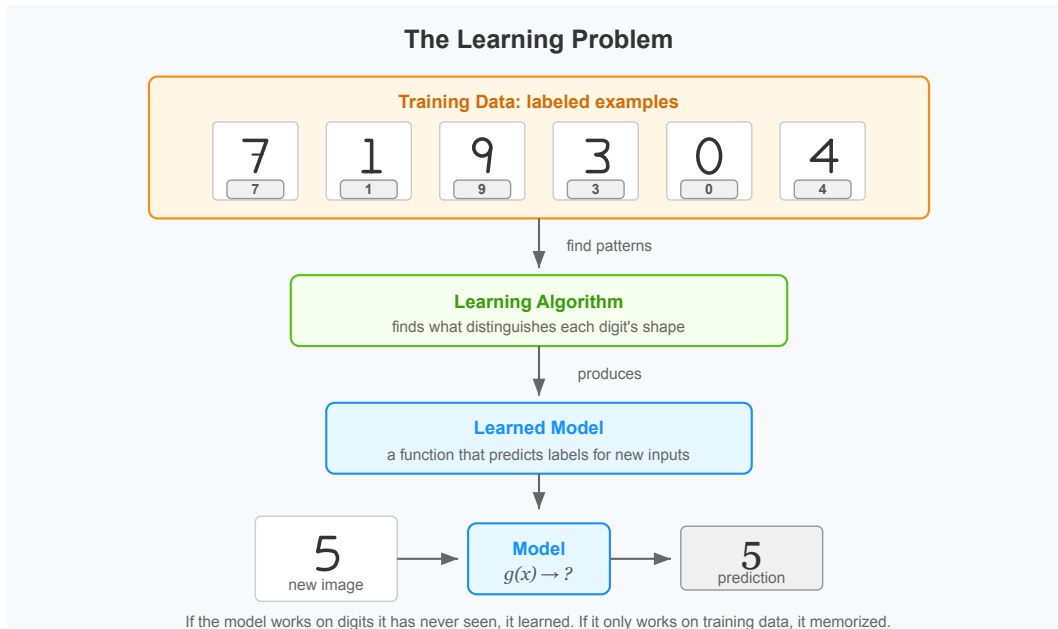


Figure 1.1: The Learning Problem

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.

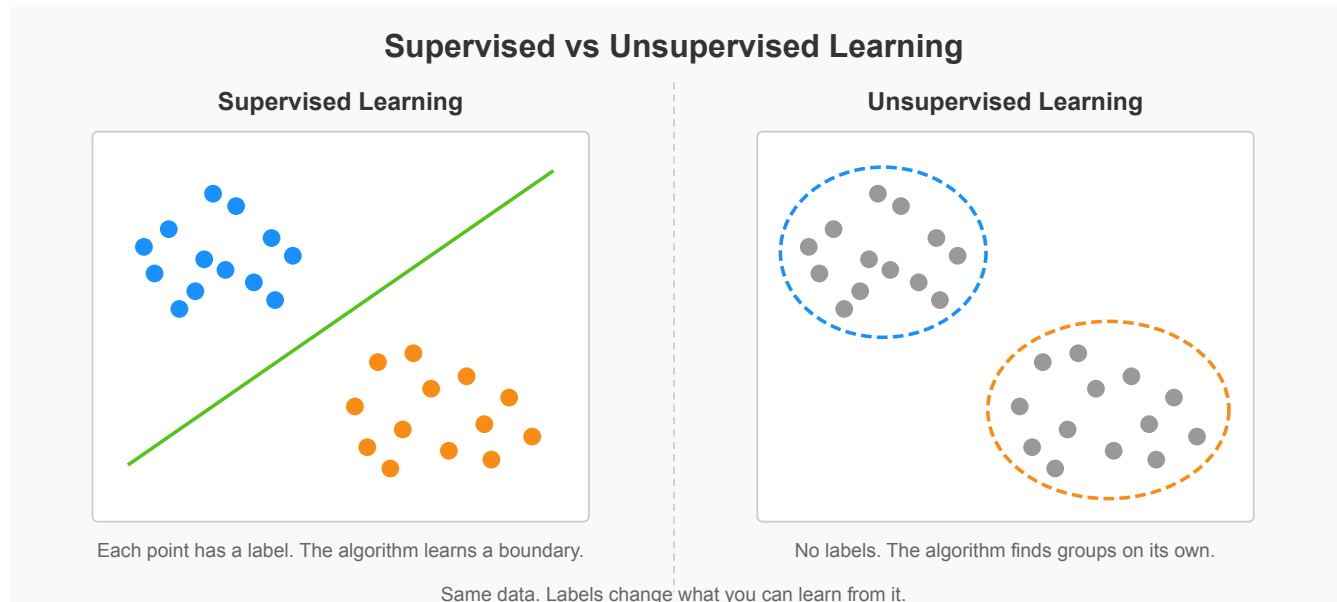


Figure 1.2: Supervised vs Unsupervised Learning

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.

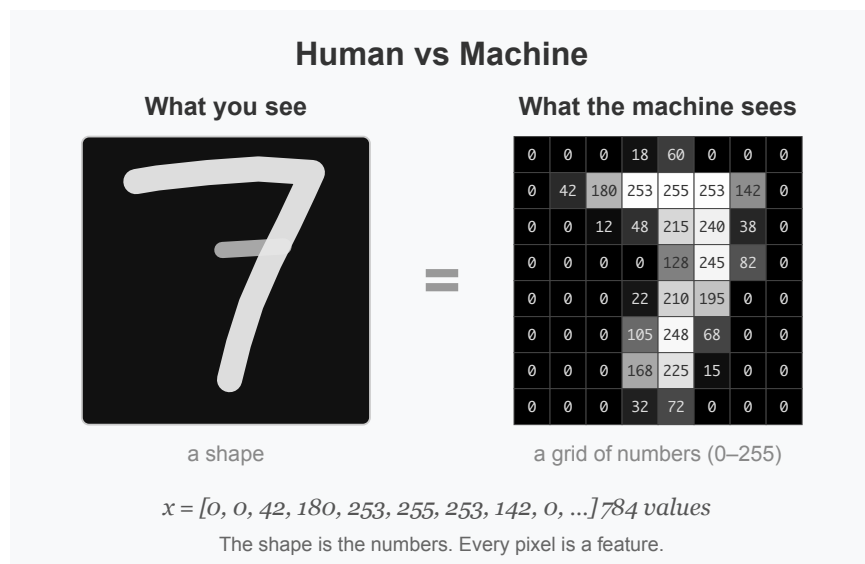


Figure 1.4: What the Machine Sees

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

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:

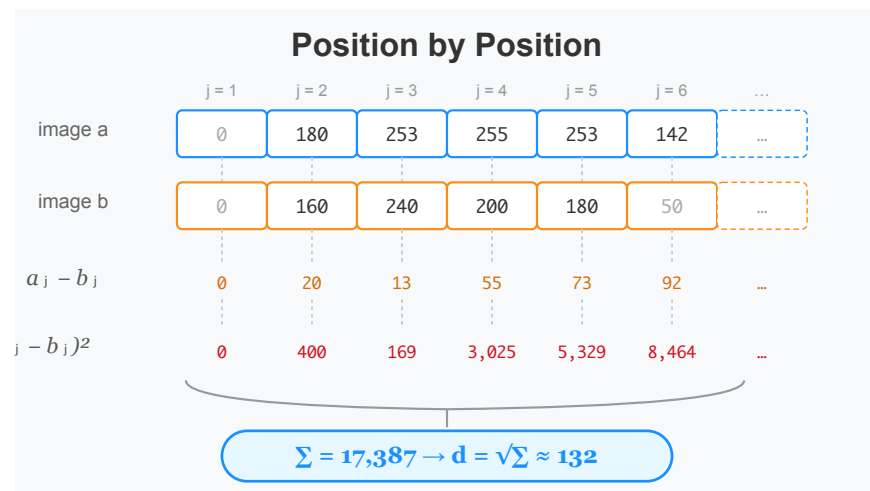


Figure 1.5: Position by Position

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.

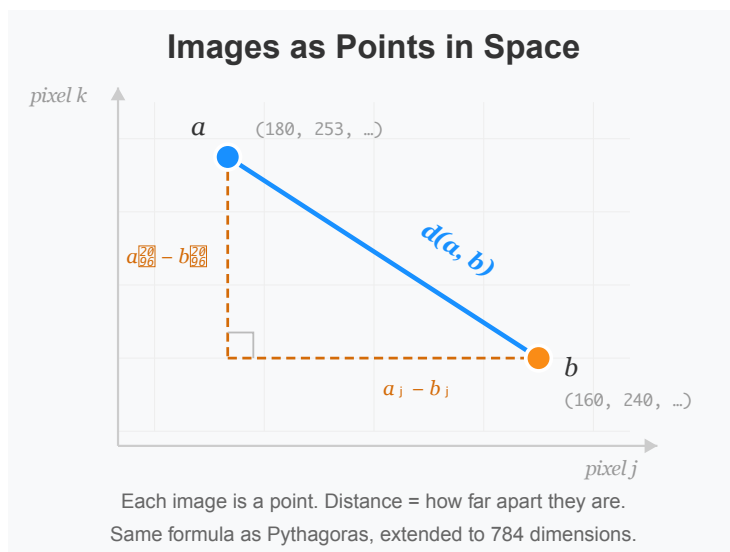


Figure 1.6: Images as Points in Space

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**.

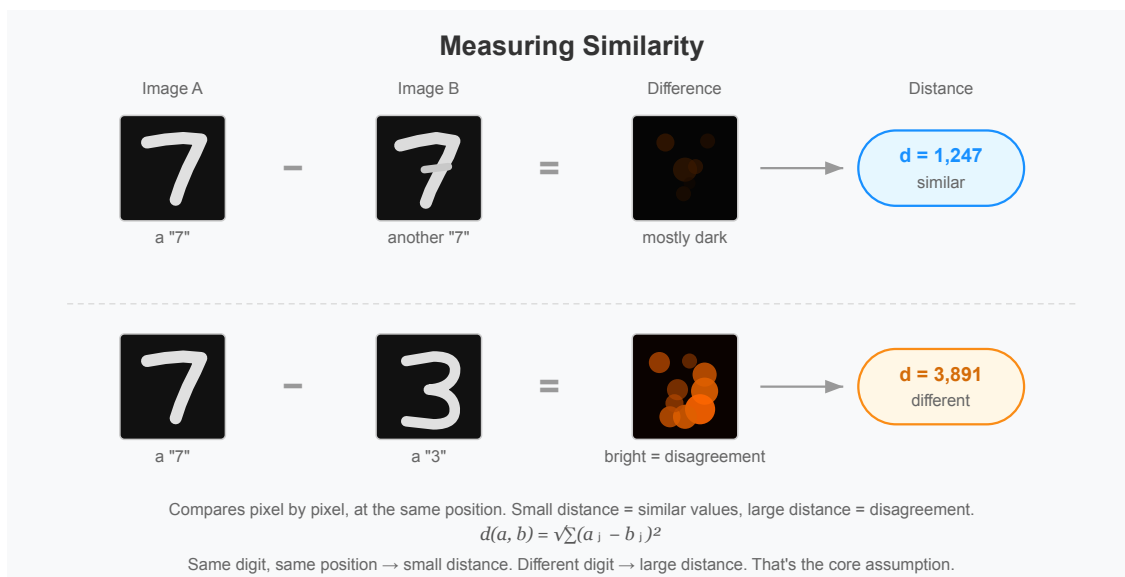


Figure 1.7: Measuring Similarity

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 $56,000 \times 14,000 = 784$ 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.

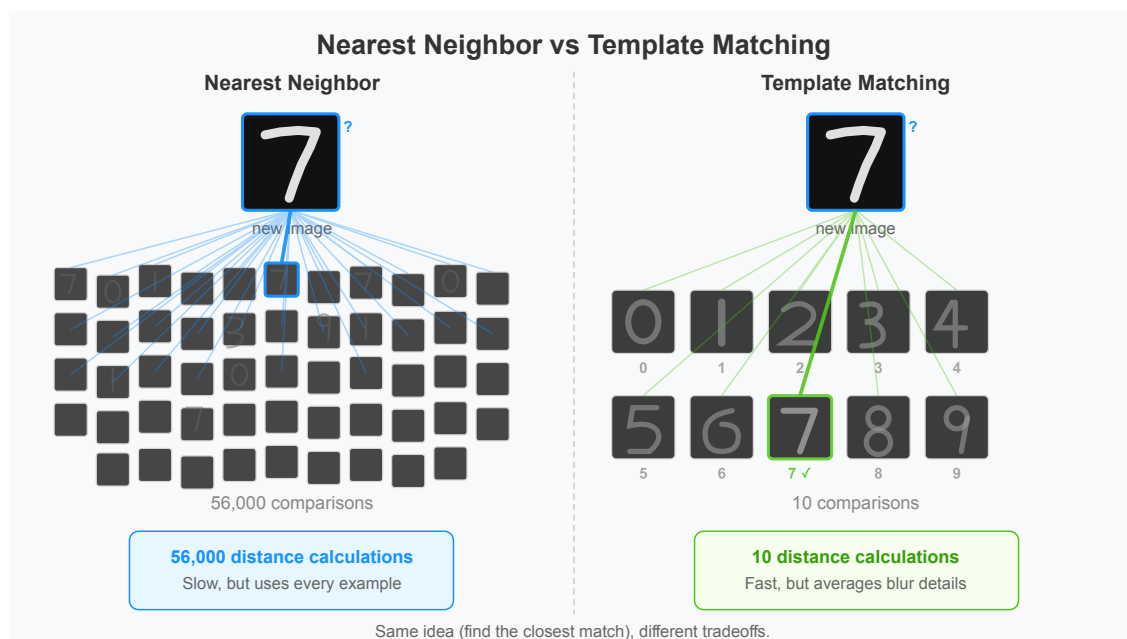


Figure 1.8: Nearest Neighbor vs Template Matching

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 percep-

tible 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.

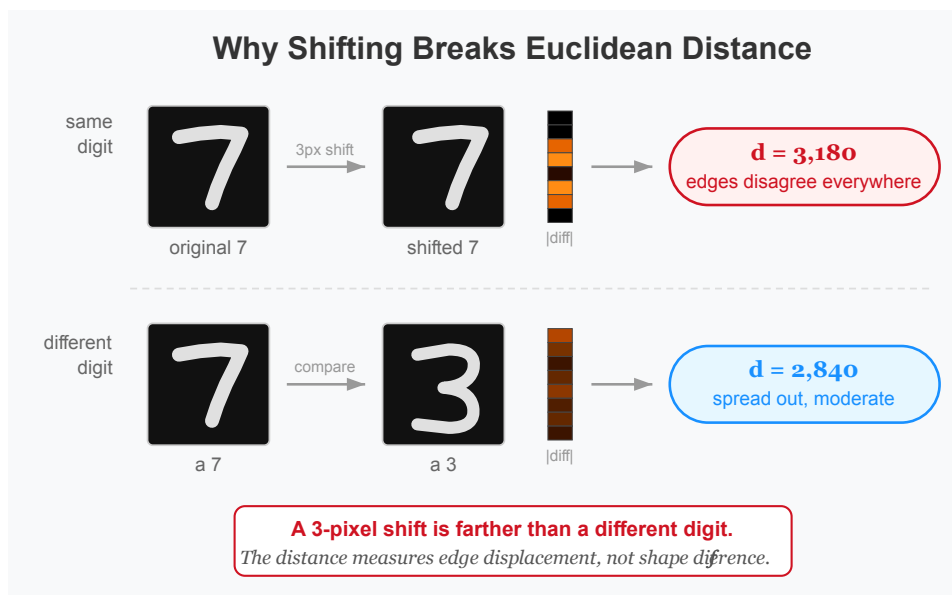


Figure 1.9: Why Shifting Breaks Euclidean Distance

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.

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):

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

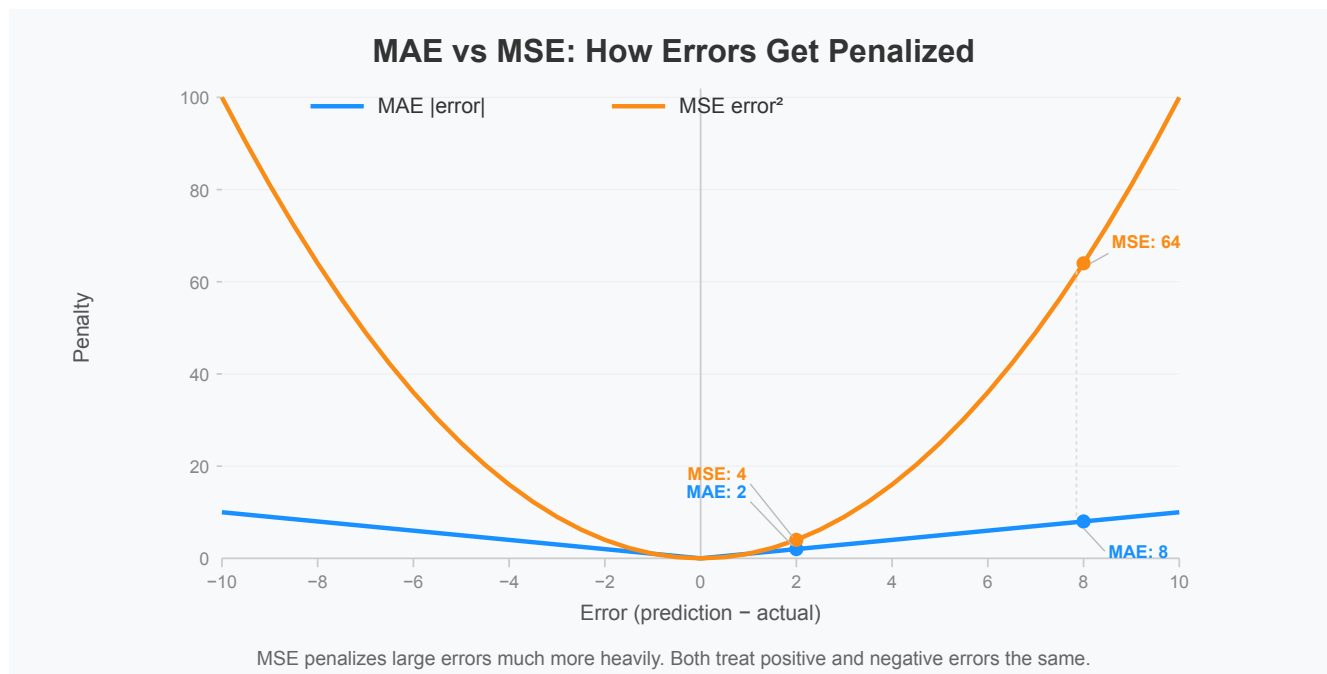


Figure 1.10: MAE vs MSE: How Errors Get Penalized

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.

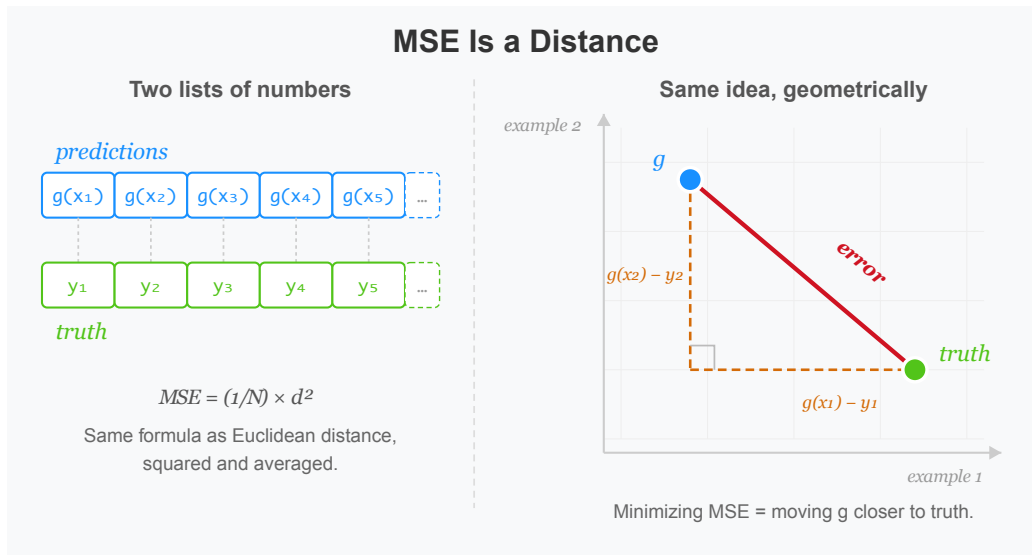


Figure 1.11: MSE Is a Distance

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.

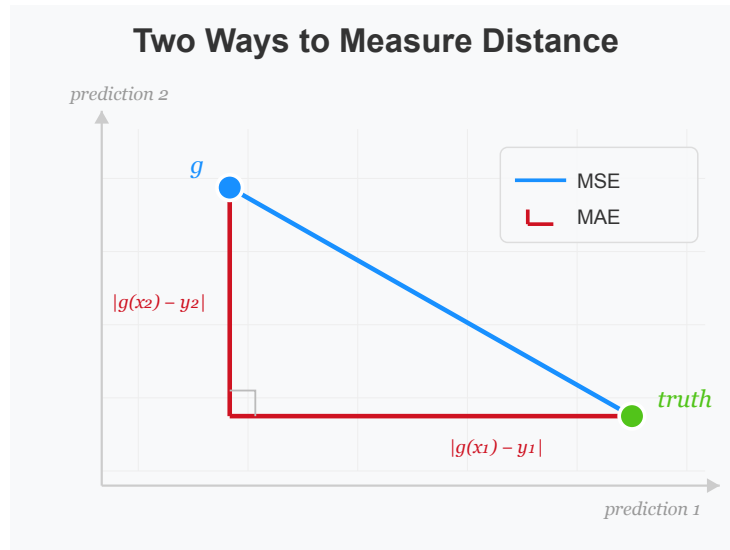


Figure 1.12: Two Ways to Measure Distance

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.

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.

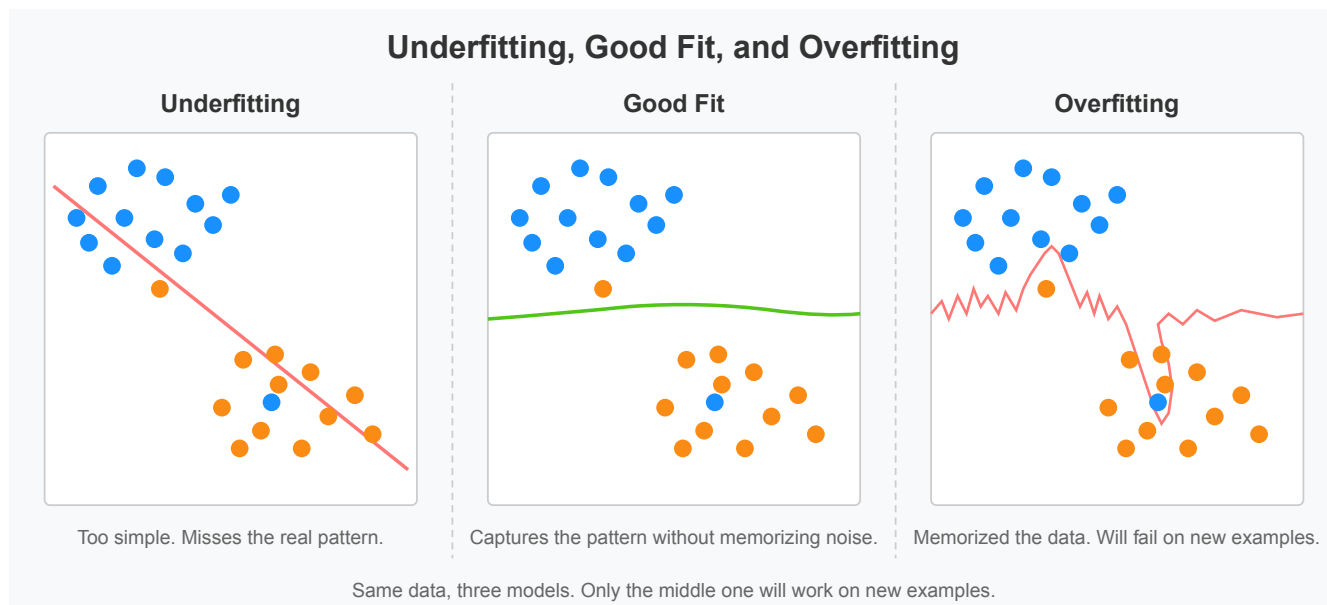


Figure 1.13: Underfitting, Good Fit, and Overfitting

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.

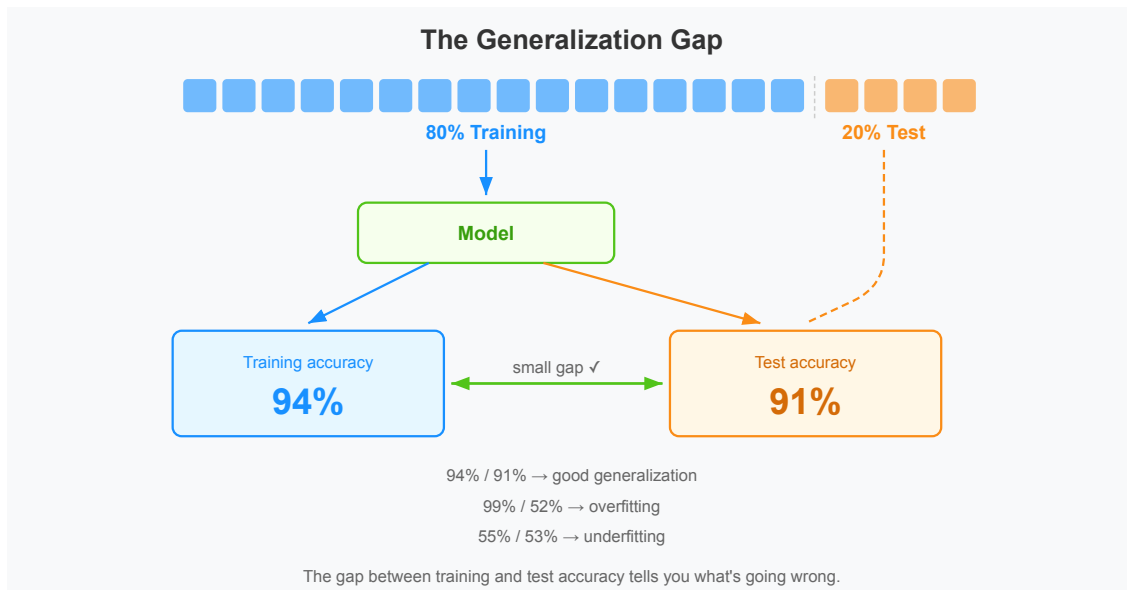


Figure 1.14: The Generalization Gap

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.

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: 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:

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

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

```
# 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:

```
# 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:

```
# 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 with Neural Networks

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."

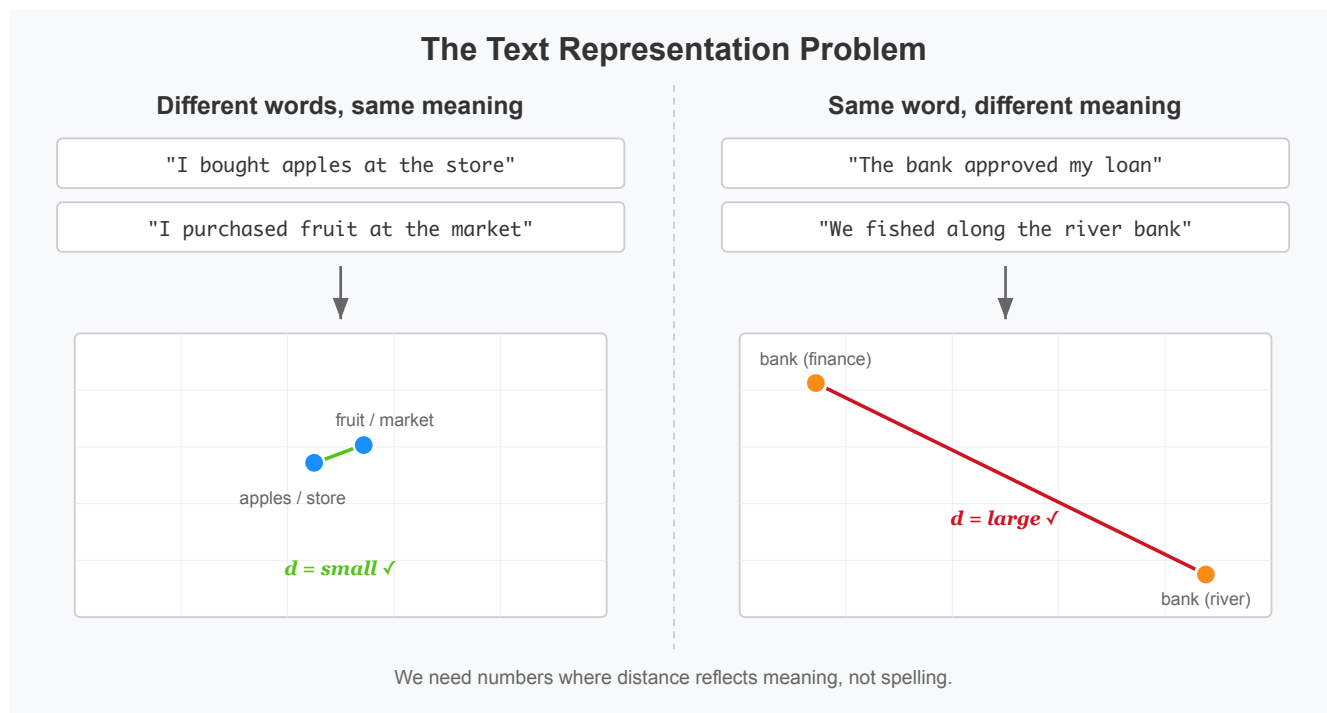


Figure 3.1: The Text Representation Problem

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:

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

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

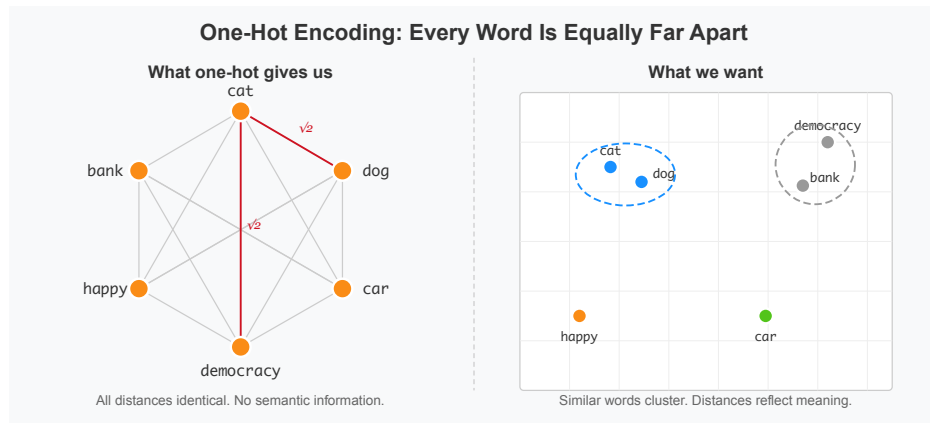


Figure 3.2: One-Hot Encoding: Every Word Is Equally Far Apart

"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."

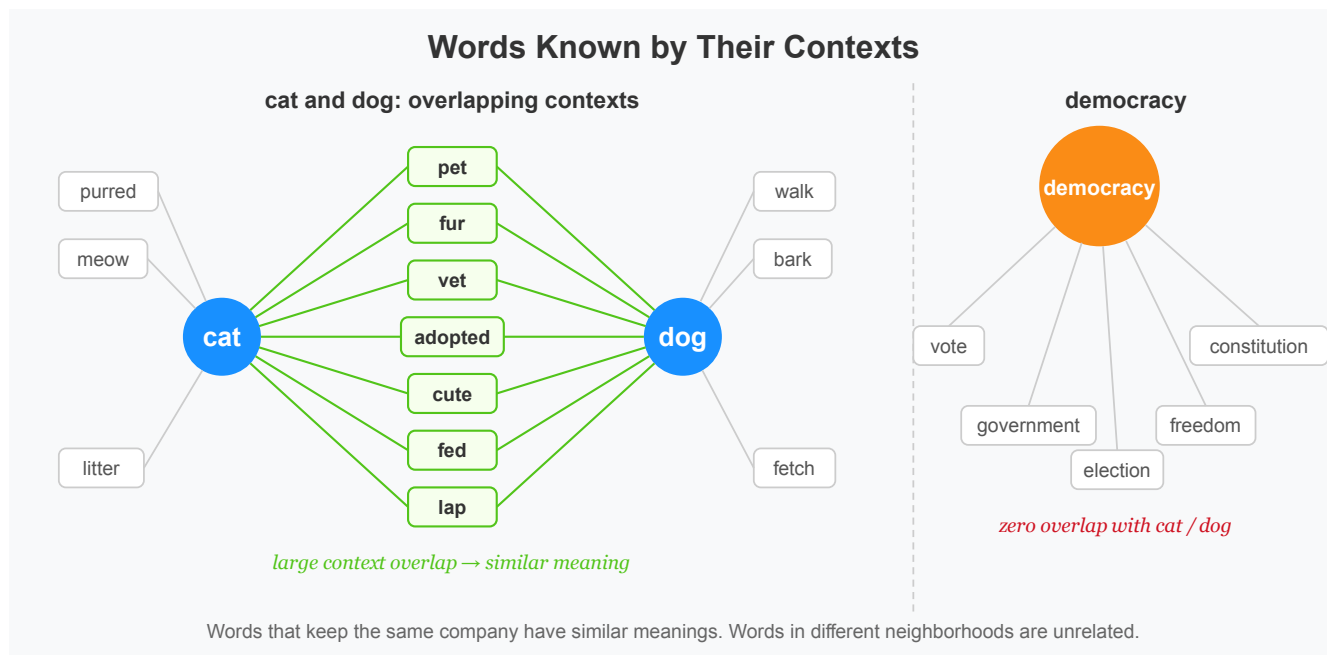


Figure 3.3: Words Known by Their Contexts

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”:

context: [the, cold, water, into, the]
target

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.)



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:

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

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.”

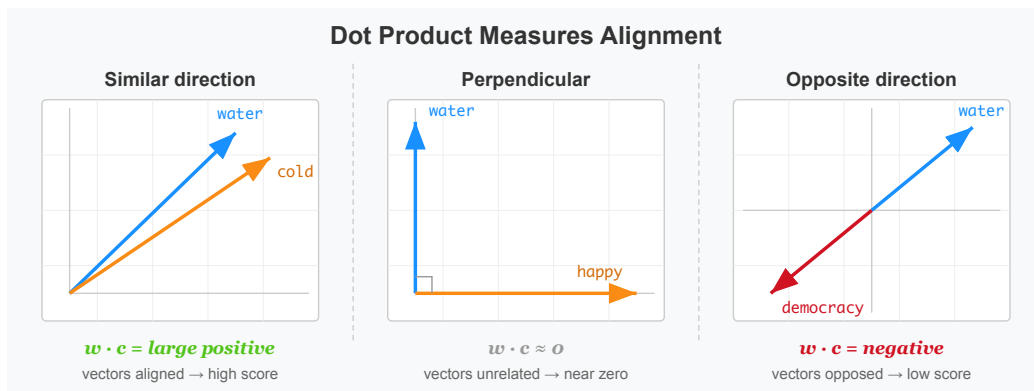


Figure 3.5: Dot Product Measures Alignment

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):

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

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.

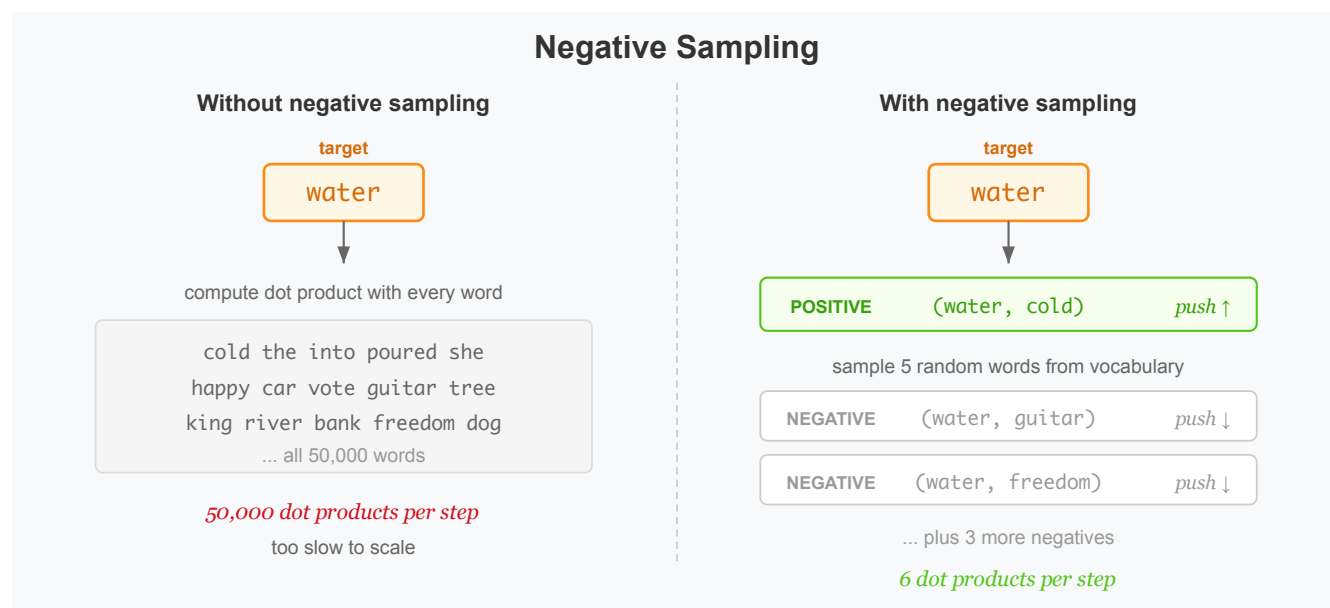


Figure 3.6: Negative Sampling: Score Real Pairs Against Random Ones

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.”

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

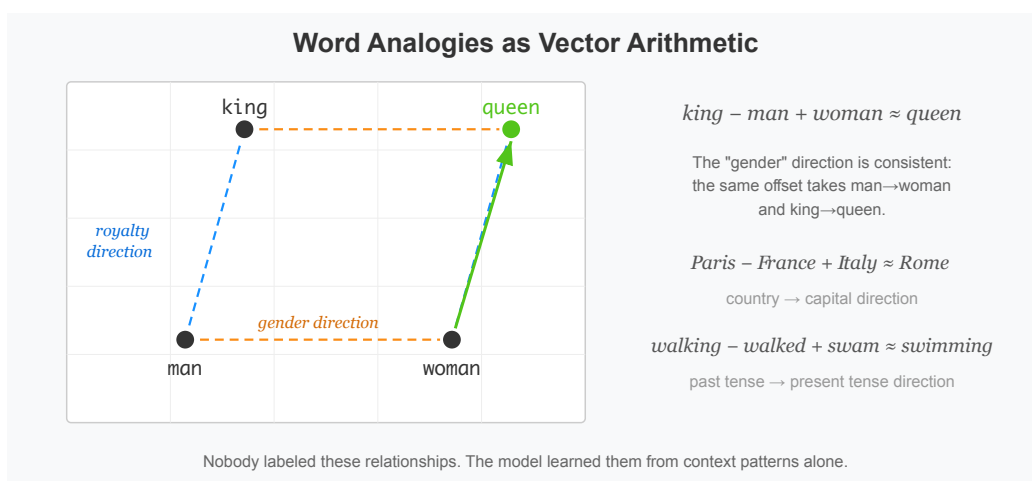


Figure 3.7: Word Analogies as Vector Arithmetic

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.

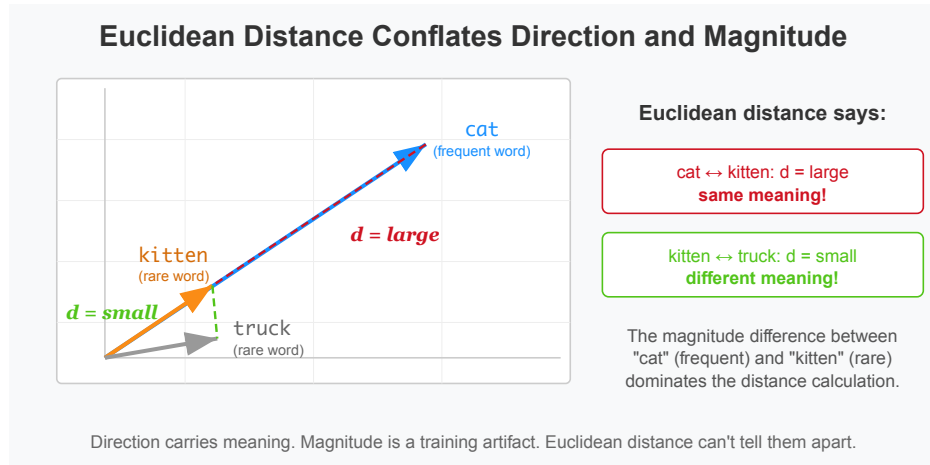


Figure 3.8: Euclidean Distance Conflates Direction and Magnitude

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**:

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

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).

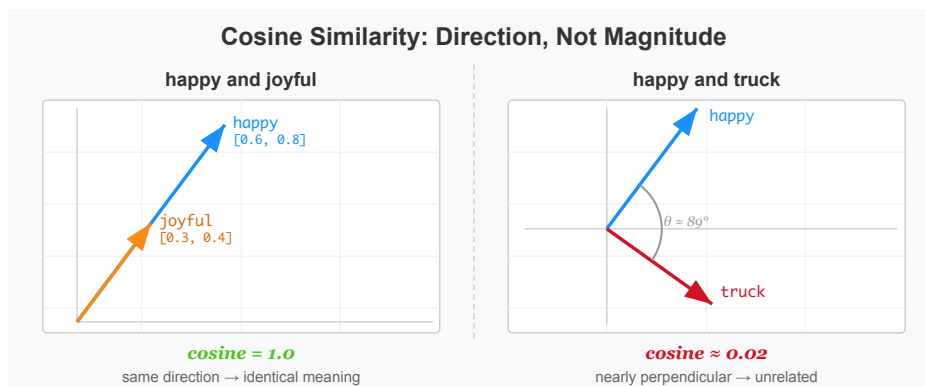


Figure 3.9: Cosine Similarity: Direction, Not Magnitude

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, a method closely related to Word2Vec that trains on global word co-occurrence statistics rather than local windows), 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)
    norm_a = np.sqrt(np.sum(a ** 2))
    norm_b = np.sqrt(np.sum(b ** 2))
    if norm_a == 0 or norm_b == 0:
        return 0.0
    return dot / (norm_a * norm_b)

# sum of a_i * b_i
# ||a||
# ||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 about what distributional embeddings actually capture: *relatedness*, not just *similarity*. “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. The vectors can’t distinguish “means the same thing” from “means the opposite thing.” Keep this in mind when we build retrieval systems in Chapter 10.

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:

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

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

    # L2-normalize every word vector:  $\hat{v} = v / ||v||$ 
    norms = np.sqrt(np.sum(vectors ** 2, axis=1))
    norms[norms == 0] = 1.0
    normalized = vectors / norms[:, np.newaxis]

    # L2-normalize the query vector
    word_norm = np.sqrt(np.sum(vec ** 2))
    word_normalized = vec / word_norm

    # Dot product of unit vectors = cosine similarity
    similarities = normalized @ word_normalized

    # Sort by similarity (highest first), skip the word itself
    top_indices = np.argsort(similarities)[::-1][1:n+1]
    return [(words[i], similarities[i]) for i in top_indices]
```

A few relevant things: `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 every vector is unit length, `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:

```
# 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 :

$$\text{answer} = \arg \max_w \text{cosine_similarity}(v_w, t)$$

```
# 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 []

    # Compute the target: t = b - a + c
    target = vb - va + vc

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

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

    # Find the closest words to the target
    similarities = normalized @ target_normalized

    # Exclude the three input words from results
    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: $\text{king} - \text{man} + \text{woman} \approx \text{queen}$ 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 $\vec{\text{king}} - \vec{\text{man}} + \vec{\text{woman}} \approx \vec{\text{queen}}$ 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` \square `walked`) be more consistent than geographic facts (`france` \square `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 recomputes normalized vectors on every call, scanning all 400,000 words each time. Measure how long a single call takes with `time.time()`. Now precompute the normalized matrix once, outside the function, and rewrite `most_similar` to use it. How much faster is the precomputed version? For a bigger speedup, compute similarities for multiple query words at once: `normalized @ normalized[query_indices].T` gives you all pairwise similarities between your queries and the full vocabulary in one matrix multiply. Time this batch version against calling the single-word function in a loop. This is the core trick behind vector databases, which we'll build with in Chapter 10.

- Bengio, Yoshua, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 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.
- Firth, John R. 1957. "A Synopsis of Linguistic Theory, 1930–1955." In *Studies in Linguistic Analysis*, 1–32. Oxford: Blackwell.
- 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.
- 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.
- 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.