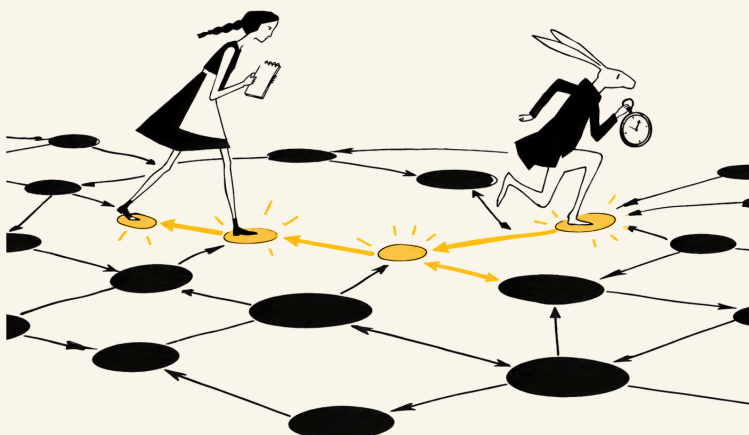


Alice's Adventures in a Differentiable Wonderland



Simone Scardapane · 2024

Illustrated audio lectures

Lecturized

Alice's Adventures in a Differentiable Wonderland

Illustrated audio lectures

Simone Scardapane · 2024

Retold as illustrated lectures by Lecturized
lecturized.com

From the original: Alice's Adventures in a Differentiable Wonderland, Volume I: A Tour
of the Land

Contents

Why Look Under the Hood

Learning as a Search

Old Ideas, New Power

Boxes of Numbers and Their Shapes

What This Book Is, and Is Not

Cutting Slices from a Tensor

Arrows, Lengths and Matching

Matrices as Batches and as Functions

How the Cost Grows

Many Products at Once, and a Shorthand for All of Them

The Slope at a Point

Slopes in Many Directions

Hidden Sizes, and the First Step Downhill

Steepest Descent, and Where It Ends

Bowls, Bumpy Landscapes and a Push of Momentum

The Price of Memory, and Where to Practise

A Practice Landscape, and What a Dataset Is

Learning Without Labels, and Reusing What Was Learned

Scoring Mistakes: the Loss

Losses That Can Be Followed, and Models That Only Remember

Predicting a Distribution Instead of a Number

Making the Data Likely, and Keeping Every Good Model

Approximations, Priors as Penalties, and Learning in Instalments

Why Look Under the Hood

Open the photo app on your phone and type a single word into its search box, say "dog". In a moment it shows you every picture with a dog in it, though nobody ever labelled your photos by hand. Now open a chat assistant and ask it to explain a recipe in simpler words; it answers in fluent sentences. Both tricks, the book tells us, are done by neural networks, and both are built from a surprisingly small set of guiding principles and components. Those few pieces are what this whole course is about.

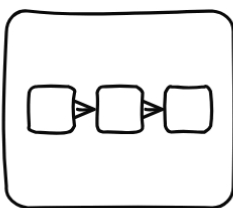
PHOTO SEARCH



CHAT



NETWORK



SCALE
UP



FEW PARTS

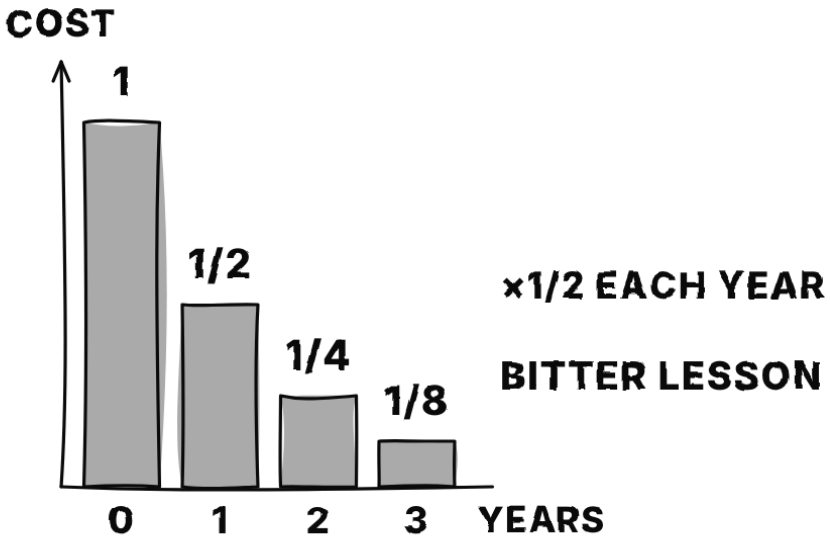


Two Apps, One Set of Parts

Neural networks have become part of everyday life, sometimes openly, as large language models, or LLMs, and sometimes hidden from view. They power drones, cars, search engines, the design of new molecules and the recommendations of what to watch next. The book's first claim is that all of this rests on very few ideas, and that the research effort has shifted from inventing new ones to scaling the old ones up to the limits of what is physically possible.

That shift has a name: neural scaling laws. Informally, for practically any task, increasing the data, the computing power and the size of the model all at once almost always gives a predictable increase in accuracy. Said the other way round, the computing power needed to reach a given accuracy shrinks by a constant factor every period of time. Here is what a constant factor means with numbers we can check: if the cost of reaching some accuracy halves every year, then after three years it is one half times one half times one half, which is one eighth of what it was. The researcher Richard Sutton called this the bitter lesson: simple, general-purpose tools combined with exponentially growing computing power keep winning over cleverer special designs.

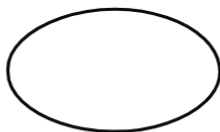
SCALING LAWS



Cost Halves Every Year

If we take scaling laws as given, the book says, we are left with an almost magical tool. In a nutshell, a neural network is optimized to approximate some probability distribution, given data drawn from it. Think of the distribution as the hidden rule that produces all the dog photos in the world; the network sees a sample of them and tries to capture the rule. In principle this can fail: modern networks are so large that they could simply memorize every example they are shown and become a trivial look-up table. Instead, trained models turn out to generalize well, even to tasks that were never explicitly in their training data. As datasets grow, the line between what is inside the training distribution and what is outside it blurs, and very large models show hints of strong generalization with surprisingly little reliance on pure memorization, which is called overfitting.

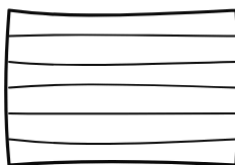
HIDDEN RULE



SAMPLE



NETWORK



OVERFIT

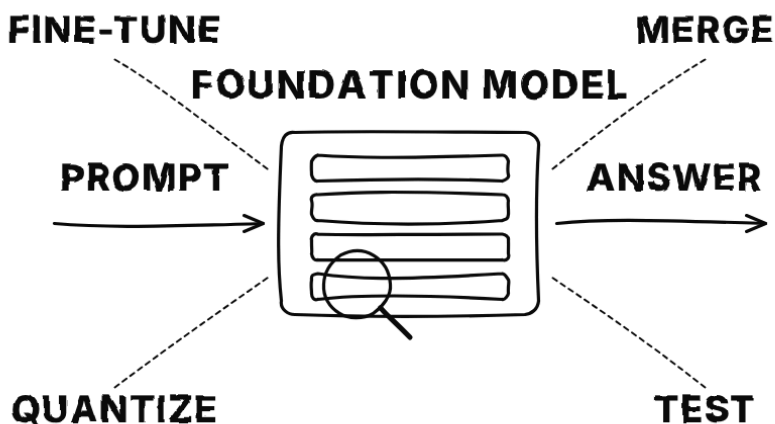
NEW



GENERALIZE

Samples of a Hidden Rule

Extremely large models that can be reused for many later tasks are sometimes called foundation models. Together with a lively open-source community, they have changed how we use these systems. Many tasks can now be solved simply by prompting, that is, by giving text or picture instructions to a pre-trained model found on the web, while its insides stay a complete black box. The book compares this to the move from writing your own libraries in a language such as C++ to relying on finished software whose source code you never see. The comparison is not far-fetched: today only a few teams in the world have the computing power and the expertise to design and release truly large models such as the Llama family of language models, just as only a few companies can build large enterprise software.



Black Box, Then Opened

But just as open-source code lets you customize programs or write new ones from scratch, ordinary consumer hardware and a bit of ingenuity give you a wide range of things to try with these models. You can fine-tune them for your own task, merge two models into one, quantize them, which means storing their numbers with fewer bits so they run on low-power hardware, test how robust they are, or design entirely new variants. For all of that you need to look under the hood and understand how these models process and transform data inside, with all the tricks and quirks that were learned from experience and debugging. The book presents itself as an entry point into that world, and it addresses its reader as someone who, like Alice, is naturally curious.

Try it yourself

Ask the same chat assistant the same question twice, once in a short plain sentence and once with a few extra instructions about tone and length. Expect the two answers to differ even though the model itself did not change at all. What changed is only the prompt; the network inside is the same black box, which is exactly the way of working the book describes, and the reason it wants you to open the box.

Remember

Neural networks now run search, photos, chat assistants and much more, yet they are all built from a small set of principles and components. Neural scaling laws say that growing data, computing power and model size together almost always raises accuracy predictably, so the cost of a given accuracy keeps shrinking by a steady factor; Sutton called this the bitter lesson. A network is optimized to approximate a probability distribution from samples, and although it could memorize, large trained models generalize surprisingly well. Foundation models are often used only by prompting, as black boxes, but customizing, fine-tuning, quantizing or inventing models requires understanding what happens inside, and that is the journey the book offers.

Questions

1. According to neural scaling laws, what happens when data, computing power and model size are all increased together?
 - a. Accuracy almost always rises in a predictable way
 - b. Accuracy rises only for tasks the model was designed for
 - c. Accuracy stops improving once the model can memorize the data
2. Why could a modern neural network, in principle, fail to learn the rule behind its data?
 - a. Because it never sees enough examples to fit them
 - b. Because it is large enough to memorize every example and become a look-up table
 - c. Because a probability distribution cannot be approximated from samples
3. Using a foundation model only by prompting is compared to what?
 - a. Writing your own libraries from scratch in a language like C++
 - b. Repairing the source code of a program you downloaded
 - c. Relying on finished software whose source code you cannot see
4. Which of these tasks requires looking under the hood of a model rather than just prompting it?
 - a. Quantizing it so that it runs on low-power hardware

- b. Asking it to rewrite a recipe in simpler words
- c. Rephrasing a question to get a different answer

Answers

- 1. **a)** For practically any task, growing data, compute and model size at once almost always gives a predictable increase in accuracy; said the other way round, the compute needed for a given accuracy keeps shrinking by a constant factor.
- 2. **b)** Modern networks are so large that they could simply store every example they are shown; the surprise is that trained models generalize instead of only memorizing.
- 3. **c)** When you only prompt a pre-trained model, its insides stay a black box, much as when you rely on software whose source code is not accessible to you.
- 4. **a)** Fine-tuning, merging, quantizing, testing robustness and designing new variants all need an understanding of how the model processes data inside, while prompting treats it as a black box.

Practice

- 1. In your own words, explain why the book calls a neural network trained under scaling laws an almost magical tool, and what could in principle spoil it.

Hint: Think about what the network is trying to approximate, and what a very large network could do instead.

Answer: A network is optimized to approximate the probability distribution behind its data, and with scaling its accuracy rises predictably. In principle a huge network could just memorize every example and become a look-up table, but trained models generalize well instead, even to tasks not explicitly in the training data.

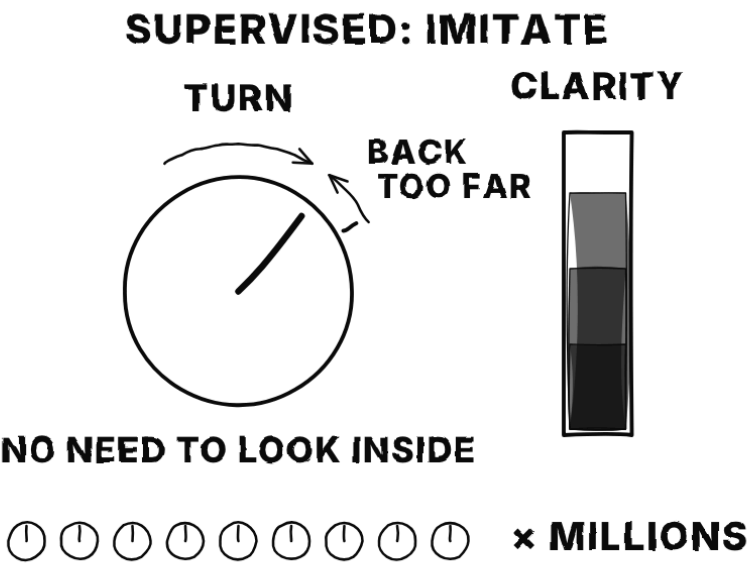
2. The book compares prompting a foundation model to using software without its source code. Explain what that comparison says about who builds such models and what everyone else can still do.

Hint: Ask how many teams can build the largest models, and what open-source code allows.

Answer: Only a few teams have the compute and expertise to release truly large models, just as few companies build large enterprise software. But, as with open-source code, ordinary hardware and some ingenuity still let anyone fine-tune, merge, quantize, test or invent models, provided they understand how the models work inside.

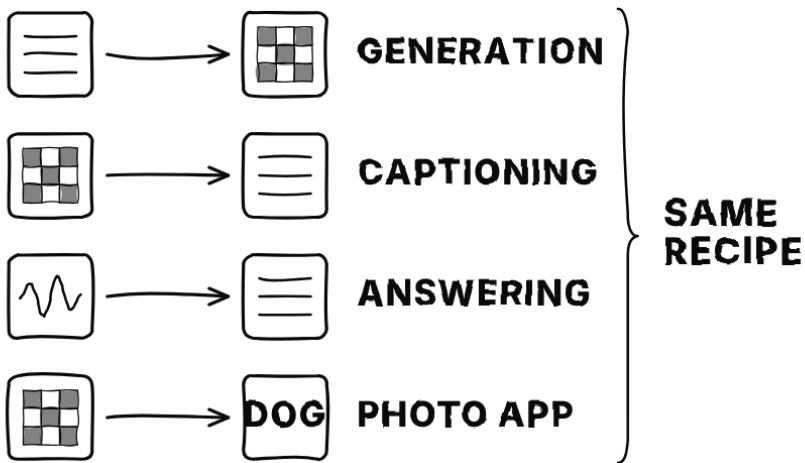
Learning as a Search

Take a radio with an old tuning knob and try to find a station you like. You turn the knob a little, listen, and if the sound gets clearer you keep turning the same way; if it gets worse, you turn back. You never need to know how the radio works inside, only which way a small turn makes things better. Hold on to that picture, because the book describes almost all of modern AI as this kind of knob-turning, done with millions of knobs at once.



Turn, Listen, Turn Again

The book assumes its readers know the basics of machine learning, ML, and more specifically supervised learning, SL. Supervised learning solves a task by gathering examples of the behaviour we want and then training, that is, optimizing, a system until it imitates that behaviour. The idea sounds simple, the book says, and it is deceptively powerful. Image generation becomes the job of collecting a very large set of pictures with their captions. Imitating the English language becomes the job of collecting a great deal of text and learning to predict each sentence from the ones before it. Reading an X-ray becomes the job of keeping a large database of scans together with the doctors' decisions about them.



NEXT SENTENCE · X-RAY → DECISION

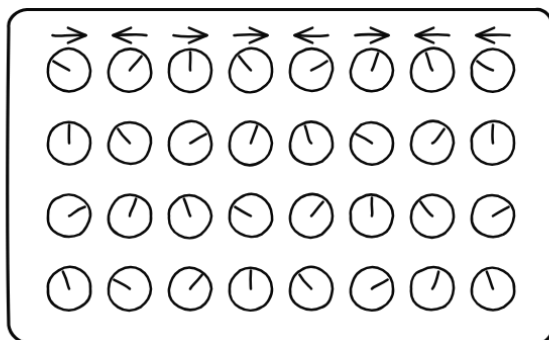
What Goes In, What Comes Out

The book's figure sorts tasks by what goes in and what must come out. Image generation wants a picture, an ordered grid of pixels, from a text, a sequence of characters; image captioning is the reverse, a caption from a picture. Answering a question about a recording wants a text from an audio clip, which is again an ordered sequence, this time of numbers. The fascinating point the book makes is that the models for all these cases are designed by following very similar specifications. Your phone's photo app is one of these: picture in, a word such as "dog" out, learned from many pictures that came with their labels.

In general, the book says, learning is a search problem. We start from a program with a very large number of degrees of freedom, which are called parameters; think of them as the knobs of our radio. Then we change the parameters until the model performs well enough. To make that practical we need efficient ways to search for the best setting even when there are millions, billions or even trillions of parameters, far too many to try one by one.

LEARNING = SEARCH

PROGRAM



$10^6 \dots 10^{12}$

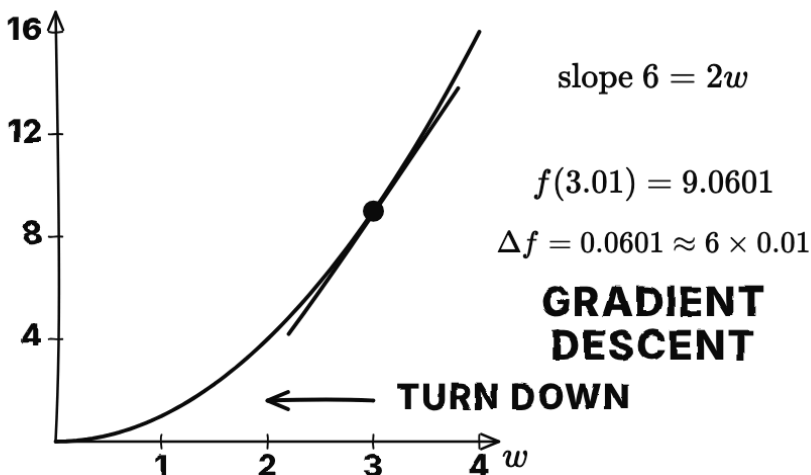
**TRY ONE
BY ONE X**

GRADIENT

PARAMETERS

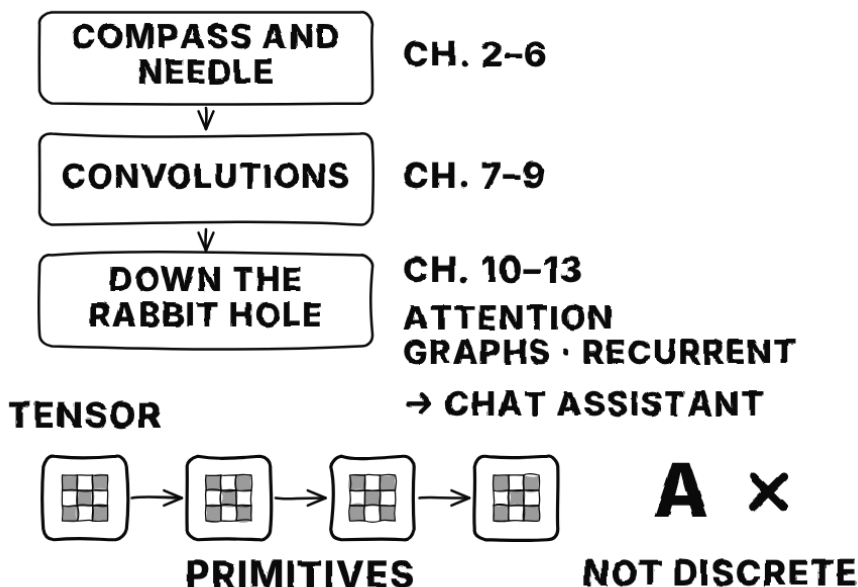
A Program With Many Knobs

Differentiable models solve this by allowing only differentiable components, mathematical functions whose derivative we can compute. Knowing the derivative of a function of many inputs, which is called a gradient, means knowing what happens when we nudge each parameter slightly. Here is the idea with a single knob and numbers we can check. Suppose the error of a model is $f(w) = w^2$, and the knob stands at $w = 3$, so the error is nine. The derivative of w^2 is $2w$, which at three is six: it says that a tiny turn of the knob changes the error about six times as much as the turn itself. Check it: turn the knob to 3.01 and the error becomes 9.0601, a change of 0.0601, almost exactly six times the turn of 0.01. So we know, without trying, that turning the knob down will lower the error, and roughly by how much.



Slope of w Squared at Three

That knowledge leads to automatic routines for optimization, most notably automatic differentiation, which computes these derivatives for a whole program, and gradient descent, which uses them to turn every knob a little in the right direction, over and over. Setting all this up is the subject of the first part of the book, called *Compass and Needle*, which runs from Chapter 2 to Chapter 6.



The Map of the Book

Once a neural network is seen as nothing more than a composition of differentiable building blocks, called primitives, the book asks two basic questions. First, what kinds of data can such a network take in and give out? Second, what kinds of primitives can we use? Differentiability is a strong demand: it rules out working directly with many ordinary data types, such as characters or whole numbers, because they jump from one value to the next and are therefore discontinuous. A letter cannot be nudged a little towards another letter. What differentiable models handle easily are large arrays of numbers, which the book will call tensors; a photo is one, a grid of brightness values, and such arrays can be transformed with algebra by chaining simple linear and nonlinear steps. The book's second figure shows exactly this: a network as a chain of primitives working on

structured arrays, each primitive sorted by what it takes in and what it gives out, which fixes the rules for joining them.

The second part of the book studies one typical differentiable component in depth, the convolution, in Chapters 7 to 9.

Convolutions can be used whenever data comes as an ordered sequence of elements: audio, images, text and video among others. Along the way the book introduces techniques for building deep models, meaning models made of many steps in sequence, and ideas such as splitting text into tokens, generating a sequence one piece after another, and causal modeling, which together are the basis of today's large language models.

The third part, called Down the Rabbit Hole, explores other designs: for sets, most importantly attention layers and transformer models in Chapters 10 and 11; for graphs in Chapter 12; and finally recurrent layers for sequences in time in Chapter 13. The chat assistant you use every day is a transformer, so that part of the journey leads straight to it.

Beyond the book, the author promises a website with further, more research-oriented chapters that can be read in any order, on generative modeling, conditional computation, transfer learning and explainability. There is also a series of guided lab sessions in notebook form, which follow much of the book and go on to advanced topics such as contrastive learning and model merging.

Remember

Supervised learning means collecting examples of a desired behaviour and optimizing a system to imitate it, and this simple recipe covers image generation, language and medical scans alike. Learning is a search over a program's parameters, its knobs, for a setting that works, even when there are billions of them. Differentiable models make the search practical: the gradient tells how each small nudge changes the error, as six times the turn did for w squared at three, and this powers automatic differentiation and gradient descent. Such models work on tensors, large arrays of numbers, not on discrete things like characters; the book's three parts cover the basics, then convolutions, then attention, graphs and recurrent layers.

Questions

1. Why does the book call learning a search problem?
 - a. Because a model looks up the closest stored example for every new input
 - b. Because training changes a program's many parameters until it performs well enough
 - c. Because supervised learning searches the web for labelled data
2. What does knowing the gradient of a model's error give us?

- a. How the error changes when each parameter is nudged slightly
 - b. The best possible setting of all parameters at once
 - c. A list of the training examples the model got wrong
3. Why can't a differentiable model work directly on characters or whole numbers?
- a. Because there are too many different characters to store
 - b. Because they are too small to be multiplied by matrices
 - c. Because they jump from one value to the next, so they cannot be nudged a little
4. Image generation and image captioning differ mainly in what?
- a. One needs labelled data and the other does not
 - b. What goes in and what must come out: text to picture versus picture to text
 - c. One is supervised learning and the other is a search problem

Answers

- 1. b)** We start from a program with a very large number of parameters, its knobs, and change them until the model performs well enough; the hard part is searching efficiently among billions of settings.
- 2. a)** The gradient tells what happens when each parameter is nudged a little, which is what lets automatic routines such as gradient descent improve the model step by step.
- 3. c)** Characters and integers are discrete and therefore discontinuous; a letter cannot be nudged slightly towards another, while large arrays of numbers such as images can.

4. b) The book sorts tasks by their input and output: generation makes a pixel grid from a text, captioning makes a text from a pixel grid, and the models follow similar designs in both cases.

Worked examples

One knob, one nudge A model's error is $f(w) = w^2$, and its single parameter stands at $w = 3$. Use the derivative to predict how much the error changes if w is turned up by 0.01, then check by computing it directly.

The derivative of w^2 is $2w$, so at $w = 3$ it equals 6: the error changes about six times as much as the nudge.

Predicted change: $6 \times 0.01 = 0.06$.

Direct check: $3.01^2 = 9.0601$, while $3^2 = 9$, so the actual change is 0.0601.

The prediction matches almost exactly, which is why a gradient tells us which way to turn the knob without trying every value.

Answer: About 0.06 (the exact change is 0.0601), so lowering w would reduce the error.

Practice

1. With the same error $f(w) = w^2$, the knob now stands at $w = 5$. Which way should it be turned to lower the error, and about how much does the error change for a turn of 0.1?

Hint: First find the derivative $2w$ at $w = 5$.

Answer: Turn it down. The derivative is $2 \times 5 = 10$, so a turn of 0.1 changes the error by about $10 \times 0.1 = 1$ (exactly: $5.1^2 - 25 = 1.01$).

2. Pick a task from your phone, such as searching photos for 'dog' or getting a reply from a chat assistant, and describe it the way the book

describes tasks: what goes in, what comes out, and what data a supervised-learning approach would collect.

Hint: Look at the examples of image generation, language and X-rays.

Answer: Photo search takes a picture in and gives a label out; one would collect many photos paired with their labels and optimize a model to reproduce them. A chat assistant takes text in and gives text out; one would collect a large amount of text and learn to predict each piece from what came before.

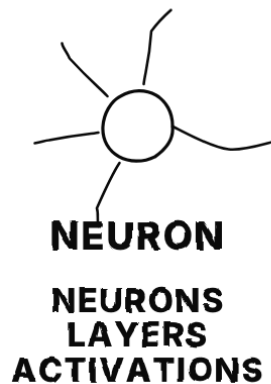
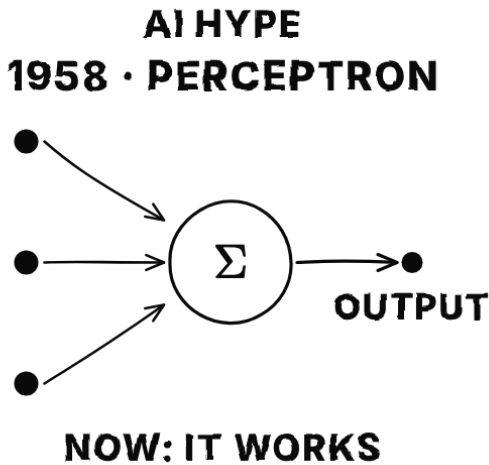
3. Explain why a photo can be fed to a differentiable model but a word, written as letters, cannot be fed to it directly.

Hint: Ask whether each can be changed by a tiny amount.

Answer: A photo is an array of numbers, a tensor of brightness values, that can be nudged slightly and transformed by linear and nonlinear steps. Letters are discrete: there is nothing between one letter and the next, so no small nudge, and therefore no derivative to follow.

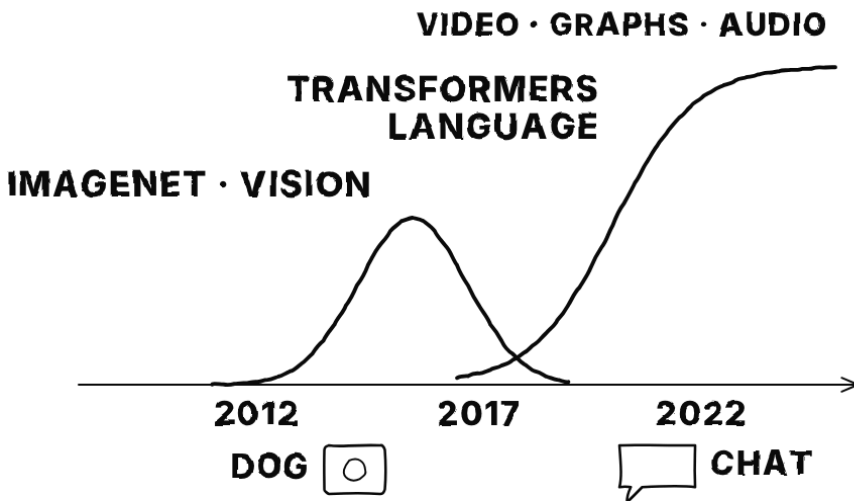
Old Ideas, New Power

Search for old newspaper stories about a "machine that learns" and you will find excited headlines that sound like this week's news about chat assistants. The surprise is the date on them. The book shows one such moment in a figure it labels, with a wink, "AI hype": the year is 1958, and the American psychologist Frank Rosenblatt is drawing wide media attention with his perceptrons, one of the first working prototypes of a neural network. So the excitement is not new; what is new is how well these machines finally work.



Neural networks have a long and rich history. Their very name goes back to early twentieth-century attempts to model biological neurons, the nerve cells of the brain. The vocabulary stuck: to match today's software frameworks, the book will still speak of neurons, layers and activations, even though the analogy with the brain has faded.

Interest came in several waves. Between 2012 and 2017 networks grew in complexity as never before, pushed by large public benchmarks and competitions, most of all the ImageNet Large Scale Visual Recognition Challenge, ILSVRC, which the book covers in Chapter 9. A second big wave came in 2017 with transformers, the subject of Chapter 10. Just as computer vision had been taken over by convolutional models a few years earlier, the processing of natural language was taken over by transformers within a very short time. Further progress followed for video, for graphs, which Chapter 12 treats, and for audio, leading to today's excitement about large language models, multimodal networks that mix text, pictures and sound, and generative models that produce new content. The photo app that finds your dog belongs to the first wave; the chat assistant that answers you belongs to the second.



Two Waves

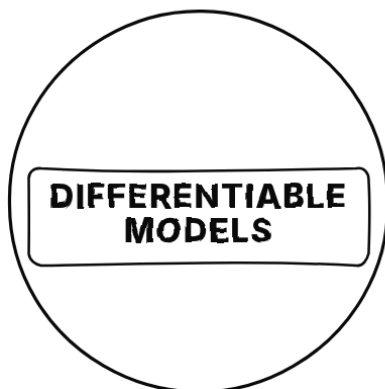
The words changed as fast as the models. In the 1980s people spoke of connectionism; later "deep learning" came to mean modern networks, in contrast with the smaller, shallower models of the past. The book finds all these terms hopelessly vague, because modern artificial networks keep almost no resemblance to biological networks or to neurology. What modern networks really have in common is that they are built from differentiable blocks, pieces whose derivatives can be computed, as we saw with the knob that could be nudged. For that reason the book prefers to call them differentiable models wherever it can.

**X CONNECTIONISM
(1980S)**

X DEEP LEARNING

VAGUE

**DIFFERENTIABLE
PROGRAMMING**



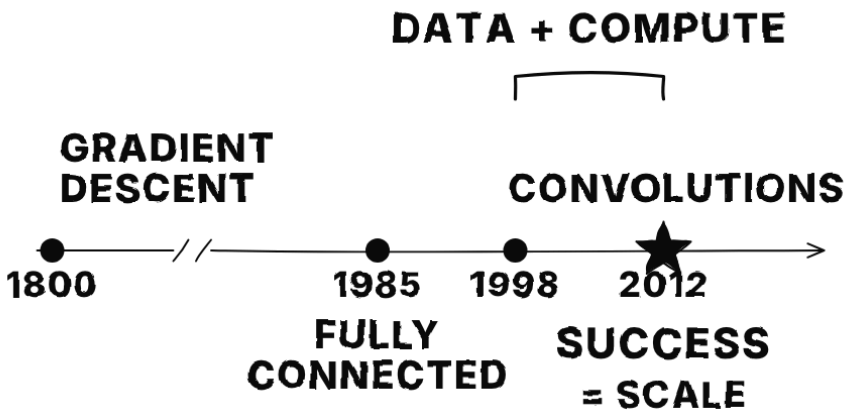
LECUN 2018 · RAFFEL 2021

Names, Then a Better Name

That view opens onto a wider field called differentiable programming, an emerging discipline that mixes computer science and optimization to study computer programs that can be differentiated in general. The book credits two short manifestos for shaping this outlook: one by Yann LeCun in 2018, arguing for this broader view of neural networks, and one by Colin Raffel in 2021, calling for models to be built the way open-source software is built.

Travelling through this land of differentiable models is also travelling through history, the book says. The numerical optimization of linear models by gradient descent, the subject of Chapter 4, was known since at least the nineteenth century. Fully-connected networks in the form the book uses later date back to the 1980s. Convolutional models were

already known and used at the end of the 1990s. Put those dates on a line and a gap appears: a method from the 1800s, another from the 1980s, another from the 1990s, yet the big successes came only after 2012.



Old Ideas, Late Success

What was missing, the book explains, was not the ideas but the means: it took many decades before there was enough data and enough computing power to see how well these models perform once they have plenty of both, and plenty of parameters. This connects back to the scaling laws of the first part: old recipes, cooked at a far larger scale, turned out to be extraordinarily good.

The book is honest about its limits. It cannot go deeply into every topic, partly because research moves so quickly. Its aim instead is to give enough foundations that the reader can find their way through the newest research papers without getting lost. For those who want the full history, it points to an overview of modern neural networks, to a book of interviews with some of the field's main figures, and to a long, opinionated annotated history by Jürgen Schmidhuber.

Try it yourself

Write three dates on a strip of paper: 1800 for gradient descent on linear models, 1985 for fully-connected networks, 1998 for convolutional models. Then add 2012, the start of the great wave, and 2017, the arrival of transformers. You will see that most of the ideas were sitting on the strip long before the successes, which is the book's point: the change came from data and computing power, not from the invention of the ideas.

Remember

Neural networks have a long history, from early models of biological neurons through Rosenblatt's perceptron in 1958 to the waves of 2012 to 2017, driven by ImageNet, and of transformers from 2017. Names like connectionism and deep learning are vague, because modern networks hardly resemble the brain; what they share is being built from differentiable blocks, so the book calls them differentiable models. This view leads to differentiable programming, a discipline at the meeting point of computer science and optimization. Gradient descent, fully-connected networks and convolutions were known decades ago; what made them shine was enough data, computing power and parameters.

Questions

1. Gradient descent, fully-connected networks and convolutions were all known long before 2012. Why did their big successes come only afterwards?
 - a. Because the old methods had errors that were corrected after 2012
 - b. Because it took decades to have enough data, computing power and parameters to show how well they work
 - c. Because the brain was finally understood well enough to copy it

2. Why does the book prefer the name 'differentiable models' to 'neural networks' or 'deep learning'?
 - a. Because modern networks hardly resemble the brain, and what they share is being built from differentiable blocks
 - b. Because the older names are protected trademarks of software frameworks
 - c. Because differentiable models are a different, newer kind of network
3. What role did transformers play in the second wave, compared with convolutional models in the first?
 - a. Transformers replaced convolutions in recognizing images during the ImageNet challenge
 - b. Transformers were an older idea that convolutional models then overtook
 - c. Transformers took over language processing in a short time, as convolutional models had taken over computer vision

Answers

1. **b)** The ideas were already there; what was missing was the means, enough data and computing power, and plenty of parameters, to reveal how well these models perform.
2. **a)** The older names are vague: modern artificial networks keep almost no resemblance to biological networks, and their essential feature is being composed of differentiable blocks.
3. **c)** Just as computer vision was overtaken by convolutional models a few years earlier, natural language processing was overtaken by transformers within a very short time after 2017.

Practice

1. Explain in your own words why the book shows a 1958 newspaper-style moment about Rosenblatt's perceptron under the label 'AI hype'.

Hint: Compare the excitement then with the excitement now, and ask what has actually changed.

Answer: Rosenblatt's perceptron, one of the first working neural networks, already drew wide media attention in 1958. The label shows that excitement about learning machines is old; what is new today is how well such machines finally work, thanks to data and computing power.

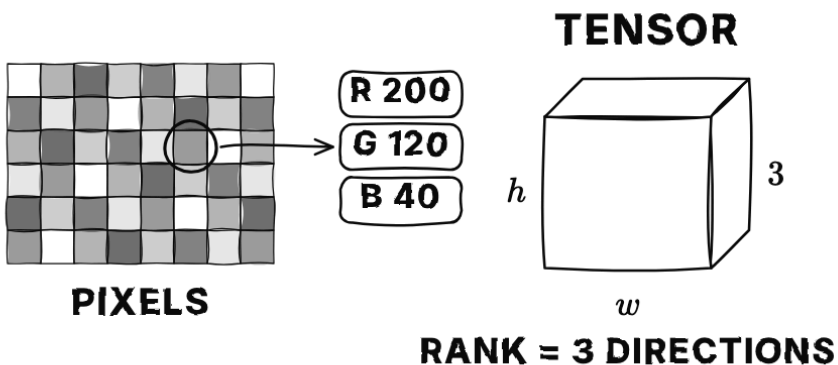
2. Place these on a timeline and say what the gap between them teaches: gradient descent for linear models, fully-connected networks, convolutional models, the 2012 to 2017 wave, transformers.

Hint: Use the dates the book gives: the nineteenth century, the 1980s, the end of the 1990s, 2012, 2017.

Answer: Gradient descent on linear models dates from at least the nineteenth century, fully-connected networks from the 1980s, convolutional models from the end of the 1990s, the great wave from 2012 to 2017, transformers from 2017. Most ideas existed long before the successes, so the change came from having enough data, computing power and parameters.

Boxes of Numbers and Their Shapes

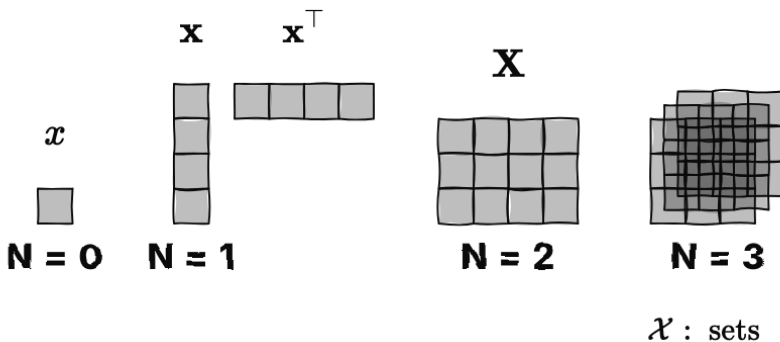
Open any photo on your phone and zoom in until you see the little squares. Each square is a pixel, and each pixel is stored as three numbers: how much red, how much green and how much blue. So the photo is not a picture to the computer at all; it is a box of numbers, so many squares high, so many wide, and three deep. Ask what such a box looks like for a whole album, and you have the first idea of this part.



A Photo Is a Box of Numbers

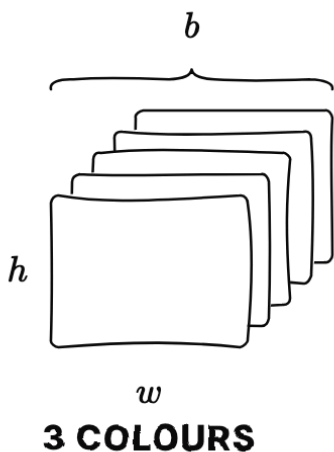
The basic data type of differentiable models is the tensor, which the book defines as an n -dimensional array of objects, usually real numbers. The book admits the name is a small misuse: in the scientific literature a tensor is a more precise thing, a multilinear operator, while here it simply means a multidimensional array; the name is so widespread that the book keeps it. With apologies to mathematicians, it calls n the rank of the tensor, that is, the number of directions in which the box extends.

The notation changes with the rank. A tensor with $n = 0$ holds a single value, a scalar, written with a lowercase letter such as x or y . Scalars are so called because any one of them can be written as a multiple, a scaling, of the number one. A tensor with $n = 1$ is a column of values, a vector, written in lowercase bold, $\mathbf{x}$. The same values laid out as a row are written $\mathbf{x}^T$, the small T meaning the column has been turned on its side; when the context makes it clear, the book drops the T for readability. The book's figure draws vectors as rows only to save space; in the text every vector is a column. A tensor with $n = 2$ is a rectangle of values, a matrix, written in uppercase bold, $\mathbf{X}$ or $\mathbf{Y}$. For ranks above two there is no special lettering, and curly calligraphic letters such as $\mathcal{X}$ are kept for sets and probability distributions.



Rank Zero to Three

Most of the time, the book says, understanding an algorithm or an operation comes down to understanding the shape of every tensor involved. To write shapes compactly it uses the sign that looks like a small wave: $X \sim (b, h, w, 3)$. In words, X is a tensor of rank four whose four sizes are b , h , w and 3 . Where it comes from is our photo album: b is the number of photos in the batch, h the height and w the width in pixels, and 3 the three colour channels. Some sizes are fixed numbers, like the 3 , and others are letters that can take any value.



$$X \sim (b, h, w, 3)$$

$$b = 32, h = w = 224$$

$$224 \times 224 \times 3 = 150,528$$

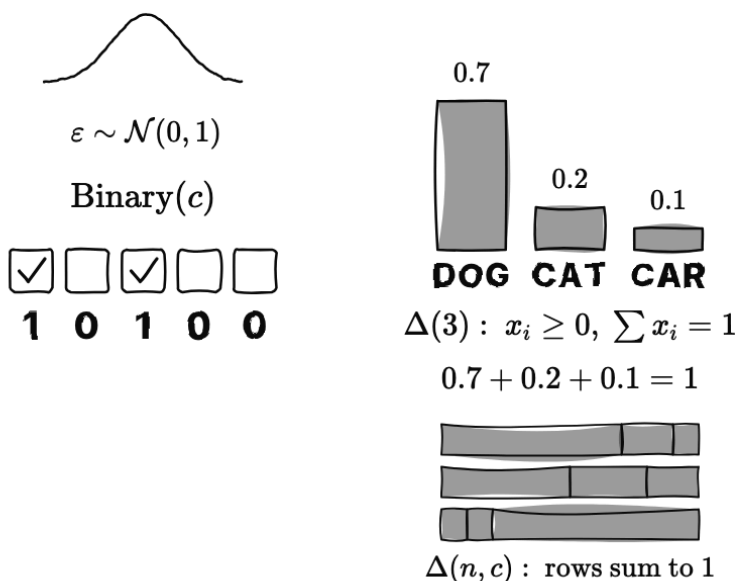
$$\times 32 = 4,816,896$$

$$\approx 5 \text{ MILLION}$$

A Batch of Photos

Let us count aloud. Take a batch of 32 photos, each 224 pixels high and 224 wide. One photo holds 224 times 224, which is 50,176 pixels, times 3 colours, which makes 150,528 numbers. The whole batch holds 32 times that: 4,816,896 numbers in a single rank-four tensor, almost five million. That is why shapes matter so much: they tell you at a glance how much is flowing through a model.

The same wave sign also means drawing a random value from a probability distribution, as in $\epsilon \sim \mathcal{N}(0, 1)$, a number drawn from the bell curve centred on zero. The book uses it that way only rarely, and the context always makes the meaning clear. So $\mathbf{x} \sim (d)$ replaces the more usual $\mathbf{x} \in \mathbb{R}^d$, a vector of d real numbers, and $\mathbf{X} \sim (n, d)$ replaces $\mathbf{X} \in \mathbb{R}^{n \times d}$, a matrix of n rows and d columns.



Rules for the Values

Sometimes the values in a tensor must obey a rule, and there is notation for that too. $\mathbf{x} \sim \text{Binary}(c)$ is a tensor of size c whose elements are only zeros and ones, for example a row of tick boxes saying which of c tags a photo has. $\mathbf{x} \sim \Delta(a)$ is a vector in the so-called simplex: every element is at least zero, $x_i \geq 0$, and all of them add up to one, $\sum_i x_i = 1$. That is exactly what a list of probabilities looks like. If your photo app says dog 0.7, cat 0.2, car 0.1, all three are at least zero and 0.7 plus 0.2 plus 0.1 is 1, so this vector lies in the simplex of size three. For higher ranks, as in $\mathbf{X} \sim \Delta(n, c)$, the rule applies along the last dimension: each of the n rows is a list of c probabilities summing to one, one row per photo.

Further notation is introduced chapter by chapter as needed.

Try it yourself

Take a sheet of squared paper and colour a tiny picture three squares high and four wide. Now write, for every square, three numbers for its red, green and blue; you have a tensor of shape three by four by three, which holds 36 numbers. Stack two such sheets and the shape becomes two by three by four by three, with 72 numbers: you have just built a small batch, the same kind of object as the book's X .

Remember

A tensor is a multidimensional array of numbers, and its rank is the number of directions it extends in: a scalar has rank zero, a vector rank one, a matrix rank two. Scalars are lowercase, vectors lowercase bold and always columns, matrices uppercase bold; curly letters are kept for sets and distributions. The wave sign gives a tensor's shape, as in $X \sim (b, h, w, 3)$ for a batch of colour photos, and a batch of 32 photos of 224 by 224 pixels holds almost five million numbers. Binary(c) means only zeros and ones, and $\Delta(a)$, the simplex, means non-negative values that sum to one, the shape of a list of probabilities.

Questions

1. A batch of colour photos is written $X \sim (b, h, w, 3)$. What is its rank, and why is the last size a fixed number?

- a. Rank three, because the batch size b does not count as a dimension
 - b. Rank four, and the 3 is fixed because every pixel has three colour values
 - c. Rank four, and the 3 is fixed because every batch holds three photos
2. Which of these vectors lies in the simplex $\Delta(3)$?
- a. (0.5, 0.5, 0.5)
 - b. (0.9, 0.3, -0.2)
 - c. (0.6, 0.3, 0.1)
3. Why does the book say that understanding an operation mostly comes down to understanding shapes?
- a. Because the shape of each tensor tells at a glance what goes in, what comes out and how much flows through
 - b. Because the values inside tensors never matter for how a model works
 - c. Because shapes are the only thing a computer can store
4. In the book's notation, how would a vector and a matrix be told apart on the page?
- a. A vector is uppercase bold and a matrix is lowercase bold
 - b. A vector is lowercase bold, a matrix uppercase bold
 - c. Both are written with curly calligraphic letters

Answers

1. **b)** The shape lists four sizes, so the rank is four; b , h and w can vary, while every pixel always has red, green and blue, which is why the last size is

3.

2. c) A vector in the simplex has every element at least zero and elements summing to one; $0.6 + 0.3 + 0.1 = 1$, while the first sums to 1.5 and the second has a negative element.

3. a) Following the shape of every tensor shows what an operation takes and gives, and how big it is, as the almost five million numbers of a batch of 32 photos show.

4. b) Vectors use lowercase bold such as bold x , matrices uppercase bold such as bold capital X , while curly letters are kept for sets and probability distributions.

Worked examples

How many numbers in a batch A batch of photos has shape $(b, h, w, 3)$ with $b = 32$, $h = 224$ and $w = 224$. How many numbers does the tensor hold?

One photo has $h \times w = 224 \times 224 = 50,176$ pixels.

Each pixel stores 3 colour values, so one photo holds $50,176 \times 3 = 150,528$ numbers.

The batch stacks 32 photos, so it holds $32 \times 150,528$ numbers.

$32 \times 150,528 = 4,816,896$.

Answer: 4,816,896 numbers, almost five million

Practice

1. A batch of 10 photos, each 100 pixels high and 50 wide, in colour. Write its shape in the book's notation, give its rank, and count its numbers.

Hint: The shape is $(b, h, w, 3)$; multiply all four sizes.

Answer: $X \sim (10, 100, 50, 3)$, rank four. It holds $10 \times 100 \times 50 \times 3 = 150,000$ numbers.

2. A classifier gives, for each of 2 photos, probabilities for 3 classes: $(0.5, 0.3, 0.2)$ and $(0.1, 0.1, 0.8)$. Write the shape of this result and check that it belongs to $\Delta(2, 3)$.

Hint: The rule of the simplex applies along the last dimension, row by row.

Answer: The result has shape $(2, 3)$. Each row has elements at least zero and sums to one: $0.5 + 0.3 + 0.2 = 1$ and $0.1 + 0.1 + 0.8 = 1$, so it belongs to $\Delta(2, 3)$.

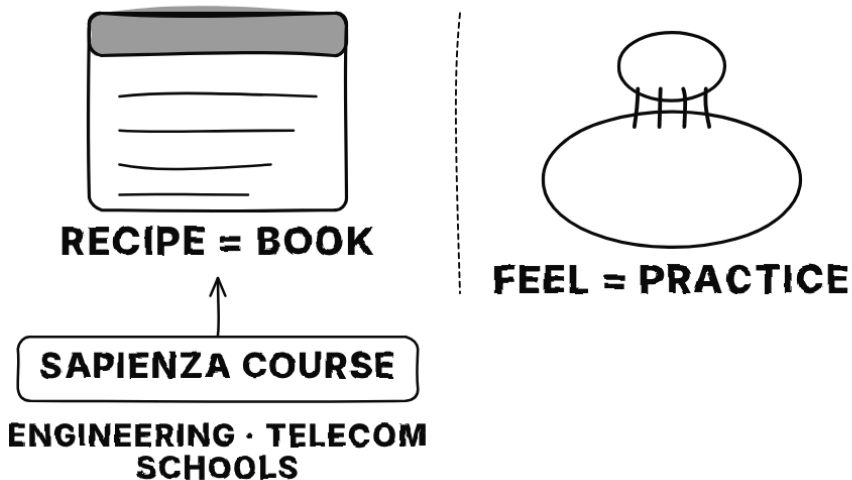
3. Rewrite $\mathbf{X} \in \mathbb{R}^{n \times d}$ and $\mathbf{x} \in \mathbb{R}^d$ in the book's shape notation, and say what each object is.

Hint: The wave sign replaces 'belongs to $\mathbb{R}$ to the ...'.

Answer: $\mathbf{X} \sim (n, d)$, a matrix of n rows and d columns of real numbers; $\mathbf{x} \sim (d)$, a column vector of d real numbers.

What This Book Is, and Is Not

Read a recipe for bread, then try to bake it, and you will learn two different things. The recipe tells you what happens to flour, water and yeast; your hands learn how wet the dough should feel, which no recipe can write down. The book draws the same line for neural networks: it gives you the recipe in full, and it is honest that the feel comes only with practice.



The Recipe and the Feel

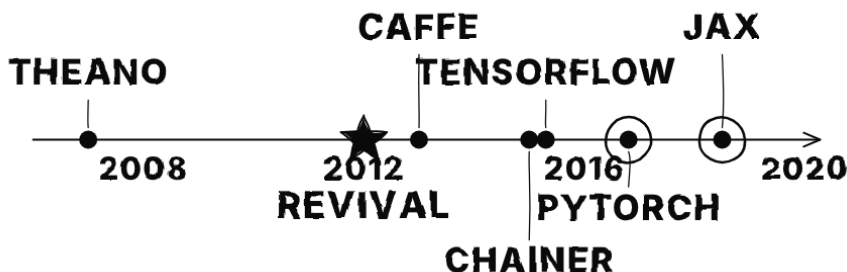
The book grew out of lectures. The author has taught a course called Neural Networks for Data Science Applications for many years in the

Master's degree in Data Science at Sapienza University of Rome, and wanted to give those lectures a coherent written form. The core chapters make up the main part of that course; the remaining chapters cover topics taught in some years and not in others. Some parts also draw on other teaching: a course on neural networks for computer engineering, an introduction to machine learning for telecommunication engineering, and assorted tutorials, doctoral courses and summer schools.

There are already several excellent and recent books on modern deep neural networks, and the book says so openly. At the start it covers much the same ground as they do; later the way of explaining, some extra parts and a few sections of the advanced chapters overlap less, following the author's own research interests. Its stated hope is to offer an additional, complementary viewpoint on material that exists elsewhere.

THEORY + CODE

DESIGN $\rightleftharpoons$ CODE



NOT A MANUAL ×

Libraries Behind the Revival

The name "differentiable programs" was chosen on purpose: understanding them comes from both theory and coding. How a model is designed and how it is implemented keep influencing each other, and automatic differentiation, which computes gradients for whole programs, is the clearest example. The book traces much of the revival of neural networks, roughly from 2012, to powerful software libraries: first Theano, then Caffe and Chainer, and then today's TensorFlow, PyTorch and JAX, among others. Wherever possible it connects its explanations to ideas from these frameworks, mainly PyTorch and JAX. But it is not a programming manual, and for a full introduction to each library it sends readers to the libraries' own documentation.

Before setting off, the book lists what it is not. First, it does not try to be complete. It picks ideas that are both common today and general enough to stay useful for a while, but it cannot see the future, and some parts may be incomplete or out of date by the time you read them. In a field where the chat assistant on your phone may be replaced by a new version every few months, that warning is simply realistic.

Second, for each idea it shows a few variations from the research literature, for instance moving from batch normalization to layer normalization, two ways of rescaling the numbers inside a network. Hundreds more variations exist, and the book invites readers to explore them on the website [Papers With Code](#).

Third, and most important, it is a book about the fundamental components of differentiable models. Building them at large scale and actually making them work also needs engineering skill and a certain amount of intuition. The book says little about hardware, and for the intuition, it says, nothing beats experience and opinionated blog posts, pointing in particular to Andrej Karpathy's recipe for training neural networks and his [Zero to Hero](#) video series. That is our bread again: the book is the recipe, and the feel of the dough is learned by baking.

Try it yourself

Open the documentation page of PyTorch or JAX and look at how many separate functions it lists. Then compare that with the few ideas this course has named so far: tensors, gradients, differentiable building blocks. You should find that the libraries are huge, while the core ideas are few; the book teaches the few, and the documentation covers the many.

Remember

The book turns the author's long-running lectures on neural networks at Sapienza University of Rome into a coherent text, with core chapters and occasional extra topics. It overlaps with other good books at the start and then follows its own path, offering a complementary viewpoint. Theory and code go together: the revival of neural networks since about 2012 owes much to libraries from Theano to PyTorch and JAX, which the book refers to, without being a programming manual. It is not complete, it shows only a few of the many variants, and it teaches the fundamental components, while building at scale also needs engineering and intuition that come from experience.

Questions

1. Why does the book say that understanding differentiable programs needs both theory and coding?

- a. Because the theory is too hard to follow without running code
 - b. Because how models are designed and how they are implemented keep shaping each other, as automatic differentiation shows
 - c. Because every chapter is written as a programming manual
2. What does the book say it leaves mostly to experience and blog posts?
- a. The intuition and engineering needed to build models at large scale and make them work
 - b. The definitions of tensors and gradients
 - c. The history of neural networks since 1958
3. Why does the book warn that some parts may be out of date when you read them?
- a. Because its examples were written for an old version of PyTorch
 - b. Because the course it comes from is no longer taught
 - c. Because it chooses ideas common today and general for a while, but the field moves fast and it does not aim at completeness

Answers

1. **b)** Design and implementation constantly interact, and automatic differentiation, which computes gradients for whole programs, is the clearest example; still, the book is not a programming manual.
2. **a)** It teaches the fundamental components, but building them at scale needs engineering skill and intuition, and for the intuition nothing beats

experience and opinionated blog posts.

3. c) It picks ideas both common now and general enough to last, but cannot foresee the future and does not strive for completeness.

Practice

1. In your own words, explain the difference the book draws between understanding the fundamental components of a network and making a large network actually work.

Hint: Think of the bread: the recipe and the feel of the dough.

Answer: The book explains the components fully, like a recipe. Building and training at large scale also needs engineering skill, knowledge of hardware and an intuition that comes from experience, for which the book points to practice and to opinionated guides such as Karpathy's.

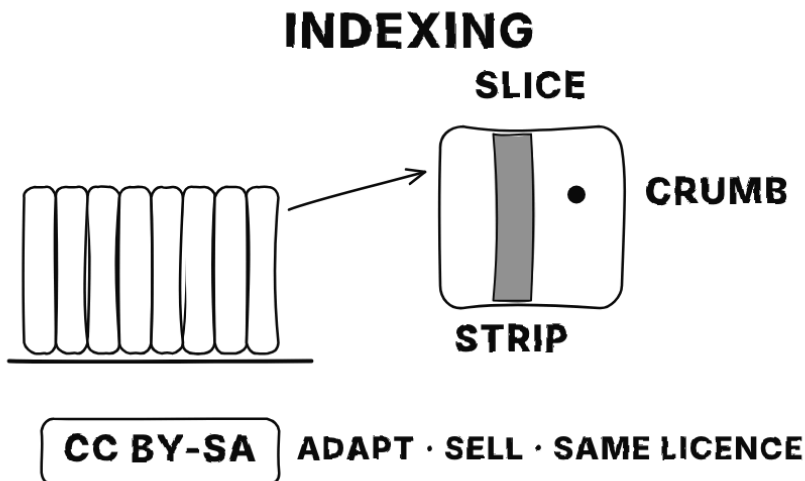
2. Why might a reader who already owns another deep-learning book still find this one useful, according to the book itself?

Hint: Look at where the book overlaps with others and where it does not.

Answer: At the start it covers much the same ground, but its way of explaining, some extra parts and a few advanced sections follow the author's research interests, so it offers an additional, complementary viewpoint on the material.

Cutting Slices from a Tensor

Take a loaf of sliced bread out of its bag. You can lift out one whole slice, you can cut a strip from the middle of that slice, or you can pinch off a single crumb at a precise spot. Each of these is a way of picking part of the loaf by saying where it sits, and that is exactly what indexing a tensor means. Before we start cutting, the book closes its introduction with thanks and a licence.



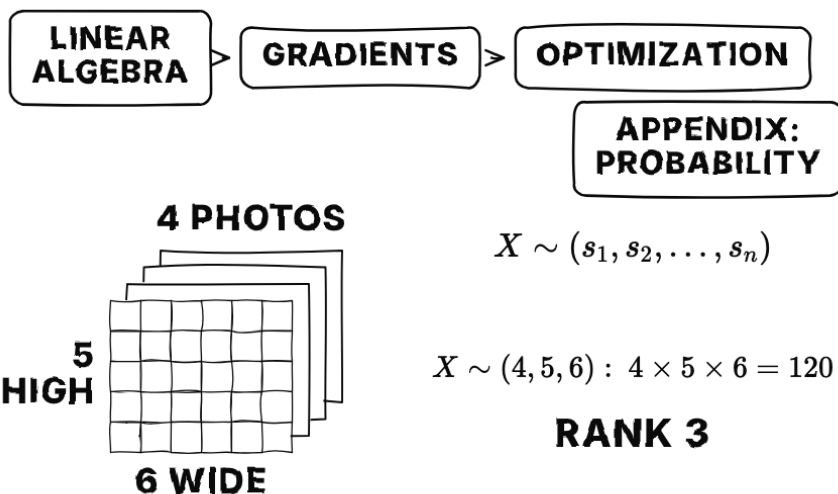
Slice, Strip, Crumb

The book thanks those who helped make it: the colouring of its equations comes from a LaTeX package by ST John; its colour

pictures of Alice and the margin symbols are licensed from a stock-image library; and the Alice pictures inside its figures reproduce John Tenniel's original illustrations, via Wikimedia. It thanks Roberto Alma for feedback on an earlier draft and for encouraging publication, and Corrado Zoccolo, Emanuele Rodolà, Marcin Słaby, Konstantin Burlachenko and Diego Sandoval for extensive corrections, along with everyone who wrote in by email.

The book is released under the Creative Commons CC BY-SA licence. In the licence's own words, it lets others distribute, remix, adapt and build on the material in any medium or format, as long as the creator is credited; it allows commercial use; and anything built from it must be released under the same terms. The course you are listening to is such an adaptation, and it carries the same licence.

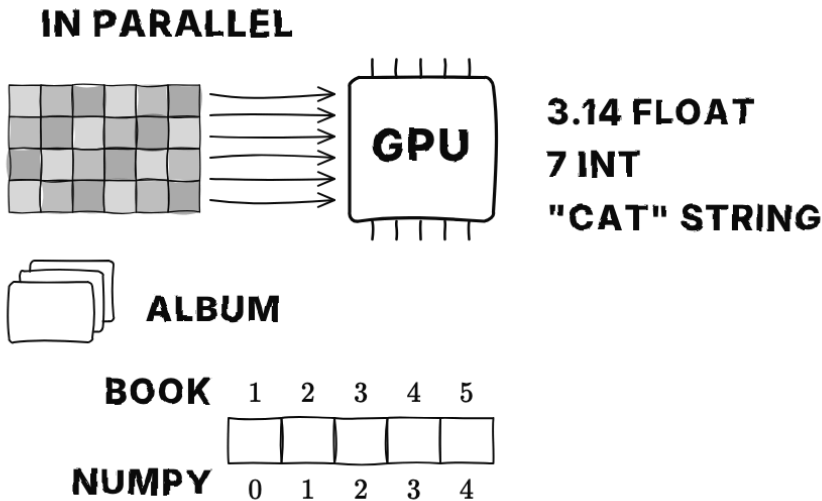
Chapter 2 gathers the mathematics the rest of the book needs. It assumes you have met these topics before and concentrates on notation and on a coherent overview, linking ideas such as tensors to how they are implemented in practice. It has three parts, each building on the one before: linear algebra, then gradients for objects with many dimensions, then how to optimize functions using those gradients. A separate appendix gives a self-contained summary of probability theory, focused on the maximum likelihood principle. The book warns that the chapter is dense with definitions, and asks the reader to bear with it.



Chapter 2, and a Tensor's Shape

The first part recalls linear algebra, partly to agree on shared notation, and it starts from the tensor. The definition: a tensor X is an n -dimensional array of elements that are all of the same type, and its shape is written $X \sim (s_1, s_2, \dots, s_n)$. In words, the list in brackets gives the size of each of the n dimensions. For example, a batch of 4 small grey photos, each 5 pixels high and 6 wide, has shape $X \sim (4, 5, 6)$, and it holds 4 times 5 times 6, which is 120 numbers. With n equal to zero we get scalars, with one vectors, with two matrices, and above that higher-dimensional arrays, keeping the lettering we already agreed on: lowercase for scalars, lowercase bold for vectors, uppercase bold for matrices.

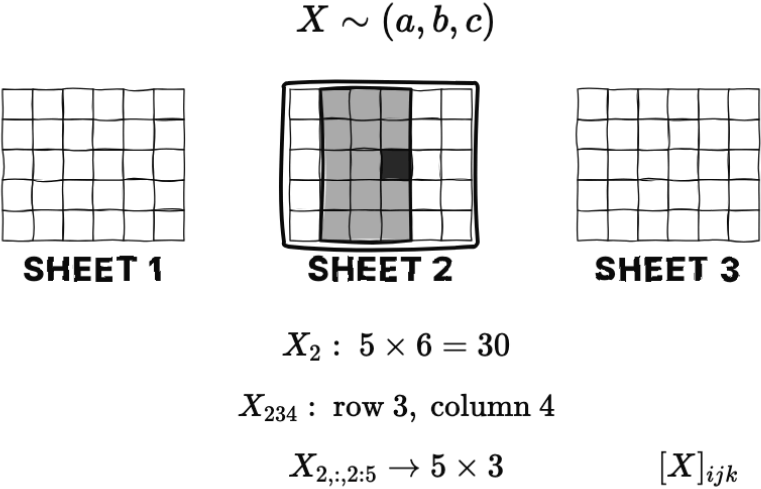
Why are tensors so central to deep learning? Because they suit massively parallel computation: graphics processors, GPUs, and more specialized chips such as TPUs and IPUs can work on all the numbers of a tensor at the same time. That is why the phone in your pocket can tag a whole album of photos quickly: it treats them as one big tensor rather than one photo after another.



All the Numbers at Once

A tensor is described by two things: the type of its elements and its shape. The book mostly uses tensors of floating-point numbers, numbers with a decimal point, whose exact format it discusses later; but tensors can also hold integers, for instance class numbers in classification, or strings, for text.

Tensors can be indexed to take slices, that is, subsets of their values, and most of the conventions of the Python library NumPy apply. One difference: for readability the book counts positions from 1, not from 0 as NumPy does. For simple expressions it writes indices as small subscripts. Take a three-dimensional tensor $X \sim (a, b, c)$. Then X_i is one slice of shape (b, c) : in our photo batch, X_2 is the whole second photo, a 5 by 6 grid of 30 numbers. And $X_{i\ j\ k}$ is a single scalar: X_{234} is the brightness of the pixel in row 3, column 4 of the second photo, one crumb of the loaf.



Cutting the Photo Batch

For more complex picks the book separates indices with commas, as in $X_{i,:;j:k}$. The colon on its own means "take everything along this dimension", and j colon k means "a range of positions", so the result

has shape $(b, k - j)$. With our photos, $X_{2,:;2:5}$ takes the second photo, all 5 rows, and a band of columns; its shape is 5 by 5 minus 2, that is, 5 by 3, a strip of 15 numbers. When an expression would get cluttered, the book writes the indexing in light grey brackets, $[X]_{i \ j \ k}$, so that the indices stand apart from the rest, and the thing inside the brackets may itself be a whole expression.

Try it yourself

Draw a grid of 5 rows and 6 columns on paper and write a number in every square; that is one photo, and two more such sheets make a batch of shape 3 by 5 by 6. Point to the second sheet: that is $X_{\text{sub } 2}$, 30 numbers. Point to row 3, column 4 on it: that is $X_{\text{sub } 2 \ 3 \ 4}$, one number. Then shade all rows of columns 2, 3 and 4 on that sheet: you have cut a 5 by 3 strip, the kind of slice written with colons.

Remember

The book thanks its helpers and is released under CC BY-SA, which allows adapting and selling as long as the author is credited and the result keeps the same licence. Chapter 2 gathers the mathematics needed later: linear algebra, gradients in many dimensions, and optimization with gradients, with probability in an appendix. A tensor is an n -dimensional array of elements of one type, with shape $X \sim (s_1, \dots, s_n)$; tensors suit parallel hardware such as GPUs. Indexing follows NumPy but counts from 1: X_i is a slice, $X_{i \ j \ k}$ a single number, colons pick whole dimensions or ranges, and grey brackets separate the indices from the rest.

Questions

1. For a tensor $X \sim (a, b, c)$, what is the difference between X_i and $X_{i \ j \ k}$?
 - a. X_i is a whole slice of shape (b, c) , while $X_{i \ j \ k}$ is a single number
 - b. X_i is a single number, while $X_{i \ j \ k}$ is a slice of shape (a, b, c)
 - c. Both are slices, but $X_{i \ j \ k}$ has three times as many numbers
2. Why are tensors so well suited to deep learning?
 - a. Because they store numbers with more precision than other formats
 - b. Because they can only hold floating-point numbers

- c. Because hardware such as GPUs can work on all their numbers in parallel
3. What does the book's CC BY-SA licence require of an adaptation such as this course?
- a. That it is never sold
 - b. That it credits the creator and is released under the same licence
 - c. That it keeps the original text without changes
4. In the book's indexing, how does it differ from NumPy?
- a. It counts positions from 1 instead of 0
 - b. It forbids colons for ranges
 - c. It always writes indices in grey brackets

Answers

1. **a)** Fixing one index leaves the other two dimensions, a slice of shape (b, c); fixing all three picks out one scalar, like the brightness of one pixel of one photo.
2. **c)** Tensors suit massively parallel implementations on GPUs and more specialized chips, which process many numbers at the same time.
3. **b)** The licence allows distributing, remixing, adapting and commercial use, as long as the creator is credited and the adapted material carries identical terms.
4. **a)** Most NumPy conventions apply, but for readability the book indexes from 1, not 0; grey brackets are used only when an expression would be cluttered.

Worked examples

Shapes of slices A batch of grey photos has shape $X \sim (4, 5, 6)$. Give the shape of X_2 , the shape of $X_{2,:,2:5}$ (which has shape $(b, k - j)$), and say what X_{234} is.

Here $a = 4$ photos, $b = 5$ rows, $c = 6$ columns.

X_2 fixes the first index, leaving shape $(b, c) = (5, 6)$: the whole second photo, 30 numbers.

$X_{2,:,2:5}$ fixes the photo, keeps all $b = 5$ rows and takes columns from $j = 2$ to $k = 5$, so its shape is $(b, k - j) = (5, 3)$, 15 numbers.

X_{234} fixes all three indices, so it is one number: row 3, column 4 of the second photo.

Answer: X_2 has shape $(5, 6)$; $X_{2,:,2:5}$ has shape $(5, 3)$; X_{234} is a single scalar.

Practice

1. A tensor has shape $(10, 8, 12)$. How many numbers does it hold, and how many does one slice X_i hold?

Hint: Multiply all sizes for the whole tensor; drop the first one for a slice.

Answer: It holds $10 \times 8 \times 12 = 960$ numbers; one slice X_i has shape $(8, 12)$ and holds 96.

2. For $X \sim (10, 8, 12)$, what is the shape of $X_{3,:,4:10}$?

Hint: Use the book's rule: the slice $X_{i,:,j:k}$ has shape $(b, k - j)$.

Answer: $(8, 6)$: all 8 positions of the middle dimension, and $10 - 4 = 6$ positions of the last.

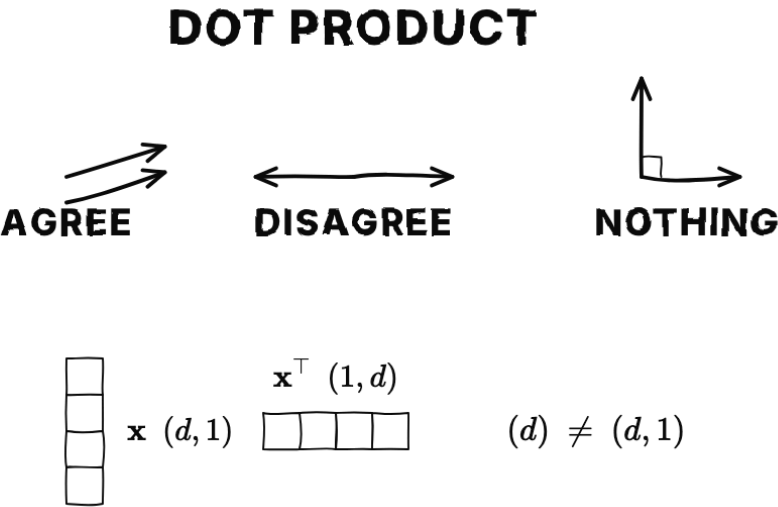
3. Explain in a few sentences what the two things are that describe a tensor, with an example of each from the lecture.

Hint: One is about what the elements are, the other about how they are arranged.

Answer: A tensor is described by the type of its elements, usually floating-point numbers but possibly integers such as class numbers or strings for text, and by its shape, the list of sizes of its dimensions, such as (4, 5, 6) for four photos of 5 by 6 pixels.

Arrows, Lengths and Matching

Stand in a room and point your arm at the door, then ask a friend to point at the window. If you both point the same way, your arms agree completely; if you point in opposite directions, they disagree completely; if your arms form a right angle, they have nothing in common. One number can measure that agreement, and it is the single most important operation of this whole course: the dot product.

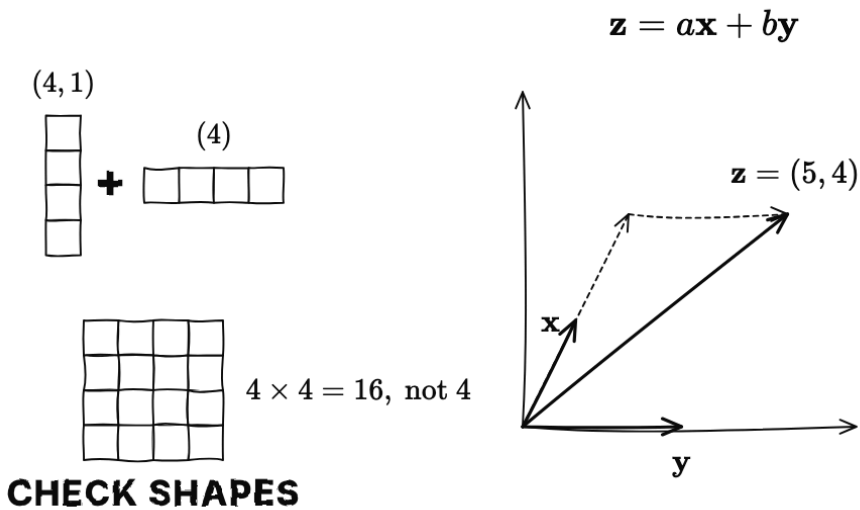


Agree, Disagree, Nothing in Common

The book begins by noting that most of its models are compositions of differentiable operations, and in fact mostly simple ones: sums,

multiplications and a few nonlinear functions such as the exponential $\exp(x)$, sines and cosines, and square roots. Vectors, written $\mathbf{x} \sim (d)$, are tensors with one dimension. Linear algebra books carefully distinguish column vectors $\mathbf{x}$ from row vectors $\mathbf{x}^T$, and the book tries to follow that convention. In code it is trickier: a row or column is a two-dimensional tensor of shape $(1, d)$ or $(d, 1)$, which is not the same as a one-dimensional tensor of shape (d) .

That difference matters because most frameworks follow broadcasting rules borrowed from NumPy. In a nutshell, broadcasting lines up two shapes from the right and repeats a tensor wherever it can, to make the shapes match. The book's code box shows a very common mistake that results: add a "column" of shape 4 by 1 to a plain vector of shape 4, expecting four sums, and you get a 4 by 4 matrix instead. The column is repeated across four columns and the vector across four rows, and every pair is added, giving sixteen numbers where you wanted four. The same thing happens in NumPy, TensorFlow, JAX and practically any framework, so the shapes have to be checked.

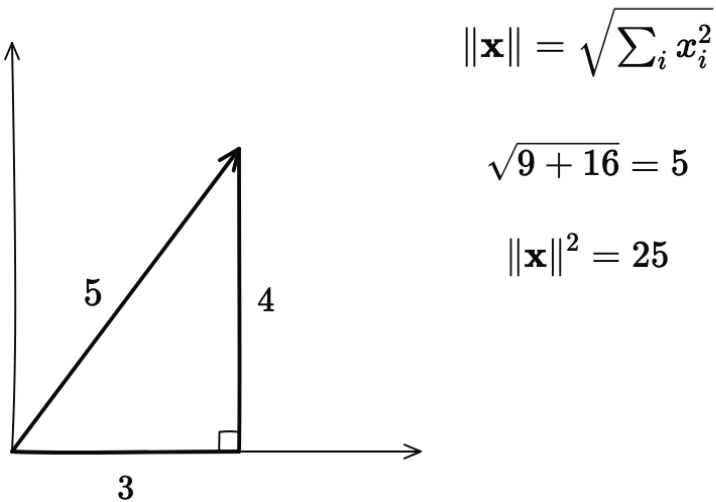


Broadcasting, and a Linear Combination

Vectors have their own algebra, called a vector space: any two vectors of the same shape can be combined linearly into a third, $\mathbf{z} = a\mathbf{x} + b\mathbf{y}$. Element by element, $z_i = ax_i + by_i$: each entry of the result is a times the entry of $\mathbf{x}$ plus b times the entry of $\mathbf{y}$. Where it comes from is simply doing the same thing in every position. With $\mathbf{x} = (1, 2)$, $\mathbf{y} = (3, 0)$, a equal to 2 and b equal to 1, we get 2 times 1 plus 3, which is 5, and 2 times 2 plus 0, which is 4: the vector $(5, 4)$.

If we picture a vector as a point, or an arrow from the origin, in d -dimensional space, adding two vectors means completing a parallelogram, and the distance of a vector from the origin is its Euclidean norm, also called the ℓ_2 norm: $\|\mathbf{x}\| = \sqrt{(\sum_i x_i^2)}$. In words, square every entry, add them up, and take the square root. It comes

from Pythagoras: in two dimensions the arrow is the long side of a right triangle whose other sides are the two entries. For $\mathbf{x} = (3, 4)$, 9 plus 16 is 25, and its root is 5. The squared norm $\|\mathbf{x}\|^2$ is especially handy, because it is just the sum of the squared entries, 25 here, with no root to take.



Length by Pythagoras

Now the central operation, the inner product, also called the dot product: multiply two vectors entry by entry and add up the results. The book's definition reads $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y} = \sum_i x_i y_i$. The angle brackets come from physics, and the book uses them now and then for clarity; the important point is that the result is a single number, a scalar. The book's own example: with $\mathbf{x} = [0.1, 0, -0.3]$ and $\mathbf{y} = [-4.0, 0.05, 0.1]$,

the products are minus 0.4, then 0, then minus 0.03, and they add up to minus 0.43.

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\top \mathbf{y} = \sum_i x_i y_i$$

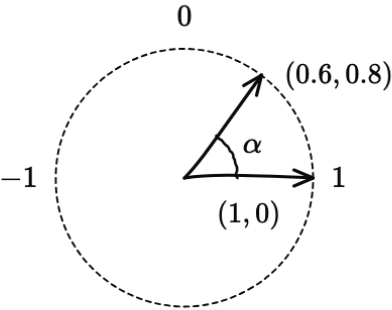
$$0.1 \times (-4.0) = -0.4$$

$$0 \times 0.05 = 0$$

$$-0.3 \times 0.1 = -0.03$$

$$\text{sum} = -0.43$$

$$\mathbf{x}^\top \mathbf{y} = \|\mathbf{x}\| \|\mathbf{y}\| \cos \alpha$$



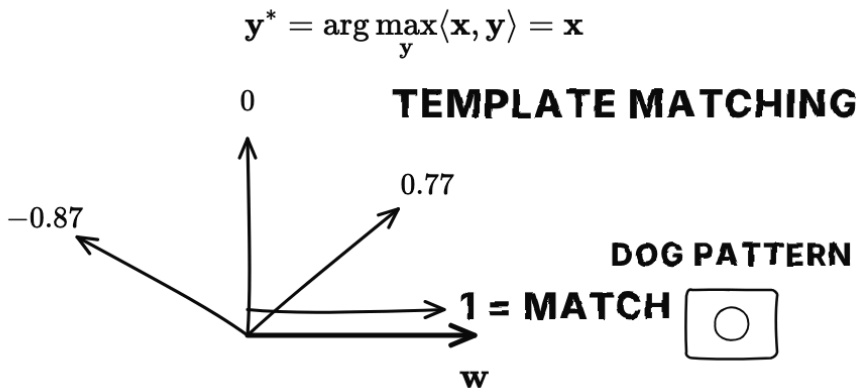
$$\cos \alpha = 0.6$$

The Dot Product, Two Ways

The dot product has a simple geometric meaning through the angle α between the two arrows: $\mathbf{x}^\top \mathbf{y} = \|\mathbf{x}\| \|\mathbf{y}\| \cos(\alpha)$. In words, it is the two lengths multiplied together, times a factor that says how much the arrows point the same way. For vectors normalized to length one, the lengths drop out and the dot product is just the cosine of the angle, which is then called the cosine similarity. It ranges from 1, arrows pointing the same way, to minus 1, arrows pointing in opposite directions, and it is exactly 0 when they are orthogonal, at right angles: your arm and your friend's again. Check one case: the unit arrows $(1, 0)$ and $(0.6, 0.8)$ have dot product 0.6, and indeed the

second has length 1, since 0.36 plus 0.64 is 1, so the cosine of their angle is 0.6.

Turn it around. Fix a unit vector $\mathbf{x}$ and ask which unit vector $\mathbf{y}$ gives the largest dot product: $\mathbf{y}^* = \operatorname{argmax}_y \langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}$. Here argmax means "the value of y that makes the output as large as possible", and the answer is y equal to x itself, since the cosine reaches 1 only when the angle is zero. This is the idea the book wants to plant: later, $\mathbf{x}$ will be an input and $\mathbf{w}$ a set of adjustable parameters, and the dot product is largest when the input resonates with the parameters, which is called template matching. Your photo app works this way in spirit: it compares a photo's numbers with a learned "dog" pattern, and a large dot product means the photo matches the pattern.



$$(2, 5, 1) \cdot (1, 1, 1) = 8 = \text{sum}$$

The Best Match Is the Template

The book closes with an observation that will be useful later: the sum of a vector's entries can itself be written as a dot product, with the vector **1** whose entries are all ones; multiplying by one and adding just adds up the entries.

Try it yourself

Take two pencils and hold them both pointing at the ceiling: they agree, and their cosine similarity is 1. Turn one to point at the floor: cosine minus 1. Lay one flat while the other still points up: they form a right angle, and the dot product of their directions is 0. Every in-between angle gives a number in between, which is all the dot product measures once the lengths are set to one.

Remember

Vectors can be combined linearly entry by entry, $z_i = ax_i + by_i$, and their length is the Euclidean norm, the square root of the sum of squared entries, 5 for (3, 4). In code, a row or column of shape (1, d) or (d, 1) differs from a plain vector of shape (d), and broadcasting can silently turn a sum of two vectors into a 4 by 4 matrix. The dot product multiplies entries and adds them, giving one number, minus 0.43 in the book's example, and it equals the two lengths times the cosine of the angle between the vectors. For unit vectors the dot product is the cosine similarity, from 1 to minus 1 with 0 for right angles, and it is largest when the two vectors coincide, which is why a parameter vector w acts as a template that inputs match.

Questions

1. In a framework, you add a tensor of shape $(4, 1)$ to one of shape (4) . Why do you get a 4 by 4 result?
 - a. Because broadcasting repeats the column across columns and the vector across rows, and adds every pair
 - b. Because the framework always turns vector sums into matrices
 - c. Because a tensor of shape (4) is secretly a matrix
2. Two unit-length vectors have a dot product of 0. What does that say about them?
 - a. They point in the same direction
 - b. They point in opposite directions
 - c. They are at right angles to each other
3. Fixing a unit vector x , which unit vector y makes the dot product as large as possible, and why does that matter later?
 - a. Any vector orthogonal to x , because orthogonal vectors are independent
 - b. y equal to x , so a parameter vector w acts as a template that matching inputs resonate with
 - c. y equal to minus x , so inputs opposite to the parameters are detected
4. Why is the squared norm often handier than the norm itself?
 - a. Because it is always smaller than the norm

- b. Because it is the plain sum of the squared entries, with no square root
- c. Because it does not depend on the entries of the vector

Answers

1. **a)** Broadcasting aligns the shapes from the right and repeats tensors to make them match, so a column plus a one-dimensional vector becomes a 4 by 4 matrix instead of four sums.
2. **c)** For unit vectors the dot product is the cosine of the angle between them; it is 0 exactly when they are orthogonal, at right angles.
3. **b)** The cosine reaches 1 only at zero angle, so the maximum is $y = x$; with x an input and w the parameters, the dot product is largest when the input matches the template w .
4. **b)** The norm is the square root of the sum of squared entries, so its square is just that sum, 25 for the vector (3, 4).

Worked examples

The book's dot product Compute the inner product of $x = [0.1, 0, -0.3]$ and $y = [-4.0, 0.05, 0.1]$.

Multiply entry by entry: $0.1 \times (-4.0) = -0.4$.

$0 \times 0.05 = 0$.

$-0.3 \times 0.1 = -0.03$.

Add the products: $-0.4 + 0 - 0.03 = -0.43$, a single number.

Answer: -0.43

A broadcasting surprise x has shape (4, 1), a 'column', and y has shape (4), a one-dimensional tensor. What is the shape of $x + y$, and how many numbers does it hold?

Broadcasting lines up the shapes from the right: $(4, 1)$ against (4) .

The size 1 in x is repeated to match 4, and y is repeated along a new first dimension of size 4.

Every entry of x is added to every entry of y .

Answer: Shape $(4, 4)$, 16 numbers, instead of the 4 sums one probably wanted.

Practice

1. Find the norm of $x = (6, 8)$ and its squared norm.

Hint: Square each entry and add.

Answer: $36 + 64 = 100$, so the squared norm is 100 and the norm is $\sqrt{100} = 10$.

2. Compute $z = 3x - 2y$ for $x = (1, 4)$ and $y = (2, 1)$.

Hint: Use $z_i = ax_i + by_i$ with $a = 3$, $b = -2$.

Answer: $z_1 = 3 \times 1 - 2 \times 2 = -1$ and $z_2 = 3 \times 4 - 2 \times 1 = 10$, so $z = (-1, 10)$.

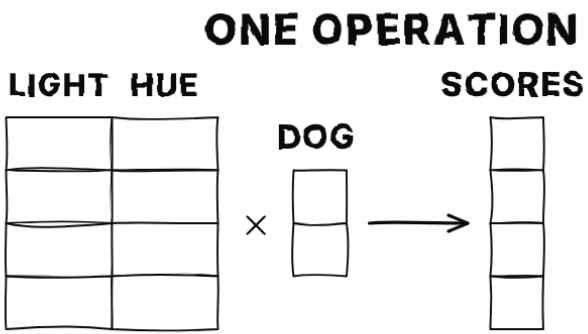
3. The unit vectors $(1, 0)$ and $(0.8, 0.6)$ and $(0, 1)$ are compared with $x = (1, 0)$. Rank them by cosine similarity with x .

Hint: For unit vectors the cosine similarity is just the dot product.

Answer: $(1, 0)$ gives 1, $(0.8, 0.6)$ gives 0.8, $(0, 1)$ gives 0; so the order is $(1, 0)$, then $(0.8, 0.6)$, then $(0, 1)$, which is orthogonal to x .

Matrices as Batches and as Functions

Open a spreadsheet and type four photos' features into it, one photo per row: brightness, colourfulness, and so on across the columns. Now you want to score every photo against the same "dog" pattern: multiply each row by the pattern and add up. You could do it row by row, four times, or with one single operation on the whole sheet. That single operation is matrix multiplication, and this part shows why it is the workhorse of every model.

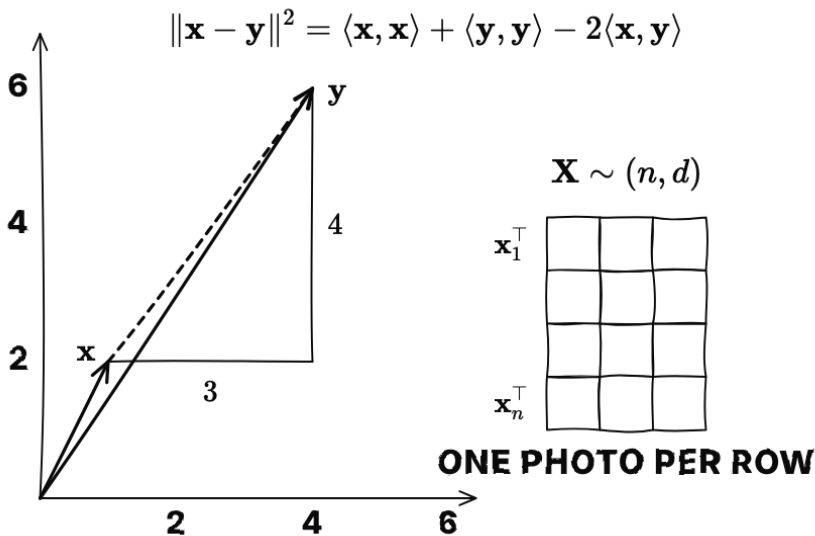


$$\langle \mathbf{x}, \mathbf{1} \rangle = (2, 5, 3) \cdot (1, 1, 1) = 10$$

Score Every Row at Once

First, two last facts about vectors that the book will use often. The sum of a vector's entries is its dot product with the all-ones vector: $\langle \mathbf{x}, \mathbf{1} \rangle = \sum_{i=1}^d x_i$. In words, multiplying every entry by one and adding them simply adds them up. For $\mathbf{x} = (2, 5, 3)$, that is 2 plus 5 plus 3, which is 10.

Second, the squared distance between two vectors can be written with dot products only: $\|\mathbf{x} - \mathbf{y}\|^2 = \langle \mathbf{x}, \mathbf{x} \rangle + \langle \mathbf{y}, \mathbf{y} \rangle - 2\langle \mathbf{x}, \mathbf{y} \rangle$. It comes from expanding a square, just like the school rule that $(a - b)^2$ is a^2 plus b^2 minus twice a times b . Check it with $\mathbf{x} = (1, 2)$ and $\mathbf{y} = (4, 6)$: directly, the differences are 3 and 4, and 9 plus 16 is 25 . With dot products, $\mathbf{x}$ with itself is 5 , $\mathbf{y}$ with itself is 52 , $\mathbf{x}$ with $\mathbf{y}$ is 16 , and 5 plus 52 minus 32 is again 25 . Setting $\mathbf{y}$ to the zero vector gives the special case $\|\mathbf{x}\|^2 = \langle \mathbf{x}, \mathbf{x} \rangle$: a vector's squared length is its dot product with itself.



Distance by Dot Products, Then a Batch

Now to two dimensions. A matrix $\mathbf{X} \sim (n, d)$ has n rows and d columns, with entries X_{11} in the top-left corner down to X_{nd} in the bottom-right. For what follows, the most useful way to see it is as a stack of n vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$, laid out as rows, each row being $\mathbf{x}_i^\top$. The book then says that $\mathbf{X}$ represents a batch of data vectors: our spreadsheet, one photo per row. Models are customarily defined, both on paper and in code, to work on batches like this.

The fundamental operation is matrix multiplication. For matrices of compatible shapes, $\mathbf{X} \sim (a, b)$ and $\mathbf{Y} \sim (b, c)$, the product $\mathbf{Z} = \mathbf{XY}$ has shape (a, c) , and each entry is $Z_{ij} = \langle \mathbf{X}_i, \mathbf{Y}_j^\top \rangle$. In words: the entry in row i and column j is the dot product of row i of the first matrix with column j of the second, which is why the inner sizes, both b , must

agree. Take the rows (1, 2) and (3, 4) as **X** and the rows (5, 6) and (7, 8) as **Y**. The top-left entry is 1 times 5 plus 2 times 7, which is 19; top-right, 1 times 6 plus 2 times 8, which is 22; bottom-left, 3 times 5 plus 4 times 7, which is 43; bottom-right, 3 times 6 plus 4 times 8, which is 50.

$$(a,b) \, (b,c) \rightarrow (a,c)$$

$$\mathbf{X} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \mathbf{Y} = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \mathbf{Z} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

$$1 \cdot 5 + 2 \cdot 7 = 19$$

$$\mathbf{W}\mathbf{x} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 10 \end{pmatrix} \qquad \mathbf{X}\mathbf{X}^\top = \begin{pmatrix} 5 & 11 \\ 11 & 25 \end{pmatrix}$$

Row Times Column

When the second factor is a single vector we get the matrix-vector product, **z** = **Wx**: each entry of **z** is the dot product of one row of **W** with **x**. With **W** having rows (1, 0) and (2, 1), and **x** equal to (3, 4), we get 3 and 10. Seen as batches, **XW^T** computes, in one linear-algebra operation, the dot product of every row of **X** with every pattern in **W**: our spreadsheet scored against the dog pattern, all photos at once. And **XX^T** ~ (n, n) computes all dot products between pairs of rows at

once: for the rows (1, 2) and (3, 4) it gives 5 and 11 in the first row and 11 and 25 in the second, every photo compared with every other.

A few more operations matter. The Hadamard product multiplies two matrices of the same shape entry by entry: $[\mathbf{X} \odot \mathbf{Y}]_{i\ j} = X_{i\ j} Y_{i\ j}$. It lacks the rich algebra of true matrix multiplication, but differentiable models use it constantly for masking, setting some entries to zero, and for scaling. Multiply the rows (1, 2) and (3, 4) entry by entry with the mask whose rows are (1, 0) and (0, 1), and you keep 1 and 4 and zero out the rest. Multiplicative interactions of this kind have also become popular in some recent model families, which the book comes back to later.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \odot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$$

MASK

$$[\exp(\mathbf{X})]_{ij} = \exp(X_{ij}), \quad \exp(0) = 1$$

$$\text{mat-exp} : 1 + 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} \approx 2.708 \approx e$$

SQUARE MATRICES ONLY

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \xrightarrow{\Sigma_j} \begin{pmatrix} 3 \\ 7 \end{pmatrix}$$

SUM ALONG AN AXIS

Entry by Entry, and Along an Axis

An expression such as $\exp(\mathbf{X})$ is meant element-wise: $[\exp(\mathbf{X})]_{i,j} = \exp(X_{i,j})$, the exponential taken of every entry separately, so a zero entry becomes 1. Compare the true matrix exponential of linear algebra, $\text{mat-exp}(\mathbf{X}) = \sum_{k=0}^{\infty} \mathbf{X}^k/k!$, the sum of the matrix's powers, each divided by k factorial. For a one-by-one matrix holding the number 1, the first terms are 1 plus 1 plus one half plus one sixth plus one twenty-fourth, about 2.708, already close to e , about 2.718. The element-wise version works for tensors of any shape, while the matrix exponential only makes sense for square matrices; that is why frameworks such as PyTorch keep such matrix-only operations, inverses and determinants too, in a separate linear-algebra module, as the book's code box shows with its two exponentials.

Reductions such as sums and means along an axis are written without limits when they run over the whole axis, $\sum_i X_i$, and in code they take an axis argument. Summing the rows (1, 2) and (3, 4) along the second axis gives one total per row, 3 and 7.

Why is matrix multiplication defined in this strange row-times-column way, rather than entry by entry like Hadamard? The book's answer: a function f of vectors is linear if $f(\alpha \mathbf{x}_1 + \beta \mathbf{x}_2) = \alpha f(\mathbf{x}_1) + \beta f(\mathbf{x}_2)$, and every such function can be written as a matrix $\mathbf{A}$. Then the matrix-vector product $\mathbf{A}\mathbf{x}$ is applying the function, and the matrix product $\mathbf{A}\mathbf{B}$ is composing two functions: $(f \circ g)(\mathbf{x}) = f(g(\mathbf{x}))$ with $g(\mathbf{x}) = \mathbf{B}\mathbf{x}$. Check: let $\mathbf{B}$ have rows (1, 1) and (0, 1), and $\mathbf{A}$ rows (2, 0) and (0, 3); for $\mathbf{x}$ equal to (1, 1), $\mathbf{B}$ gives (2, 1) and then $\mathbf{A}$ gives (4, 3). The product $\mathbf{A}\mathbf{B}$ has rows (2, 2) and (0, 3), and applied to (1, 1) it gives (4, 3) directly: doing one function after the other is the same as multiplying their matrices first.

$$f(\alpha \mathbf{x}_1 + \beta \mathbf{x}_2) = \alpha f(\mathbf{x}_1) + \beta f(\mathbf{x}_2)$$

$$(f \circ g)(\mathbf{x}) = f(g(\mathbf{x}))$$

$$(1, 1) \xrightarrow{\mathbf{B} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}} (2, 1) \xrightarrow{\mathbf{A} = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}} (4, 3)$$

=

$$(1, 1) \xrightarrow{\mathbf{AB} = \begin{pmatrix} 2 & 2 \\ 0 & 3 \end{pmatrix}} (4, 3)$$

Multiplying Matrices Composes Functions

Try it yourself

On squared paper, write the matrix with rows (1, 2) and (3, 4) on the left and the one with rows (5, 6) and (7, 8) on the right. Run a finger along the first row of the left matrix and down the first column of the right one, multiplying and adding: you should get 19. Do the other three entries and check 22, 43 and 50; you have done by hand what a phone's chip does billions of times when it tags your photos.

Remember

The sum of a vector's entries is its dot product with the ones vector, and the squared distance between two vectors can be written with dot products, $\|x - y\|^2 = \langle x, x \rangle + \langle y, y \rangle - 2\langle x, y \rangle$. A matrix is best seen as a batch of row vectors, and the entry Z_{ij} of a product XY is the dot product of row i of X with column j of Y , as 19, 22, 43, 50 in our example. XW^T scores every row of a batch against every pattern at once, and XX^T compares all pairs of rows; the Hadamard product multiplies entry by entry and is used for masking and scaling. Functions like $\exp$ act element-wise on tensors, unlike the true matrix exponential of square matrices; and matrix multiplication is defined the way it is because it composes linear functions.

Questions

1. In a matrix product $Z = XY$, what is the entry Z_{ij} ?
 - a. The product of the entries X_{ij} and Y_{ij}
 - b. The dot product of row i of X with column j of Y
 - c. The sum of row i of X plus column j of Y
2. Why is matrix multiplication defined row-times-column rather than entry by entry?
 - a. Because then multiplying matrices composes the linear functions they represent
 - b. Because it is faster to compute than entry-by-entry products

- c. Because entry-by-entry products are not allowed for square matrices
3. If X is a batch of photos, one per row, what does XX^T give?
 - a. The sum of all photos
 - b. Each photo multiplied entry by entry with itself
 - c. All dot products between pairs of photos, as an n by n matrix
 4. Why do frameworks keep the matrix exponential in a separate linear-algebra module, while $\exp$ works on any tensor?
 - a. Because the matrix exponential is only an approximation of $\exp$
 - b. Because element-wise $\exp$ works for tensors of any shape, while the matrix exponential is defined only for square matrices
 - c. Because $\exp$ cannot be applied to matrices at all

Answers

1. **b)** Each entry is the dot product of row i of the first matrix with column j of the second, which is why the inner sizes must agree; entry-by-entry multiplication is the Hadamard product.
2. **a)** Every linear function is a matrix; applying it is a matrix-vector product, and the matrix product AB is the function that applies $g = B$ first and then $f = A$.
3. **c)** XX^T has shape (n, n) , and its entry i, j is the dot product of rows i and j : every photo compared with every other in one operation.
4. **b)** $\exp(X)$ applies the exponential to every entry, valid for any shape, while the true matrix exponential sums powers of the matrix, which only makes sense for square matrices.

Worked examples

Multiplying two small matrices Compute XY for X with rows (1, 2) and (3, 4), and Y with rows (5, 6) and (7, 8).

Top-left: row 1 of X with column 1 of Y , $1 \times 5 + 2 \times 7 = 19$.

Top-right: row 1 with column 2, $1 \times 6 + 2 \times 8 = 22$.

Bottom-left: row 2 with column 1, $3 \times 5 + 4 \times 7 = 43$.

Bottom-right: row 2 with column 2, $3 \times 6 + 4 \times 8 = 50$.

Answer: XY has rows (19, 22) and (43, 50).

Distance from dot products For $x = (1, 2)$ and $y = (4, 6)$, compute $\|x - y\|^2$ directly and with $\langle x, x \rangle + \langle y, y \rangle - 2\langle x, y \rangle$.

Directly: $x - y = (-3, -4)$, and $9 + 16 = 25$.

$\langle x, x \rangle = 1 + 4 = 5$ and $\langle y, y \rangle = 16 + 36 = 52$.

$\langle x, y \rangle = 4 + 12 = 16$, so $2\langle x, y \rangle = 32$.

$5 + 52 - 32 = 25$, the same as the direct computation.

Answer: 25 both ways

Practice

1. Compute Wx for W with rows (2, 1) and (0, 3), and $x = (4, 5)$.

Hint: Each entry is the dot product of one row of W with x .

Answer: Row 1: $2 \times 4 + 1 \times 5 = 13$; row 2: $0 \times 4 + 3 \times 5 = 15$; so $Wx = (13, 15)$.

2. Mask the matrix with rows (5, 6) and (7, 8) using the Hadamard product with the mask whose rows are (0, 1) and (1, 0). What remains?

Hint: Multiply entry by entry.

Answer: The rows (0, 6) and (7, 0): the entries where the mask is 1 survive, the others become 0.

3. Let $g(x) = Bx$ with B having rows (1, 0) and (1, 1), and $f(x) = Ax$ with A having rows (3, 0) and (0, 2). Compute $f(g(x))$ for $x = (2, 1)$ in two ways: one function after the other, and with the single matrix AB .

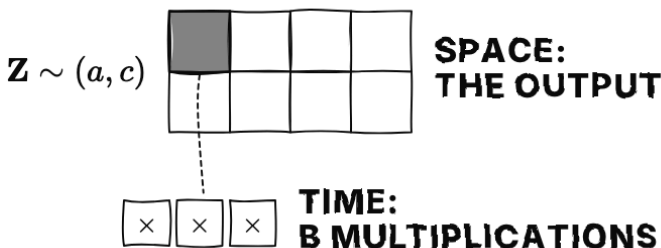
Hint: First Bx , then A times the result; then compute AB row by column.

Answer: $Bx = (2, 3)$, then A gives (6, 6). AB has rows (3, 0) and (2, 2), and AB times (2, 1) is (6, 6): the same, because the matrix product composes the functions.

How the Cost Grows

Time how long your phone takes to tag one small album, then imagine an album ten times wider in every direction. You might guess ten times longer, but for matrix multiplication the answer is about a thousand times longer. Counting how work grows with size, before you run anything, is the topic of this part.

10× WIDER → 1000× WORK



$$a = 2, b = 3, c = 4 : \quad 8 \times 3 = 24$$

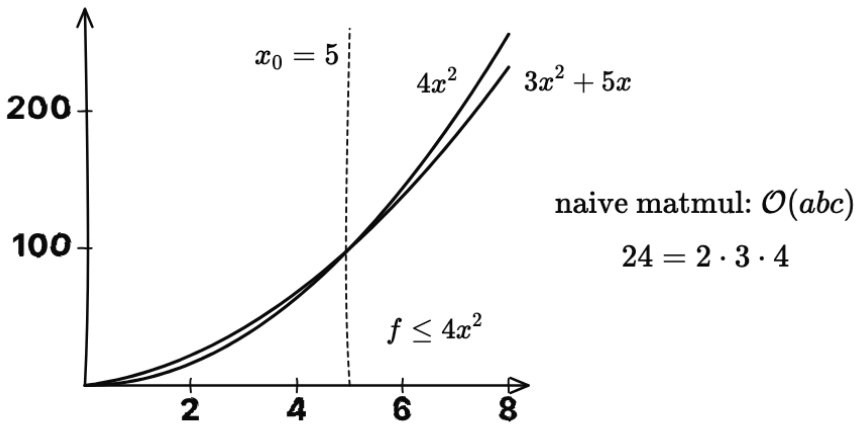
Counting the Work

The book uses matrix multiplication to introduce the complexity of an operation. To compute $\mathbf{Z} \sim (a, c)$ from $\mathbf{X} \sim (a, b)$ and $\mathbf{Y} \sim (b, c)$ straight from the definition, we need ac inner products, one per entry

of Z , each of length b . That count of work is called the time complexity. The memory a simple, sequential implementation needs is just the size of the output matrix, and that is called the space complexity. With a equal to 2, b equal to 3 and c equal to 4, Z has 8 entries, each a dot product of 3 multiplications, so 24 multiplications in all, and 8 numbers to store.

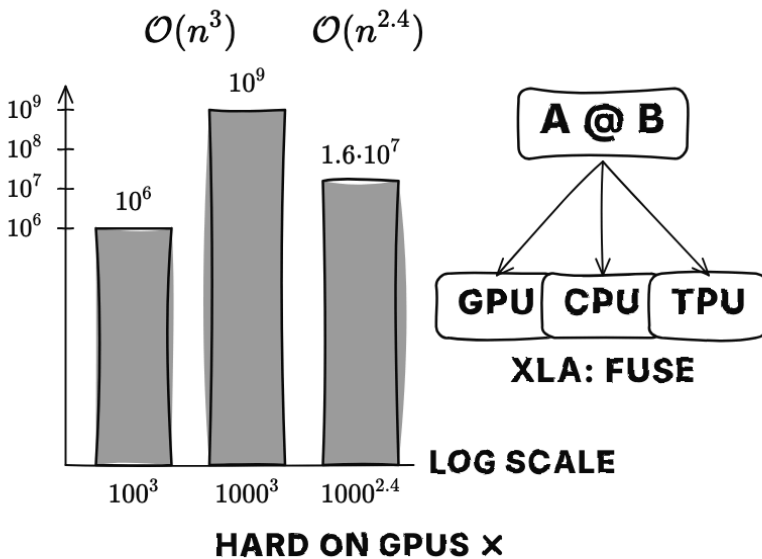
To get away from the details of particular hardware, computer science uses big-O notation, from the German word *Ordnung*, order of approximation. The definition: a function $f(x)$ is $\mathcal{O}(g(x))$, with inputs and outputs taken as non-negative, if there are a constant c and a starting value x_0 such that $f(x) \leq cg(x)$ for any $x \geq x_0$. In words: once x is large enough, f never exceeds a fixed multiple of g , so for large inputs everything except g can be ignored. That is called an asymptotic analysis. For example, $f(x) = 3x^2 + 5x$ is $\mathcal{O}(x^2)$: take c equal to 4 and x_0 equal to 5, because $3x^2 + 5x$ is at most $4x^2$ exactly when $5x$ is at most x^2 , which holds from 5 on.

$$3x^2 + 5x \leq 4x^2 \iff 5x \leq x^2$$



Big O: Eventually Below c·g

So a naive matrix multiplication is $\mathcal{O}(abc)$: the work grows in proportion to each of the three sizes, as our 24 equals 2 times 3 times 4 shows. For two square matrices of size (n, n) that is $\mathcal{O}(n^3)$, cubic in the size. With n equal to 100 it is a million multiplications; with n equal to 1,000, a billion: ten times wider, a thousand times the work, the number we started from.



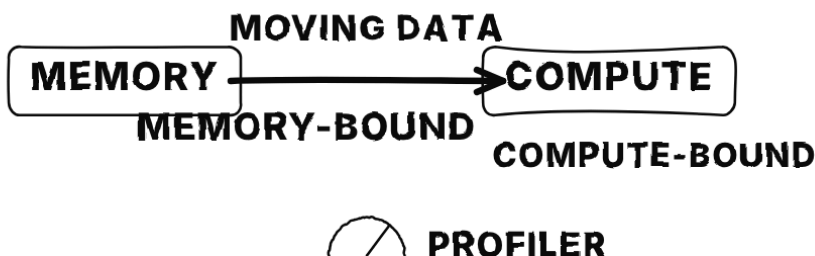
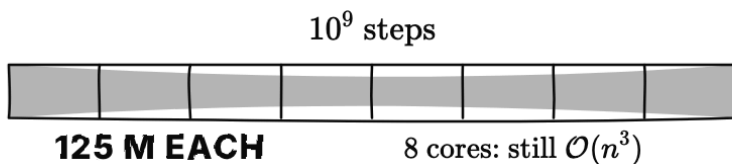
Cubic Cost, and What Runs It

Asymptotic thinking is important and elegant, the book says, but choosing an algorithm by its big-O alone does not guarantee real speed, which depends on the hardware, on how much parallel work it supports, and much else. When you call matrix multiplication in a framework, written `A @ B` in PyTorch, the low-level routine actually run, called the kernel, is chosen at run time for your hardware, a process known as dispatching; the same line of code can run on a GPU, a CPU or a TPU kernel. Compilers such as XLA make this even more involved, fusing and reshaping operations for a specific target chip.

The book's example: the best known algorithms for multiplying two n by n matrices scale as $\mathcal{O}(n^c)$ with a constant c below 2.4, much better

than the naive $\mathcal{O}(n^3)$. For n equal to 1,000, n to the 2.4 is about 16 million, against a billion for the cube. Yet these clever algorithms are hard to run efficiently in parallel on hardware such as GPUs, so in practice they are rarely used.

Two more limits of asymptotic thinking. First, having k processors does not change the big- $\mathcal{O}$ at all: at best it divides the time by k , a constant factor; eight cores turn a billion steps into 125 million each, still cubic in n . Second, big- $\mathcal{O}$ ignores the time spent moving data from one place to another, which is often the real bottleneck. When that happens the implementation is called memory-bound, as opposed to compute-bound, and the only practical way to tell which you have is to run a profiler, a tool that measures where the time actually goes. The book promises that judging an algorithm's cost will prove far from trivial, because asymptotic complexity and observed speed pull in different directions.



Two More Limits

Try it yourself

Take two 2 by 2 grids of numbers and multiply them by hand, counting multiplications: you will count 8. Now do two 3 by 3 grids: you will count 27. Doubling from 2 to 4 would give 64, eight times as many: the size went up two times, the work two times two times two, which is what cubic means.

Remember

Computing $Z \sim (a, c)$ from $X \sim (a, b)$ and $Y \sim (b, c)$ by definition takes ac dot products of length b , its time complexity, and needs memory for the output, its space complexity. Big-O says $f(x)$ is $O(g(x))$ when f stays below c times g from some x_0 on, so only the fastest-growing part matters; naive matrix multiplication is $O(abc)$, cubic for square matrices. Real speed depends on hardware, kernels and compilers: algorithms below n to the 2.4 exist but run poorly on GPUs, and k processors only divide time by a constant. Moving data can dominate, making code memory-bound rather than compute-bound, and only a profiler tells which.

Questions

1. Two square matrices grow from size 100 to size 1,000. Roughly how much more work does naive multiplication need?
 - a. About 10 times more
 - b. About 100 times more
 - c. About 1,000 times more
2. Why are the algorithms that multiply matrices in less than n to the 2.4 steps rarely used in practice?
 - a. Because they give slightly wrong results
 - b. Because they are hard to parallelize efficiently on hardware such as GPUs

- c. Because they need more memory than the output matrix
3. Why does having k processors not change the big-O complexity of an algorithm?
- a. Because at best it divides the time by the constant k , and big-O ignores constants
 - b. Because processors cannot share memory
 - c. Because big-O only counts the slowest processor
4. Code is memory-bound. What does that mean, and how would you find out?
- a. It runs out of memory; you find out from an error message
 - b. It uses too many multiplications; big-O analysis reveals it
 - c. Moving data, not computing, is the bottleneck; a profiler shows it

Answers

1. **c)** Naive multiplication of n by n matrices is $O(n^3)$: ten times the size gives ten times ten times ten, a thousand times the work, from a million to a billion multiplications.
2. **b)** Their asymptotic cost is much better, but real performance depends on hardware and parallelism, and these algorithms run poorly on highly parallel chips.
3. **a)** Parallelism can give at most a constant $1/k$ speedup, and asymptotic analysis ignores constant factors.
4. **c)** Big-O ignores data movement, which often dominates; such code is memory-bound rather than compute-bound, and only running a profiler shows which it is.

Worked examples

Counting the work of a product X has shape (2, 3) and Y has shape (3, 4). How many dot products, and how many multiplications, does computing $Z = XY$ by definition need, and how many numbers must be stored?

Z has shape $(a, c) = (2, 4)$, so it has $ac = 8$ entries.

Each entry is a dot product of length $b = 3$, so 3 multiplications each.

In all, $8 \times 3 = 24$ multiplications, which equals $abc = 2 \times 3 \times 4$.

A sequential implementation stores the output, 8 numbers.

Answer: 8 dot products, 24 multiplications, 8 numbers stored

Checking a big-O claim Show that $f(x) = 3x^2 + 5x$ is $O(x^2)$ by finding c and x_0 .

Try $c = 4$: we need $3x^2 + 5x \leq 4x^2$.

Subtract $3x^2$: the condition becomes $5x \leq x^2$.

For positive x , divide by x : $5 \leq x$, so it holds for every $x \geq 5$.

Answer: $c = 4$ and $x_0 = 5$ work, so f is $O(x^2)$.

Practice

1. How many multiplications does a naive product of two 10 by 10 matrices need, and of two 20 by 20 matrices?

Hint: Use n^3 for square matrices.

Answer: $10^3 = 1,000$ and $20^3 = 8,000$: doubling the size multiplies the work by eight.

2. A product of $X \sim (1000, 50)$ and $Y \sim (50, 10)$ is computed by definition. What is its time complexity in multiplications, and how many numbers does the output hold?

Hint: abc multiplications; the output has shape (a, c).

Answer: $1000 \times 50 \times 10 = 500,000$ multiplications; the output has shape (1000, 10), 10,000 numbers.

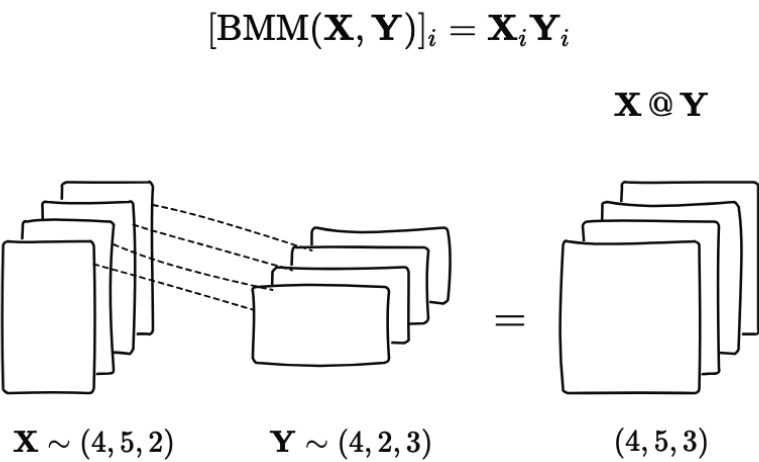
3. Explain why choosing between two algorithms only by their big-O can mislead you on a real phone or GPU.

Hint: Think of parallel hardware, kernels and moving data.

Answer: Big-O ignores constant factors, parallelism and data movement. A theoretically faster algorithm may parallelize badly on a GPU, the kernel actually run depends on the hardware, and moving data may be the real bottleneck, which only a profiler reveals.

Many Products at Once, and a Shorthand for All of Them

Picture a stack of four sheets of squared paper, each holding a small table of numbers, and next to it a second stack of four. You want to multiply sheet one by sheet one, sheet two by sheet two, and so on, four separate matrix products done side by side. Nothing new is happening inside each pair; the only new thing is the stack, and this part shows how frameworks handle such stacks, and a neat way to write any of these operations as a short string of letters.



$$4 \times (5 \cdot 2 \cdot 3) = 4 \times 30 = 120$$

One Product per Sheet

Vectors and matrices are special, the book says, because many operations are defined for them that are undefined or awkward in more dimensions: the matrix exponential, matrix multiplication, determinants. In higher dimensions, most operations we need are either batched versions of matrix operations, or combinations of matrix operations with reductions such as sums.

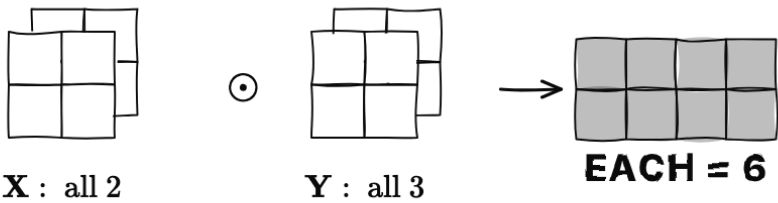
First the batched kind. Take two tensors $X \sim (n, a, b)$ and $Y \sim (n, b, c)$, two stacks of n matrices each. Batched matrix multiplication, BMM, is $[\text{BMM}(X, Y)]_i = X_i Y_i \sim (n, a, c)$. In words: for each position i in the stack, multiply the i -th matrix of X by the i -th matrix of Y , and stack the n results. Where it comes from is nothing more than our sheets of paper, one ordinary matrix product per sheet. The book's code box uses stacks of 4: X of shape 4 by 5 by 2 and Y of shape 4 by 2 by 3 give a result of shape 4 by 5 by 3. Counting the work: each product is 5 times 2 times 3, which is 30 multiplications, and there are 4 of them, so 120 in all.

Frameworks handle such stacks transparently: they assume the batch dimensions come first, the leading dimensions, and in PyTorch batched multiplication is written exactly like ordinary multiplication, with `matmul` or the `@` sign. Your photo app does this constantly: a whole batch of photos goes through the same layer, each photo's matrix multiplied by the same weights, all in one call.

Now the reduction kind. For two tensors of the same shape, $X, Y \sim (a, b, c)$, a generalized dot product, GDT, is $\text{GDT}(X, Y) = \sum_{i, j, k} [X \odot Y]_{i j k}$. In words: multiply the two tensors entry by entry, then add up everything. It is simply the ordinary dot product of the two tensors after flattening each into one long vector. For two tensors of shape 2 by 2 by 2, with every entry of X equal to 2 and every entry of Y equal

to 3, there are 8 products of 6, so the result is 48. These two kinds, the book says, cover most of the tensor operations used in the rest of the book.

$$\text{GDT}(\mathbf{X}, \mathbf{Y}) = \sum_{i,j,k} [\mathbf{X} \odot \mathbf{Y}]_{ijk}$$



FLATTEN, THEN DOT PRODUCT

$$8 \times 6 = 48$$

EINSUM →

Multiply Everything, Add Everything

The next section, marked as optional, introduces einsum, a convention that can express practically every tensor operation, sums, products and reductions, with a simple text string. Write the two examples with every index visible. Batched multiplication: $M_{ijk} = \sum_z A_{ijz} B_{zjk}$. Generalized dot product: $M = \sum_i \sum_j \sum_k X_{ijk} Y_{ijk}$.

$$M_{ijk} = A_{ijz} B_{izk}$$

$$M = X_{ijk} Y_{ijk}$$

MISSING FROM THE LEFT → SUMMED

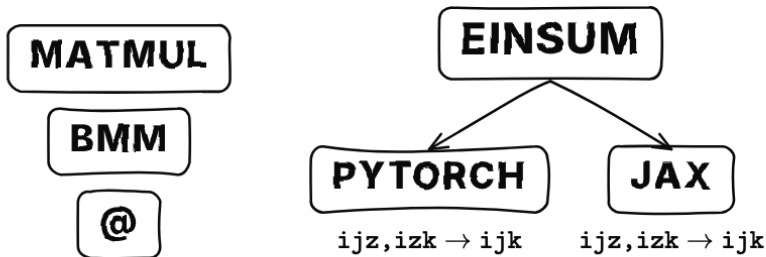
$$\begin{array}{ccc} \mathbf{ijz, izk} & \rightarrow & \mathbf{ijk} \\ \mathbf{A} & \mathbf{B} & \mathbf{OUT} \\ \mathbf{ijk, ijk} & \rightarrow & \end{array}$$

From Sums to a String

Following Einstein's notation, in a simplified form that ignores the difference between upper and lower indices, we drop the sum signs, with one rule: any index that appears on the right-hand side but not on the left is summed over. So $M_{ijk} = A_{ijz} B_{izk}$ means "sum over z", because z is missing from M, and $M = X_{ijk} Y_{ijk}$ means "sum over all three", because M has no index at all.

Then we keep only the index letters, operands first, result after an arrow. Batched matrix multiply becomes 'ijz,izk→ijk'. Generalized dot product becomes 'ijk,ijk→'. Read the first aloud: the first tensor has axes i, j, z; the second i, z, k; the output keeps i, j, k; z disappears, so it is summed. There is a one-to-one match between the formulas and

the strings, and most frameworks implement this as an einsum operation.



DIFFERENT RULES

$$A \sim (n, a, b), B \sim (n, c, b)$$

EINOPS

$$ijz, ikz \rightarrow ijk$$

One Syntax Everywhere

The advantage, the book says, is that you no longer need to remember each framework's interface. PyTorch alone has several multiplication functions, such as `matmul` and `bmm`, with different broadcasting rules and shape demands, while `einsum` gives one uniform syntax for all of them. Moving between frameworks becomes transparent too: the `einsum` string for batched multiplication is exactly the same in JAX as in PyTorch, as the book's two code boxes show.

Transposed axes are easy as well. For $A \sim (n, a, b)$ and $B \sim (n, c, b)$, multiplying each matrix of A by the transpose of the matching matrix

of B only needs the letters of B swapped: 'ijz,ikz→ijk'. The summed letter z now sits last in both inputs, which is exactly what multiplying by a transpose means. For these reasons einsum and its relatives, such as the popular einops package, have become widely used.

Try it yourself

Write the string 'ijz,ikz→ijk' on a card and, underneath, the shapes 4 by 5 by 2 and 4 by 2 by 3. Match letters to sizes: i is 4, j is 5, z is 2, k is 3. Read off the output i, j, k: 4 by 5 by 3, the same shape the book's code prints, and z, the letter that vanished, is the size 2 that was summed away.

Remember

In more than two dimensions, most operations are batched matrix operations or matrix operations combined with reductions. Batched matrix multiplication multiplies matching matrices in two stacks, (n, a, b) with (n, b, c) giving (n, a, c), and frameworks treat leading dimensions as batches automatically. The generalized dot product multiplies two same-shaped tensors entry by entry and sums everything, a dot product of the flattened tensors. Einsum writes any such operation as a string of index letters, summing every letter missing from the output, as in 'ijz,ikz→ijk' or 'ijk,ijk→', with the same syntax across frameworks.

Questions

1. X has shape (n, a, b) and Y has shape (n, b, c). What does batched matrix multiplication produce?
 - a. One matrix of shape (a, c), the sum of all n products
 - b. A stack of n matrices of shape (a, c), one product per position
 - c. A single number, the dot product of the flattened tensors
2. In the einsum string 'ijz,izk→ijk', why is the index z summed over?
 - a. Because it appears in both inputs but not in the output
 - b. Because it is the last letter of the alphabet used
 - c. Because it is the batch dimension
3. What does the string 'ijk,ijk→' compute?
 - a. A tensor of the same shape with entries multiplied
 - b. A batched matrix product
 - c. The generalized dot product: multiply entry by entry and sum everything into one number
4. Why does the book find einsum convenient across frameworks?
 - a. Because it runs faster than any built-in multiplication
 - b. Because one uniform string replaces each framework's different functions and is the same in PyTorch and JAX
 - c. Because it avoids the need for batch dimensions

Answers

1. **b)** For each position i in the stack, the i -th matrix of X is multiplied by the i -th matrix of Y , and the n results are stacked into shape (n, a, c) .
2. **a)** The convention is that any index appearing on the right-hand side but not in the result is summed; z is missing from ijk , so it is summed away.
3. **c)** The output has no index at all, so every index is summed over: entry-by-entry products added up, the dot product of the flattened tensors.
4. **b)** PyTorch alone has several multiplication functions with different rules, while einsum gives one syntax for all of them, and the same string works in JAX.

Worked examples

The book's batched product In the book's code box, X has shape $(4, 5, 2)$ and Y has shape $(4, 2, 3)$. Find the shape of $X @ Y$ and count the multiplications done by definition.

The leading dimension 4 is the batch: there are 4 separate matrix products.

Each product multiplies a 5 by 2 matrix by a 2 by 3 matrix, giving 5 by 3.

So the result has shape $(4, 5, 3)$.

Each product needs $5 \times 2 \times 3 = 30$ multiplications, and $4 \times 30 = 120$ in all.

Answer: Shape $(4, 5, 3)$; 120 multiplications

Practice

1. X and Y both have shape $(2, 2, 2)$; every entry of X is 1 and every entry of Y is 5. What is their generalized dot product?

Hint: Count the entries and multiply each pair.

Answer: There are $2 \times 2 \times 2 = 8$ entries, each product is $1 \times 5 = 5$, so the sum is 40.

2. A has shape (8, 10, 6) and B has shape (8, 6, 4). Using 'ijz,izk→ijk', give the size of each letter and the output shape.

Hint: Match the letters of each operand to its sizes.

Answer: i = 8, j = 10, z = 6, k = 4; the output keeps i, j, k, so it has shape (8, 10, 4), and z = 6 is summed over.

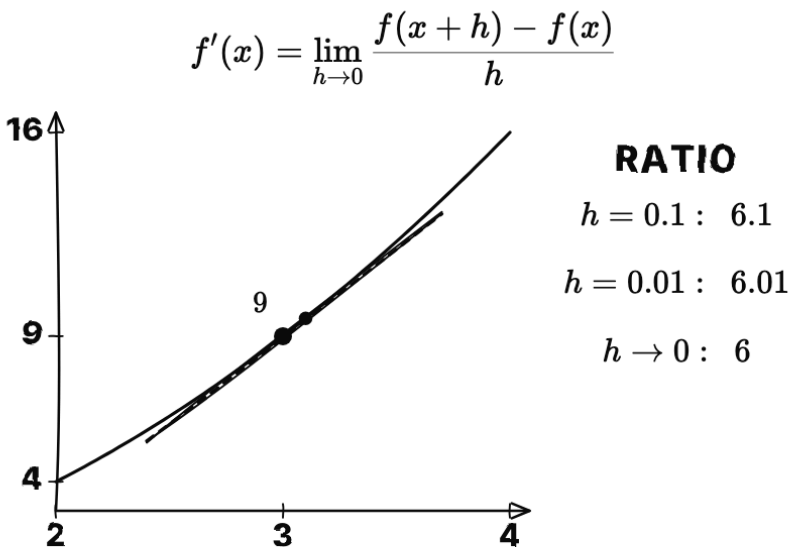
3. A has shape (n, a, b) and B has shape (n, c, b). Which einsum string multiplies each matrix of A by the transpose of the matching matrix of B, and what is the output shape?

Hint: The summed letter must sit on the axis of size b in both inputs.

Answer: 'ijz,ikz→ijk': z is the shared size b, summed over, and the output has shape (n, a, c).

The Slope at a Point

Open a calculator, type 3, and square it: 9. Now type 3.1 and square it: 9.61, which grew by 0.61 for a step of 0.1, about six times the step. Try 3.01: 9.0601, a growth of 0.0601, closer still to six times the step. The ratio is settling on a single number, and that number is the derivative, the foundation of everything the book calls differentiable.



Ratios Settling on 6

Gradients play a central role in the book, as the word differentiable suggests: they let models be optimized by semi-automatic methods derived from gradient descent. This section recalls the basic

definitions for differentiating functions, focusing on what will be needed later, sometimes at the cost of full mathematical rigour.

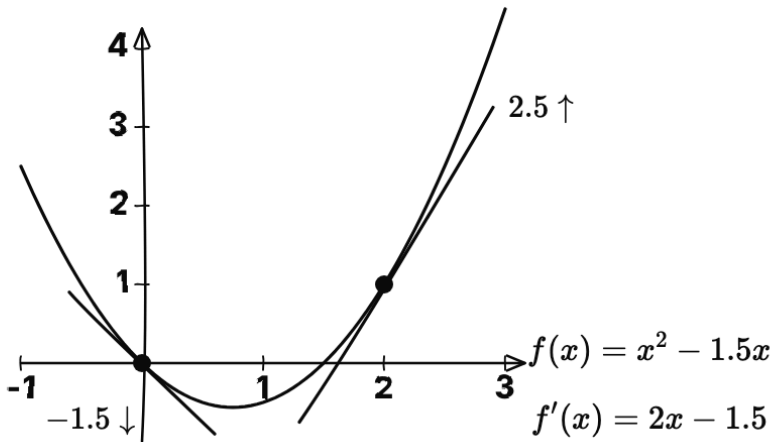
Start with the simplest case, $y = f(x)$, one number in and one number out. Its derivative is defined as $f'(x) = \lim_{h \rightarrow 0} (f(x + h) - f(x)) / h$. In words: step a little, by h , see how much the output changed, divide by the step, and let the step shrink towards zero. That is exactly what our calculator did: with f equal to x squared at 3, steps of 0.1 and 0.01 gave ratios of 6.1 and 6.01, closing in on 6.

The book uses several notations. The symbol ∂ stands for derivatives and gradients of any dimension; ∂_x , or ∂ over ∂x , names the variable we differentiate with respect to when that needs saying; and $f'(x)$, called Lagrange's notation, is kept for functions of one number. The book simply assumes the derivatives it needs exist, which is not guaranteed even for continuous functions, and returns to the point only for functions with corners, such as the absolute value $f(x) = \text{abs}(x)$ at zero, in Chapter 6.

For simple functions, applying the definition gives the familiar rules. For a power, $\partial x^p = p x^{p-1}$: the exponent comes down in front and drops by one; for x cubed at 2, that is 3 times 2 squared, which is 12. For the logarithm, $\partial \log(x) = 1/x$: at 2 it is one half. For the sine, $\partial \sin(x) = \cos(x)$: at zero it is the cosine of zero, which is 1.

Geometrically, the derivative is the slope of the tangent line through a point, or equally the best straight-line approximation of the function at that point. That is the view that matters: the slope tells how the function behaves close by. With a positive slope the function rises to the right and falls to the left, for a small enough interval; with a negative slope the reverse. The book's figure plots $f(x) = x^2 - 1.5x$ with tangents at two points. Its derivative is $2x - 1.5$: at zero the slope is

minus 1.5, so the curve is going down there, and at 2 it is plus 2.5, so the curve is going up. This reading of the slope will carry over to functions of many variables.



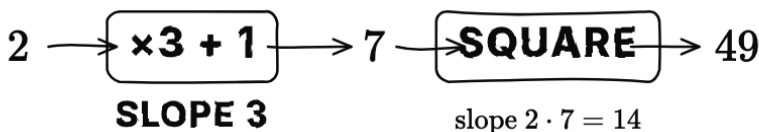
Down at Zero, Up at Two

Three properties extend to many dimensions. Linearity: the derivative of a sum is the sum of the derivatives, $\partial[f(x) + g(x)] = f'(x) + g'(x)$; for x squared plus sine x , the derivative is $2x$ plus cosine x . The product rule: $\partial[f(x)g(x)] = f'(x)g(x) + f(x)g'(x)$; with f equal to x and g equal to x squared, that is 1 times x squared plus x times $2x$, which is $3x$ squared, exactly the derivative of x cubed.

$$\partial[f + g] = f' + g'$$

$$\partial[fg] = f'g + fg'$$

$$\partial f(g(x)) = f'(g(x)) g'(x)$$



$$3 \times 14 = 42$$

A NETWORK IS A LONG CHAIN

Rates Multiply Along a Chain

And the chain rule, the one this whole book runs on: $\partial[f(g(x))] = f'(g(x)) g'(x)$. In words, when one function feeds another, the rates multiply. Take g equal to $3x$ plus 1 and f equal to the square, at x equal to 2: g gives 7 and has slope 3, the square has slope 2 times 7, which is 14, at 7, and the product is 42. That is the calculator chain, two, times three, plus one, squared: a tiny change in the starting 2 moves the final 49 forty-two times as much. A neural network is a long chain of such functions, and the chain rule is how its slopes are found.

Try it yourself

With a calculator, compute 2 times 3 plus 1, squared: 49. Now start from 2.01 instead: 2.01 times 3 plus 1 is 7.03, and squared it is 49.4209. The output grew by 0.4209 for a step of 0.01, about 42 times the step, just as the chain rule predicted by multiplying 14 by 3.

Remember

The derivative $f'(x)$ is the limit of the change in output divided by a shrinking step h , 6 for x squared at 3. Familiar rules follow: x^p gives px^{p-1} , $\log x$ gives 1 over x , sine gives cosine; the notation ∂ covers any dimension and f' is for scalar functions. The derivative is the slope of the tangent, the best straight-line approximation: positive means rising, negative falling, as for $x^2 - 1.5x$ at 2 and at 0. Derivatives are linear, follow the product rule, and obey the chain rule, $f'(g(x))$ times $g'(x)$, which multiplies rates along a chain, as 14 times 3 gives 42.

Questions

1. The derivative of $f(x) = x^2 - 1.5x$ is $2x - 1.5$. What does it say about the function near $x = 0$?
 - a. The function is falling there, since the slope is -1.5
 - b. The function is rising there, since the slope is 1.5

- c. The function is flat there, since x is zero
2. Why is the chain rule the rule 'this whole book runs on'?
 - a. Because neural networks contain no sums or products
 - b. Because a network is a long chain of functions, and the chain rule multiplies their rates to find the slope of the whole
 - c. Because it is the only rule valid for functions with corners
 3. In the definition of the derivative, what role does h play?
 - a. It is the height of the function at x
 - b. It is a fixed step of 0.1 used in every calculation
 - c. It is a step that shrinks towards zero, and the ratio of change in output to h approaches the derivative

Answers

1. **a)** At 0 the slope is $2 \times 0 - 1.5 = -1.5$; a negative slope means the function decreases to the right and increases to the left, in a small enough interval.
2. **b)** When one function feeds another, the derivative is $f'(g(x))$ times $g'(x)$; a network is a long composition, so its slopes come from multiplying rates along the chain.
3. **c)** The derivative is the limit of $(f(x + h) - f(x)) / h$ as h goes to zero; for x^2 at 3 the ratios 6.1 and 6.01 close in on 6.

Worked examples

The derivative by shrinking steps For $f(x) = x^2$ at $x = 3$, compute the ratio $(f(x + h) - f(x)) / h$ for $h = 0.1$ and $h = 0.01$, and compare with the rule $\partial x^p = p x^{p-1}$.

$$h = 0.1: 3.1^2 = 9.61, \text{ so } (9.61 - 9) / 0.1 = 6.1.$$

$h = 0.01$: $3.01^2 = 9.0601$, so $(9.0601 - 9) / 0.01 = 6.01$.

The ratios approach 6 as h shrinks.

The power rule gives $2 \times 3^1 = 6$, the limit.

Answer: 6.1, then 6.01, approaching the derivative 6

The chain rule on the calculator chain With $g(x) = 3x + 1$ and $f(u) = u^2$, find the derivative of $f(g(x))$ at $x = 2$.

$g(2) = 7$, and $g'(x) = 3$.

$f'(u) = 2u$, so $f'(g(2)) = 2 \times 7 = 14$.

Chain rule: $f'(g(2)) \times g'(2) = 14 \times 3 = 42$.

Check: from 2.01, $(3 \times 2.01 + 1)^2 = 7.03^2 = 49.4209$, a growth of 0.4209 for a step of 0.01, about 42 times.

Answer: 42

Practice

1. Use the power and linearity rules to find the derivative of $x^3 + 4x^2$ at $x = 1$.

Hint: Differentiate each term and add.

Answer: $3x^2 + 8x$, which at 1 is $3 + 8 = 11$.

2. Use the product rule on $x \cdot \log(x)$ and evaluate the derivative at $x = 1$.

Hint: $f = x$ has $f' = 1$; $g = \log x$ has $g' = 1/x$.

Answer: $1 \times \log(x) + x \times (1/x) = \log(x) + 1$, which at 1 is $0 + 1 = 1$.

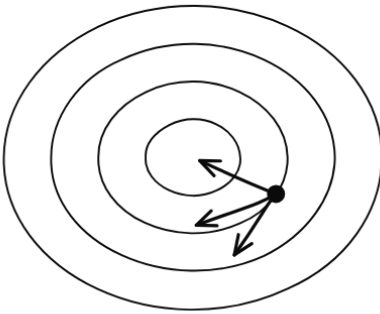
3. Find the derivative of $\sin(2x)$ at $x = 0$ with the chain rule.

Hint: The outer function is sine, the inner $2x$.

Answer: $\cos(2x) \times 2$, which at 0 is $\cos(0) \times 2 = 2$.

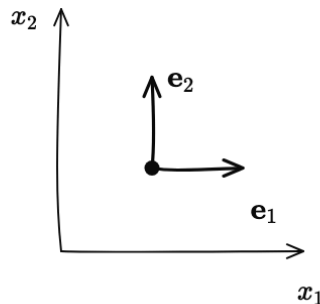
Slopes in Many Directions

Stand on a hillside and ask how steep it is. The honest answer is: it depends which way you face. Straight uphill it is steep, along the slope it is flat, and every direction in between has its own steepness. With a function of one number there were only two directions, left and right; with a function of many numbers there are infinitely many, and this part shows that one vector, the gradient, answers for all of them.



— **STEEP**
— **FLAT**

$$\partial_{x_i} f = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}$$



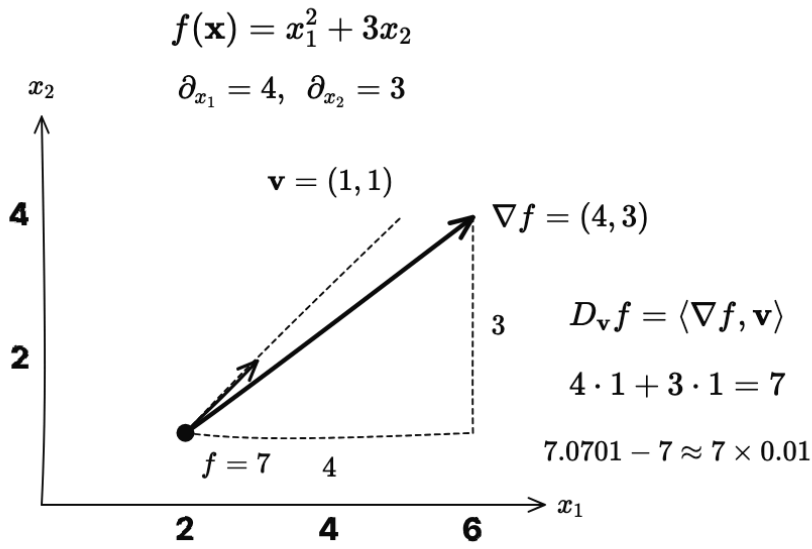
PARTIAL DERIVATIVE

Steepness Depends on Direction

Take a function $y = f(\mathbf{x})$ whose input is a vector $\mathbf{x} \sim (d)$. A tiny nudge means nothing until we say which way. The simplest choice is to move

along one axis, the i -th, keeping all the other inputs fixed: $\partial_{x_i} f(\mathbf{x}) = \partial y / \partial x_i = \lim_{h \rightarrow 0} (f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})) / h$. Here $\mathbf{e}_i$ is the i -th basis vector, the i -th row of the identity matrix: its j -th entry is 1 when j equals i and 0 otherwise, so adding h times it nudges only the i -th input. This is called a partial derivative: the ordinary derivative, taken along one axis.

Let us use a small function throughout: $f(\mathbf{x}) = x_1^2 + 3x_2$, at the point $\mathbf{x} = (2, 1)$, where f is 4 plus 3, which is 7. Moving only x_1 , the slope is $2x_1$, which is 4; moving only x_2 , the slope is 3.



The Gradient Answers Every Direction

Stacking all the partial derivatives into one column gives a d -dimensional vector, the gradient: $\nabla f(\mathbf{x}) = \partial f(\mathbf{x}) = [\partial_{x_1} f(\mathbf{x}), \dots, \partial_{x_d} f(\mathbf{x})]$

$f(\mathbf{x})]^T$. Because it is so fundamental, it gets its own symbol, the upside-down triangle ∇ . For our function the gradient at (2, 1) is (4, 3).

What about a nudge in a general direction $\mathbf{v}$? That gives the directional derivative, $D_{\mathbf{v}} f(\mathbf{x}) = \lim_{h \rightarrow 0} (f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})) / h$.

Since any movement can be split into separate movements along each axis, the directional derivative turns out to be a dot product: $D_{\mathbf{v}} f(\mathbf{x}) = \langle \nabla f(\mathbf{x}), \mathbf{v} \rangle = \sum_i \partial_{x_i} f(\mathbf{x}) v_i$. Each term is the slope along one axis times how far we move along that axis. For $\mathbf{v} = (1, 1)$: 4 times 1 plus 3 times 1, which is 7. Check it directly: at (2.01, 1.01), f is 4.0401 plus 3.03, which is 7.0701, a change of 0.0701 for a step of 0.01, about 7 times the step. So knowing the gradient is enough to know the slope in every possible direction, which is the hillside answered by a single vector.

Now the general case: a vector input $\mathbf{x} \sim (d)$ and a vector output $\mathbf{y} \sim (o)$, which the book says is the most general case we will need. With several outputs we take one gradient per output, and stacking them as rows gives an o by d matrix, the Jacobian: $\partial f(\mathbf{x}) \sim (o, d)$, whose entry in row j and column i is $\partial y_j / \partial x_i$. Take $f(\mathbf{x}) = (x_1 x_2, x_1 + x_2)$ at (2, 3): the first output has slopes 3 and 2, the second 1 and 1, so the Jacobian has rows (3, 2) and (1, 1). With one output, o equal to 1, the Jacobian is the gradient; with one input and one output it is the ordinary derivative.

$$\partial f(\mathbf{x}) \sim (o, d), \quad [\partial f]_{ji} = \frac{\partial y_j}{\partial x_i}$$

$$f(\mathbf{x}) = (x_1 x_2, x_1 + x_2) \text{ at } (2, 3) : \begin{pmatrix} 3 & 2 \\ 1 & 1 \end{pmatrix}$$

O = 1: THE GRADIENT

$$\partial[f(g(\mathbf{x}))] = [\partial f] \partial g$$

$$g(3, 1) = (4, 2), \quad \partial f = (2, 4)$$

$$\begin{pmatrix} 2 & 4 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = (6, -2)$$

$$\nabla(x_1^2 - x_2^2) \text{ at } (3, 1) = (6, -2)$$

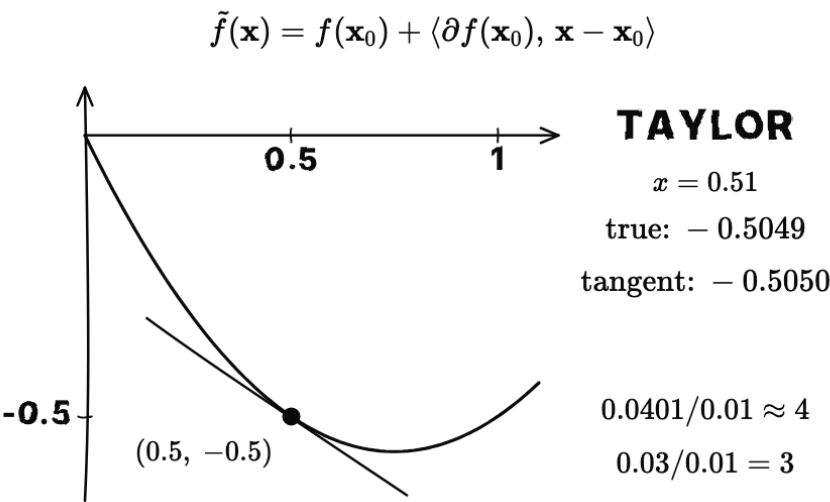
Jacobians Multiply

Jacobians inherit the properties of derivatives, and above all the chain rule becomes a matrix product: $\partial[f(g(\mathbf{x}))] = [\partial f(\bullet)] \partial g(\mathbf{x})$, with the first evaluated at $g(\mathbf{x}) \sim (h)$. Check with $g(\mathbf{x}) = (x_1 + x_2, x_1 - x_2)$ and $f(\mathbf{u}) = u_1 u_2$, at $\mathbf{x}$ equal to $(3, 1)$. g gives $\mathbf{u}$ equal to $(4, 2)$, with Jacobian rows $(1, 1)$ and $(1, -1)$; f has gradient (u_2, u_1) , which is $(2, 4)$ there.

Multiplying, 2 times 1 plus 4 times 1 is 6, and 2 times 1 plus 4 times minus 1 is minus 2: the row $(6, -2)$. Directly, f of g is $x_1^2 - x_2^2$, whose gradient at $(3, 1)$ is $(6, -2)$ as well.

As in one dimension, gradients and Jacobians are linear functions tangent at a point. The gradient gives the best first-order approximation near $\mathbf{x}_0$: $\tilde{f}(\mathbf{x}) = f(\mathbf{x}_0) + \langle \partial f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle$, the value at the point plus the slope times the displacement. This is Taylor's theorem.

The book's code box checks it for $f(x) = x^2 - 1.5x$ at 0.5, where f is minus 0.5 and the slope is minus 0.5: one step of 0.01 gives a true value of minus 0.5049 and an approximation of minus 0.5050, almost identical.



The Tangent Approximation

Try it yourself

On a calculator, evaluate $x_1^2 + 3x_2$ at $(2, 1)$, giving 7, and then at $(2.01, 1)$, giving 7.0401, and at $(2, 1.01)$, giving 7.03. The two changes, 0.0401 and 0.03, divided by the step 0.01, are about 4 and 3, the partial derivatives. Now step both together to $(2.01, 1.01)$: 7.0701, a change of about 7 times the step, the sum the gradient predicted.

Remember

For a vector input, a partial derivative nudges one input along its axis; stacking all of them gives the gradient $\nabla f(x)$, $(4, 3)$ for $x_1^2 + 3x_2$ at $(2, 1)$. The directional derivative along any v is the dot product of the gradient with v , so the gradient gives the slope in every direction. For a vector output, one gradient per output stacks into the Jacobian, an o by d matrix, and the chain rule becomes a product of Jacobians, as $(6, -2)$ showed both ways. Near a point, f is approximated by its value plus the gradient dotted with the displacement, Taylor's theorem, checked as -0.5049 against -0.5050 .

Questions

1. Why is knowing the gradient enough to find the slope of a function in any direction?

- a. Because the directional derivative along v is the dot product of the gradient with v
 - b. Because a function of many inputs has only one direction that matters
 - c. Because the gradient lists the slopes in every possible direction one by one
2. A function takes a vector of size 5 and returns a vector of size 3. What is the shape of its Jacobian?
- a. (5, 3)
 - b. (3, 5)
 - c. (15)
3. How does the chain rule look for Jacobians?
- a. The Jacobians of the two functions are added
 - b. The Jacobians are multiplied entry by entry
 - c. The Jacobian of the outer function, evaluated at $g(x)$, is matrix-multiplied by the Jacobian of g
4. What does Taylor's theorem, in its first-order form, say about f near a point x_0 ?
- a. That f is equal to its value at x_0 everywhere nearby
 - b. That f is best approximated by its value at x_0 plus the gradient dotted with the displacement
 - c. That f grows like the square of the distance from x_0

Answers

1. a) Any movement splits into movements along each axis, so the directional derivative is the sum of each partial derivative times the

movement along that axis, the dot product $\langle \nabla f(x), v \rangle$.

2. b) The Jacobian stacks one gradient per output as rows, so it has $o = 3$ rows and $d = 5$ columns.

3. c) The derivative of a composition becomes a matrix product, ∂f evaluated at $g(x)$ times $\partial g(x)$, as the row $(6, -2)$ showed for $u_1 u_2$ after $(x_1 + x_2, x_1 - x_2)$.

4. b) The best linear approximation is $f(x_0) + \langle \partial f(x_0), x - x_0 \rangle$, which in the book's example gives -0.5050 against the true -0.5049 .

Worked examples

The book's linearization For $f(x) = x^2 - 1.5x$, approximate $f(0.51)$ with the tangent at 0.5 and compare with the true value.

$$f(0.5) = 0.25 - 0.75 = -0.5.$$

$$f'(x) = 2x - 1.5, \text{ so } f'(0.5) = -0.5.$$

$$\text{Approximation: } -0.5 + (-0.5) \times 0.01 = -0.5050.$$

$$\text{True value: } 0.51^2 - 1.5 \times 0.51 = 0.2601 - 0.765 = -0.5049.$$

Answer: -0.5050 approximated against -0.5049 true

A gradient and a directional derivative For $f(x) = x_1^2 + 3x_2$ at $(2, 1)$, find the gradient and the directional derivative along $v = (1, 1)$.

$$\text{Partial along } x_1: 2x_1 = 4.$$

$$\text{Partial along } x_2: 3.$$

$$\text{Gradient: } (4, 3).$$

$$\text{Along } v: 4 \times 1 + 3 \times 1 = 7; \text{ check: } f(2.01, 1.01) = 7.0701, \text{ a change of } 0.0701 \text{ for a step of } 0.01.$$

Answer: $\nabla f = (4, 3)$; directional derivative 7

Practice

1. For $f(x) = x_1x_2 + x_2^2$ at $(1, 2)$, find the gradient and the directional derivative along $v = (1, -1)$.

Hint: Differentiate with respect to x_1 holding x_2 fixed, then the other way round.

Answer: $\partial/\partial x_1 = x_2 = 2$ and $\partial/\partial x_2 = x_1 + 2x_2 = 5$, so $\nabla f = (2, 5)$; along $(1, -1)$: $2 - 5 = -3$.

2. Find the Jacobian of $f(x) = (3x_1, x_1x_2)$ at $(1, 4)$.

Hint: One row per output, one column per input.

Answer: Row 1: $(3, 0)$; row 2: $(x_2, x_1) = (4, 1)$. The Jacobian has rows $(3, 0)$ and $(4, 1)$, shape $(2, 2)$.

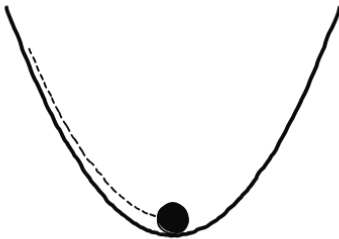
3. With $f(x) = x_1^2 + 3x_2$ at $x_0 = (2, 1)$, use the first-order approximation to estimate $f(2.1, 0.9)$, then compute the true value.

Hint: $f(x_0)$ plus the gradient $(4, 3)$ dotted with the displacement.

Answer: Displacement $(0.1, -0.1)$: $7 + 4 \times 0.1 + 3 \times (-0.1) = 7.1$. True: $2.1^2 + 3 \times 0.9 = 4.41 + 2.7 = 7.11$.

Hidden Sizes, and the First Step Downhill

Put a marble on a curved salad bowl and let go. It does not know where the bottom is, yet it gets there: at every instant it rolls the way the bowl tilts beneath it. Training a model is that marble, and this part sets up the rule it follows, after a short but important warning about the sizes of Jacobians.



TRAINING

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

$$\partial_{\mathbf{x}}[\mathbf{W}\mathbf{x}] = \mathbf{W}$$

$$\mathbf{W} = \begin{pmatrix} 1 & 2 & \boxed{0} \\ 0 & 1 & \boxed{4} \end{pmatrix}$$

**NUDGE $x_3 \rightarrow$
OUTPUTS MOVE 0 AND 4**

Marble in a Bowl; $\mathbf{W}$ as a Rate

The book calls the warning pedantic, but it matters later. Take $\mathbf{y} = \mathbf{W}\mathbf{x}$, a matrix times a vector, the core of every layer of a network. Seen

as a function of $\mathbf{x}$, its derivative is, as before, an o by d matrix, and it is simply $\partial_{\mathbf{x}}[\mathbf{W}\mathbf{x}] = \mathbf{W}$. In words, the rate at which the output changes with the input is the matrix itself, just as the derivative of $3x$ is 3 . For $\mathbf{W}$ with rows $(1, 2, 0)$ and $(0, 1, 4)$, nudging the third input by a small step moves the outputs by 0 and 4 times that step: the third column of $\mathbf{W}$.

Seen as a function of $\mathbf{W}$, though, the input is itself an o by d matrix, and the "Jacobian" then has shape (o, o, d) , one rate for every output and every entry of $\mathbf{W}$. We can always switch to an equivalent function that takes the flattened version of $\mathbf{W}$, $\text{vect}(\mathbf{W}) \sim (od)$, a single long vector, and then the Jacobian is an ordinary matrix of shape (o, od) . For our 2 by 3 example, the shape is 2 by 2 by 3 , which is 12 numbers, or 2 by 6 after flattening, the same 12 .

$$(o, o, d) = (2, 2, 3) \quad \text{or} \quad (o, od) = (2, 6)$$

$$y_i = \sum_j W_{ij} x_j$$

$$\partial y_i / \partial W_{ij} = x_j$$

1	2	3
---	---	---

0	0	0
---	---	---

OUTPUT 1

0	0	0
---	---	---

1	2	3
---	---	---

OUTPUT 2

$$1000 \times 1000 \times 1000 = 10^9$$

NEVER STORED: VECTOR-JACOBIAN PRODUCTS

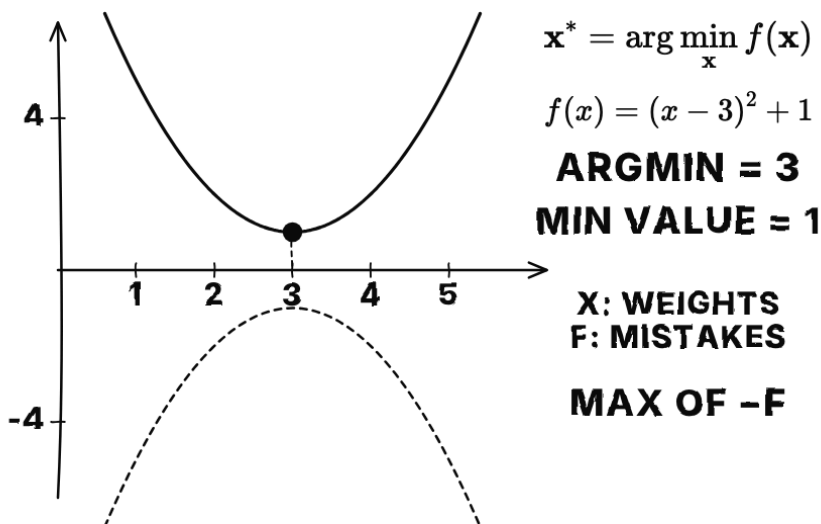
The Jacobian With Respect to W

The book's box works out what those numbers are. Write the product entry by entry: $y_i = \sum_j W_{ij} x_j$. Then immediately $\partial y_i / \partial W_{ij} = x_j$: nudging one weight in row i changes output i by that weight's input value, and leaves every other output untouched. With $x = (1, 2, 3)$, output 1 moves by 1, 2 or 3 per unit nudge of the weights in row 1, and does not move at all for the weights in row 2. So of our 12 numbers, 6 are zero and the other 6 are just x copied twice.

That is the point: storing the Jacobian explicitly would mean keeping a great deal of repeated values. For a single layer with 1,000 inputs and 1,000 outputs, W holds a million weights, and its Jacobian would hold 1,000 times 1,000 times 1,000, a billion numbers, mostly zeros and copies. As Chapter 6 will show, we never need to build it, because

in practice we only ever apply the Jacobian to another tensor, through so-called vector-Jacobian products, a view that can also be formalized by treating Jacobians as abstract linear maps. The book keeps vector notation because it is enough, but warns that in Chapter 6 matrix Jacobians will hide their true sizes inside the shapes of inputs and outputs, especially the batch sizes.

Now the reason gradients are worth all this: section 2.3, gradient descent. The task is to minimize a function $f(\mathbf{x})$ with $\mathbf{x} \sim (d)$: $\mathbf{x}^* = \operatorname{argmin} f(\mathbf{x})$. Like argmax , argmin means "the value of x that gives the lowest possible f ". For $f(x) = (x - 3)^2 + 1$, the argmin is 3, where f is 1; the minimum value is 1, but the argmin is the place, 3. The book assumes a single output, single-objective optimization, and no restrictions on where x may go.



Argmin Is a Place

In the rest of the book, $\mathbf{x}$ will hold the parameters of a model and f will measure how badly the model does on the data, the supervised-learning setup of the next chapter. For your phone's photo app, $\mathbf{x}$ is its millions of weights and f counts its mistakes on labelled photos. Minimizing loses nothing compared with maximizing, since minimizing f is maximizing minus f : flip a curve upside down across the horizontal axis and its valleys become peaks.

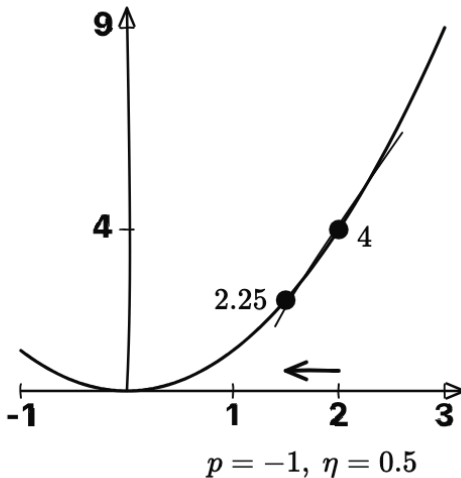
Rarely, the minimum can be written down as a formula, in closed form; the book will meet one such case with least squares in Section 4.1.2. Usually we must iterate. Start from a random guess $\mathbf{x}_0$ and at every iteration take a step made of a length η_t and a direction $\mathbf{p}_t$: $\mathbf{x}_t = \mathbf{x}_{t-1} + \eta_t \mathbf{p}_t$. In words, the new guess is the old guess plus a

displacement. η_t is called the step size or, in machine learning, the learning rate.

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \eta_t \mathbf{p}_t$$

η : learning rate

$\mathbf{p}$: direction



$$f(\mathbf{x}_t) \leq f(\mathbf{x}_{t-1})$$

$$D_{\mathbf{p}} f \leq 0 : -4$$

One Step Downhill

A direction is a descent direction if some step length along it makes f no larger: $f(\mathbf{x}_t) \leq f(\mathbf{x}_{t-1})$. If we can pick a descent direction at every iteration and choose step sizes carefully, the procedure reaches a minimum, in a sense the book makes precise shortly. For differentiable functions the directional derivative names every such direction exactly: $\mathbf{p}_t$ descent $\Rightarrow D_{\mathbf{p}_t} f(\mathbf{x}_{t-1}) \leq 0$, a direction in which f does not increase, at least to first order. Try $f(x) = x^2$ from 2: the derivative is 4, so the direction minus 1 has directional derivative minus 4, which is negative, and a step of 0.5 along it lands at 1.5, where f falls from 4 to 2.25.

Try it yourself

Draw the curve of x squared on paper and put a coin on it at x equal to 2. Move the coin half a unit to the left, the direction where the slope says the curve goes down, and read the height: 2.25 instead of 4. Move it to the right instead and the height grows to 6.25; the sign of the slope told you in advance which way was downhill.

Remember

For $y = Wx$, the derivative with respect to x is just W , an o by d matrix. With respect to W the Jacobian has shape (o, o, d) , or (o, od) after flattening, with $\partial y_i / \partial W_{i,j} = x_j$: mostly zeros and copies, so Chapter 6 avoids storing it. Optimization seeks $x^* = \operatorname{argmin} f(x)$; in the book x holds a model's parameters and f its error, and minimizing is the same as maximizing $-f$. Iterative methods update $x_t = x_{t-1} + \eta_t p_t$ with a learning rate η_t , and a descent direction is one with a non-positive directional derivative, as the step from 2 to 1.5 on x^2 showed.

Questions

1. For $y = Wx$ with W of shape (o, d) , why is the Jacobian with respect to W never stored in practice?
 - a. Because it is always the identity matrix

- b. Because it has shape (o, o, d), mostly zeros and copies of x, and we only need to apply it to another tensor
 - c. Because it cannot be computed for matrix inputs
- 2. What is the difference between the minimum of $f(x) = (x - 3)^2 + 1$ and its argmin?
 - a. The minimum is 1, the lowest value; the argmin is 3, the place where it is reached
 - b. The minimum is 3 and the argmin is 1
 - c. They are the same number
- 3. Why does the book lose nothing by always talking about minimizing?
 - a. Because no useful function has a maximum
 - b. Because gradient descent cannot find maxima
 - c. Because maximizing f is the same as minimizing $-f$
- 4. When is a direction p a descent direction for a differentiable f at the current guess?
 - a. When it points away from the starting guess x_0
 - b. When the directional derivative of f along p is at most zero
 - c. When its length equals the learning rate

Answers

1. **b)** Since $\partial y_i / \partial W_{ij} = x_j$, the Jacobian repeats x and is otherwise zero; for a 1,000 by 1,000 layer it would hold a billion numbers, and Chapter 6 avoids it with vector-Jacobian products.
2. **a)** argmin names the input that gives the lowest value of f ; here f is smallest, equal to 1, at $x = 3$.

3. c) Flipping a function across the horizontal axis turns its peaks into valleys, so any maximization can be written as a minimization.

4. b) A descent direction is one along which f does not increase; for differentiable functions this means $D_p f \leq 0$, as the direction -1 for x^2 at 2 , with directional derivative -4 .

Worked examples

The Jacobian of Wx with respect to W W has shape $(2, 3)$ and $x = (1, 2, 3)$. Using $\partial y_i / \partial W_{ij} = x_j$, give the shape of the Jacobian of $y = Wx$ with respect to W and say how many of its entries are zero.

$o = 2$ and $d = 3$, so the Jacobian has shape $(o, o, d) = (2, 2, 3)$, 12 entries.

Output y_1 depends only on row 1 of W : its rates are $x_1, x_2, x_3 = 1, 2, 3$, and 0 for the three weights of row 2.

Likewise y_2 has rates 1, 2, 3 for row 2 and 0 for row 1.

So 6 entries are zero, and the other 6 are x copied twice.

Answer: Shape $(2, 2, 3)$, or $(2, 6)$ after flattening; 6 of the 12 entries are zero.

Practice

1. For W with rows $(1, 2, 0)$ and $(0, 1, 4)$, what is the derivative of Wx with respect to x , and how do the outputs change if only x_2 is nudged by 0.1?

Hint: The derivative with respect to x is W itself; look at its second column.

Answer: It is W . Nudging x_2 by 0.1 moves the outputs by $2 \times 0.1 = 0.2$ and $1 \times 0.1 = 0.1$, the second column of W times the step.

2. $f(x) = x^2$ and the current guess is 3. Take the direction $p = -1$ with step size $\eta = 0.5$. Is it a descent direction, and what are the new guess

and the new value of f ?

Hint: Check the sign of $f'(3) \times p$, then apply $x_t = x_{t-1} + \eta p$.

Answer: $f'(3) = 6$, so the directional derivative is $6 \times (-1) = -6 \leq 0$: a descent direction. New guess $3 + 0.5 \times (-1) = 2.5$, and f falls from 9 to 6.25.

3. A layer has 500 inputs and 200 outputs. How many numbers would its Jacobian with respect to W hold if stored explicitly?

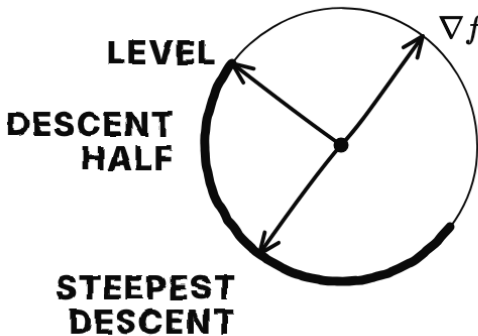
Hint: The shape is (o, o, d) .

Answer: $200 \times 200 \times 500 = 20,000,000$ numbers, although W itself has only 100,000 weights.

Steepest Descent, and Where It Ends

Go back to the hillside and face uphill, where the slope is steepest. Now turn round: you face straight down the steepest way. Turn a quarter instead and you walk level, neither up nor down; anything between the quarter-turns on the downhill side takes you lower. This simple compass, built from the gradient and the cosine, is the whole of gradient descent.

$$D_{\mathbf{p}}f = \|\nabla f\| \|\mathbf{p}\| \cos \alpha$$



$$\nabla f(3, 4) = (6, 8), \quad \|\nabla f\| = 10$$

$$(-0.6, -0.8) : -10$$

$$(-0.8, 0.6) : 0$$

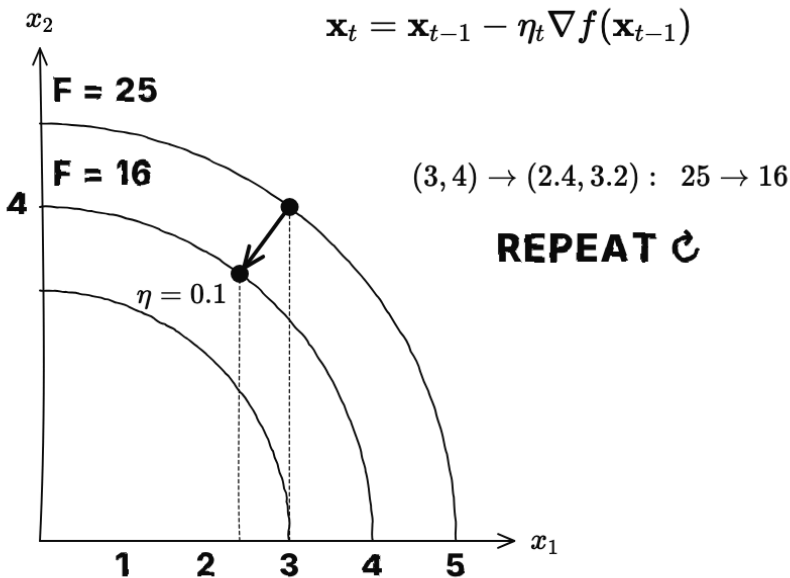
A Compass Made of the Gradient

Combine the directional derivative with the cosine form of the dot product: $D_{\mathbf{p}_t} f(\mathbf{x}_{t-1}) = \langle \nabla f(\mathbf{x}_{t-1}), \mathbf{p}_t \rangle = \|\nabla f(\mathbf{x}_{t-1})\| \|\mathbf{p}_t\| \cos(\alpha)$, where α is

the angle between the step direction and the gradient. The first length does not depend on the direction we choose, and since the direction only says which way to go, we can fix its length to one. So everything depends on the cosine.

The cosine is negative for angles between $\pi/2$ and $3\pi/2$, a quarter-turn and three quarter-turns, so every direction in that range is a descent direction. Among them the one at angle π , pointing exactly against the gradient, $\mathbf{p}_t = -\nabla f(\mathbf{x}_{t-1})$, gives the most negative directional derivative, and it is called the steepest descent direction. Take $f(\mathbf{x}) = x_1^2 + x_2^2$ at $(3, 4)$: the gradient is $(6, 8)$, of length 10. The unit direction against it, $(-0.6, -0.8)$, gives a directional derivative of minus 10, the steepest possible; the sideways direction $(-0.8, 0.6)$ gives minus 4.8 plus 4.8, which is zero: walking level.

Putting this into the iterative step gives an algorithm that can minimize any differentiable function, steepest gradient descent. Given a starting point $\mathbf{x}_0$ and a sequence of step sizes η_t , repeat $\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla f(\mathbf{x}_{t-1})$. In words: from where you are, step a little against the gradient. With our function, a step size of 0.1 from $(3, 4)$ moves to 3 minus 0.6 and 4 minus 0.8, that is $(2.4, 3.2)$, and f drops from 25 to 5.76 plus 10.24, which is 16.



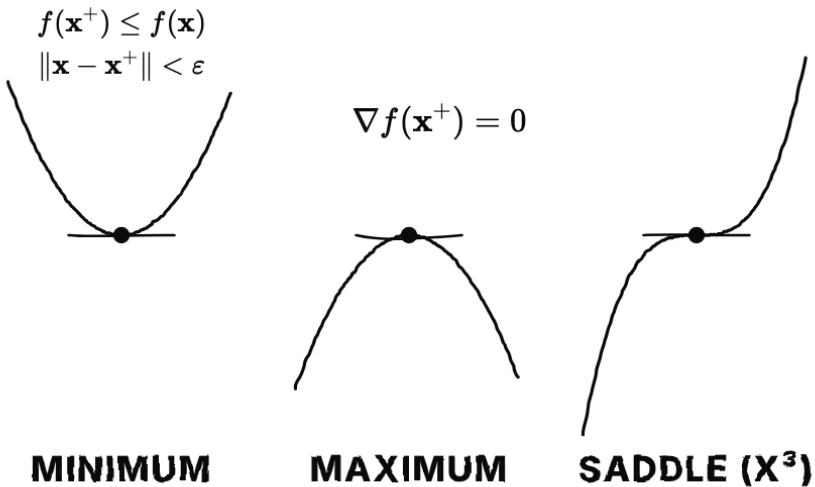
One Step of Gradient Descent

The book sets aside how to choose the step size and simply assumes it is small enough for each step to reduce f . It also notes that gradient descent is only as fast as our way of computing the gradient, which is why Chapter 6 introduces a general algorithm for it. When your photo app was trained, this was the loop, repeated an enormous number of times: compute the gradient of the error with respect to millions of weights, step a little against it.

Where does gradient descent end up? First we must say what a minimum is. A local minimum is a point $\mathbf{x}^+$ such that, for some small radius $\varepsilon > 0$, $f(\mathbf{x}^+) \leq f(\mathbf{x})$ for all $\mathbf{x}$ with $\|\mathbf{x} - \mathbf{x}^+\| < \varepsilon$. In words: within a small enough ball around it, nothing is lower. Intuitively, at such a

point the tangent is flat and the gradient around it points uphill, away from it.

GD REACHES SOME STATIONARY POINT

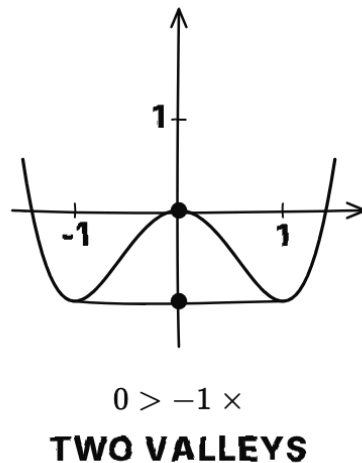
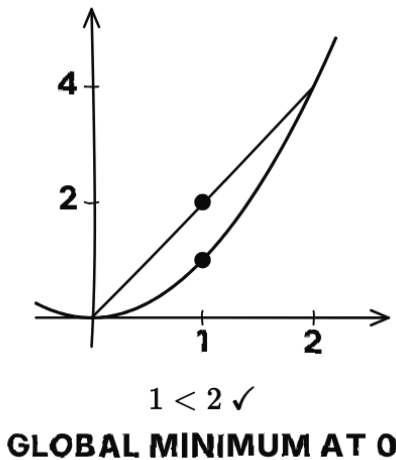


Stationary Points

The flat-tangent idea gets its own name: a stationary point is one where $\nabla f(\mathbf{x}^+) = 0$. But stationary points are not only minima: they can be maxima, which are minima of minus f , or saddle points, where the curvature of the function changes. The book's figure shows a saddle; in one dimension, x cubed at zero is the classic case: its slope, $3x$ squared, is zero there, yet the curve keeps rising on both sides. Without assumptions on f , gradient descent can only be proved to reach some stationary point, which one depending on where it starts.

Can we do better? Picture a parabola. It has no saddle points and a single minimum, and that minimum is special: the lowest value over the whole domain. A global minimum is a point $\mathbf{x}^*$ with $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for any $\mathbf{x}$. On a parabola gradient descent reaches it from any starting point, because every gradient points towards it.

$$f(\alpha \mathbf{x}_1 + (1 - \alpha) \mathbf{x}_2) \leq \alpha f(\mathbf{x}_1) + (1 - \alpha) f(\mathbf{x}_2)$$



Below Every Chord

Convexity generalizes the parabola. A function is convex if for any two points $\mathbf{x}_1$ and $\mathbf{x}_2$ and any $\alpha \in [0, 1]$, $f(\alpha \mathbf{x}_1 + (1 - \alpha) \mathbf{x}_2) \leq \alpha f(\mathbf{x}_1) + (1 - \alpha) f(\mathbf{x}_2)$. In words: on the stretch between two points, the function never rises above the straight chord joining their values. For x squared with points 0 and 2 and alpha equal to one half: the midpoint is 1, where f is 1, while the chord's midpoint is the average of 0 and 4, which is 2; 1 is below 2, as convexity demands. For $x^4 - 2x^2$, with

points minus 1 and 1, the midpoint 0 gives 0, but the chord's average of minus 1 and minus 1 is minus 1: the curve pokes above the chord, so it is not convex, and indeed it has two separate valleys.

Try it yourself

Draw a bowl-shaped curve, x squared, and mark two points on it, at 0 and at 2. Join their heights with a straight ruler line: every point of the curve between them lies under the ruler. Now draw a W-shaped curve with two dips and put the ruler across both dips: the bump in the middle sticks up above it, which is exactly what convexity forbids.

Remember

The directional derivative equals the gradient's length times the step's length times $\cos \alpha$, so every direction more than a quarter-turn from the gradient goes downhill, and the steepest is minus the gradient. Gradient descent repeats $\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla f(\mathbf{x}_{t-1})$: for $x_1^2 + x_2^2$ from (3, 4) with step 0.1 it reaches (2.4, 3.2), and f falls from 25 to 16. A local minimum is lowest in a small ball; a stationary point has zero gradient and may be a minimum, a maximum or a saddle, and in general gradient descent only reaches some stationary point. A global minimum is lowest everywhere; convex functions, which lie below every chord, generalize the parabola, on which gradient descent reaches the global minimum from anywhere.

Questions

1. Why is minus the gradient called the steepest descent direction?
 - a. Because it is the only direction in which f decreases
 - b. Because at an angle of π to the gradient the cosine is -1 , giving the most negative directional derivative
 - c. Because it always leads straight to the global minimum
2. Gradient descent stops at a point where the gradient is zero. What can you say about that point in general?
 - a. It is a stationary point, which could be a minimum, a maximum or a saddle point
 - b. It is always the global minimum
 - c. It is always a local minimum
3. What does convexity guarantee that makes gradient descent behave like on a parabola?
 - a. That the step size can be chosen freely
 - b. That the gradient is never zero
 - c. That the function never rises above the chord between two points, ruling out separate valleys and saddles
4. A direction is exactly at right angles to the gradient. What does a small step along it do to f , to first order?
 - a. It decreases f as fast as possible
 - b. It leaves f unchanged, since the cosine of a right angle is zero
 - c. It increases f as fast as possible

Answers

1. **b)** With unit-length directions, the directional derivative is the gradient's length times $\cos \alpha$; every angle between $\pi/2$ and $3\pi/2$ goes downhill, but $\cos \alpha$ is smallest, -1 , at $\alpha = \pi$.
2. **a)** A zero gradient defines a stationary point, and without assumptions on f , gradient descent can only be proved to reach some stationary point, depending on where it started.
3. **c)** A convex function lies below every chord, generalizing the parabola, whose single minimum is global and towards which all gradients point.
4. **b)** The directional derivative is proportional to $\cos \alpha$, which is 0 at a quarter-turn, as $(-0.8, 0.6)$ against the gradient $(6, 8)$ gave $-4.8 + 4.8 = 0$.

Worked examples

One step of gradient descent Minimize $f(x) = x_1^2 + x_2^2$ starting from $(3, 4)$ with step size 0.1. Find the gradient, the new point and the new value of f .

The gradient is $(2x_1, 2x_2) = (6, 8)$ at $(3, 4)$.

Update: $(3, 4) - 0.1 \times (6, 8) = (3 - 0.6, 4 - 0.8) = (2.4, 3.2)$.

Old value: $9 + 16 = 25$.

New value: $2.4^2 + 3.2^2 = 5.76 + 10.24 = 16$.

Answer: New point $(2.4, 3.2)$; f falls from 25 to 16

Testing convexity with one chord Check the convexity inequality for $f(x) = x^4 - 2x^2$ with $x_1 = -1$, $x_2 = 1$ and $\alpha = 0.5$.

The mixed point is $0.5 \times (-1) + 0.5 \times 1 = 0$, and $f(0) = 0$.

$f(-1) = 1 - 2 = -1$ and $f(1) = -1$, so the chord value is $0.5 \times (-1) + 0.5 \times (-1) = -1$.

Convexity would need $0 \leq -1$, which is false.

Answer: The inequality fails, so $x^4 - 2x^2$ is not convex; it has two separate valleys.

Practice

1. For $f(x) = x^2$ starting at 5 with step size 0.25, compute two steps of gradient descent.

Hint: The gradient is $2x$; update $x \leftarrow x - 0.25 \times 2x$.

Answer: Step 1: $5 - 0.25 \times 10 = 2.5$. Step 2: $2.5 - 0.25 \times 5 = 1.25$. Each step halves x , heading to the minimum at 0.

2. Is $x = 0$ a stationary point of $f(x) = x^3$? Is it a minimum?

Hint: Compute the derivative at 0, then look at f on either side.

Answer: $f'(x) = 3x^2$, which is 0 at 0, so it is stationary. It is not a minimum: f is negative just to the left and positive to the right, so it is a point where the curve flattens and keeps rising, like a saddle in one dimension.

3. Check the convexity inequality for $f(x) = x^2$ with $x_1 = 1$, $x_2 = 3$ and $\alpha = 0.25$.

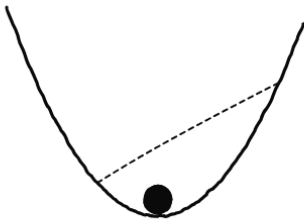
Hint: Compare f at $0.25 \times 1 + 0.75 \times 3$ with $0.25 f(1) + 0.75 f(3)$.

Answer: The point is $0.25 + 2.25 = 2.5$, and $f(2.5) = 6.25$. The chord gives $0.25 \times 1 + 0.75 \times 9 = 7$. Since $6.25 \leq 7$, the inequality holds, as convexity requires.

Bowls, Bumpy Landscapes and a Push of Momentum

Roll a ball into a smooth salad bowl and it always ends at the bottom, wherever you drop it. Roll it across an egg carton and it settles in whichever cup happens to be nearest. And give the ball a little weight, so that it keeps some of its speed, and it rolls more smoothly and faster to the bottom. Those three pictures are this part: what convexity guarantees, what we lose without it, and how momentum speeds things up.

CONVEX



NON-CONVEX



ANY START → GLOBAL MIN

STRICT → UNIQUE

ONLY A STATIONARY POINT

Bowl or Egg Carton

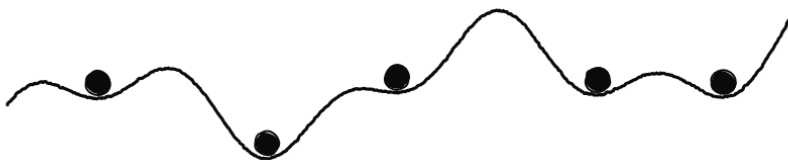
The convexity inequality compares two things. On the left is the value of f anywhere on the stretch from x_1 to x_2 ; on the right is the corresponding point on the straight line joining $f(x_1)$ and $f(x_2)$. If the function always lies below every such line, it is convex; an upward-opening parabola is the standard example, the salad bowl.

Convexity tells us how easy optimization is. For a general non-convex function, gradient descent reaches a stationary point and nothing more can be said without looking at higher derivatives, the derivatives of the derivatives. For a convex function, gradient descent reaches a global minimum whatever the starting point. And if the inequality holds strictly, with "less than" instead of "less than or equal", which is called strict convexity, that global minimizer is also unique.

The difference is severe. To be sure of the global minimum of a non-convex problem with gradient descent, you would have to run it from every possible starting point, infinitely many times, which makes the task NP-hard, among the hardest kinds of problems to solve. That is the egg carton: each start finds a cup, but nothing tells you which cup is the lowest.



GLOBAL MIN: NP-HARD



**NETWORKS:
NON-CONVEX**

SVM: CONVEX

**✓ GOOD POINTS
IN PRACTICE** **CHAT · PHOTOS**

Which Cup Is Lowest?

This matters historically. As Chapter 5 will show, every non-trivial neural network is non-convex, so its optimization can have many stationary points. Other learning methods, such as support vector machines, keep non-linear behaviour while still having a convex optimization problem, which was long seen as a strong advantage. Yet, interestingly, complex differentiable models work well anyway: started from a reasonable initialization, their optimization reaches points that perform well in practice. The chat assistant and the photo app on your phone were trained exactly this way, on egg cartons with billions of dimensions, and they work.

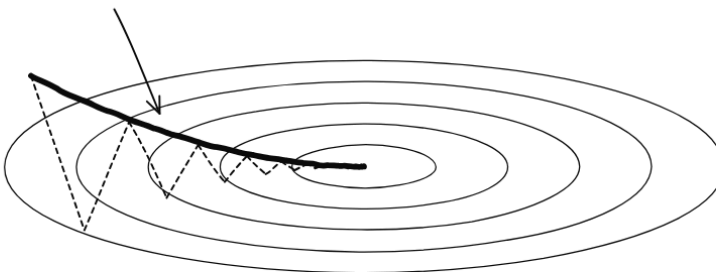
Now acceleration. The negative gradient is the steepest way down only in an infinitely small neighbourhood of the current point, and, as

Chapter 5 will show with stochastic optimization, these directions can be very noisy for large models. Many techniques choose better descent directions; for computational reasons the book prefers those that need no higher derivatives, such as the Hessian, the matrix of second derivatives, and no extra calls to the function.

$$\mathbf{g}_t = -\eta_t \nabla f(\mathbf{x}_{t-1}) + \lambda \mathbf{g}_{t-1}, \quad \mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{g}_t$$

↑ PART OF THE LAST STEP

MOMENTUM



PLAIN GD

Zigzag Versus Momentum

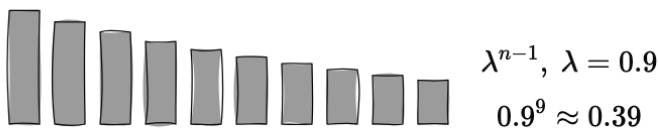
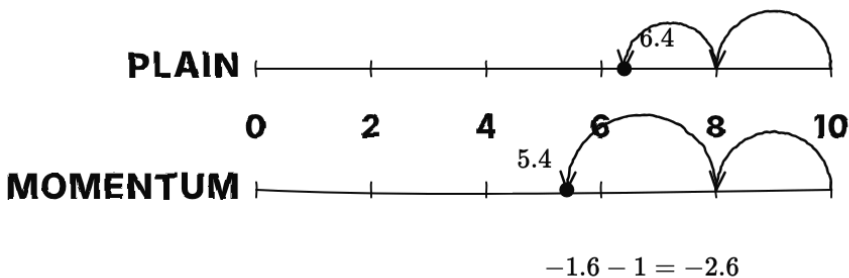
The one it describes is momentum. Picture gradient descent as a ball rolling downhill: its path is jerky, because each gradient may point somewhere quite different; in fact, with a perfectly chosen step on a convex function, two successive gradients are at right angles.

Momentum smooths this by keeping part of the previous step: $\mathbf{g}_t = -\eta_t \nabla f(\mathbf{x}_{t-1}) + \lambda \mathbf{g}_{t-1}$, then $\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{g}_t$, starting from $\mathbf{g}_0 = \mathbf{0}$. In words:

the new step is the usual steepest-descent step plus a fraction λ of the previous step.

Let us run it. Take $f(x) = x^2$, start at 10, with a step size of 0.1 and $\lambda = 0.5$. First step: the gradient is 20, so the step is minus 2 and x becomes 8, the same as plain gradient descent. Second step: the gradient is 16, the plain part is minus 1.6, and momentum adds half of the previous minus 2, giving minus 2.6, so x becomes 5.4. Plain gradient descent would have reached only 6.4: the remembered push made the ball roll further.

$$\mathbf{g}_t = -\eta_t \nabla f(\mathbf{x}_{t-1}) - \lambda \eta_t \nabla f(\mathbf{x}_{t-2}) + \lambda^2 \mathbf{g}_{t-2}$$



Momentum Rolls Further

Unrolling two steps shows how the memory fades: $\mathbf{g}_t = -\eta_t \nabla f(\mathbf{x}_{t-1}) - \lambda \eta_t \nabla f(\mathbf{x}_{t-2}) + \lambda^2 \mathbf{g}_{t-2}$. In general, the gradient from n steps back is

damped by a factor λ^{n-1} . With λ equal to 0.9, a gradient from ten steps back still counts with 0.9 to the ninth, about 0.39: momentum has a long memory. The book's figure compares plain and momentum gradient descent on $x \sin(2x)$, starting just past 1 with λ equal to 0.3, and momentum shows the smoother path.

Another common idea is to adapt the step size for each parameter separately, according to how large its gradients have been. The widely used optimizer called Adam combines several of these ideas. One of its advantages is that it is fairly robust to the choice of its hyperparameters, the settings chosen by the user rather than learned by gradient descent, such as the step size and λ ; the frameworks' default values are a good start in most cases. Designing a new optimizer that can unseat Adam, or its variants such as AdamW, as the default in deep learning is still an open research problem, including recent work that designs optimizers for neural networks from first principles.

Try it yourself

On a calculator, run plain gradient descent on x squared from 10 with step 0.1: each step multiplies x by 0.8, giving 8, 6.4, 5.12. Now run momentum with λ equal to 0.5: 8, then 5.4, then compute the third step yourself, the gradient 10.8 gives minus 1.08, plus half of minus 2.6, total minus 2.38, so x is 3.02. Compare 3.02 with plain descent's 5.12 after the same three steps.

Remember

On a convex function gradient descent reaches a global minimum from any start, and with strict convexity that minimizer is unique; on a non-convex one it only reaches a stationary point, and finding the global minimum becomes NP-hard. Neural networks are non-convex, unlike support vector machines, yet from a reasonable start they reach points that work well in practice. Momentum adds a fraction λ of the previous step, $g_t = -\eta_t \nabla f(x_{t-1}) + \lambda g_{t-1}$, so old gradients fade by λ^{n-1} ; on x^2 from 10 it reached 5.4 instead of 6.4 in two steps. Adam combines momentum with per-parameter step sizes, is robust to its hyperparameters, and remains the default that new optimizers try to beat.

Questions

1. What does gradient descent guarantee on a convex function that it cannot guarantee on a non-convex one?
 - a. Reaching a global minimum from any starting point
 - b. Taking fewer steps than momentum
 - c. Never meeting a point with zero gradient
2. Why is finding the global minimum of a non-convex problem with gradient descent called NP-hard?
 - a. Because each gradient step is too expensive

- b. Because one would have to run it from every possible starting point to be sure
 - c. Because non-convex functions have no minimum
- 3. In momentum, $g_t = -\eta_t \nabla f(x_{t-1}) + \lambda g_{t-1}$. What does λ control?
 - a. How large the current gradient step is
 - b. How many gradients are computed per step
 - c. How much of the previous step is kept, so older gradients fade by powers of λ
- 4. Why is Adam often a convenient choice?
 - a. Because it makes neural networks convex
 - b. Because it is fairly robust to its hyperparameters, and the defaults work in most cases
 - c. Because it needs the Hessian, which makes it exact

Answers

1. **a)** On a convex function gradient descent reaches a global minimum irrespective of initialization; on a non-convex one it only reaches some stationary point.
2. **b)** Each run finds some stationary point, like a ball settling in one cup of an egg carton, and only trying every start would guarantee the lowest one.
3. **c)** λ is the fraction of the previous step carried over; unrolling shows the gradient from n steps back weighted by λ^{n-1} .
4. **b)** Adam combines momentum with per-parameter step sizes and is found relatively robust to the choice of its hyperparameters, so framework defaults are a good start.

Worked examples

Two steps with momentum Minimize $f(x) = x^2$ from $x = 10$ with step size 0.1 and $\lambda = 0.5$, starting with $g_0 = 0$. Compare with plain gradient descent after two steps.

Step 1: gradient $2 \times 10 = 20$, so $g_1 = -0.1 \times 20 + 0.5 \times 0 = -2$ and $x = 8$.

Step 2: gradient 16 , so $g_2 = -1.6 + 0.5 \times (-2) = -2.6$ and $x = 8 - 2.6 = 5.4$.

Plain descent: $10 \rightarrow 8 \rightarrow 8 - 1.6 = 6.4$.

Momentum is closer to the minimum at 0 after the same two steps.

Answer: Momentum reaches 5.4 , plain descent 6.4

Practice

1. Continue the momentum example for a third step from $x = 5.4$ with $g_2 = -2.6$.

Hint: The gradient is 2×5.4 ; add half of g_2 .

Answer: Gradient 10.8 , plain part -1.08 , plus $0.5 \times (-2.6) = -1.3$, so $g_3 = -2.38$ and $x = 5.4 - 2.38 = 3.02$, against 5.12 for plain descent.

2. With $\lambda = 0.9$, by what factor is the gradient from 4 steps back weighted in the current momentum step?

Hint: The gradient from n steps back is damped by λ^{n-1} .

Answer: $0.9^3 = 0.729$.

3. Explain why neural networks are trained successfully even though their optimization is non-convex.

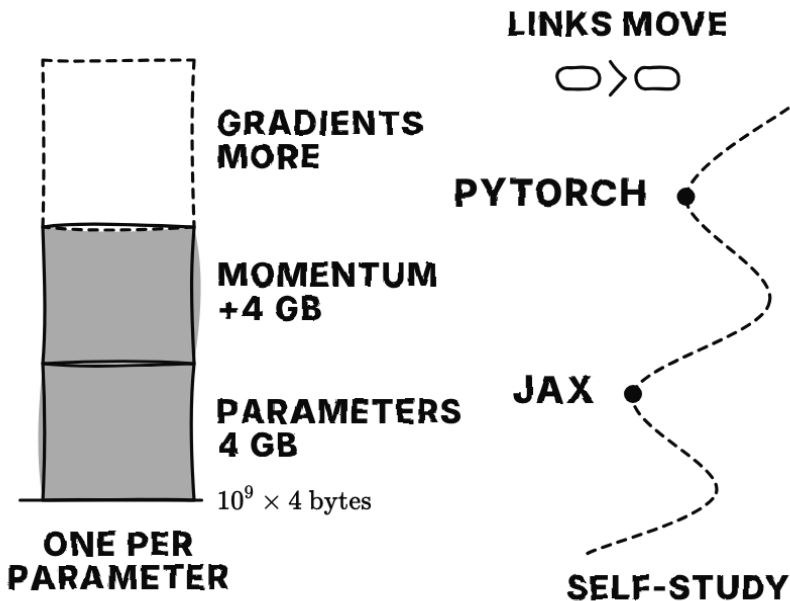
Hint: Think of what the book says about starting points and empirical performance.

Answer: Non-convexity means many stationary points and no guarantee of the global minimum. But, as the book observes, when started from a reasonable initialization, optimization reaches points with good

performance in practice, so the global guarantee is not needed for models to work well.

The Price of Memory, and Where to Practise

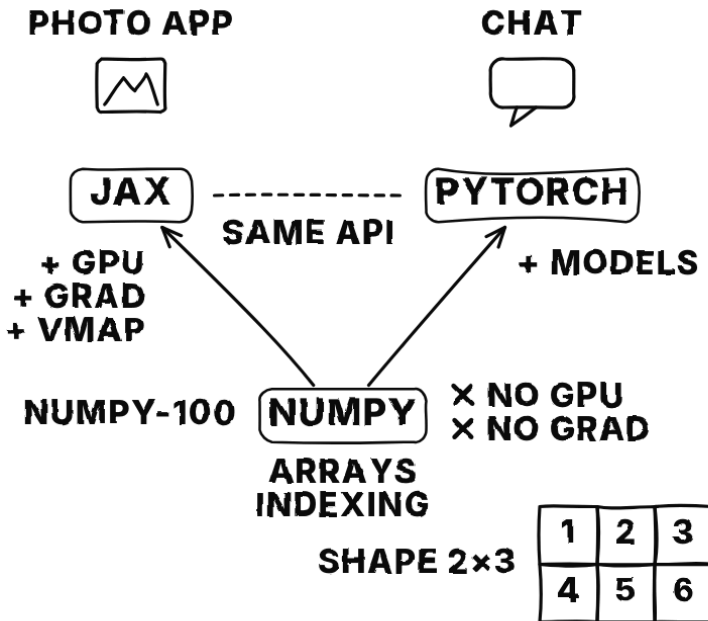
A ball with momentum remembers its last push, and remembering costs space. Momentum keeps the previous step in memory, one number for every parameter, which doubles the space the optimizer itself needs. Put numbers on it: a model with a billion parameters, each stored in 4 bytes, takes 4 gigabytes; momentum's memory of the last step adds another 4. The book adds that, in most cases, the memory needed to compute the gradient matters even more, a point it returns to in Section 6.3.



Momentum Costs Memory

Then the chapter turns from theory to practice. The book has no classical end-of-chapter exercises, since many textbooks already offer those. Instead it proposes a self-learning path through two frameworks, JAX and PyTorch, that grows as the book goes on, with solutions to be published on the book's website. It warns that these sections are full of links to online material, which may have moved or expired by the time you look.

The starting point for anyone designing differentiable models, the book says, is a careful study of NumPy. NumPy provides a general set of functions for handling multidimensional arrays, what the book calls tensors, together with functions to index and transform their contents. You should be comfortable handling arrays in NumPy, above all indexing them, the slices and colons we met earlier. The library's quick-start guide is the place to begin, and a collection of a hundred NumPy exercises, the `rougier/numpy-100` repository, offers a slow-paced way to test yourself.



NumPy, Then JAX and PyTorch

For all its influence, NumPy has limits. Without extra libraries it has little support for parallel hardware such as GPUs, and it has no automatic differentiation, the machinery of Chapter 6. JAX copies NumPy's interface and adds broader hardware support, automatic computation of gradients, and further transformations such as the vectorized map, `jax.vmap`, which applies a function across a whole batch at once. PyTorch also has a NumPy-like core, with small changes in naming and behaviour, and adds high-level tools for building differentiable models.

The book suggests skimming the documentation of `jax.numpy.array` and `torch.tensor` to see how much both share with NumPy. High-level modules such as `torch.nn` can be ignored for now; the book explains

how these frameworks are designed in Chapter 6, once their gradient machinery has been introduced. The models behind your photo app and your chat assistant were almost certainly written in one of these two frameworks, so this path leads straight to the tools of real systems.

Try it yourself

If you have Python at hand, create a small array with NumPy holding the numbers 1 to 6 in 2 rows of 3, and ask for its shape: it should answer 2 by 3. Do the same with `jax.numpy` or `torch`, and notice that the commands are almost identical. That near-identity is what the book means when it says these frameworks replicate NumPy's interface.

Remember

Accelerated optimizers cost memory: momentum stores the previous step, doubling the optimizer's space, 4 more gigabytes for a billion parameters, though computing gradients usually needs even more. Instead of classical exercises, the book offers a self-learning path through JAX and PyTorch, with solutions online. Start with NumPy, the standard library for arrays, and practise indexing, for instance with the `numpy-100` exercises. NumPy lacks easy GPU support and automatic differentiation; JAX copies its interface and adds both, plus transformations like `vmap`, while PyTorch offers a NumPy-like core with tools for building models.

Questions

1. Why does momentum increase the memory an optimizer needs?
 - a. Because it stores the previous step, one number per parameter, doubling the optimizer's space
 - b. Because it computes second derivatives
 - c. Because it keeps a copy of the whole dataset
2. What does JAX add to NumPy's interface?
 - a. A different way of indexing arrays
 - b. Support for strings and integers in arrays
 - c. Broader hardware support, automatic gradients and transformations such as vmap
3. Why does the book start its practical path with NumPy rather than with a full framework?
 - a. Because NumPy is faster than JAX and PyTorch on GPUs
 - b. Because handling and indexing arrays in NumPy is the foundation that JAX and PyTorch copy
 - c. Because NumPy computes gradients automatically

Answers

1. a) Momentum has to remember the previous gradient step for every parameter, doubling the space of the optimization algorithm, although computing the gradient usually needs even more memory.

2. c) NumPy lacks easy GPU support and automatic differentiation; JAX replicates its interface while adding both, plus transformations like the vectorized map.

3. b) NumPy is the standard for manipulating multidimensional arrays, and both JAX and PyTorch build on a NumPy-like interface, so comfort with NumPy carries over.

Worked examples

Memory cost of momentum A model has 1 billion parameters, each stored in 4 bytes. How much memory do the parameters take, and how much does momentum add?

Parameters: $1,000,000,000 \times 4 \text{ bytes} = 4,000,000,000 \text{ bytes}$, about 4 gigabytes.

Momentum stores the previous step, one number per parameter, also 4 bytes each.

So it adds another 4 gigabytes, doubling the optimizer's space.

Answer: 4 GB for the parameters, plus 4 GB for momentum

Practice

1. A model has 7 billion parameters stored in 2 bytes each. How much memory does momentum's stored step take?

Hint: One number per parameter, same size as the parameters.

Answer: $7,000,000,000 \times 2 \text{ bytes} = 14,000,000,000 \text{ bytes}$, about 14 gigabytes.

2. In your own words, explain why the book suggests ignoring modules like torch.nn for now.

Hint: Think of what the book has not yet introduced.

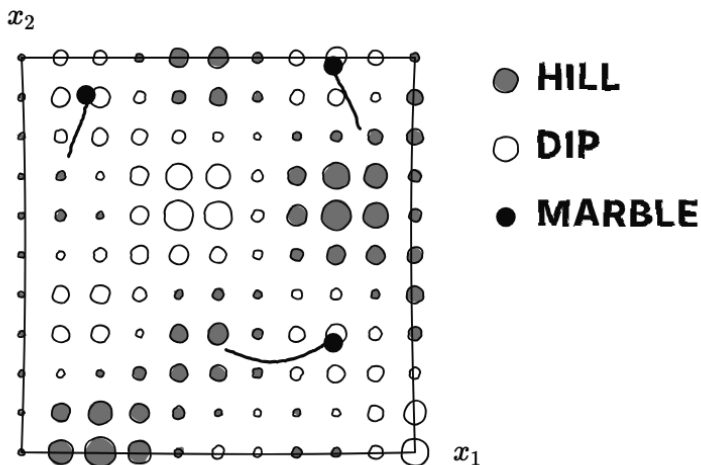
Answer: They are high-level tools for building models, and the book explains how frameworks are designed only in Chapter 6, after introducing how gradients are computed; for now the aim is to see how much `jax.numpy.array` and `torch.tensor` share with NumPy.

A Practice Landscape, and What a Dataset Is

Imagine a sheet of egg carton laid flat on a table, ten units by ten, with bumps and dips scattered over it. Drop a few marbles at different places and watch where each one ends up. That is the exercise the book sets before it moves on, and right after it comes the question every trained model depends on: where do the examples come from, and can we trust them?

$$f(\mathbf{x}) = \sin x_1 \cos x_2 + \sin 0.5x_1 \cos 0.5x_2$$

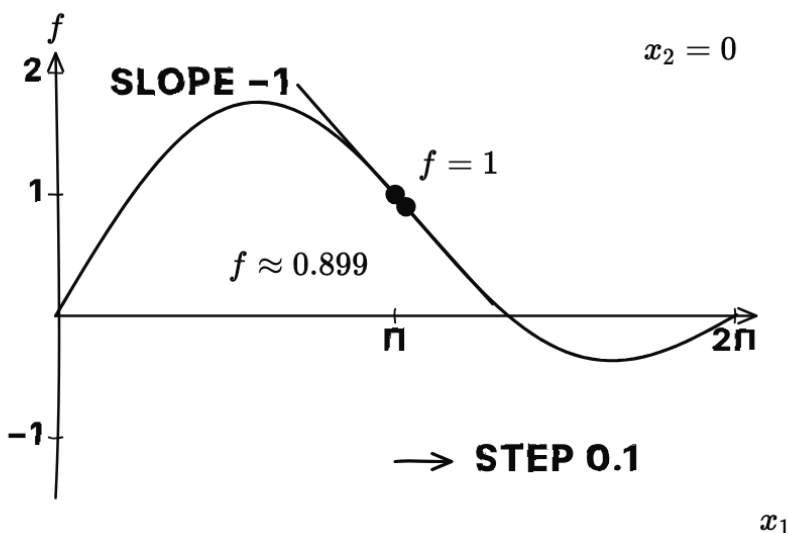
FAST WAVE SLOW WAVE



A Landscape of Bumps and Dips

The book asks the reader to repeat one exercise three times, in NumPy, in JAX and in PyTorch, each taking only minutes once the syntax is familiar. The landscape is a function of two inputs, $\mathbf{x} \sim (2)$, over the square from 0 to 10 in both directions: $f(\mathbf{x}) = \sin(x_1)\cos(x_2) + \sin(0.5x_1)\cos(0.5x_2)$. In words, two waves multiplied together, plus a slower copy of the same pattern, which makes a terrain with several peaks and valleys. The book adds an honest footnote: this function was generated by asking ChatGPT for one with several minima and maxima, and nothing else in the book was produced by a language model.

The tasks come in four steps. First, implement the function in vectorized form: given a matrix $\mathbf{X} \sim (n, 2)$ of n input points, return a vector of n values, $[f(\mathbf{X})]_i = f(\mathbf{X}_i)$, all at once rather than in a loop. Second, write a function for the gradient by hand, since automatic differentiation has not been introduced yet. Third, write a basic gradient descent and draw the paths from several starting points, the marbles on the carton; in JAX or PyTorch this is a good place to try `vmap`. Fourth, add momentum and plot the length of the gradient, which should shrink towards zero as the method approaches a stationary point.



One Step Down the Slope

Let us do the gradient by hand at one point, to see the exercise is within reach. The partial derivative along x_1 is $\cos(x_1)\cos(x_2) + 0.5\cos(0.5x_1)\cos(0.5x_2)$, the chain rule giving the factor one half for the slow wave. At the point $(\pi, 0)$, f is 0 plus sine of $\pi/2$, which is 1; the first partial is minus 1 plus one half times 0, which is minus 1, and the second partial is 0. One gradient-descent step of 0.1 therefore moves x_1 up to about 3.24, and f drops to about 0.899, a fall of roughly 0.1, just as the slope of minus 1 predicted.

Now Chapter 3, which formalizes supervised learning: datasets, losses, empirical risk minimization, the basic assumptions, and finally a probabilistic formulation built on maximum likelihood. The book calls this short chapter the backbone of everything that follows.

SUPERVISED

x	y	
	DOG	} N PAIRS
	CAT	
	DOG	
	CAT	

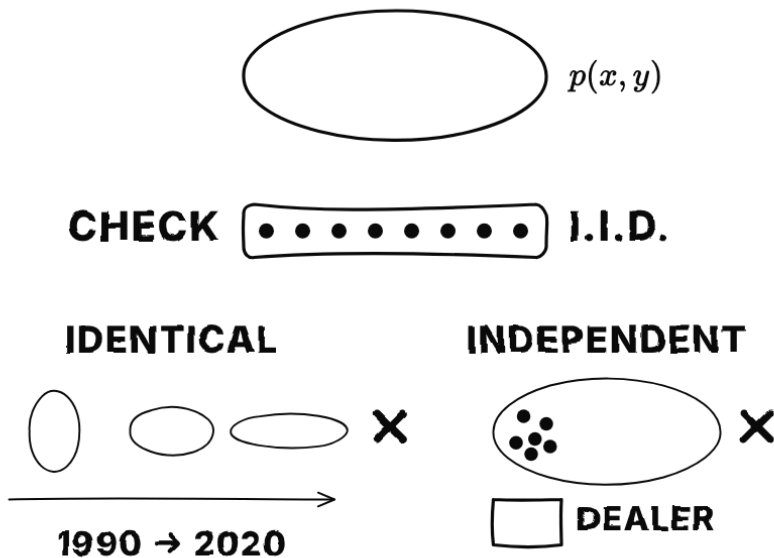
$n = 10,000$

A Dataset Is a List of Pairs

The starting situation: writing the function by hand is impossible, recognizing objects in real photos, for instance, but collecting examples of the desired behaviour is easy enough. Speech recognition and robot navigation are further examples. The definition: a supervised dataset $\mathcal{S}_n$ of size n is a set of n pairs (x_i, y_i) , each an example of the input-output relation we want to model. Your phone's photo app could start from 10,000 such pairs: a photo as x , and the word "dog" or "cat" as y .

One more assumption is added, and it is crucial: each example is an independent and identically distributed draw, i.i.d., from some unknown, and in fact unknowable, probability distribution $p(x, y)$. It sounds technical, but it is what makes the task meaningful. Identically

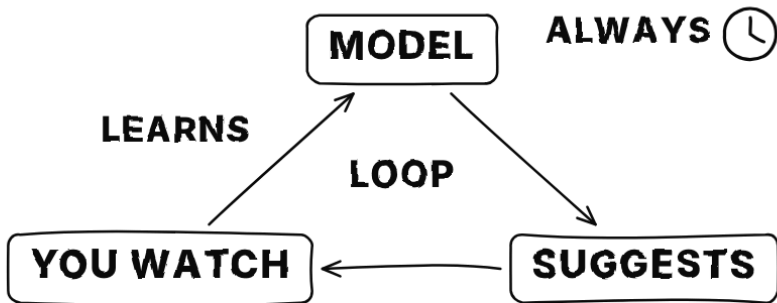
distributed means we are approximating something stable that does not change over time. The book's example is recognizing car models from photos: collected over a short period, the assumption holds; collected over several decades, it fails, because the models changed, and a model trained on such data will miss new cars or perform poorly.



Same Source, Fairly Drawn

Independently distributed means the collection has no bias and represents the whole distribution. Gather car photos next to a Tesla dealership and you get far too many of one kind and too few of every other maker. Validity also depends on where the model is used: a car dataset collected in Italy may serve in Rome or Milan, but not in Tokyo or Taiwan. So the i.i.d. assumption must always be checked.

And for modern language models, trained on enormous distributions of data, even telling which tasks are inside the distribution and which are outside it, and how far the models generalize, becomes blurred.



DOMAIN SHIFT



A Feedback Loop

The book's box adds that checking is never done once and for all; it must continue for the whole life of a model. Left unchecked, slow changes in which cars are on the road will degrade a car classifier, an example of domain shift. A recommender system is trickier still: it changes how people use an app, because users start reacting to its own suggestions, which creates feedback loops that require the system and the app to be re-evaluated constantly. The videos your streaming app suggests are a live case: what you watch shapes what it learns, and what it learns shapes what you watch.

Try it yourself

On a calculator set to radians, evaluate $\sin(3.14159)$ plus $\sin(1.5708)$: you get about 0 plus 1, which is 1. Now evaluate $\sin(3.24159)$ plus $\sin(1.62080)$: about minus 0.0998 plus 0.9988, which is 0.899. The landscape dropped by about 0.1 when you stepped 0.1 against the gradient, exactly what one step of gradient descent promises.

Remember

The suggested exercise implements $f(x) = \sin(x_1)\cos(x_2) + \sin(0.5x_1)\cos(0.5x_2)$ in NumPy, JAX and PyTorch: vectorized over a batch, with a hand-coded gradient, gradient descent paths from several starts, and momentum with a shrinking gradient norm. A supervised dataset is a set of n input-output pairs, used when writing the function by hand is impossible but examples are easy to collect. The pairs are assumed i.i.d. from an unknown distribution: identically distributed means stable over time, independently distributed means collected without bias, as the decades of car photos and the Tesla dealership show. The assumption must be watched for the model's whole life: domain shift degrades models over time, and recommender systems create feedback loops by changing the behaviour they learn from.

Questions

1. A car classifier is trained on photos collected over the last forty years. Which part of the i.i.d. assumption is broken, and why?
 - a. Identically distributed, because car models changed over time, so the relation is not stable
 - b. Independently distributed, because the photos were taken in one city
 - c. Neither, because more years give more data
2. Why would collecting car photos next to one brand's dealership break the i.i.d. assumption?
 - a. Because the photos would be too similar in lighting
 - b. Because the collection would be biased, with too many cars of one kind and too few of others
 - c. Because dealerships only sell new cars
3. Why does a recommender system need constant re-evaluation?
 - a. Because its gradients become noisy over time
 - b. Because its dataset is always too small
 - c. Because users react to its own suggestions, creating feedback loops that change the data it learns from
4. In the practice exercise, what should happen to the length of the gradient as gradient descent proceeds?
 - a. It should shrink towards zero as the method approaches a stationary point
 - b. It should grow, because momentum adds to every step
 - c. It should stay constant if the step size is fixed

Answers

1. **a)** Identically distributed means the thing being approximated is stable over time; over decades car models change, and the trained model will miss new ones.
2. **b)** Independently distributed means the collection has no bias and represents the whole distribution; a dealership over-represents one maker.
3. **c)** A recommender changes how users interact with an app, so the data shifts in response to the system itself, which calls for continuous re-evaluation.
4. **a)** At a stationary point the gradient is zero, so as the optimizer approaches one, the norm of the gradient converges to zero.

Worked examples

One step on the practice landscape For $f(x) = \sin(x_1)\cos(x_2) + \sin(0.5x_1)\cos(0.5x_2)$ at $(\pi, 0)$, compute f and the gradient, then take one gradient-descent step of size 0.1.

$$f(\pi, 0) = \sin(\pi) \cdot 1 + \sin(\pi/2) \cdot 1 = 0 + 1 = 1.$$

$$\partial f / \partial x_1 = \cos(x_1)\cos(x_2) + 0.5\cos(0.5x_1)\cos(0.5x_2) = -1 + 0.5 \times 0 = -1.$$

$$\partial f / \partial x_2 = -\sin(x_1)\sin(x_2) - 0.5\sin(0.5x_1)\sin(0.5x_2) = 0 \text{ at } x_2 = 0.$$

Step: $x_1 = \pi - 0.1 \times (-1) \approx 3.2416$, x_2 stays 0; $f \approx -0.0998 + 0.9988 \approx 0.899$.

Answer: f drops from 1 to about 0.899 after one step

Practice

1. Write the partial derivative of $f(x) = \sin(x_1)\cos(x_2) + \sin(0.5x_1)\cos(0.5x_2)$ with respect to x_2 , and evaluate it at $(\pi/2, \pi/2)$.

Hint: The derivative of $\cos$ is minus $\sin$, and the chain rule brings a factor 0.5 for the slow wave.

Answer: $\partial f / \partial x_2 = -\sin(x_1)\sin(x_2) - 0.5\sin(0.5x_1)\sin(0.5x_2)$. At $(\pi/2, \pi/2)$: $-1 \times 1 - 0.5 \times \sin(\pi/4)^2 = -1 - 0.5 \times 0.5 = -1.25$.

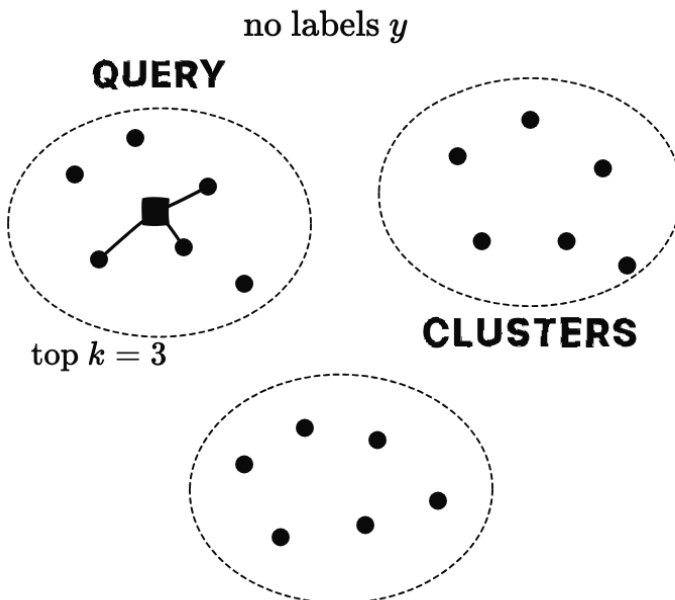
2. A team trains a face-unlock model on photos from one country and ships it worldwide. Explain, with the book's terms, what could go wrong.

Hint: Think of the car dataset collected in Italy and used in Tokyo.

Answer: The data may be valid where it was collected but not elsewhere: the deployment distribution differs, so the i.i.d. assumption fails and performance can drop for users unlike those in the training set; it must be checked, and re-checked over time.

Learning Without Labels, and Reusing What Was Learned

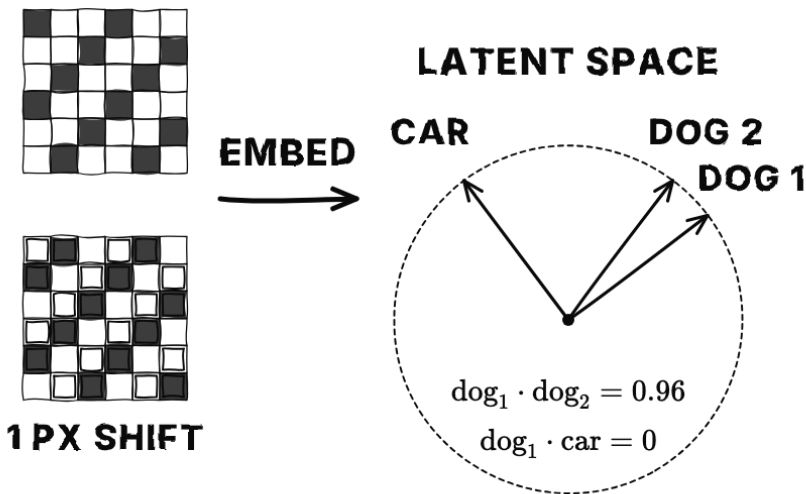
Open your photo app, choose a picture of your dog, and look for a button that says something like "show similar". It brings back other dog photos, including ones nobody ever labelled, and puts the photos of your car far away. Nobody told the app what is similar to what; something inside a trained model did. This part explains that something, and the many ways a trained model can be reused.



Grouping and Finding Similar

Most successful applications use supervised learning in some form, but there are many variations. When the dataset has no targets y_i at all, we speak of unsupervised learning. A classic use is clustering: grouping inputs so that points in the same cluster are similar and points in different clusters are not. Another is retrieval: searching a large database for the top k elements most similar to a user's query.

With complex data like images this is hard, because distances computed on raw pixels mean little: even a tiny change can alter millions of pixels. A phone photo of 12 million pixels, shifted sideways by a single pixel, looks identical to us, yet almost every one of its 36 million colour numbers changes.



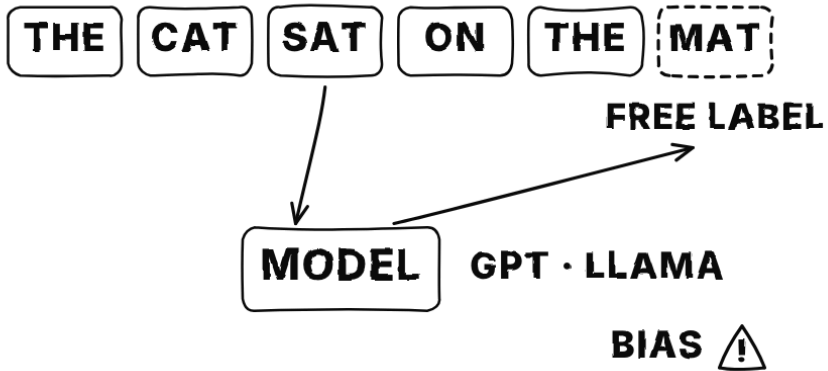
From Pixels to a Latent Space

Now suppose we already have a differentiable model optimized for some other, fairly general task, such as image classification; this is called a pre-trained model. As the book will show, the internal states of such a model can be read as vectors in a high-dimensional space, and often they have useful geometry: things that are similar in meaning are sent, embedded, to points that are close together. The book's figure calls this embedding the data: moving it from a space without a meaningful distance, raw images, into a space with one, the latent space.

Here the dot product of the earlier parts returns. Suppose, in a tiny two-dimensional latent space, one dog photo lands at $(0.8, 0.6)$, another dog at $(0.6, 0.8)$, and a car at $(-0.6, 0.8)$, all of length one. The cosine similarity of the two dogs is 0.48 plus 0.48 , which is 0.96 , nearly the same direction; the first dog and the car give minus 0.48 plus 0.48 , which is 0 , unrelated. That is how "show similar" works, and the same latent vectors can be fed to standard clustering methods such as Gaussian mixture models.

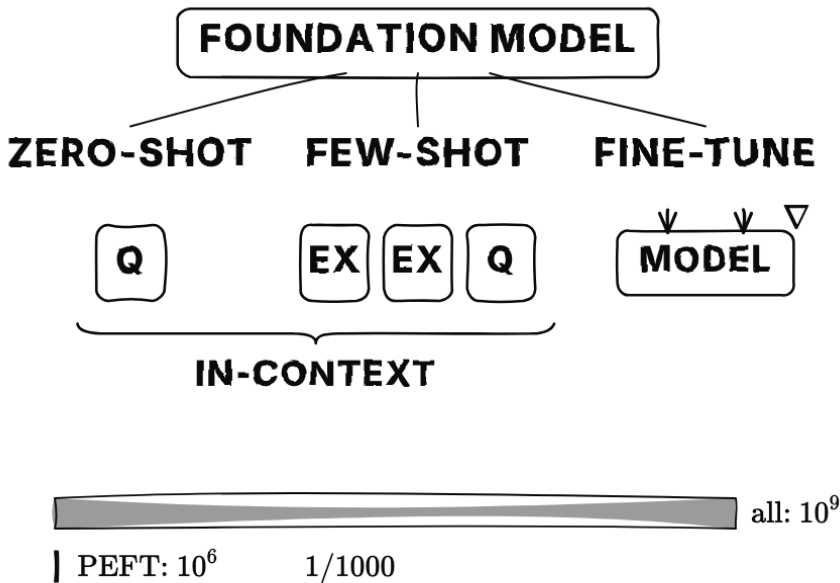
And if no pre-trained model is available? A common variant of unsupervised learning is self-supervised learning, SSL. Its idea is to find a supervised task hidden inside unlabelled data, and use it to train a model that serves many later, downstream, tasks. With a large body of text, for example, we can always train a program to predict how a short piece of text continues: the next words are free labels. The discovery that neural networks pre-trained this way also produce efficient embeddings of text had a profound impact on the field. The book adds a warning: large web datasets are full of biases, profanity and vulgar content, models trained on them absorb these biases, and this is a major criticism of closed-source foundation models.

SELF-SUPERVISED



Labels Hidden in Text

Large language models, as Chapters 8 and 10 will show, are modern versions of this idea: models such as GPT or Llama always begin with self-supervised training to predict the next token, a small piece of text. Such models are sometimes called foundation models, and there are three ways to use them, which the book's figure compares. Zero-shot: give the model a new task directly, such as a question, and use its answer. Few-shot prompting: put a few examples of the new task into the prompt first; both of these work only if the model generalizes very well. Fine-tuning: take the pre-trained model and optimize its parameters by gradient descent on a small dataset for the new task, much as if training from scratch.



Three Ways to Use a Model

When you ask a chat assistant to translate one sentence, that is zero-shot; when you first show it two translated examples in the style you want, that is few-shot prompting; and a company that trains it further on its own documents is fine-tuning it. When the examples in the prompt are enough and no fine-tuning is needed, the book says the model is performing in-context learning; few-shot learning can also be done by fine-tuning. The book itself focuses on building models from scratch, noting that fine-tuning works by similar means.

Fine-tuning is especially easy today thanks to large open-source repositories of models. It can update all the parameters of the starting model, or only a small subset, or a small number of added parameters; the last options are called parameter-efficient fine-

tuning, PEFT. To get a feeling for the saving: training one million added numbers on top of a model of one billion means adjusting one thousandth as many values as full fine-tuning would. The book leaves PEFT techniques to its next volume.

Try it yourself

Draw three arrows of length one from the centre of a sheet: one towards the upper right for "dog one", one slightly steeper for "dog two", and one towards the upper left for "car". Measure the angles with a protractor: the two dog arrows are close, about 16 degrees apart, while the car arrow is at a right angle to the first dog. A small angle means a large cosine, that is, "similar", which is all a retrieval system needs to rank its results.

Remember

Without targets we have unsupervised learning, such as clustering and retrieval, but raw-pixel distances are meaningless for images. A pre-trained model's internal vectors embed data into a latent space where similar things lie close, so dot products and clustering work there, as 0.96 for two dogs and 0 for a dog and a car showed. Self-supervised learning invents labels from unlabelled data, like predicting the next words of text; it is how language models such as GPT and Llama start, and it also imports the biases of web data. Foundation models can be used zero-shot, with few-shot prompts, or fine-tuned by gradient descent, either fully or by parameter-efficient fine-tuning of a small set of parameters.

Questions

1. Why is it hard to find similar images by measuring distances directly on their pixels?
 - a. Because pixels cannot be stored as numbers
 - b. Because even a tiny change, like a one-pixel shift, alters millions of pixel values
 - c. Because images have no labels
2. What makes the internal vectors of a pre-trained model useful for retrieval and clustering?
 - a. Things similar in meaning are embedded close together in the latent space
 - b. They are always two-dimensional
 - c. They contain the original pixels in compressed form
3. How does self-supervised learning get labels from a large body of unlabelled text?
 - a. By asking people to label a small part of it
 - b. By clustering the text first
 - c. By training the model to predict how a piece of text continues, so the next words serve as labels
4. You show a chat assistant two examples of your preferred summary style, then ask for a third summary, without changing its parameters. What is this called?
 - a. Fine-tuning
 - b. Few-shot prompting, an example of in-context learning

c. Zero-shot use

Answers

1. **b)** Distances on raw pixels are ill-defined: small perturbations modify millions of pixels while the picture looks the same to us.
2. **a)** The model's internal states often have useful geometry, sending semantically similar objects to nearby points, so distances and dot products become meaningful there.
3. **c)** SSL finds a supervised objective inside unlabelled data; with text one can always train a model to predict the continuation, which is how models like GPT and Llama start.
4. **b)** Giving a few examples in the prompt is few-shot prompting; when no fine-tuning is needed, the model is performing in-context learning.

Worked examples

Similarity in a latent space In a two-dimensional latent space, dog one is at $(0.8, 0.6)$, dog two at $(0.6, 0.8)$ and a car at $(-0.6, 0.8)$, all of length one. Compute the cosine similarity of dog one with dog two and with the car.

For unit vectors the cosine similarity is the dot product.

Dog one with dog two: $0.8 \times 0.6 + 0.6 \times 0.8 = 0.48 + 0.48 = 0.96$.

Dog one with the car: $0.8 \times (-0.6) + 0.6 \times 0.8 = -0.48 + 0.48 = 0$.

So the two dogs point almost the same way, and the car is at a right angle to dog one.

Answer: 0.96 for the two dogs, 0 for dog and car

Practice

1. A photo has 4,000 by 3,000 pixels with 3 colour values each. How many numbers describe it, and why does this make pixel distances a poor measure of similarity?

Hint: Multiply the three sizes; think of a one-pixel shift.

Answer: $4,000 \times 3,000 \times 3 = 36,000,000$ numbers. Shifting the photo by one pixel changes almost all of them, so two photos that look the same can be very far apart in pixel distance.

2. A base model has 1 billion parameters. Full fine-tuning updates all of them, while a parameter-efficient method trains 2 million added parameters. What fraction of the parameter count does the parameter-efficient method train?

Hint: Divide the two numbers.

Answer: $2,000,000 / 1,000,000,000 = 0.002$, that is, two thousandths of the base model's size.

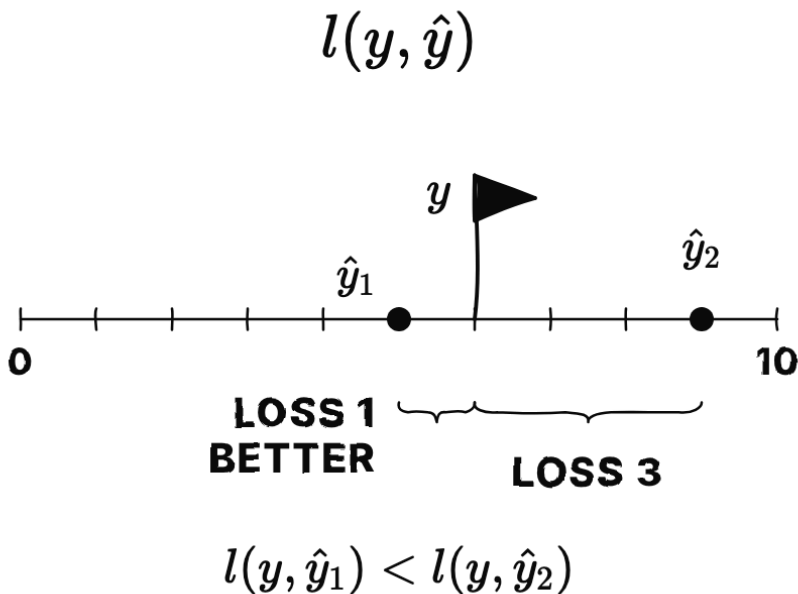
3. Classify each use of a language model as zero-shot, few-shot prompting or fine-tuning: (a) asking it directly to translate a sentence; (b) training it further on a company's support tickets; (c) showing it three solved riddles before asking a fourth.

Hint: Ask whether examples are in the prompt and whether parameters change.

Answer: (a) zero-shot; (b) fine-tuning, since its parameters are optimized by gradient descent; (c) few-shot prompting, since the examples are in the prompt and parameters do not change.

Scoring Mistakes: the Loss

Play a guessing game with a friend: they think of a number and you guess. After each guess they say how far off you were, and you try to make that number small. Without that one number, "how far off", you would have no way to improve; with it, improving becomes a matter of arithmetic. A loss function is that number, and training a model is the game played millions of times.



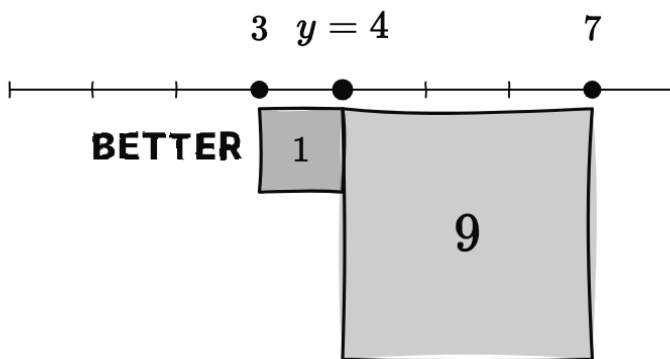
How Far Off Is the Guess

First, the book completes its list of learning scenarios with a few pointers. When only part of a dataset is labelled, the setting is semi-

supervised, with examples coming in Chapter 12. When there are several datasets from similar distributions, or one distribution changing over time, countless problems arise depending on the order in which tasks or data arrive: domain adaptation, meta-learning, continual learning, metric learning, unlearning and more, some of which the next volume treats.

Section 3.2 turns to loss functions, which make the idea of "approximating" a behaviour precise. The definition: given a target y and the model's prediction $\hat{y} = f(x)$, a loss $l(y, \hat{y})$ is a single real number, differentiable, whose value tracks how well the model does. Concretely, $l(y, \hat{y}_1) < l(y, \hat{y}_2)$ means prediction one is better than prediction two for that target.

A common example is the squared error, the difference squared. If a house's true price is 4 and two models predict 3 and 7, the losses are 1 squared, which is 1, and 3 squared, which is 9; the smaller loss, 1, marks the better prediction. A loss turns our understanding of the task, and our preferences among solutions, into a single real number an optimizer can work with. And because it is differentiable, it turns learning into a mathematical optimization problem that gradient descent can solve, by minimizing the average loss over the dataset.



$$f^* = \arg \min_f \frac{1}{n} \sum_{i=1}^n l(y_i, f(x_i)) \quad \text{ERM}$$

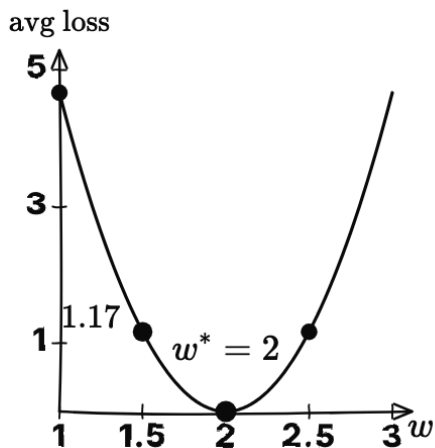
Squared Error, Then the Average

So, given a dataset $\mathcal{S}_n$ of pairs and a loss l , a sensible goal is the model with the smallest average loss: $f^* = \arg \min_f (1/n) \sum_{i=1}^n l(y_i, f(x_i))$. In words: try the model on every example, score each prediction, average the scores, and pick the model with the lowest average. For historical reasons this is called empirical risk minimization, ERM, where "risk" is just another word for loss.

Let us compute an empirical risk. Three examples have targets 2, 4 and 6, and a model predicts 3, 4 and 4. The squared losses are 1, 0 and 4; their average is 5 divided by 3, about 1.67.

y	$\hat{y}$	loss
2	3	1
4	4	0
6	4	4

$$\text{avg} = 5/3 \approx 1.67$$



$$w^* = \arg \min_w \frac{1}{n} \sum_{i=1}^n l(y_i, f(x_i, w))$$

Average Loss and Its Valley

That formula searches over all possible functions of x . In practice, as the book will show, a model is described by a set of tensors w , its parameters, and we write $f(x, w)$. Training then searches for the best parameters by numerical optimization, minimizing the empirical risk with gradient descent: $w^* = \operatorname{argmin}_w (1/n) \sum_{i=1}^n l(y_i, f(x_i, w))$. The only change is that the search is now over the parameter tensor w .

Make it concrete with the simplest model, $f(x, w) = wx$, on inputs 1, 2, 3 with targets 2, 4, 6. With w equal to 1.5, the predictions are 1.5, 3 and 4.5, the squared losses 0.25, 1 and 2.25, and the average about 1.17. With w equal to 2, every prediction is exact and the average loss is 0, so here w^* is 2. Your phone's photo app was trained the same

way, only with millions of photos, millions of parameters and a loss suited to labels instead of prices.

Try it yourself

Take the three pairs (1, 2), (2, 4), (3, 6) and try a few values of w in the model w times x : 1, 1.5, 2, 2.5. For each, compute the three squared errors and their average. You should see the average fall towards 0 at w equal to 2 and rise again beyond it, a small valley whose bottom is the trained parameter.

Remember

Beyond plain supervised learning lie semi-supervised learning and many multi-dataset settings such as domain adaptation, continual learning and unlearning. A loss $l(y, \hat{y})$ is a differentiable real number that is lower for better predictions, like a squared error of 1 against 9. Empirical risk minimization picks the model with the lowest average loss over the dataset, about 1.67 for predictions 3, 4, 4 against targets 2, 4, 6. With a parameterized model $f(x, w)$, training searches for $w^* = \operatorname{argmin}_w$ of the average loss, using gradient descent; for wx on our three pairs, w^* is 2.

Questions

1. Why must a loss function be differentiable?

- a. So that it can be printed as a real number
 - b. So that learning becomes an optimization problem gradient descent can solve
 - c. So that it is always positive
2. What does empirical risk minimization choose?
- a. The model with the smallest loss on one chosen example
 - b. The model with the largest number of parameters
 - c. The model with the smallest average loss over the whole dataset
3. For a target of 4, prediction A is 3 and prediction B is 7. With the squared error, which is better and why?
- a. A, because its loss is 1 against B's 9
 - b. B, because it is larger than the target
 - c. They are equally good, since both are wrong
4. What changes when we move from minimizing over all functions f to minimizing over parameters w ?
- a. The loss function is no longer needed
 - b. The search becomes a numerical optimization over the parameter tensors of a fixed model family
 - c. The dataset is replaced by a single example

Answers

1. **b)** Being differentiable, the loss turns learning into a mathematical optimization problem solved by gradient descent, minimizing the average loss on the dataset.

2. c) ERM scores every prediction on the dataset, averages the losses, and picks the model, or the parameters, with the lowest average; risk is a synonym for loss.

3. a) A lower loss means a better prediction: $(4 - 3)^2 = 1$ is smaller than $(4 - 7)^2 = 9$.

4. b) Models are described by tensors w , so training searches for w^* = argmin over w of the average loss by numerical optimization such as gradient descent.

Worked examples

Empirical risk of a simple model The model $f(x, w) = wx$ is tried on the pairs (1, 2), (2, 4), (3, 6) with the squared loss. Compute the empirical risk for $w = 1.5$ and for $w = 2$.

$w = 1.5$: predictions 1.5, 3, 4.5; squared errors 0.25, 1, 2.25.

Average: $(0.25 + 1 + 2.25) / 3 = 3.5 / 3 \approx 1.17$.

$w = 2$: predictions 2, 4, 6, all exact; squared errors 0, 0, 0; average 0.

So $w = 2$ minimizes the empirical risk on this dataset.

Answer: About 1.17 for $w = 1.5$, and 0 for $w = 2$, which is w^*

Practice

1. Targets are 1, 3 and 5; a model predicts 2, 3 and 3. Compute the empirical risk with the squared loss.

Hint: Square each difference, add, divide by 3.

Answer: Squared errors 1, 0 and 4; sum 5; empirical risk $5/3 \approx 1.67$.

2. For $f(x, w) = wx$ on the pairs (1, 2), (2, 4), (3, 6), compute the empirical risk for $w = 2.5$ and compare with $w = 1.5$.

Hint: Predictions 2.5, 5, 7.5.

Answer: Squared errors 0.25, 1 and 2.25, average ≈ 1.17 , the same as $w = 1.5$: the risk rises symmetrically on both sides of the minimum at $w = 2$.

3. A spam filter is trained on a dataset where only 5% of emails are labelled. Which learning setting is this, and what does the book say about scenarios where data arrive over time from changing distributions?

Hint: Think of partial labels, and of the list of multi-dataset problems.

Answer: Partially labelled data is semi-supervised learning. Data from similar or changing distributions arriving over time give rise to problems such as domain adaptation and continual learning, some treated in the next volume.

Losses That Can Be Followed, and Models That Only Remember

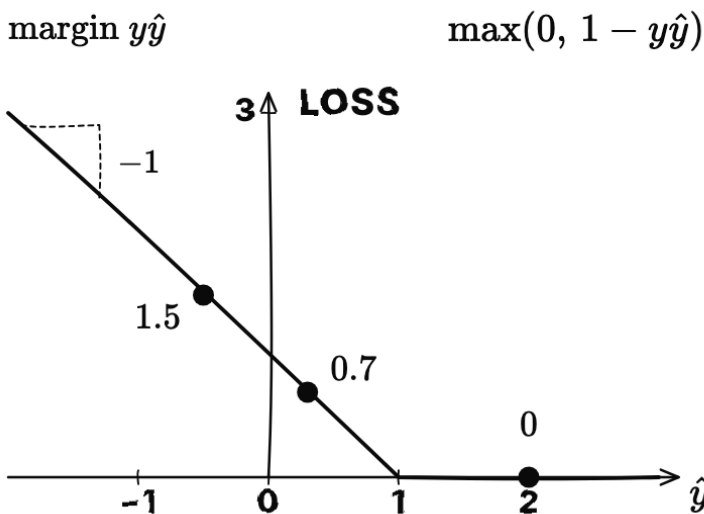
Play the guessing game again, but now your friend only ever says "right" or "wrong". You guess 30 and hear "wrong"; you guess 31 and hear "wrong" again. You learn nothing about which way to move, because the answer does not change when you nudge your guess. A loss with that flaw cannot guide gradient descent, and this part shows how to avoid it, before turning to a deeper trap: a model that scores perfectly by simply remembering.



The Right-or-Wrong Loss Is Flat

First observation: the differentiability of the loss is essential. Take a binary classification task, where the target $y = \pm 1$ takes only two values, and a model $f(x)$ that outputs any real number. We can read the model's decision as the sign of its output, $\text{sign}(f(x))$, and define the 0/1 loss: $l(y, \hat{y}) = 0$ if $\text{sign}(\hat{y}) = y$, and 1 otherwise. It matches our intuition of being right, but it is useless for training. Its gradient is zero almost everywhere, except exactly where the sign flips, so gradient descent never moves from its starting point: right-or-wrong, with nothing in between.

A less intuitive but more useful quantity is the margin, $y\hat{y}$. It is positive when the model's sign agrees with the target and negative when it does not, and, unlike the 0/1 loss, it changes smoothly. One loss built on it is the hinge loss, $l(y, \hat{y}) = \max(0, 1 - y\hat{y})$, used to train support vector machines.



The Hinge Loss Has a Slope

Compute it for a target of plus 1. A prediction of 0.3 is right in sign, so the 0/1 loss is 0, but the hinge loss is 1 minus 0.3, which is 0.7: correct, yet not confidently. A prediction of 2 gives a margin of 2 and a hinge loss of 0. A prediction of minus 0.5 gives a margin of minus 0.5 and a hinge loss of 1.5, larger because it is wrong. And below a margin of 1, the hinge loss falls by one for every unit the prediction rises: a slope gradient descent can follow. The book draws the lesson: there is a built-in tension between losses that express our idea of good performance and losses that work well for numerical optimization.

A box notes the history: empirical and expected risk minimization are linked with the work of the Russian computer scientist Vladimir Vapnik, who founded statistical learning theory. That theory studies

how well empirical risk approximates expected risk from finitely many samples, for restricted families of functions and measures of their complexity. Modern neural networks, which keep generalizing well long after overfitting would have been expected, have opened many new research questions in that theory.

Second observation: empirical risk is always trivial to minimize.

Define a model that looks up its input in the training set: if the pair is there, return its y ; otherwise return a fixed default such as 0. This look-up table makes the training loss as low as it can be, yet it is useless, because it knows nothing about any input it has not seen. That is the difference between memorizing and learning. We optimize the average loss on training data, but our real goal is low loss on future inputs not yet seen; the training set is only a stand-in.

LOOK-UP

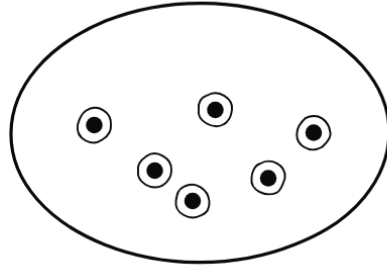
$1 \rightarrow 2$
$2 \rightarrow 4$
$3 \rightarrow 6$

TRAIN LOSS 0

$5 \rightarrow 0$ ✗

MEMORIZING

$$ER[f] = \mathbb{E}_{p(x,y)}[l(y, f(x))]$$



ALL PAIRS

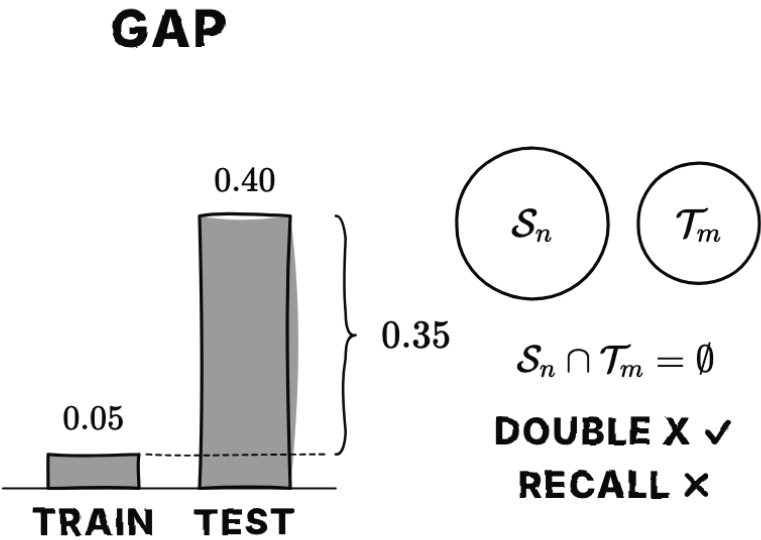
EMPIRICAL $\approx$

Remembering Versus Expected Risk

The book formalizes this as the expected risk: $ER[f] = \mathbb{E}_{p(x,y)}[l(y, f(x))]$. In words: the average loss over every input-output pair the model could ever meet, weighted by how likely each one is, for instance every email a spam filter could ever receive. A model with low expected risk would be guaranteed to work, but computing it is impossible, since we can never list and label every possible input. Empirical risk estimates it from the dataset we have, as a Monte Carlo approximation: an average over random samples standing in for an average over everything.

The difference between expected and empirical risk is the generalization gap. The look-up table has a huge gap: it overfits its training data. In practice we test generalization with a separate test

set $\mathcal{T}_m$ of m points never used in training, sharing nothing with the training set: $\mathcal{S}_n \cap \mathcal{T}_m = \emptyset$. If a photo classifier reaches an average loss of 0.05 on its training photos but 0.40 on held-out test photos, the gap of 0.35 is a measure of how much it overfits.



The Generalization Gap

Try it yourself

Write down five pairs, such as (1, 2), (2, 4), (3, 6), (4, 8), (5, 10), and hide two of them. Build a "model" that just recalls the three you kept, and another that says "double the input". Check both on the two hidden pairs: the recall model has no answer and falls back on 0, the doubling model gets both right; that is memorization against learning, measured on a test set.

Remember

The loss must be differentiable: the 0/1 loss matches being right but has zero gradient almost everywhere, so gradient descent cannot move. The margin $\hat{y}\hat{y}$ varies smoothly, and the hinge loss $\max(0, 1 - \hat{y}\hat{y})$ gives 0.7 for a correct but timid 0.3, 0 for a confident 2, and 1.5 for a wrong -0.5 . Empirical risk can be driven to its minimum by a look-up table that only memorizes; what we really want is low expected risk, the average loss over all possible inputs, which can only be estimated. The gap between expected and empirical risk measures overfitting, and a separate test set never used in training estimates it in practice.

Questions

1. Why is the 0/1 loss useless for training with gradient descent?
 - a. Because it gives the wrong answer for correct predictions

- b. Because its gradient is zero almost everywhere, so gradient descent never moves
 - c. Because it can only be used with support vector machines
- 2. For a target of +1, which prediction gets a hinge loss of 0.7?
 - a. 0.3
 - b. 2
 - c. -0.5
- 3. A look-up table that returns the stored answer for every training input achieves the lowest empirical risk. Why is it still useless?
 - a. Because looking things up is too slow
 - b. Because its loss is not differentiable
 - c. Because it only memorizes, and has nothing useful to say about inputs it has not seen
- 4. How is overfitting measured in practice?
 - a. By comparing the loss on the training set with the loss on a separate test set never used in training
 - b. By computing the expected risk over all possible inputs
 - c. By counting the parameters of the model

Answers

- 1. **b)** The 0/1 loss only changes where the sign of the prediction flips; everywhere else nudging the prediction changes nothing, so the gradient is zero and the optimizer stays at its start.
- 2. **a)** The hinge loss is $\max(0, 1 - y\hat{y})$: for 0.3 it is $1 - 0.3 = 0.7$, for 2 it is 0, and for -0.5 it is 1.5.

3. c) Our true goal is low loss on future, unseen inputs, the expected risk; the look-up table falls back on a default and overfits, with a huge generalization gap.

4. a) The expected risk cannot be computed, so a held-out test set estimates it; the difference between training and test loss approximates the generalization gap.

Worked examples

Hinge loss against 0/1 loss For a target $y = +1$, compute the margin, the 0/1 loss and the hinge loss $\max(0, 1 - y\hat{y})$ for predictions $\hat{y} = 0.3, 2$ and -0.5 .

$\hat{y} = 0.3$: margin 0.3; sign agrees, so 0/1 loss 0; hinge $1 - 0.3 = 0.7$.

$\hat{y} = 2$: margin 2; 0/1 loss 0; hinge $\max(0, -1) = 0$.

$\hat{y} = -0.5$: margin -0.5 ; sign disagrees, so 0/1 loss 1; hinge $1 + 0.5 = 1.5$.

The 0/1 loss cannot tell 0.3 from 2, while the hinge loss still pushes 0.3 towards a confident margin.

Answer: Hinge losses 0.7, 0 and 1.5; 0/1 losses 0, 0 and 1

Practice

1. For a target $y = -1$, compute the hinge loss for predictions $\hat{y} = -3, -0.2$ and 0.4 .

Hint: First compute the margin $y\hat{y}$.

Answer: Margins 3, 0.2 and -0.4 ; hinge losses $\max(0, -2) = 0, 0.8$ and 1.4.

2. A model has average loss 0.12 on its training set and 0.15 on a test set; another has 0.01 on training and 0.45 on test. Which overfits more, and what is each generalization gap estimate?

Hint: Subtract training loss from test loss.

Answer: Gaps: 0.03 and 0.44. The second model overfits much more: it almost memorizes its training data but does poorly on unseen data.

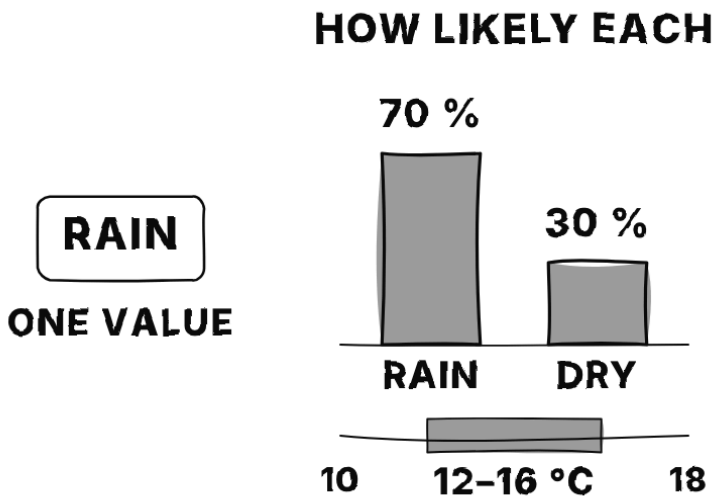
3. Explain in your own words why the empirical risk is called a Monte Carlo approximation of the expected risk.

Hint: Think of what the expected risk averages over and what the dataset provides.

Answer: The expected risk averages the loss over every possible input-output pair, weighted by the distribution, which cannot be computed. The dataset is a random sample from that distribution, and averaging the loss over it estimates the full average, the way Monte Carlo methods estimate averages from random samples.

Predicting a Distribution Instead of a Number

Ask a weather app about tomorrow and it rarely says simply "rain". It says "70 percent chance of rain", or "between 12 and 16 degrees". That is a richer kind of answer: not one value, but how likely each value is. The book now shows that a model can be built to answer this way, and that doing so tells us which loss to use.



PROBABILISTIC

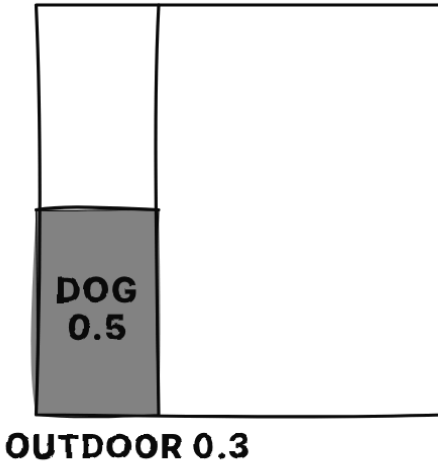
One Value or How Likely Each

The book suggests that this is a good moment to study or skim its appendix on probability, especially distributions, sufficient statistics

and maximum likelihood estimation. The loss, it says, encodes our prior knowledge about the task and has a big effect on performance. Sometimes simple reasoning about the problem is enough to design one, as with the hinge loss. But there is a more principled route: restate the whole of training in purely probabilistic terms, a viewpoint many books prefer and which can be more intuitive or more useful in some settings.

The key observation starts from the assumption that examples come from a distribution $p(x, y)$. By the product rule of probability, $p(x, y) = p(x) p(y | x)$. In words: the chance of seeing a particular input and output together is the chance of the input, times the chance of that output once the input is known. If 30 percent of photos are taken outdoors, and half of the outdoor photos show a dog, then the chance of an outdoor photo with a dog is 0.3 times 0.5, which is 0.15.

ALL PHOTOS



$$p(x, y) = p(x) p(y | x) \\ = p(x | y) p(y)$$

$$0.3 \times 0.5 = 0.15$$

$p(y | x)$: discriminative

$p(x)$: generative

The Product Rule as Areas

The book adds that the product can also be split the other way, $p(x, y) = p(x | y) p(y)$. Methods that estimate $p(x)$ or $p(x | y)$ are called generative, and those that estimate $p(y | x)$ are called discriminative; apart from language modeling, this book focuses on the discriminative kind and leaves generative modeling to its next volume.

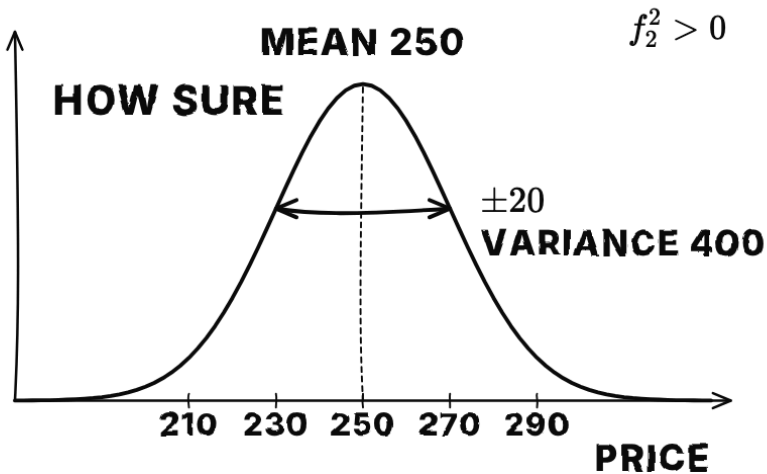
Approximating $p(y | x)$ with a single function $f(x)$ that outputs one prediction makes sense only if, for each input, almost all the probability sits on one value of y . Such a distribution is close to what mathematicians call a Dirac delta, a spike at a single point, and assuming it simplifies the problem enormously.

The relaxation: let the model's outputs not be the prediction itself, but the sufficient statistics, the parameters, of a conditional distribution $p(y \mid f(x))$ over possible outputs. For a classification problem with three classes, numbered 1, 2 and 3, the model can have three outputs that define a categorical distribution: $p(y \mid f(x)) = \prod_{i=1}^3 f_i(x)^{y_i}$. Here $y \sim \text{Binary}(3)$ is the one-hot encoding of the class: all zeros except a 1 in the position of the class, and $f(x) \sim \Delta(3)$ lists the predicted probability of each class.

Let us read the formula with numbers. Suppose the photo app predicts (0.7, 0.2, 0.1) for dog, cat and car, and the true class is dog, one-hot (1, 0, 0). The product is 0.7 to the power 1, times 0.2 to the power 0, times 0.1 to the power 0, and since anything to the power 0 is 1, the result is simply 0.7: the probability the model gave to the right class. If the photo were in fact a car, one-hot (0, 0, 1), the same formula would pick out 0.1. The exponents work as a switch that selects the probability of the true class.

For regression, predicting a real number $y \in \mathbb{R}$, the model can output two values, $f(x) \sim (2)$, the mean and the spread of a Gaussian, a bell curve: $p(y \mid f(x)) = \mathcal{N}(y \mid f_1(x), f_2^2(x))$. The second output is squared so that the variance is always positive. If a house-price model outputs 250 and minus 20, the forecast is a bell curve centred on 250 with variance 400, a typical spread of 20; the minus sign disappears when squared.

$$p(y \mid f(x)) = \mathcal{N}(y \mid f_1(x), f_2^2(x))$$



A Bell Curve for a Price

This setup is very general and includes everything said so far. It also gives the designer more freedom, because choosing a form for $p(y \mid x)$ can be easier than inventing a loss $l(y, \hat{y})$ directly. And it models uncertainty naturally: the variance of the Gaussian says how sure the model is, just as the weather app's percentage does.

Try it yourself

Write three probabilities that add up to one, say 0.6, 0.3 and 0.1, for three possible answers to a quiz question. Pick the true answer and write its one-hot vector, such as (0, 1, 0). Raise each probability to the matching 0 or 1 and multiply: you get 0.3, the probability you assigned to the truth, which is the number the coming parts will try to make large.

Remember

The loss carries our prior knowledge of the task; instead of designing it by hand, we can restate training in probabilistic terms. By the product rule, $p(x, y) = p(x) p(y | x)$; estimating $p(y | x)$ is discriminative, estimating $p(x)$ or $p(x | y)$ is generative, and the book focuses on the discriminative kind. A model can output the parameters of a distribution: three class probabilities for a categorical, where the one-hot exponents pick the probability of the true class, 0.7 for a dog predicted at 0.7. For regression it can output a Gaussian's mean and a squared spread, which keeps the variance positive and expresses the model's uncertainty.

Questions

1. A classifier outputs probabilities (0.7, 0.2, 0.1) and the true class is the third one. What does the categorical formula $\prod_i f_i(x)^{y_i}$ give?

- a. 0.7
 - b. 0.1
 - c. 1.0
2. In the Gaussian regression model, why is the second output squared?
- a. To make the mean larger
 - b. To ensure the predicted variance is always positive
 - c. To turn regression into classification
3. When does it make sense to predict a single value $f(x)$ rather than a whole distribution over y ?
- a. When, for each input, almost all the probability sits on one value of y , like a Dirac delta
 - b. When the dataset is very large
 - c. When the task has more than two classes
4. Which kind of method estimates $p(y | x)$?
- a. Generative
 - b. Unsupervised
 - c. Discriminative

Answers

- 1. b)** The one-hot vector $(0, 0, 1)$ puts exponent 1 only on the third probability, and anything to the power 0 is 1, so the product picks out 0.1, the probability given to the true class.
- 2. b)** A variance cannot be negative; squaring the second output, $f_2^2(x)$, keeps it positive whatever the model outputs, so -20 gives 400.

3. a) A single prediction is a good summary only if $p(y \mid x)$ is concentrated around one point; otherwise modelling the distribution captures the uncertainty.

4. c) Methods estimating $p(x)$ or $p(x \mid y)$ are generative, those estimating $p(y \mid x)$ discriminative; apart from language modeling, the book focuses on the latter.

Worked examples

The categorical formula as a switch A model predicts $f(x) = (0.7, 0.2, 0.1)$ for dog, cat, car. Compute $p(y \mid f(x)) = \prod_i f_i(x)^{y_i}$ when the true class is dog, and when it is car.

Dog has one-hot $(1, 0, 0)$: $0.7^1 \times 0.2^0 \times 0.1^0$.

Any number to the power 0 is 1, so the product is 0.7.

Car has one-hot $(0, 0, 1)$: $0.7^0 \times 0.2^0 \times 0.1^1 = 0.1$.

Answer: 0.7 for dog, 0.1 for car: the probability the model gave to the true class

The product rule with numbers 30% of photos are outdoors, and half of the outdoor photos show a dog. What is the probability of an outdoor photo showing a dog?

$p(x) = 0.3$ for an outdoor photo.

$p(y \mid x) = 0.5$ for a dog given an outdoor photo.

Product rule: $p(x, y) = p(x) p(y \mid x) = 0.3 \times 0.5$.

Answer: 0.15

Practice

1. A model with four classes predicts $(0.1, 0.4, 0.4, 0.1)$; the true class is the second. Write its one-hot vector and compute $\prod_i f_i(x)^{y_i}$.

Hint: Only the exponent at the true class is 1.

Answer: One-hot (0, 1, 0, 0); the product is 0.4.

2. A regression model for temperatures outputs $f(x) = (14, -3)$. Describe the predicted distribution.

Hint: The first output is the mean; the second is squared to give the variance.

Answer: A Gaussian centred on 14 with variance $(-3)^2 = 9$, that is, a typical spread of 3 degrees.

3. Explain why predicting a distribution can be more useful than predicting one number, using an example from daily life.

Hint: Think of what a weather forecast tells you beyond the most likely outcome.

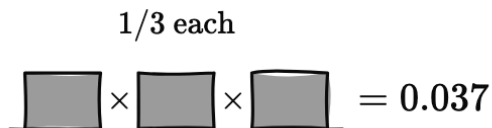
Answer: A distribution also says how certain the prediction is: '70% chance of rain' or a temperature range lets you decide whether to take an umbrella, while a single 'rain' hides the uncertainty. The variance of the Gaussian plays that role in the book's regression model.

Making the Data Likely, and Keeping Every Good Model

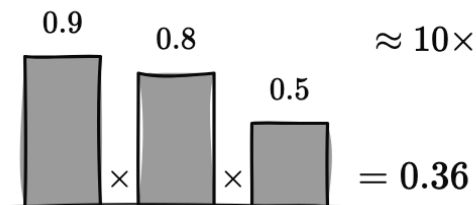
Imagine two weather forecasters who both predicted last week. For each day you look up the chance each of them gave to the weather that actually happened, and you multiply those chances together. The forecaster with the larger product made last week look least surprising, and you would trust that one more. That product is the likelihood, and choosing the model that makes it largest is how probabilistic models are trained.

$$p(\mathcal{S}_n | f) = \prod_{i=1}^n p(y_i | f(x_i))$$

RANDOM



TRAINED



Likelihood: Multiply the Chances

Recall the assumption that the dataset $\mathcal{S}_n$ is a set of independent, identically distributed samples. Because the samples are independent, the probability a given model f assigns to the whole dataset is the product over the examples: $p(\mathcal{S}_n | f) = \prod_{i=1}^n p(y_i | f(x_i))$. This quantity is called the likelihood of the dataset. A model chosen at random spreads its probability more or less evenly over all possible outputs, so it gives our particular data a small likelihood.

Take the photo app with three classes and three labelled photos. A random model gives each true class a probability of one third, so the likelihood is one third cubed, about 0.037. A trained model gives the true classes 0.9, 0.8 and 0.5, so the likelihood is 0.9 times 0.8 times 0.5, which is 0.36, about ten times larger.

The sensible strategy is therefore to pick the model that makes the dataset as likely as possible, which is the maximum likelihood principle. The definition: given a dataset and a family of distributions $p(y | f(x))$ parameterized by f , the maximum likelihood solution is $f^* = \operatorname{argmax}_f \prod_{i=1}^n p(y_i | f(x_i))$. It is again an optimization problem, but now, once the distributions are chosen, it follows from the laws of probability, while before the loss was something we had to design.

MAXIMUM LIKELIHOOD

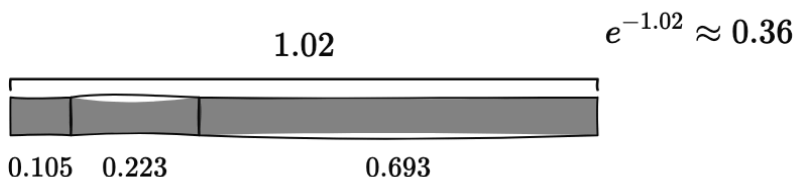
$$f^* = \arg \max_f \prod_{i=1}^n p(y_i | f(x_i))$$

log
↓

$$\arg \min_f \sum_{i=1}^n -\log p(y_i | f(x_i))$$

PSEUDO-LOSS

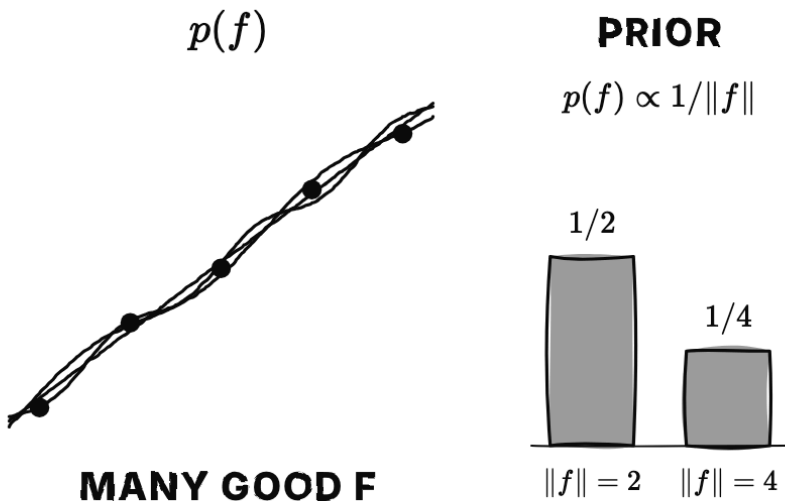
$$(1/2)^{1000} \approx 10^{-301} \quad \mathbf{X} \quad 1000 \times 0.693 = 693$$



From a Product to a Sum of Logs

The two views are closely connected. Take logarithms, which turn products into sums, and switch from maximizing to minimizing by flipping the sign: $\arg \max_f \log \prod_i p(y_i | f(x_i)) = \arg \min_f \sum_i -\log p(y_i | f(x_i))$. The logarithm keeps the order of numbers, so the best model is the same, and sums are far kinder to computers than long products: a thousand factors of one half multiply to a number with about three hundred zeros after the decimal point, while their minus logs simply add up to about 693. For our trained model, minus log of 0.9, 0.8 and 0.5 are about 0.105, 0.223 and 0.693, adding to about 1.02, and e to the minus 1.02 is indeed about 0.36. So the two formulations coincide if we treat $-\log p(y | f(x))$ as a pseudo-loss, and the book says that every loss used in practice arises this way, from a specific choice of that term.

Section 3.3 goes one step further, to Bayesian learning and Bayesian neural networks, of which the book gives only the general idea. A probability over outputs already lets a model express several plausible predictions. But training still returns one single function f , while many different parameter settings may all be good, and having all of them could give more faithful predictions.

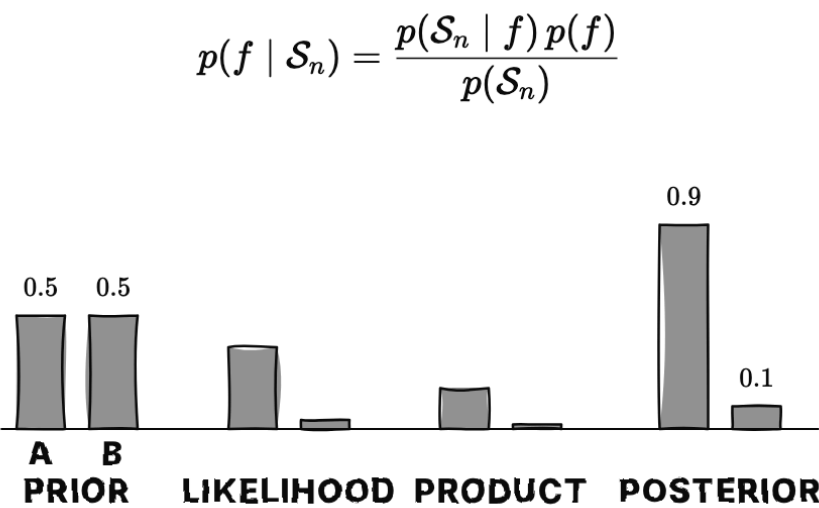


Many Good Models, and a Prior

Again the solution is another probability distribution, this time over functions. We start with a prior $p(f)$, our preference before seeing any data; since f is described by a finite set of parameters, this is in practice a prior over the weights. For example, functions with a smaller norm are often preferred because they are more stable, and

then one might take $p(f) \propto 1/\|f\|$: a function of norm 2 would be twice as probable a priori as one of norm 4.

After observing the data, the probability over functions shifts according to the prior and the likelihood, by Bayes' theorem: $p(f | \mathcal{S}_n) = p(\mathcal{S}_n | f) p(f) / p(\mathcal{S}_n)$. The left side is the posterior, the belief after the data; the denominator, the evidence, only makes the posterior add up to one. Suppose there are just two candidate models, A and B, equally likely a priori, with likelihoods 0.36 and 0.04. Prior times likelihood gives 0.18 and 0.02; the evidence is their sum, 0.2; so the posterior is 0.9 for A and 0.1 for B.



$$0.9 \times 0.8 + 0.1 \times 0.3 = 0.75$$

Unlike the single best model, the posterior can keep preferring several functions at once, which may predict better. For a new input we average over models, each weighted by its posterior: $p(y | x) = \int p(y | f(x)) p(f | \mathcal{S}_n) df$, which in practice is approximated by averaging over a finite sample of k models. With our two models, if A says "dog" with probability 0.8 and B with 0.3, the averaged prediction is 0.9 times 0.8 plus 0.1 times 0.3, which is 0.75: mostly A's view, softened a little by B's doubt.

Try it yourself

Write down three days of weather and two forecasters' probabilities for what actually happened, for example 0.9, 0.8 and 0.5 for the first and 0.4, 0.5 and 0.2 for the second. Multiply each forecaster's three numbers: 0.36 against 0.04. Then add the minus logarithms on a calculator instead: about 1.02 against 3.22; the smaller sum belongs to the same forecaster as the larger product, which is why maximizing likelihood and minimizing the log-loss are the same choice.

Remember

For i.i.d. data, the likelihood of a dataset under a model is the product of the probabilities it gives to each true output, 0.36 for a trained model against 0.037 for a random one. Maximum likelihood picks the model that makes the data most likely; taking logs turns this into minimizing the sum of $-\log p(y_i | f(x_i))$, a pseudo-loss from which practical losses derive. Bayesian learning keeps a distribution over models: a prior $p(f)$, updated by Bayes' theorem into a posterior, with the evidence as normalizer, 0.9 and 0.1 for our two models. Predictions then average all models weighted by the posterior, approximated by a finite sample of models, like 0.75 from 0.8 and 0.3.

Questions

1. Why can the likelihood of a dataset be written as a product over its examples?
 - a. Because the examples are assumed independent, so their probabilities multiply
 - b. Because the model has one parameter per example
 - c. Because the loss is always a product
2. Why is maximizing the likelihood the same as minimizing the sum of $-\log p(y_i | f(x_i))$?
 - a. Because the logarithm reverses the order of numbers

- b. Because the logarithm keeps the order of numbers and turns the product into a sum, and the minus sign turns maximizing into minimizing
 - c. Because both are always equal to zero at the best model
- 3. What does Bayesian learning keep that maximum likelihood throws away?
 - a. The training data itself
 - b. The loss function
 - c. A whole distribution over models, so several good functions can contribute to predictions
- 4. In Bayes' theorem, what is the role of the evidence $p(S_n)$?
 - a. It normalizes the posterior so that it adds up to one
 - b. It is the prior preference for small norms
 - c. It is the likelihood of the best model

Answers

- 1. **a)** The samples are i.i.d., so the probability the model assigns to the whole dataset is the product of the probabilities it assigns to each true output.
- 2. **b)** The log is increasing, so the best model is unchanged; it turns products into sums, and flipping the sign gives a minimization of a pseudo-loss.
- 3. **c)** Maximum likelihood returns one function, while the posterior can prefer several functions at once, and predictions average them by their posterior weight.
- 4. **a)** The posterior is likelihood times prior divided by the evidence, which ensures the posterior is a proper distribution, as 0.18 and 0.02 divided by

0.2 gave 0.9 and 0.1.

Worked examples

Likelihood and log-loss A classifier gives the true classes of three photos probabilities 0.9, 0.8 and 0.5. Compute the likelihood and the negative log-likelihood, and compare with a random model giving $1/3$ each.

Likelihood: $0.9 \times 0.8 \times 0.5 = 0.36$.

Negative log-likelihood: $-\ln 0.9 - \ln 0.8 - \ln 0.5 \approx 0.105 + 0.223 + 0.693 \approx 1.02$.

Check: $e^{(-1.02)} \approx 0.36$.

Random model: $(1/3)^3 \approx 0.037$, about ten times smaller.

Answer: Likelihood 0.36, negative log-likelihood about 1.02; the random model scores about 0.037

A posterior over two models Models A and B have equal priors 0.5 and likelihoods 0.36 and 0.04. Find their posteriors, and the averaged probability of 'dog' if A predicts 0.8 and B predicts 0.3.

Prior times likelihood: $0.5 \times 0.36 = 0.18$ and $0.5 \times 0.04 = 0.02$.

Evidence: $0.18 + 0.02 = 0.2$.

Posteriors: $0.18 / 0.2 = 0.9$ for A and $0.02 / 0.2 = 0.1$ for B.

Averaged prediction: $0.9 \times 0.8 + 0.1 \times 0.3 = 0.72 + 0.03 = 0.75$.

Answer: Posteriors 0.9 and 0.1; averaged probability of dog 0.75

Practice

1. A model gives the true outputs of four examples probabilities 0.5, 0.5, 0.8 and 0.25. Compute the likelihood.

Hint: Multiply the four numbers.

Answer: $0.5 \times 0.5 \times 0.8 \times 0.25 = 0.05$.

- 2.** Models A and B have priors 0.7 and 0.3, and likelihoods 0.1 and 0.4. Compute their posteriors.

Hint: Multiply prior by likelihood for each, then divide by the sum.

Answer: 0.07 and 0.12; evidence 0.19; posteriors ≈ 0.37 for A and ≈ 0.63 for B. The data shifted belief from A to B.

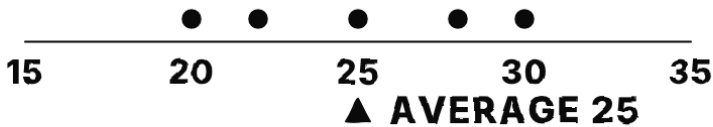
- 3.** With the prior $p(f) \propto 1/\|f\|$, how much more probable a priori is a function of norm 1 than one of norm 5?

Hint: Compare $1/1$ with $1/5$.

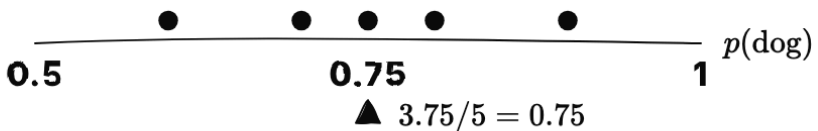
Answer: Five times more probable, since $1/1$ is five times $1/5$; smaller-norm functions are preferred as more stable.

Approximations, Priors as Penalties, and Learning in Instalments

Ask five friends to estimate how long a walk will take: 20, 25, 30, 22 and 28 minutes. Their average, 25, is usually a better guess than any one of them. Bayesian prediction is that idea made precise: ask many plausible models and average them, and this part shows how that is done in practice, and what simpler methods hide inside it.



K MODELS FROM THE POSTERIOR



POSTERIOR: RARELY EXACT MCMC · VARIATIONAL

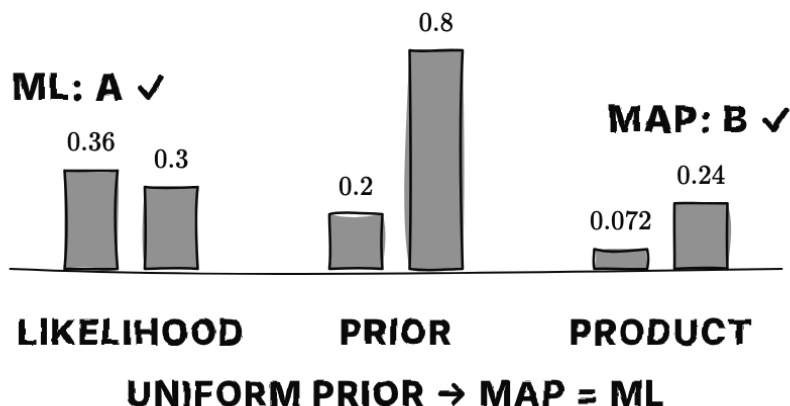
Averaging Many Guesses

The averaged prediction over all models is an integral, and in practice it is replaced by a Monte Carlo average over k models drawn at

random from the posterior, $f_k \sim p(f \mid \mathcal{S}_n)$. If five sampled models give a photo the chance of being a dog of 0.8, 0.7, 0.9, 0.6 and 0.75, the averaged prediction is their sum, 3.75, divided by 5, which is 0.75. The beauty of this setup, the book says, is spoiled by one fact: the posterior can almost never be computed exactly, except for very particular choices of prior and likelihood. One must approximate it, either with Markov chain Monte Carlo methods or with variational inference, and Section 9.3.1 will show one Bayesian treatment of a model's parameters, Monte Carlo dropout.

The book closes the chapter with two remarks. First, suppose we only want the single function with the highest posterior density. Then the evidence can be ignored, and the problem splits into two terms: $f^* = \operatorname{argmax}_f p(\mathcal{S}_n \mid f) p(f) = \operatorname{argmax}_f [\log p(\mathcal{S}_n \mid f) + \log p(f)]$. The first term is the likelihood, the second a regularization term, and the answer is called the maximum a posteriori, or MAP, solution.

$$f^* = \arg \max_f [\log p(\mathcal{S}_n | f) + \log p(f)]$$



Likelihood or Likelihood Times Prior

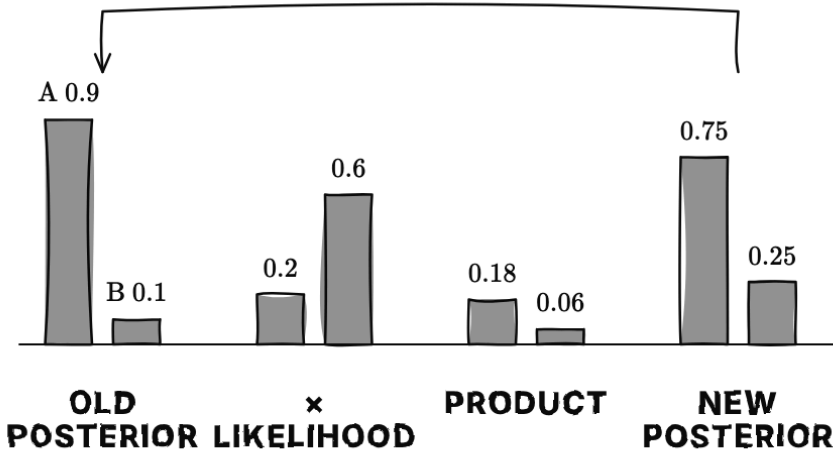
Compare the two with numbers. Model A explains the data with likelihood 0.36 but has prior 0.2; model B has likelihood 0.30 and prior 0.8. Maximum likelihood picks A, since 0.36 beats 0.30. MAP multiplies by the prior: 0.36 times 0.2 is 0.072, while 0.30 times 0.8 is 0.24, so MAP picks B, the model that fits slightly worse but is far more plausible beforehand.

If every function has the same prior weight, a uniform prior, the second term is constant and MAP reduces to maximum likelihood. Otherwise MAP penalizes functions that stray too far from the prior, which helps against overfitting and lets us impose constraints on f . That is why $\log p(f)$ is called a regularizer: it pulls the solution towards the region the prior favours. A footnote links this to the two

interpretations of probability: loosely, maximum likelihood treats the parameters as fixed unknowns and the data as random, the frequentist view, while the Bayesian view treats the data as fixed and the parameters as random.

Second remark: the full Bayesian treatment absorbs new data easily. Given a new dataset $\mathcal{S}'_n$ from the same distribution, we use the posterior computed on the first data as the new prior, and update it with the new likelihood. The book suggests seeing the original prior as the posterior after observing nothing at all.

POSTERIOR → NEW PRIOR: NO FORGETTING



Yesterday's Posterior, Today's Prior

Take models A and B with posteriors 0.9 and 0.1 after the first batch. A new batch has likelihood 0.2 under A and 0.6 under B: multiplying

gives 0.18 and 0.06, whose sum is 0.24, so the new posteriors are 0.75 and 0.25. Belief moved towards B, but the first batch was not forgotten: A still leads. This is why the Bayesian view can ease a well-known problem of models trained online, on data arriving over time: catastrophic forgetting, where learning from new data wipes out what was learned before. A chat assistant updated only on this month's conversations, with nothing carrying over, would be exposed to exactly that risk.

Try it yourself

Hold two coins and decide in advance that you believe 9 to 1 that the first one is the fair one. Flip them a few times and, for each coin, multiply your belief by how likely its results look, then rescale the two numbers so they add up to one. Do it again after more flips, starting from the beliefs you just computed rather than from 9 to 1: that is the posterior becoming the next prior.

Remember

Bayesian predictions average over models drawn from the posterior, a Monte Carlo approximation such as 0.75 from five samples, but the posterior itself usually needs approximation by Markov chain Monte Carlo or variational inference. Keeping only the most probable function gives the MAP solution, the $\arg \max$ of log likelihood plus log prior, where the log prior acts as a regularizer; with a uniform prior it equals maximum likelihood. MAP can prefer a slightly worse-fitting model that is more plausible a priori, as B over A with 0.24 against 0.072. New data are absorbed by using the old posterior as the new prior, which helps against catastrophic forgetting in online training.

Questions

1. When does the MAP solution reduce to the maximum likelihood solution?
 - a. When the prior is uniform, giving every function the same weight
 - b. When the dataset is very small
 - c. When the posterior is computed exactly
2. Why is $\log p(f)$ called a regularizer?
 - a. Because it makes the likelihood larger
 - b. Because it pulls the solution towards functions the prior favours, penalizing those that stray far from it

- c. Because it removes the evidence from Bayes' theorem
- 3. How does the Bayesian treatment absorb a new batch of data?
 - a. By retraining from the original prior on the new batch only
 - b. By averaging the old and new datasets
 - c. By using the posterior from the old data as the prior for the new data
- 4. Why must Bayesian predictions usually be approximated?
 - a. Because the posterior can rarely be computed in closed form
 - b. Because the prior is always unknown
 - c. Because Monte Carlo averages are exact

Answers

- 1. **a)** MAP maximizes log likelihood plus log prior; with a uniform prior the second term is a constant, leaving maximum likelihood.
- 2. **b)** The log prior penalizes functions deviating from the prior, which helps against overfitting and imposes constraints on f .
- 3. **c)** The previous posterior becomes the starting belief, updated with the new likelihood, so old information is kept, which helps against catastrophic forgetting.
- 4. **a)** Except for very specific priors and likelihoods, the posterior has no closed form, so one uses Markov chain Monte Carlo or variational inference, and averages over sampled models.

Worked examples

Maximum likelihood against MAP Model A has likelihood 0.36 and prior 0.2; model B has likelihood 0.30 and prior 0.8. Which does

maximum likelihood choose, and which does MAP choose?

Maximum likelihood compares likelihoods: $0.36 > 0.30$, so it picks A.

MAP compares likelihood times prior: A gives $0.36 \times 0.2 = 0.072$.

B gives $0.30 \times 0.8 = 0.24$.

$0.24 > 0.072$, so MAP picks B.

Answer: Maximum likelihood picks A; MAP picks B

Updating in instalments After a first batch, models A and B have posteriors 0.9 and 0.1. A new batch has likelihood 0.2 under A and 0.6 under B. Compute the updated posteriors.

Use the old posterior as the new prior.

Prior times likelihood: $0.9 \times 0.2 = 0.18$ and $0.1 \times 0.6 = 0.06$.

Evidence: $0.18 + 0.06 = 0.24$.

Posteriors: $0.18 / 0.24 = 0.75$ and $0.06 / 0.24 = 0.25$.

Answer: 0.75 for A and 0.25 for B

Practice

1. Four models sampled from the posterior predict probabilities 0.6, 0.9, 0.7 and 0.8 that an email is spam. What is the Monte Carlo prediction?

Hint: Average the four predictions.

Answer: $(0.6 + 0.9 + 0.7 + 0.8) / 4 = 3.0 / 4 = 0.75$.

2. With a uniform prior over two models with likelihoods 0.4 and 0.1, which model does MAP choose, and does it agree with maximum likelihood?

Hint: A uniform prior multiplies both by the same number.

Answer: Both are multiplied by the same prior, so MAP picks the model with likelihood 0.4, exactly as maximum likelihood does.

3. Explain catastrophic forgetting and how using the posterior as the next prior can help.

Hint: Think of a model trained on data arriving over time.

Answer: When a model trained online learns only from new data, it can lose what it learned before. Using the posterior from earlier data as the prior for the new data carries the old information forward, so the update shifts belief without erasing it.