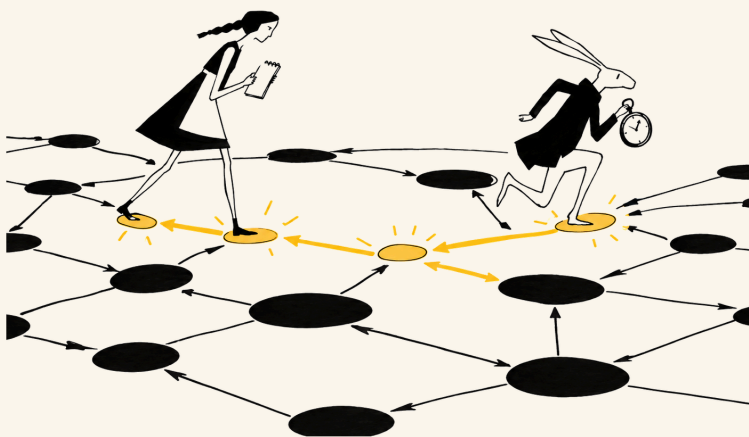


Alice's Adventures in a Differentiable Wonderland



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

A Few Simple Pieces, Made Very Large

Learning as a Search Among Dials

Old Ideas Waiting for Enough Data

Boxes of Numbers and Their Shapes

What This Book Is, and What It Is Not

Credits, a Licence, and Cutting Slices

Arrows, Lengths and the Dot Product

Tables of Numbers That Act

Counting the Work

Stacks of Tables and a Shorthand by Letters

The Slope at a Point

Slopes in Every Direction

Where the Ground Falls Fastest

Walking Downhill, and Where You End Up

A Ball That Remembers

From Paper to Code

What Makes a Good Pile of Examples

Learning Without Labels, and Reusing What Was Learned

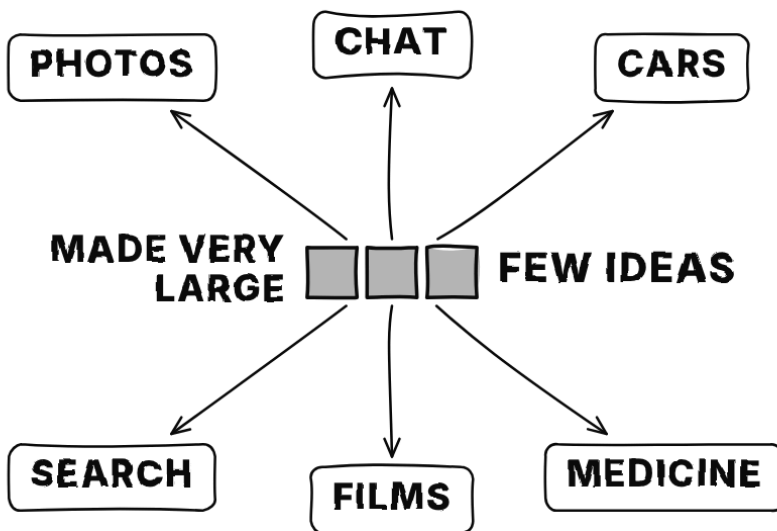
Scoring a Prediction

Memorising Is Not Learning

The Most Likely Explanation, and Many Explanations at Once
A Penalty from the Prior, and the Simplest Block

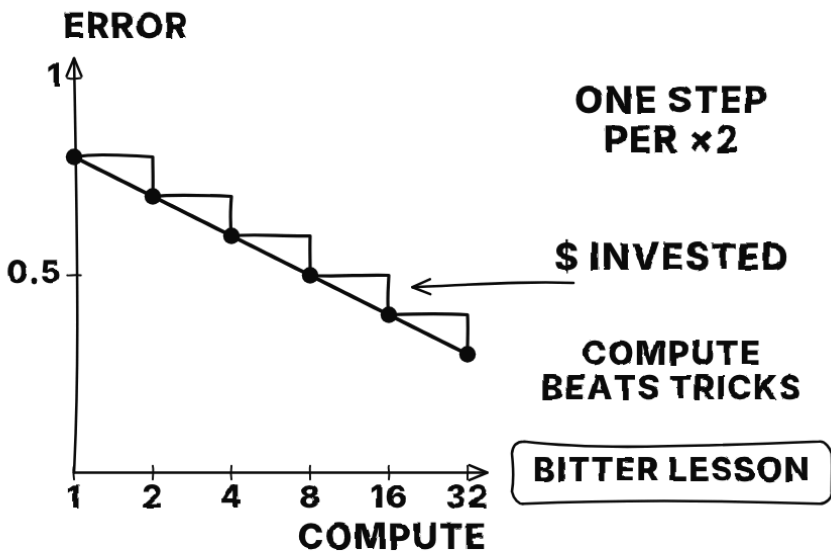
A Few Simple Pieces, Made Very Large

Open the photo app on your phone and type the word "dog" into its search box. Within a second it shows you every dog you have ever photographed, although you never labelled a single picture. Ask a chat assistant to explain a recipe or fix a sentence, and it answers in fluent paragraphs. Both feats are done by neural networks, and so are many things you never see: the steering help in cars, the ranking of search results, the films a streaming service suggests, even the design of new molecules for medicines. The surprising news the book opens with is that all of this rests on a very small set of guiding ideas and building blocks.



Few Blocks, Many Uses

What changed in recent years is not so much the ideas as their size. Researchers noticed a pattern called neural scaling laws: for almost any task, if you increase three things at once, the amount of data, the computing power and the size of the model, the accuracy improves, and it improves predictably. Said the other way round, the computing power needed to reach a given accuracy keeps falling by a steady factor as the years go by. Picture a staircase you can see in advance: each time you double the budget, you know roughly how many steps up you will climb. That predictability is what persuaded companies to invest enormous sums in artificial intelligence. The researcher Richard Sutton gave the lesson a name, the bitter lesson: simple, general tools combined with ever more computing power have beaten clever hand-made tricks again and again.

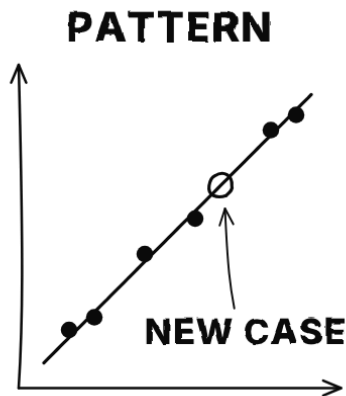


The Scaling-Law Staircase

Take the scaling laws as given, and you are left with an almost magical tool. In one sentence, a neural network is tuned so that it imitates the pattern hidden in a large collection of examples; in the book's language, it approximates a probability distribution from data drawn from it. This could easily go wrong. Modern networks are so large that they could simply memorise every example they are shown, like a pupil who learns the answers to last year's exam by heart and is lost when the questions change. A network that did that would be nothing but a look-up table. Yet trained networks do something better: they generalise, handling cases that were never in their training data. And as the collections of examples grow huge, the line between familiar and unfamiliar cases blurs, and the largest models learn surprisingly little on rote memory.

MEMORISE	
INPUT	ANSWER
2	4
3	6
5	10
4	?

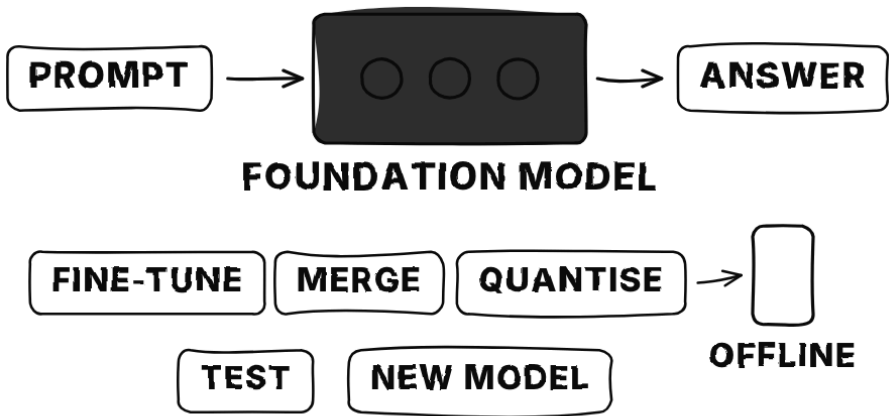
LOOK-UP TABLE



Memorise or Generalise

These very large models, trained once and then used for many different jobs, are sometimes called foundation models. Together with a lively open-source community, they have changed how people use networks. Many tasks no longer need any programming at all: you simply prompt a model, giving it instructions in text or pictures, much as you type a request to a chat assistant. The model itself stays a black box, a closed machine whose insides you never look at. The book compares this to ordinary software: instead of writing every library yourself in a language such as C++, you rely on programs written by others, often without seeing their code. The comparison is apt, because only a few teams in the world have the computing power and the skill to build truly large models, such as the Llama family of

language models, just as only a few companies can build the large business software that firms run on.



A Black Box You Can Adapt

But the comparison also points the other way. Open-source software lets anyone adapt a program or write a new one from scratch, and in the same way an ordinary computer and some ingenuity open many doors with these models. You can fine-tune a model, training it a little more on your own examples; merge two models into one; quantise a model, storing its numbers more coarsely so that it runs on a small, low-power device; test how robust it is against tricky inputs; or invent new kinds of models altogether. The voice assistant that answers you offline on your phone is often exactly such a shrunk, quantised model. For all of this you have to look under the hood: to understand

how these models take in and transform data, along with the tricks and quirks that practitioners learned through experience and debugging. The book presents itself as the entry point to that world, and its author hopes that a reader who is naturally curious, like Alice, will enjoy the journey.

Try it yourself

Open the photo app on your phone and search for an object you have never labelled, such as "bicycle", "beach" or "birthday cake". Notice that it finds most of them, and also notice the occasional odd mistake, a lamp taken for a cake or a dog taken for a teddy bear. Those mistakes are a glimpse of a model that learned a pattern from examples rather than a list of rules, which is exactly the kind of machine this book takes apart.

Remember

Neural networks now power chat assistants, phone photo apps, cars, search, recommendations and science, yet they are built from a very small set of principles. Scaling laws say that more data, more computing power and bigger models together give a predictable gain in accuracy, which drove huge investment. A network is tuned to imitate the pattern behind its examples; it could memorise them, but large trained networks generalise to cases they never saw. Foundation models are often used as black boxes through prompts, but adapting, shrinking or inventing models requires understanding what happens inside them. That understanding of the inner workings is what the book sets out to give.

Questions

1. What do neural scaling laws say happens when data, computing power and model size are all increased together?
 - a. Accuracy rises, and by an amount that can be predicted in advance
 - b. Accuracy rises at first and then always falls as models memorise
 - c. Accuracy stays the same, but the model runs faster
 - d. Accuracy rises only for language tasks, not for pictures

2. Why would a network that memorised all its training examples be of little use?
 - a. It would need far more computing power to answer
 - b. It would be a look-up table, lost on any case it had not seen
 - c. It could no longer be prompted with text
 - d. It would forget the examples after a while
3. In the book's comparison with software, what does prompting a foundation model correspond to?
 - a. Writing every library yourself in C++
 - b. Building your own enterprise business software
 - c. Using someone else's program without seeing its source code
 - d. Reading the source code of an open-source library
4. Which of these jobs requires looking under the hood of a model rather than only prompting it?
 - a. Asking a chat assistant to fix a sentence
 - b. Searching your photos for the word dog
 - c. Typing a question into a search engine
 - d. Quantising a model so that it runs on a small device

Answers

1. **a)** The scaling laws describe a predictable gain in accuracy for almost any task when the three grow together; that predictability is what drove the investment.
2. **b)** Memorising turns the network into a look-up table, like a pupil who knows last year's answers by heart; what makes trained networks useful is that they generalise to new cases.

- 3. c)** Prompting uses the model as a black box, just as one relies on programs written by others without access to their code.
- 4. d)** Fine-tuning, merging, quantising or testing a model all need an understanding of how it processes data internally, which is what the book offers.

Practice

- 1.** A friend says that a model trained on millions of photos must simply be remembering them. Using what this part said, how would you answer?

Hint: Think about what a model that only remembered would do with a photo it had never seen.

Answer: A model that only remembered would be a look-up table and would fail on new photos, yet trained networks recognise objects in pictures they were never shown. They generalise, and the largest models rely surprisingly little on pure memorisation.

- 2.** Explain in your own words why scaling laws persuaded companies to spend so much on artificial intelligence.

Hint: What does it mean for a gain to be predictable before you pay for it?

Answer: Scaling laws say that more data, computing power and model size give a predictable rise in accuracy. A company can therefore estimate in advance what a bigger budget will buy, which makes a large investment look like a safe bet rather than a gamble.

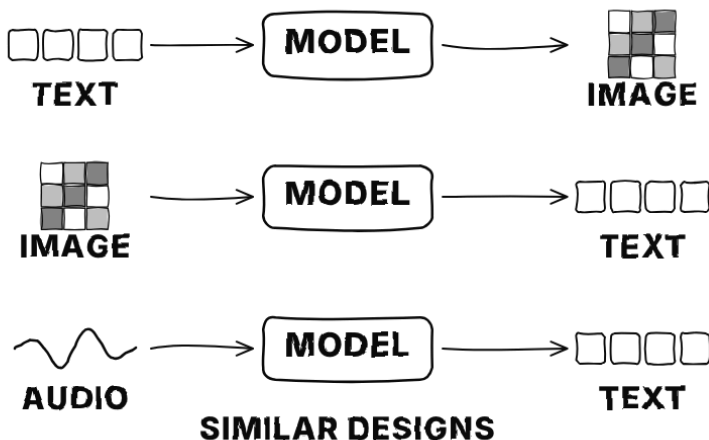
- 3.** Name two things you could do with an existing model on an ordinary computer, and say why each needs an understanding of the model's insides.

Hint: Look at the list of options the part gave after the open-source comparison.

Answer: For example, fine-tuning it on your own examples, which means changing its internal numbers sensibly, and quantising it to run on a small device, which means knowing which numbers can be stored more coarsely without breaking it. Both act on the inside of the model, not just on its answers.

Learning as a Search Among Dials

Imagine you want a program that tells photos of cats from photos of dogs. You could try writing rules: pointed ears, whiskers, a certain shape of nose. Within an hour you would be stuck, because there is always a cat that breaks the rules and a dog that fits them. The book assumes you already know the other way, the way that works: collect thousands of photos, each labelled "cat" or "dog", and let a system adjust itself until its answers match the labels. This is supervised learning, the core of what is called machine learning: gather examples of the behaviour you want, then train, which simply means optimise, a system until it imitates that behaviour.

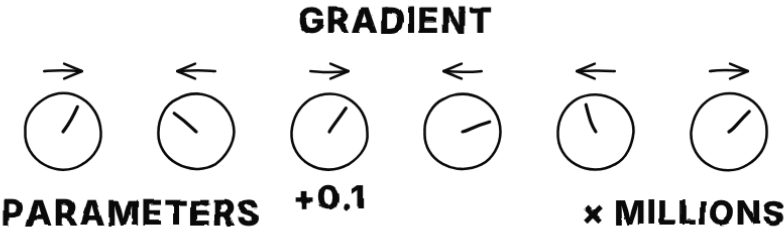


What Goes In, What Comes Out

The idea sounds almost too simple, yet it reaches remarkably far. Generating pictures from a description becomes the job of collecting an enormous set of images, each with its caption. Imitating the English language, the job behind a chat assistant, becomes the job of gathering a vast amount of text and learning to predict each sentence from the ones before it. Reading an X-ray becomes the job of keeping a large archive of scans together with the decision a doctor made about each. The book's figure sorts such tasks by what goes in and what comes out. Generating an image takes a text, a row of characters, and returns an image, an ordered grid of coloured dots called pixels. Captioning does the reverse, from image to text, and answering a spoken question goes from audio, another ordered row, this time of

numbers, to text. Fascinatingly, the book notes, the models for all these jobs are designed along very similar lines.

At heart, the book says, learning is a search. You start with a program that has a great many adjustable numbers, like a mixing desk with millions of dials, and these dials are called parameters. Then you turn the dials until the program performs well enough. The difficulty is the number of dials: millions in a small model, billions or even trillions in the largest ones, far too many to try settings at random. The photo app on your phone was tuned in exactly this way, its dials turned over and over on huge piles of labelled pictures.



ERROR **11.5**

$$\text{rate} = \frac{0.5}{0.1} = 5$$

AUTODIFF GRADIENT DESCENT

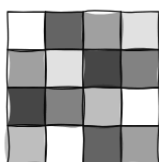
Turning Dials Downhill

Differentiable models make the search possible by allowing only one kind of building block: mathematical functions that can be differentiated, which means that for each one we can work out how its output changes when its input changes a little. For a function with many inputs, the collection of these rates of change is called the gradient. Knowing the gradient means knowing, for every dial at once, what happens to the program's performance if that dial is nudged slightly. Here is a small picture of it. Suppose the program's error is twelve, and you learn that nudging one particular dial up by a tenth would lower the error by half a unit: the rate is five units of error per unit of dial, so you know to turn it up, and roughly by how much. With such a rate for every dial, turning the dials becomes an automatic routine, and the two routines the book builds on are called automatic differentiation, which computes the rates, and gradient descent, which uses them. The first part of the book, called Compass and Needle, chapters two to six, sets up exactly this machinery.

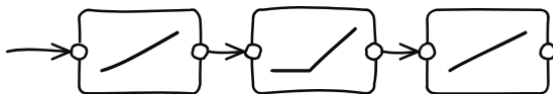
A B C x
JUMPS

 
SOUND IMAGE


TEXT



TENSOR



CONVOLUTION

IN / OUT SHAPES

Blocks That Fit Like Plugs

Seeing a network as a chain of differentiable building blocks raises two questions, and the book's second figure illustrates them. First, what kinds of data can go in and come out? Second, what kinds of blocks can we use? Being differentiable is a strong requirement. Letters and whole numbers jump from one value to the next with nothing in between, and a quantity that jumps cannot be nudged a little, so such data cannot be handled directly. What differentiable models handle easily is data written as large arrays of numbers, which the book calls tensors, such as an image, a grid of brightness values. These arrays are transformed by chaining simple steps, some straight-line and some bent, which the book calls linear and nonlinear transformations. In the figure, every block is described by what it takes in and what it gives out, and those descriptions decide which

blocks can be joined to which, much like the shapes of plugs and sockets.

The second part of the book is devoted to one typical block, the convolution, in chapters seven to nine. A convolution fits any data that comes as an ordered row of elements: sound, images, text and video. Along the way the book introduces ways to build deep models, deep meaning made of many steps in a row, and several ideas that underpin today's large language models: splitting text into pieces called tokens, generating a sequence one piece at a time, each from the ones before, and keeping the model from peeking at what comes later. The third part, called Down the Rabbit Hole, turns to other kinds of data: sets, handled by attention layers and the transformer models of chapters ten and eleven; graphs, networks of connected points, in chapter twelve; and recurrent layers for sequences in time, in chapter thirteen.

Outside the printed pages, the book is accompanied by a website where the author plans to collect further chapters on topics not tied to one kind of data: generating new data, computation that switches parts of a model on and off, transfer of what was learned from one task to another, and explaining what a model has learned. These chapters lean more towards current research and can be read in any order. There are also guided lab sessions in notebook form, covering much of the book as well as advanced topics such as contrastive learning and model merging, the combining of two trained models into one.

Remember

The book assumes supervised learning: collect examples of the wanted behaviour and train a system until it imitates them, an idea that covers image generation, chat assistants and X-ray diagnosis alike. Tasks can be sorted by what goes in and what comes out, and the models for them are designed in strikingly similar ways. Learning is a search over a huge number of adjustable numbers, the parameters, and it becomes practical by building models only from differentiable blocks. The gradient tells, for every parameter at once, how the performance changes when it is nudged, which allows automatic differentiation and gradient descent. The book's three parts cover this machinery, then convolutions, then attention, transformers, graphs and recurrent layers, with extra chapters and lab notebooks online.

Questions

1. Why does the book describe learning as a search?
 - a. Because the model looks up each answer in a database of examples
 - b. Because training turns a huge number of adjustable numbers until the program performs well
 - c. Because the programmer searches the internet for ready-made rules
 - d. Because the model tries every possible setting of its dials one by one

2. Why can a differentiable model not work directly with letters or whole numbers?
 - a. Because they take up too much memory
 - b. Because they cannot be put into a tensor at all
 - c. Because they jump from one value to the next, so they cannot be nudged a little
 - d. Because computers store them as text rather than numbers
3. An error of twelve would drop by half a unit if one dial were turned up by a tenth. What does that tell you?
 - a. The rate is five units of error per unit of dial, so turning it up helps
 - b. The rate is half a unit, so the dial does not matter
 - c. The error will reach zero after twenty-four nudges
 - d. The dial should be turned down to lower the error
4. Which kind of data does a convolution suit, according to the book?
 - a. Data with no order at all, such as a set
 - b. Data arranged as a network of connected points
 - c. Only photographs
 - d. Data that comes as an ordered row of elements, such as sound, images, text and video

Answers

1. **b)** Learning starts from a program with very many parameters and adjusts them until the performance is good enough; with millions of dials, trying settings at random is hopeless, which is why gradients are needed.

- 2. c)** A differentiable block needs to know how its output changes for a small change of input, and a quantity that jumps has no small changes.
- 3. a)** Half a unit of change for a tenth of a nudge is a rate of five, and since the error falls, the dial should be turned up; the gradient gives such a rate for every dial at once.
- 4. d)** Convolutions apply whenever data can be written as an ordered sequence of elements; sets and graphs are left to attention layers and graph models in the third part.

Worked examples

Reading a rate of change A program's error is twelve. Nudging one dial up by one tenth would lower the error by half a unit. What is the rate of change of the error with respect to that dial, and which way should the dial be turned?

The change in the dial is one tenth, and the change in the error is a drop of half a unit.

The rate is the change of error divided by the change of the dial: half divided by one tenth is five.

Because the error falls when the dial goes up, the rate says the error drops five units per unit of dial in that direction.

So the dial should be turned up, and a small turn of one tenth is worth about half a unit of error.

Answer: A rate of five units of error per unit of dial; turn the dial up.

Practice

1. Using the book's input and output framing, describe the task 'turning a photo into a caption' and the task 'answering a spoken question'.

Hint: For each, say what kind of data goes in and what comes out.

Answer: Captioning takes an image, an ordered grid of pixels, and returns a text, a row of characters. Answering a spoken question takes audio, an ordered row of numbers, and returns a text. The book notes that models for such tasks are designed along very similar lines.

2. Explain in your own words why a chat assistant can be built by supervised learning.

Hint: What examples of the wanted behaviour can be collected for language?

Answer: Imitating language can be turned into predicting each sentence from the ones before it, and huge amounts of text supply endless examples of exactly that. Training the system to make those predictions well is supervised learning on text.

3. A model has a billion parameters. Why is it hopeless to improve it by trying random settings, and what does the gradient offer instead?

Hint: Think about how many combinations a billion dials have, and what one gradient tells you.

Answer: The number of possible settings is astronomically large, so random tries would almost never land on a good one. The gradient tells, for every parameter at once, which way and how strongly a small nudge changes the performance, so the dials can be turned automatically in a useful direction.

Old Ideas Waiting for Enough Data

Picture a newspaper page from 1958. A psychologist, Frank Rosenblatt, is in the headlines for a machine called the perceptron, one of the first working prototypes of a neural network, and reporters are already promising machines that will walk, talk and think. The book shows a picture of that moment with a wry caption: artificial intelligence hype, except that it is 1958. If that sounds familiar, it is because the same kind of headline now greets every new chat assistant. The scene is a reminder that the ideas behind today's networks are much older than the excitement around them.

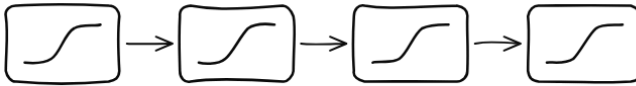
The very name neural network is a leftover from those early attempts to imitate biological neurons, the nerve cells of the brain, in the twentieth century. The vocabulary has stuck: to stay consistent with the software people actually use, the book too will speak of neurons, layers and activations. Interest in such networks came in several waves. Between 2012 and 2017 networks grew in complexity as never before, driven by large public contests, above all the ImageNet challenge, in which programs competed to recognise the objects in over a million photos; the book returns to it in chapter nine. A second great wave began in 2017 with the transformer, the subject of chapter ten. Just as convolutional models had taken over computer vision a few years earlier, transformers took over the processing of human language within a very short time. Progress followed for video, for graphs and for sound, leading to today's excitement about large language models, models that mix several kinds of data, and models that generate new pictures, text and sound.

The words changed along with the waves. In the 1980s people spoke of connectionism; later, deep learning came to mean modern networks, as opposed to the smaller and shallower models of the past. Yet all these terms stay vague, because modern artificial networks hardly resemble biological neurons at all. What truly defines them, the book argues, is that they are built from differentiable blocks, pieces whose output changes smoothly when their input is nudged. That is why the book prefers the name differentiable models wherever it can. Seen this way, neural networks belong to a wider field called differentiable programming, a young discipline that mixes computer science and optimisation to study computer programs that can be differentiated. The book credits a short manifesto by Yann LeCun, posted in 2018, and a second one by Colin Raffel in 2021, on building models the way open-source software is built, as sources of this view.

CONNECTIONISM → **DEEP LEARNING**
1980S



↓
DIFFERENTIABLE MODELS



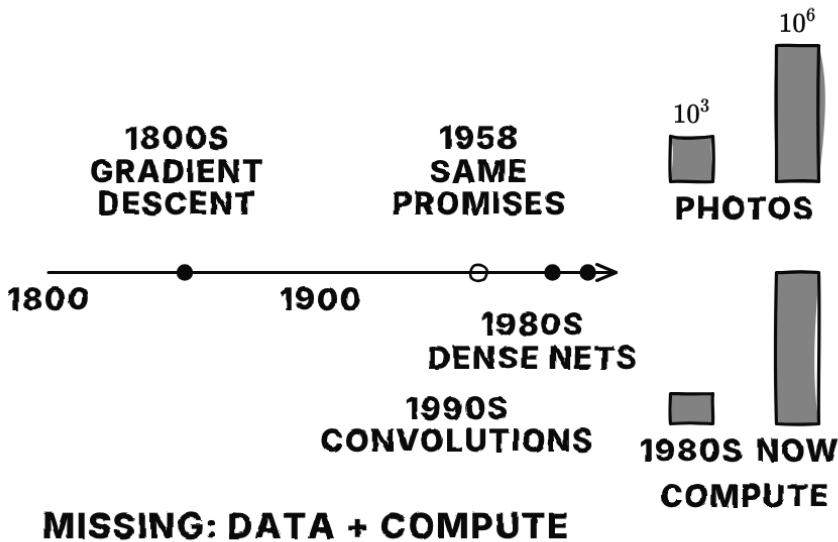
DIFFERENTIABLE BLOCKS

DIFFERENTIABLE PROGRAMMING

Names Change, Blocks Stay

Travelling through the land of differentiable models is also a trip through history, and the dates are worth hearing. Fitting a simple straight-line model by gradient descent, the topic of chapter four, was known since at least the nineteenth century. Fully-connected networks of the kind the book uses later date back to the 1980s. Convolutional models were already in use at the end of the 1990s. So the main pieces existed decades ago, and what was missing was something else: enough data and enough computing power to show how well these models perform once they have plenty of both, and plenty of parameters. Here is a way to feel the gap: a phone in your pocket today does more arithmetic in a second than the computers of the 1980s could do in many hours, and the photo collections used to

train its apps hold millions of pictures where early researchers had a few thousand.



Old Pieces, New Fuel

Try it yourself

Search the web for the 1958 newspaper reports on Rosenblatt's perceptron, and then search for the headlines about a recent chat assistant. Put the two side by side and compare the promises: machines that learn, that understand, that will soon think. You will see how closely the language of excitement repeats itself, while what actually changed underneath was mostly the amount of data and computing power.

The book closes this short history with a modest promise. Research moves so fast that no book can cover every topic in depth, but the hope is to give the reader enough grounding to find their way through the newest research papers on their own.

Remember

Neural networks are old: the name comes from twentieth-century attempts to imitate brain cells, and in 1958 the perceptron already caused a wave of hype. Interest came in waves, with a leap in network size between 2012 and 2017 driven by contests such as ImageNet, and a second leap with transformers from 2017, leading to today's language and generative models. Modern networks barely resemble the brain; their real defining trait is that they are built from differentiable blocks, so the book calls them differentiable models, part of differentiable programming. Gradient descent goes back to the nineteenth century, fully-connected networks to the 1980s and convolutional models to the 1990s; what was missing for decades was enough data and computing power.

Questions

1. Why does the book prefer the name differentiable models to neural networks?
 - a. Because modern networks hardly resemble brain cells, and their real defining trait is being built from differentiable

blocks

- b. Because the word neural is protected by a trademark
 - c. Because differentiable models are a newer invention than neural networks
 - d. Because neural networks cannot be used for language
2. Fully-connected networks date from the 1980s and convolutional models from the 1990s. Why did their success come so much later?
- a. Because the ideas had to be reinvented from scratch
 - b. Because enough data and computing power to show their strength arrived only much later
 - c. Because programmers refused to use them until 2017
 - d. Because they only work on photographs
3. What happened to language processing after transformers appeared in 2017?
- a. It slowly abandoned neural networks
 - b. It switched back to hand-written rules
 - c. Transformers took it over within a very short time, as convolutional models had done for vision
 - d. It became limited to translation only
4. What does the book's 1958 picture of the perceptron suggest about today's excitement over chat assistants?
- a. That the perceptron was secretly a chat assistant
 - b. That nothing has improved since 1958
 - c. That hype is a new phenomenon of the internet age

- d. That waves of hype around neural networks are not new; the ideas are older than the excitement

Answers

1. **a)** The old names stay vague, since artificial networks barely resemble biological neurons; what all modern networks share is that they are built from differentiable blocks.
2. **b)** The main pieces existed decades ago; what was missing was enough data, computing power and parameters to reveal how well they perform.
3. **c)** The book draws the parallel: convolutional models overtook computer vision, and a few years later transformers overtook natural language processing very quickly.
4. **d)** The caption calls it AI hype, except in 1958, a reminder that such waves have happened before and that the ideas predate today's attention.

Practice

1. Put these in the order in which the book says they appeared: transformers, gradient descent for linear models, convolutional models, fully-connected networks.

Hint: One goes back to the nineteenth century; one is from 2017.

Answer: Gradient descent for linear models (nineteenth century), fully-connected networks (1980s), convolutional models (end of the 1990s), transformers (2017).

2. In your own words, what changed between the 1990s and the 2010s that made old network designs suddenly succeed?

Hint: The book names two ingredients, plus the size of the models.

Answer: The amount of available data and the computing power grew enormously, which made it possible to train models with many

parameters and show how well the old designs perform when given enough of everything.

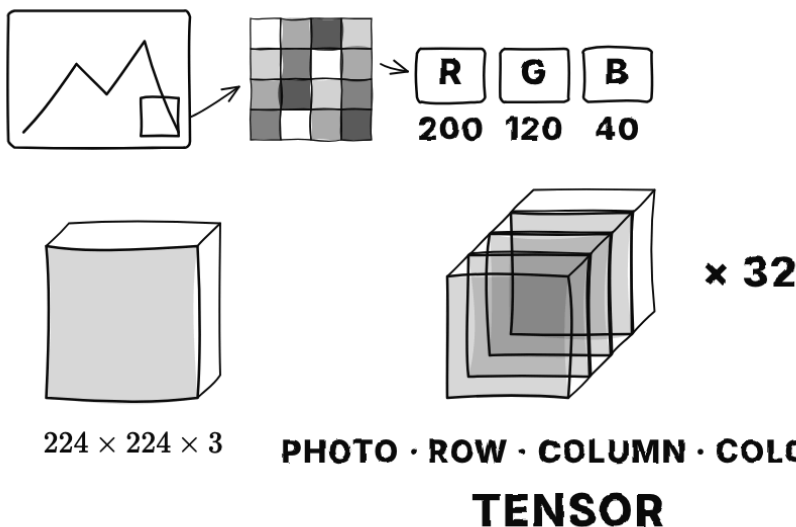
3. Explain what differentiable programming is, using the idea from the previous part that a gradient tells how a quantity responds to small nudges.

Hint: What must every block of such a program allow?

Answer: It is the study of computer programs built so that every step can be differentiated, meaning one can compute how the program's output responds to small nudges of its inputs and parameters. Neural networks are one example, which is why the book treats them as differentiable models.

Boxes of Numbers and Their Shapes

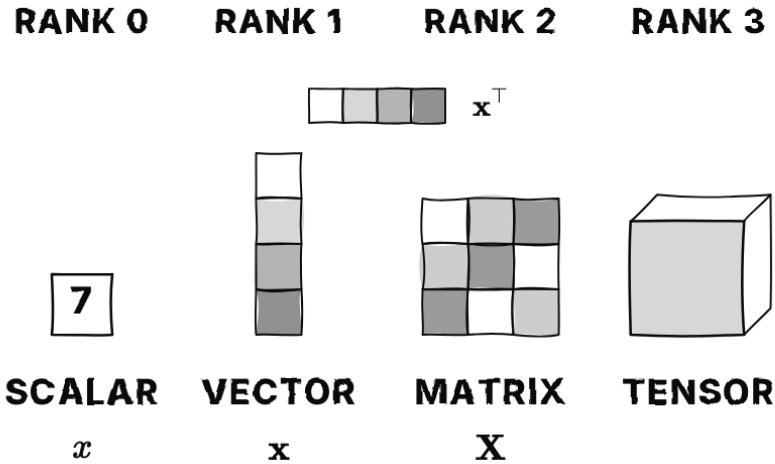
Take one photo from your phone and ask what it really is to a computer. Zoom in far enough and it breaks into tiny squares, each with three numbers: how much red, how much green and how much blue. A photo of 224 squares by 224 squares is therefore a block of numbers 224 high, 224 wide and 3 deep. When the photo app learns from your pictures, it takes a pile of such blocks at once, say thirty-two of them, and that pile is a box of numbers with four directions: which photo, which row, which column, which colour. The book calls every such box a tensor, and it is the basic thing that differentiable models handle.



A Photo Is a Box of Numbers

A tensor, in this book, is simply an array of numbers laid out in some number of directions, and that number of directions is called the rank of the tensor. The book admits that mathematicians use the word tensor for something more refined, a kind of operator, and apologises for borrowing it, but the borrowed meaning is so widespread that it keeps it. A single number, rank zero, is called a scalar, and the book writes it with an ordinary small letter. A column of numbers, rank one, is a vector, written with a small bold letter; when the same numbers are laid out as a row, the book marks it with a small raised T, called the transpose, though it drops the mark when the meaning is clear. The book's figure draws the vector as a row only for convenience; in the text every vector is a column. A rectangle of numbers, rank two, is a matrix, written with a capital bold letter. For

ranks above two there is no special lettering, and curly, calligraphic capitals are kept for sets and for probability distributions.



SHAPE MATTERS

Rank: Zero, One, Two, Three

The book makes one point that will matter throughout: understanding an operation usually comes down to understanding the shape of every tensor involved. So it uses a short way of stating a shape: the name of the tensor, a wavy sign, and the list of its sizes in brackets. The pile of photos from before would be written as a tensor whose shape is the batch size, the height, the width, and three. Some sizes are fixed numbers, like the three colours, and others are left as letters, like the batch size, because they can change from one use to the next. Let us count what such a shape means: thirty-two photos, each 224 by 224 squares with 3 colours, is 224 times 224, which is 50,176 squares,

times 3, which is 150,528 numbers per photo, times 32, which is 4,816,896 numbers in the whole pile. That is almost five million numbers for one small batch, which is why shapes deserve attention.

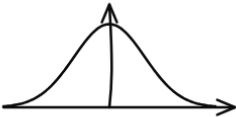
$$X \sim (b, 224, 224, 3)$$

CAN CHANGE

FIXED

$$\begin{aligned} 224 \times 224 &= 50,176 \\ \times 3 &= 150,528 \\ \times 32 &= 4,816,896 \\ &\approx \mathbf{5 \text{ MILLION}} \end{aligned}$$

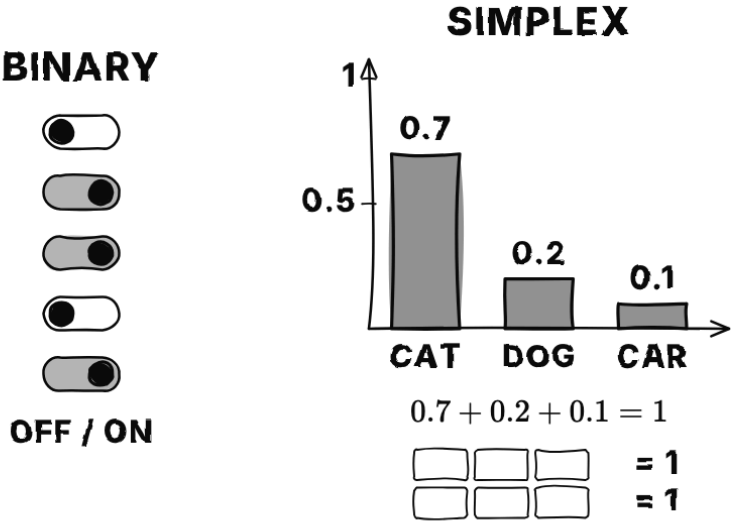
$$x \sim \mathcal{N}(0, 1)$$



Writing and Counting a Shape

The same wavy sign is also used, rarely, to say that a number is drawn at random from a probability distribution, for example a number drawn from the familiar bell curve centred at zero. The context always tells which meaning is meant. The book uses the shape notation in place of the more usual mathematical way of saying that a vector is a list of a certain number of real numbers; it is shorter and keeps the eye on the shape.

Two more labels say what values a tensor may hold. One marks a tensor whose entries are only zeros and ones, called binary, like a row of switches that are either off or on. The other marks a vector that lies in the simplex: its entries are never negative and they add up to exactly one. That is precisely what a chat assistant produces when it guesses the next word: chances such as seven tenths for "cat", two tenths for "dog" and one tenth for "car", which add up to one. For a tensor of higher rank, such as a table with one row per example, the book applies the rule along the last direction, so each row on its own adds up to one.



Binary and Simplex

Finally, the book places three small pictures in its margins to guide the reader.

Remember

A tensor here means an array of numbers in some number of directions, the rank: a scalar has rank zero, a vector rank one, a matrix rank two. Scalars are small letters, vectors small bold letters (columns, with a raised T for the row form), matrices capital bold letters; curly capitals are kept for sets and distributions. Understanding an operation mostly means knowing the shape of each tensor, written with a wavy sign and a list of sizes, some fixed and some left as letters; a batch of thirty-two colour photos of 224 by 224 holds almost five million numbers. Extra labels mark tensors of only zeros and ones, and vectors in the simplex, whose entries are not negative and add up to one.

Questions

1. A batch of photos is stored with four directions: which photo, which row, which column, which colour. What is its rank?
 - a. Three, because colour does not count
 - b. Four, one for each direction
 - c. One, because it is a single batch
 - d. Two, because a photo is flat
2. Which of these lists of numbers lies in the simplex?
 - a. Seven tenths, two tenths and one tenth
 - b. Half, half and half

- c. One, zero and minus one... no, minus one is not allowed:
zero, one and one
 - d. Six tenths, three tenths and two tenths
3. Why does the book leave some sizes in a shape as letters, such as the batch size, while writing others as numbers?
- a. Because letters take less memory
 - b. Because the sizes written as letters are always larger
 - c. Because those sizes can change from one use to the next, while others, like three colours, are fixed
 - d. Because letters mark sizes that are not yet known to anyone
4. In a table with one row per example and the simplex label, what must add up to one?
- a. All the numbers in the whole table together
 - b. Each column on its own
 - c. The first row only
 - d. Each row on its own, along the last direction

Answers

- 1. b)** The rank is the number of directions of the array; photo, row, column and colour make four.
- 2. a)** A vector is in the simplex when no entry is negative and the entries add up to exactly one; seven tenths, two tenths and one tenth add up to one, like a chat assistant's next-word chances.
- 3. c)** Fixed sizes such as the three colours are written as numbers; sizes that vary between uses, such as how many photos are in a batch, are left as letters.

4. **d)** For tensors of higher rank the book applies the rule along the last direction, so each row separately must add up to one.

Worked examples

Counting the numbers in a shape A batch holds 8 colour photos, each 64 squares high and 64 wide, with 3 colour values per square. How many numbers does the tensor hold, and what is its rank?

The directions are photo, row, column and colour, so the rank is four.

One photo has 64 times 64 squares, which is 4,096 squares.

Each square has 3 colour values, so one photo holds 4,096 times 3, which is 12,288 numbers.

Eight photos hold 12,288 times 8, which is 98,304 numbers.

Answer: 98,304 numbers in a tensor of rank four.

Practice

1. Give the rank of each: the temperature outside right now; the daily temperatures of one week; a table of daily temperatures for each week of a year.

Hint: Count how many directions you need to find one number.

Answer: One number is rank zero, a scalar. Seven daily values in a column are rank one, a vector. A table with one row per week and one column per day is rank two, a matrix.

2. A model outputs the chances five tenths, three tenths and three tenths for three possible words. Is this vector in the simplex? Explain.

Hint: Check the two conditions: no negative entries, and the total.

Answer: No. None of the entries is negative, but they add up to eleven tenths, not one, so the vector is not in the simplex.

3. A single photo of 100 by 200 squares in colour is stored as a tensor. How many numbers does it hold?

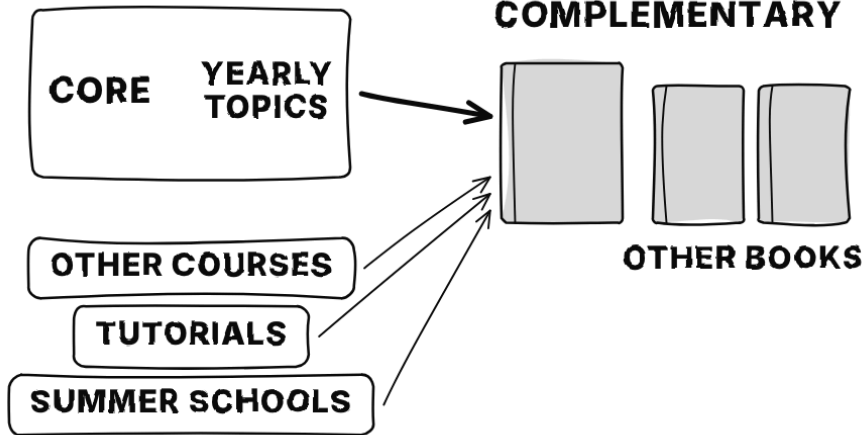
Hint: Multiply the sizes of all directions, including the colours.

Answer: 100 times 200 is 20,000 squares, and with 3 colour values each that is 60,000 numbers, in a tensor of rank three.

What This Book Is, and What It Is Not

Before a long walk, a good guide tells you three things: where the path came from, what you will see, and what you will not. The book does exactly that before it begins. It grew out of a course the author has taught for many years in the master's programme in data science at Sapienza University of Rome, called Neural Networks for Data Science Applications. The core chapters are the heart of that course; the other chapters are topics that come and go from year to year. Pieces were added from other teaching too: a course on neural networks for computer engineers, an introduction to machine learning for telecommunications engineers, tutorials, doctoral courses and summer schools.

SAPIENZA COURSE

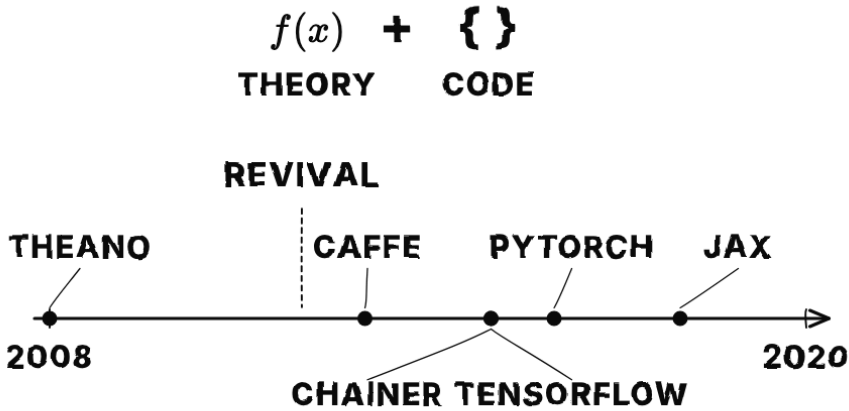


Where the Book Came From

There are already several excellent recent books on modern deep networks, and the book says so openly. At the start it covers much the same ground as they do; later, the way things are explained and some extra sections depart from them, following the author's own research interests. The aim is to offer an additional, complementary point of view rather than to replace the others.

The book's choice of the name differentiable programs carries a message: understanding them takes both theory and code. How a model is designed and how it is programmed constantly shape each other, and automatic differentiation, the automatic computing of rates of change met earlier, is the best example. In fact, the revival of neural networks from about 2012 owes a great deal to software:

libraries such as Theano, Caffe and Chainer, and then today's TensorFlow, PyTorch and JAX, which made it practical for anyone to build and train networks. The photo app on your phone and the chat assistant you use were almost certainly built with tools from this family. Wherever it can, the book connects its ideas to these tools, especially PyTorch and JAX, but it is not a programming manual, and for learning the libraries themselves it points to their own documentation.



Software Brought Networks Back

Then comes an honest list of what the book is not. It does not try to be complete. It picks concepts that are common today and general enough to stay useful for a while, but nobody can foresee the future, and some parts may be incomplete or out of date by the time you read

them. For each idea it shows a few of the variants found in research papers, for example two different ways of normalising the numbers inside a network, but hundreds more exist, and the book invites you to browse a large online catalogue of papers and their code for the rest.

Finally, the book is about the basic components of differentiable models, not about running them at large scale. Making a big model actually work needs engineering skill and some intuition, and the book says little about hardware. For the intuition, it says, nothing beats experience, plus opinionated blog posts by practitioners; it recommends a well-known recipe for training networks by Andrej Karpathy, and Karpathy's video series that builds networks from zero. Think of it like learning to cook: the book teaches what each ingredient does, while the knack of cooking for a hundred guests comes from practice in a busy kitchen.

Remember

The book grew out of a long-running master's course at Sapienza University of Rome, extended with material from other courses, tutorials and summer schools. It overlaps with other good books at first, then follows its own path, aiming to be a complementary point of view. Understanding differentiable programs means joining theory and code; the comeback of neural networks since 2012 owes much to libraries such as TensorFlow, PyTorch and JAX, which the book links to without being a programming manual. It is not complete, it may age, it shows only a few of the many variants of each idea, and it leaves large-scale engineering, hardware and practical intuition to experience and practitioners' writing.

Questions

1. Why does the book say that understanding differentiable programs takes both theory and code?
 - a. Because model design and its programming constantly shape each other, as automatic differentiation shows
 - b. Because every chapter is a programming exercise
 - c. Because the theory alone is too short to fill a book
 - d. Because the libraries cannot be used without reading the book

2. What does the part credit, in large part, for the revival of neural networks from about 2012?
 - a. A new theory of the brain
 - b. The end of the ImageNet challenge
 - c. Powerful software libraries such as TensorFlow, PyTorch and JAX
 - d. A ban on hand-written rules
3. Which of these does the book say it does not try to do?
 - a. Connect its ideas to PyTorch and JAX
 - b. Explain the basic components of differentiable models
 - c. Offer a point of view complementary to other books
 - d. Cover every variant of every idea and the engineering of very large models
4. Where does the book send the reader for practical intuition about training networks?
 - a. To its own lab notebooks only
 - b. To experience and to opinionated blog posts by practitioners, such as Karpathy's recipe
 - c. To the documentation of TensorFlow
 - d. To the history of the perceptron

Answers

1. a) How models are designed and how they are implemented influence each other all the time, and automatic differentiation is the clearest example.

- 2. c)** The revival owes a great deal to software libraries, from Theano, Caffe and Chainer to today's TensorFlow, PyTorch and JAX.
- 3. d)** It shows only a few variants of each idea, says it may be incomplete or outdated, and leaves large-scale engineering, hardware and intuition to experience.
- 4. b)** For intuition, nothing beats experience, plus practitioners' blog posts; the book recommends Andrej Karpathy's training recipe and video series.

Practice

- 1.** In your own words, explain the difference between knowing what the components of a model do and being able to train a very large model that works.

Hint: Think of the cooking comparison in this part.

Answer: Knowing the components is like knowing what each ingredient does; the book teaches that. Training a large model that works is like cooking for a hundred guests: it needs engineering skill, knowledge of hardware and intuition that come mainly from practice.

- 2.** Why might parts of the book be out of date by the time you read it, and what does it do to limit this?

Hint: Look at how it picks its concepts.

Answer: Research moves fast and nobody can foresee which methods will last. The book limits the risk by choosing concepts that are both common today and general enough to stay useful for a while.

- 3.** The book is not a programming manual, yet it often mentions PyTorch and JAX. Why do both statements make sense together?

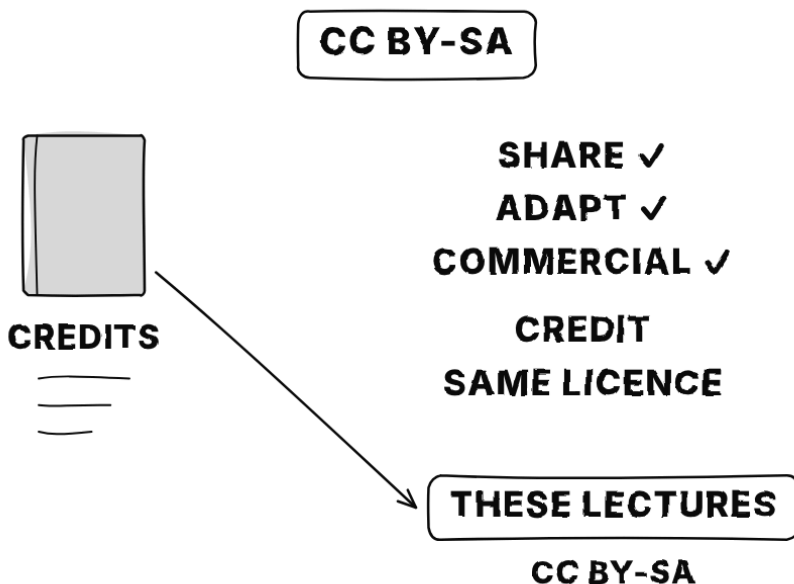
Hint: What role do the libraries play in the explanations?

Answer: The book uses the libraries to connect ideas with how they are actually implemented, since theory and code shape each other. It does not

teach the libraries from scratch; for that it points to their own documentation.

Credits, a Licence, and Cutting Slices

Every book has a page few people read, and this one has something worth hearing on it. The coloured highlighting of its equations comes from a software package written by S. T. John. The colour pictures of Alice and the small black-and-white symbols in the margins are licensed from a stock-image company, Shutterstock, while the Alice pictures inside the figures reproduce John Tenniel's original illustrations, obtained through Wikimedia. The author 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 many corrections, as well as everyone who sent feedback by e-mail.



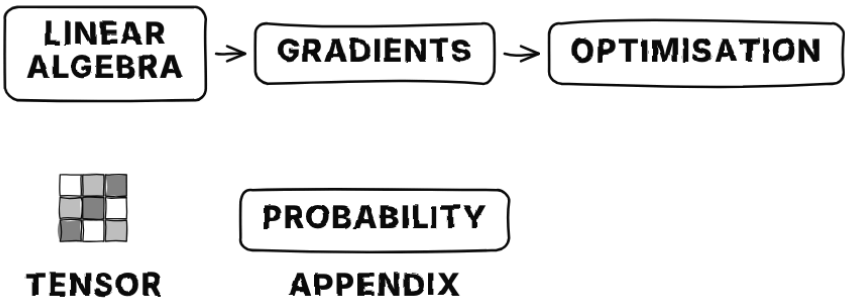
A Licence That Allows Adapting

Then comes the licence, and it matters directly to you. The book is released under a Creative Commons licence called Attribution-ShareAlike. In its own words, it lets anyone "distribute, remix, adapt, and build upon the material in any medium or format", provided the creator is credited; commercial use is allowed; and whatever is built on it must be released under the same terms. The lectures you are listening to are exactly such an adaptation, which is why they too carry that licence.

With that, the book opens chapter two, on the mathematical background. It is a compressed review rather than a first course: it assumes you have met these topics, and it mainly fixes the notation and shows how the pieces fit, with an eye on how they are

implemented in practice. The chapter has three stages, each building on the one before: linear algebra, then gradients for objects with many dimensions, then how to optimise a function using those gradients. Probability theory, with a focus on a principle called maximum likelihood, is reviewed separately in an appendix. The book warns, cheerfully, that the chapter is full of definitions and asks for a little patience.

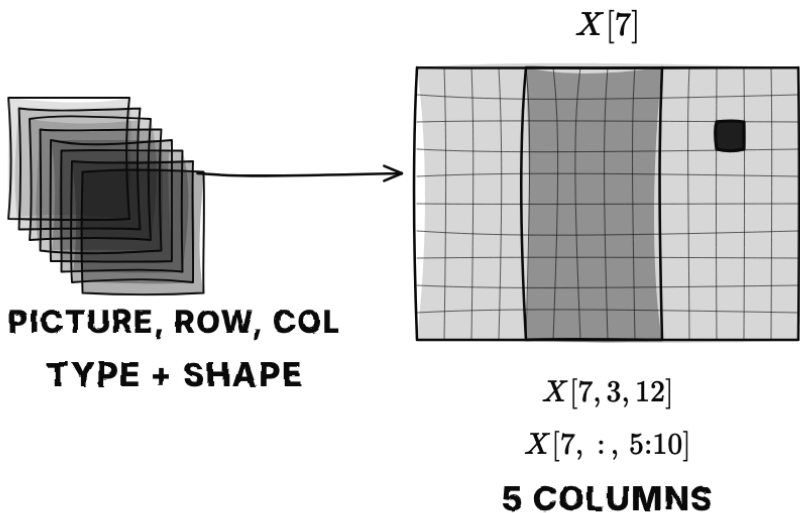
MANY DEFINITIONS



Chapter Two in Three Stages

The first stage, linear algebra, starts again from the tensor, the box of numbers met earlier, now defined formally: an array in some number of directions whose elements all have the same type. Its shape is written with the wavy sign and the list of sizes, one size per direction. Tensors are central in deep learning for a practical reason: the same

simple operation applied to millions of numbers at once is exactly what graphics chips do well, which is why your phone's photo app and the servers behind a chat assistant run on such chips, called GPUs, or on even more specialised ones. A tensor is described by two things, the type of its elements and its shape. Most of the time the elements are numbers with a decimal point, stored in a format the book discusses later, but tensors can also hold whole numbers, for example class labels, or strings of text.



Cutting Slices from a Tensor

Often you need only part of a tensor, a slice, and the book follows the conventions of the popular NumPy library, with one difference: for readability it counts positions from one, not from zero. Picture a tensor holding a batch of grey-scale pictures: which picture, which

row, which column. Fixing only the first position, say picture number seven, gives a slice that is one whole picture, a rectangle of rows and columns. Fixing all three positions, picture seven, row three, column twelve, gives a single number, the brightness of one dot. For more complex slices the book separates the positions with commas and uses a colon to mean everything along a direction, and a range from one position up to, but not including, another. Let us take one: picture seven, all rows, and columns from five up to but not including ten. That keeps every row and ten minus five, that is five, columns, so the slice is a strip of all the rows by five columns. When the indexing would clutter an equation, the book draws square brackets in light grey around an expression, with the positions written small at the lower right, so the eye can separate what is indexed from how.

Remember

The equation colouring, the stock and Tenniel images, and the many readers who sent corrections are credited, and the book is released under Creative Commons Attribution-ShareAlike, which allows adaptation and commercial use with credit and the same licence, as these lectures do. Chapter two is a compact review of linear algebra, gradients and optimisation, with probability in an appendix. A tensor is an array in some number of directions with elements of one type; it is described by that type and its shape, and it suits the massively parallel chips used for deep learning. Slices follow NumPy's conventions but count from one: fixing some positions gives a smaller tensor, fixing all gives one number, and a range such as five up to ten keeps five positions.

Questions

1. Under the book's licence, which of these is allowed?
 - a. Selling an adapted version, if the author is credited and the adaptation keeps the same licence
 - b. Selling an adapted version under a stricter licence of your own
 - c. Adapting the book without crediting the author
 - d. Nothing beyond reading it privately
2. A tensor holds grey-scale pictures by picture, row and column. What do you get by fixing only the picture number?
 - a. A single brightness value
 - b. One whole picture, a rectangle of rows and columns
 - c. One column across all pictures
 - d. A tensor with the same three directions
3. Why are tensors so central in deep learning, according to the part?
 - a. Because they are the only way to store text
 - b. Because they make networks resemble the brain
 - c. Because the same operation on millions of numbers at once suits GPUs and similar chips
 - d. Because they never use decimal numbers
4. A slice takes picture seven, all rows, and columns from five up to but not including ten. How many columns does it keep?
 - a. Six

- b. Ten
- c. Four
- d. Five

Answers

1. **a)** Attribution-ShareAlike allows distribution, remixing and commercial use, provided the creator is credited and the adapted material carries the same licence.
2. **b)** Fixing one position removes that direction and leaves a slice of the other two: one complete picture.
3. **c)** Tensors suit massively parallel hardware, such as graphics chips, which apply one operation to very many numbers at the same time.
4. **d)** A range up to, but not including, the end keeps ten minus five, that is five, positions.

Worked examples

The size of a slice A tensor holds 50 grey-scale pictures, each 28 rows by 28 columns. You take picture three, rows from 1 up to but not including 15, and all columns. How many numbers are in the slice, and what shape does it have?

Fixing picture three removes the first direction, so the slice has rows and columns left.

Rows from 1 up to but not including 15 keep 15 minus 1, that is 14 rows.

All columns keeps all 28 columns.

The slice is 14 rows by 28 columns, which is 14 times 28, that is 392 numbers.

Answer: A rectangle of 14 rows by 28 columns, 392 numbers.

Practice

1. In the same tensor of 50 pictures of 28 by 28, how many numbers do you get by fixing the picture, the row and the column?

Hint: What is left when every direction is fixed?

Answer: One number: the brightness of a single dot. Fixing every position leaves no direction, so the result is a scalar.

2. You take all 50 pictures, row 10 only, and columns from 3 up to but not including 13. What is the shape of the slice and how many numbers does it hold?

Hint: Fixing the row removes that direction; count the columns as end minus start.

Answer: The row direction disappears, leaving 50 pictures by 13 minus 3, that is 10 columns. The slice is 50 by 10, which holds 500 numbers.

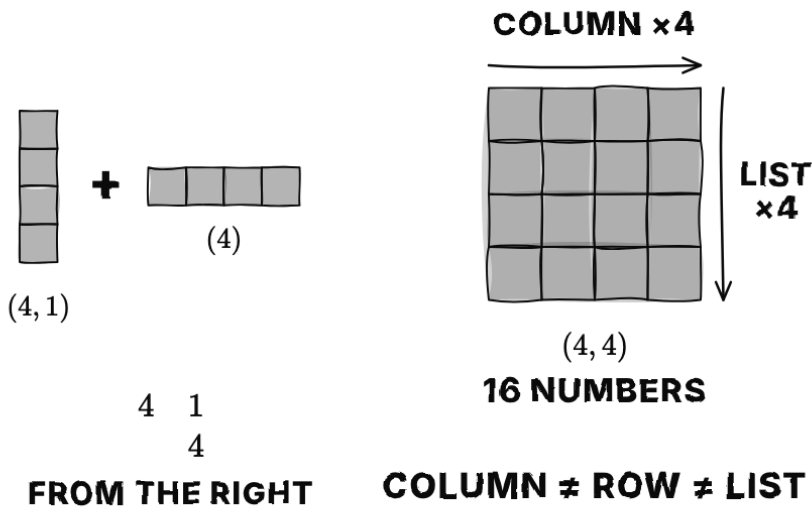
3. Someone adapts this book into a paid video course. According to the licence, what two things must they do?

Hint: Look at the two conditions attached to the permission.

Answer: They must credit the creator of the book, and they must release their adapted material under the same Attribution-ShareAlike terms. Commercial use itself is allowed.

Arrows, Lengths and the Dot Product

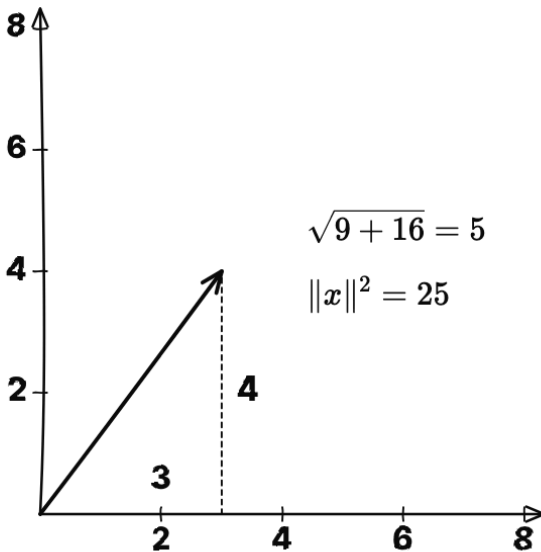
Here is a small trap that catches programmers every day. Take four numbers stacked as a column, and four numbers lying as a plain list, and ask a library such as PyTorch to add them. You expect four sums, one for each position. What you get is a table of four rows by four columns, sixteen numbers, without any error message. The book shows exactly this in a short code box, and the culprit has a name: broadcasting.



Broadcasting: Four Plus Four Gives Sixteen

Broadcasting is a rule, borrowed from the NumPy library, by which a library lines up the shapes of two tensors from the right and quietly repeats one of them wherever that makes the shapes match. A column of four is, for the library, a two-direction tensor of four rows by one column; a plain list of four has one direction only. Lined up from the right, the single column is stretched across four columns and the list is copied down four rows, and every pair gets added. Mathematics books carefully separate column vectors from row vectors, and the book follows that habit, but in code a column, a row and a plain list are three different shapes, and mixing them is how such surprises happen. The book's advice is simply to keep this in mind, because almost every framework behaves the same way.

Most of the models in the book are built from very plain ingredients: sums, multiplications, and a few bent functions such as the exponential, sines and cosines, and square roots. Vectors, the tensors with one direction, have their own arithmetic, which mathematicians call a vector space. Any two vectors of the same shape can be combined by scaling each and adding them, position by position, to give a third vector. For instance, take the vector two, one and add twice the vector one, three: position one gives two plus two, which is four, and position two gives one plus six, which is seven, so the result is four, seven.



Vectors as Arrows

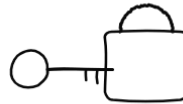
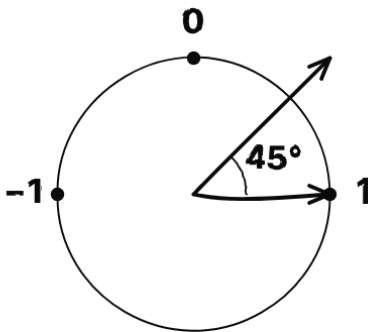
It helps to see a vector as an arrow from the origin to a point. Adding two arrows then means placing them head to tail, which completes a parallelogram, and the length of an arrow, its distance from the origin, is called its norm: square each entry, add the squares, and take the square root. For the arrow three, four, the squares are nine and sixteen, they add up to twenty-five, and the square root is five. The squared norm, simply the sum of the squared entries, twenty-five here, turns out to be especially handy.

The most important operation on vectors is the inner product, also called the dot product: multiply the two vectors position by position and add up the results. The answer is always a single number. The book's own example multiplies the vector one tenth, zero, minus three

tenths by the vector minus four, five hundredths, one tenth. The three products are minus four tenths, zero, and minus three hundredths, and their sum is minus forty-three hundredths.

$$(0.1)(-4) + (0)(0.05) + (-0.3)(0.1) \\ = -0.4 + 0 - 0.03 = -0.43$$

$$a \cdot b = \|a\| \|b\| \cos \theta$$



TEMPLATE MATCHING

$$\cos \theta = \frac{1}{1.41} \approx 0.71$$

Dot Product and the Angle

The dot product has a beautiful geometric meaning: it equals the length of the first arrow, times the length of the second, times the cosine of the angle between them. So if both arrows have length one, the dot product is just the cosine of their angle, and it is then called the cosine similarity. It is one when the arrows point the same way, minus one when they point in opposite directions, and zero when they stand at right angles, which is called orthogonal. Take the arrows one, zero and one, one: their dot product is one, their lengths are one and

about one point four one, so the cosine is about seven tenths, which is the cosine of forty-five degrees, exactly the angle between them.

Turn this around. If one arrow of length one is fixed, which other arrow of length one gives the largest dot product? The answer is the arrow itself: the cosine is largest, one, when the angle is zero. This is the reason the dot product is everywhere in the book. Later the first vector will be an input and the second a set of adjustable parameters, and their dot product is largest when the input resonates with the parameters, matching them like a key in a lock. The book calls this template matching. A common way search features find related items, such as photos similar to one you pick, is to turn each item into a vector and compare them with exactly this cosine similarity.

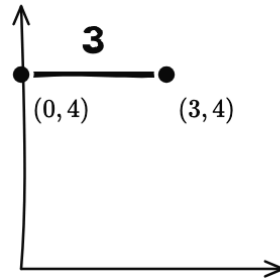
Two last observations will save effort later. The sum of all entries of a vector is its dot product with a vector made entirely of ones: for two, five, three, multiplying by one, one, one and adding gives ten. And the squared distance between two vectors can be written with dot products alone: the dot product of the first with itself, plus that of the second with itself, minus twice the dot product of the two. Check it with the arrows three, four and zero, four: twenty-five plus sixteen minus twice sixteen gives nine, and indeed the two points are three apart, and three squared is nine. When the second vector is all zeros, the rule says that the squared length of a vector is its dot product with itself.

$$(2, 5, 3) \cdot (1, 1, 1) = 10$$

$$\|a - b\|^2 = a \cdot a + b \cdot b - 2a \cdot b$$

$$25 + 16 - 2 \cdot 16 = 9 = 3^2$$

$$b = 0, \quad \|a\|^2 = a \cdot a$$



Distance from Dot Products

Try it yourself

On a sheet of squared paper, draw the arrow from the origin to the point three, four and measure it with a ruler: it is five squares long. Then draw the arrows one, zero and one, one and measure the angle between them with a protractor: forty-five degrees. Finally multiply three, four by itself, position by position, and add: twenty-five, which is five squared, the squared length you measured.

Remember

Libraries broadcast: they line up shapes from the right and repeat tensors to match, so adding a column of four to a plain list of four silently gives a four by four table. Vectors can be scaled and added position by position; seen as arrows, adding completes a parallelogram and the norm is the length, the square root of the sum of squares. The dot product multiplies two vectors position by position and adds, giving one number; it equals the two lengths times the cosine of the angle, so for unit vectors it is the cosine similarity, from one to minus one, zero meaning orthogonal. For a fixed unit vector the dot product is largest with the vector itself, the template matching that inputs and parameters will rely on. The sum of entries is the dot product with a vector of ones, and squared distances can be written with dot products alone.

Questions

1. Why does adding a column of four numbers to a plain list of four numbers give sixteen numbers in a library such as PyTorch?
 - a. Because the library lines the shapes up from the right and repeats each tensor to match, producing a four by four table
 - b. Because the library multiplies instead of adding
 - c. Because a column always holds sixteen numbers

- d. Because the library reports an error and fills the gap with zeros
2. Two arrows of length one have a dot product of zero. What does this tell you?
- a. They point the same way
 - b. They point in opposite directions
 - c. They are orthogonal, standing at right angles
 - d. One of them is all zeros
3. With one unit arrow fixed, which unit arrow gives the largest dot product with it?
- a. Any arrow at right angles to it
 - b. The arrow pointing the opposite way
 - c. The arrow of all ones
 - d. The arrow itself
4. How can the sum of all the entries of a vector be written as a dot product?
- a. As its dot product with itself
 - b. As its dot product with a vector made entirely of ones
 - c. As its dot product with a vector of zeros
 - d. As the square root of its dot product with itself

Answers

1. a) Broadcasting aligns shapes from the right and silently repeats tensors; the column of four rows by one column and the list of four combine into four rows by four columns.

2. c) For unit vectors the dot product is the cosine of the angle, and a cosine of zero means a right angle.

3. d) The cosine is largest, one, when the angle is zero, so the best match is the same arrow; this is the template matching that inputs and parameters will use.

4. b) Multiplying each entry by one and adding just adds the entries: for two, five, three this gives ten.

Worked examples

The book's dot product Find the dot product of the vector one tenth, zero, minus three tenths with the vector minus four, five hundredths, one tenth.

Multiply the first entries: one tenth times minus four is minus four tenths.

Multiply the second entries: zero times five hundredths is zero.

Multiply the third entries: minus three tenths times one tenth is minus three hundredths.

Add the three products: minus four tenths plus zero minus three hundredths is minus forty-three hundredths.

Answer: Minus forty-three hundredths, a single number.

Practice

1. What is the length, the norm, of the arrow six, eight?

Hint: Square each entry, add the squares, then take the square root.

Answer: Ten: six squared is thirty-six, eight squared is sixty-four, they add up to one hundred, and the square root of one hundred is ten.

2. Use the dot-product rule for distances to find how far apart the points one, two and four, six are.

Hint: Compute each vector's dot product with itself and the dot product of the two, then combine: first plus second minus twice the mixed one.

Answer: The first with itself gives one plus four, five; the second with itself gives sixteen plus thirty-six, fifty-two; the mixed product is four plus twelve, sixteen. Five plus fifty-two minus thirty-two is twenty-five, the squared distance, so the points are five apart.

3. What is the cosine similarity of the arrows one, zero and zero, one, and what angle does it mean?

Hint: Both arrows already have length one.

Answer: Their dot product is one times zero plus zero times one, which is zero. Since both have length one, the cosine similarity is zero: they are orthogonal, at ninety degrees.

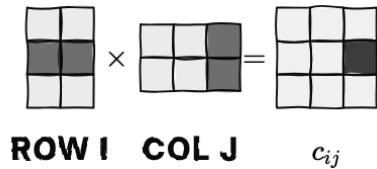
Tables of Numbers That Act

Write down a small table: three flats for sale, and for each one two numbers, its floor area and its number of rooms. You have three rows and two columns, a rectangle of numbers, which mathematicians call a matrix. Read it row by row and something useful appears: each row is one flat, a little vector of its own, and the table is just those three vectors stacked on top of each other. The book calls such a stack a batch of data vectors, and from now on models will be written, both on paper and in code, to work on whole batches at once. When the photo app on your phone learns, it never looks at one picture at a time; it takes a batch, a stack of pictures, and processes the whole stack in one go.

$$(a \times b) (b \times c) = (a \times c)$$

80	3
55	2
120	4

A BATCH OF 3 VECTORS



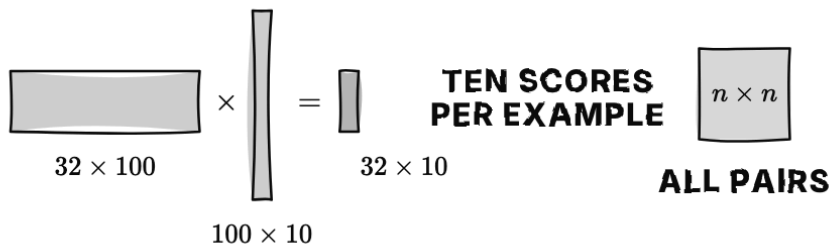
$$c_{ij} = \text{row}_i \cdot \text{col}_j$$

A Table Is a Stack of Vectors

The key operation on matrices is multiplication, and it has a rule about shapes. You can multiply a table of a rows and b columns by a table of b rows and c columns, and the result has a rows and c columns: the two inner sizes must agree, and they disappear. Each entry of the result is a dot product: the entry in row i and column j is the dot product of row i of the first table with column j of the second. Let us do one by hand. Take the first table with rows one, two and three, four, and the second with rows five, six and seven, eight. The top-left entry is row one, two times column five, seven: five plus fourteen, nineteen. The top-right entry is one, two times six, eight: six plus sixteen, twenty-two. The bottom row gives three times five plus four times seven, forty-three, and three times six plus four times

eight, fifty. So the product is nineteen, twenty-two on top and forty-three, fifty below.

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



Multiplying by Hand, Then in Bulk

When the second factor is a single vector, the product is a matrix times a vector, which gives a vector. Now the batch view pays off. If each row of a table is one data vector and you multiply the table by the turned-over table of some weights, you compute the dot product of every data vector with the weights, all in one single operation. A batch of thirty-two examples with a hundred numbers each, multiplied by weights shaped a hundred by ten, gives thirty-two by ten: ten scores for each of the thirty-two examples, at once. And a table multiplied by its own turned-over version, which gives a square table of n rows and

n columns, holds the dot product of every pair of rows, so every pair of examples is compared in one go.

A second kind of product is much simpler: the Hadamard product, which multiplies two tables of the same shape position by position. It lacks the rich properties of true matrix multiplication, but networks use it all the time to mask, that is to set some entries to zero, or to scale entries. Multiply the table one, two over three, four position by position by the table one, zero over zero, one, and the off-diagonal entries vanish: one, zero over zero, four. Some recent families of models rely heavily on such multiplicative interactions, and the book will come back to them.

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

MASK

ENTRY BY ENTRY

$$e^0 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

MATRIX EXP

$$\exp(0) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

TORCH.LINALG

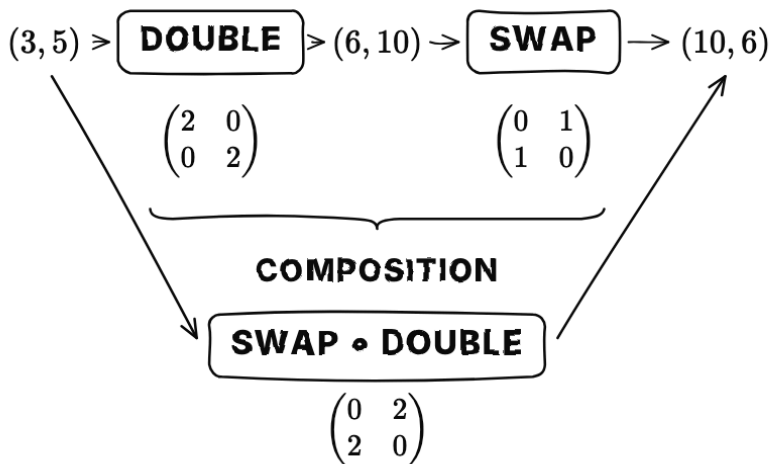


SUM EACH ROW → PER FLAT
SUM EACH COLUMN → TOTALS

The book also warns about a trap of notation. Writing the exponential of a matrix usually means applying the exponential to each entry separately, and that works for tensors of any shape. But linear algebra also has a true matrix exponential, defined only for square matrices as an endless sum of the matrix multiplied by itself more and more times, each term divided by a growing factorial. The two are very different. For a table full of zeros, the entry-by-entry exponential gives a table full of ones, while the true matrix exponential gives ones only on the diagonal and zeros elsewhere. That is why libraries such as PyTorch keep the genuinely matrix-specific operations, such as the matrix exponential, inverses and determinants, in a separate linear-algebra module, as the book's code box shows.

Finally, a reduction such as a sum or an average runs along one whole direction of a table, unless limits are given. In code, this is a method of the tensor with an argument naming the direction: summing the flats table along the columns gives, for each flat, the area plus the rooms; summing along the rows gives the totals over all three flats.

Why is matrix multiplication defined in this strange row-by-column way, and not position by position? The book answers in a box. A function on vectors is linear when scaling and adding the inputs first gives the same result as scaling and adding the outputs. Every such function can be written as a matrix, and applying the function to a vector is exactly multiplying the matrix by the vector. Multiplying two matrices then corresponds to doing one function after the other, which is called composition. For instance, if one matrix doubles every entry and another swaps the two entries, their product is a single matrix that doubles and then swaps: from three, five you get ten, six, whichever way you compute it.



Matrices Compose Functions

Remember

A matrix is a table of rows and columns, best seen as a stack of data vectors, a batch, and models are written to work on whole batches at once. Matrices multiply when the inner sizes agree; each entry of the result is the dot product of a row of the first with a column of the second, so one product computes many dot products at once. The Hadamard product multiplies position by position and is used for masking and scaling; an exponential written on a matrix means entry by entry, unlike the true matrix exponential of square matrices. Reductions such as sums run along a whole direction, chosen in code by an axis argument. Matrix multiplication is defined this way because matrices are linear functions: multiplying by a vector applies the function, and multiplying two matrices composes two functions.

Questions

1. A table of 32 rows and 100 columns is multiplied by a table of 100 rows and 10 columns. What is the shape of the result?
 - a. 100 rows by 100 columns
 - b. 32 rows by 10 columns
 - c. 10 rows by 32 columns
 - d. The tables cannot be multiplied
2. What does each entry of a matrix product hold?

- a. The dot product of a row of the first table with a column of the second
 - b. The product of the two entries in the same position
 - c. The sum of all entries of both tables
 - d. The largest entry of the row
3. For a table full of zeros, how do the entry-by-entry exponential and the true matrix exponential differ?
- a. Both give a table of zeros
 - b. Both give a table of ones
 - c. The entry-by-entry one gives ones only on the diagonal
 - d. The entry-by-entry one gives all ones; the matrix exponential gives ones on the diagonal and zeros elsewhere
4. Why is matrix multiplication defined row by column rather than position by position?
- a. Because it is faster on graphics chips
 - b. Because position-by-position products do not exist
 - c. Because matrices stand for linear functions, and this product corresponds to applying one function after another
 - d. Because it always gives a square result

Answers

1. **b)** The inner sizes, 100 and 100, agree and disappear; the result keeps the outer sizes, 32 by 10, ten scores for each of thirty-two examples.
2. **a)** The entry in row i and column j is the dot product of row i of the first table with column j of the second.

3. d) Each entry's exponential of zero is one, while the true matrix exponential of the zero matrix leaves only the diagonal ones; the two operations are very different.

4. c) Every linear function is a matrix; multiplying by a vector applies it, and multiplying two matrices composes the two functions.

Worked examples

Multiplying two small tables Multiply the table with rows two, zero and one, three by the table with rows one, four and two, one.

Top-left: row two, zero with column one, two gives two times one plus zero times two, which is two.

Top-right: row two, zero with column four, one gives eight plus zero, which is eight.

Bottom-left: row one, three with column one, two gives one plus six, which is seven.

Bottom-right: row one, three with column four, one gives four plus three, which is seven.

Answer: The product has rows two, eight and seven, seven.

Practice

1. Can a table of 3 rows by 2 columns be multiplied by another table of 3 rows by 2 columns? What about by its own turned-over version?

Hint: Check whether the inner sizes agree.

Answer: No: the inner sizes would be 2 and 3, which differ. Yes with its turned-over version, which is 2 by 3: the inner sizes are both 2, and the result is 3 by 3, the dot products of every pair of rows.

2. Multiply the table five, six over seven, eight position by position, the Hadamard product, by the mask zero, one over one, zero.

Hint: Each entry is multiplied only by the entry in the same position.

Answer: Zero, six over seven, zero: the mask keeps the off-diagonal entries and sets the diagonal ones to zero.

3. One matrix doubles both entries of a vector; another swaps its two entries. Apply 'double, then swap' to the vector three, five, and check that 'swap, then double' gives the same result here.

Hint: Do each step in turn.

Answer: Doubling gives six, ten, and swapping gives ten, six. Swapping first gives five, three, and doubling gives ten, six. Both compositions give ten, six, as their single combined matrix would.

Counting the Work

Multiply two tables by hand, one of two rows and three columns and one of three rows and four columns. The result has two rows and four columns, eight entries, and each entry is a dot product of three pairs, so you make eight times three, twenty-four multiplications. Now double every size: four rows by six, times six by eight. The result has thirty-two entries of six multiplications each, one hundred and ninety-two in all, eight times as many as before, although each size only doubled. How the work grows as the sizes grow is what computer scientists call complexity, and the book uses matrix multiplication to introduce it.

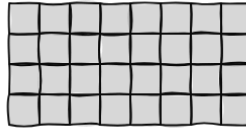
$$(2 \times 3)(3 \times 4)$$



$$8 \times 3 = 24$$

× 8

$$(4 \times 6)(6 \times 8)$$



$$32 \times 6 = 192$$

$$(a \times b)(b \times c)$$

A·C DOT PRODUCTS, LENGTH B

$$\text{time} \sim a b c$$

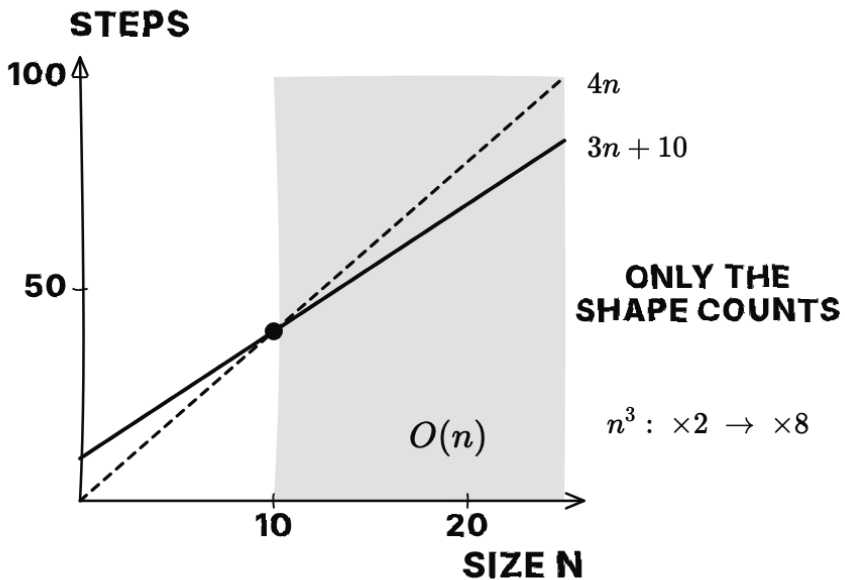
$$\text{space} \sim a c$$

Double the Sizes, Eight Times the Work

Applying the definition directly, a table of a rows and b columns times one of b rows and c columns needs a times c dot products, each of length b . The number of operations is called the time complexity. The memory the computation needs, for a straightforward one-step-at-a-time program, is just the size of the result table, and that is called the space complexity.

To talk about growth without getting lost in the details of particular machines, computer science uses the big-O notation, the O coming from the German word *Ordnung*, order, as in order of approximation. Saying that the work of a program is of order g means this: from some size onwards, the work never exceeds some fixed multiple of g . For example, a program that needs three times the size plus ten steps is of

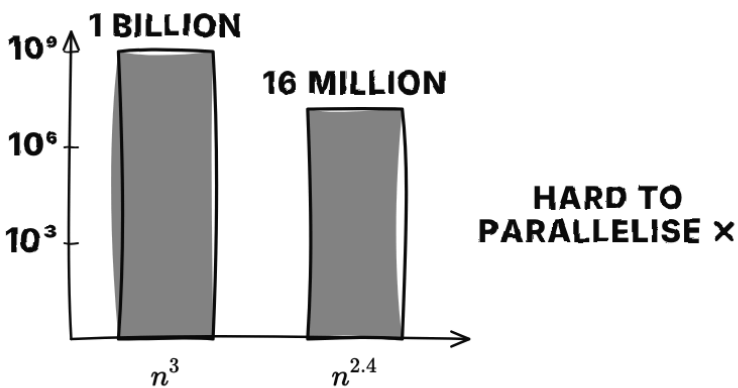
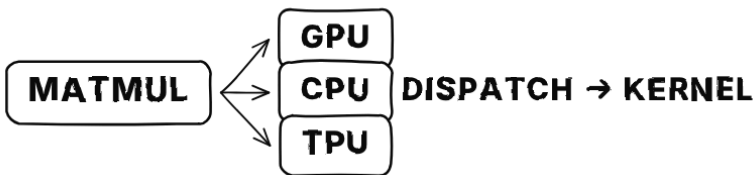
order the size, because once the size reaches ten, three times the size plus ten is never more than four times the size. Beyond that point, constants and small extra terms stop mattering, and only the overall shape of the growth counts; this is called asymptotic analysis. In this language, the naive matrix multiplication is of order a times b times c, growing in proportion to each of the three sizes. For two square tables of n rows and n columns that is n times n times n, and we say multiplication is cubic: double n and the work grows eightfold, exactly as in the hand count.



Big-O: Bounded from Some Size On

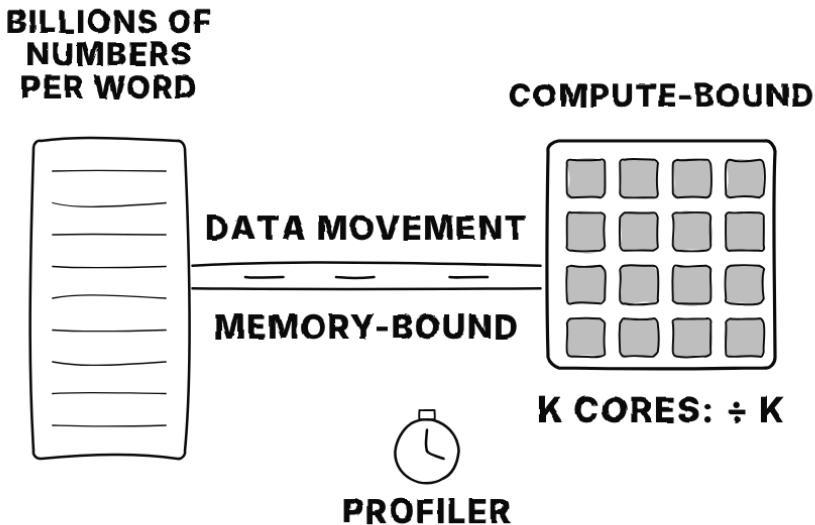
Big-O reasoning is elegant and important, but the book warns that it does not settle which method is actually faster. That depends on details such as the hardware and the kind of parallel work it supports.

When you ask a library such as PyTorch to multiply two matrices, the library picks a specific low-level routine, called a kernel, for the hardware your program runs on, whether a graphics chip, an ordinary processor or a specialised chip, in a step called dispatching; compilers such as XLA go further and fuse several operations into one tuned for that hardware. A striking example: the best known methods for multiplying two square tables have a work of order n raised to a power below two point four, far better than the cube of the naive method. For tables of a thousand by a thousand, the cube is a billion operations, while a thousand to the power two point four is only about sixteen million. Yet these clever methods are hard to spread efficiently across the thousands of small cores of a graphics chip, so in practice they are rarely used.



Better Growth, Rarely Used

Two more limits of big-O. First, having k processors working in parallel changes nothing in the order of growth: at best it divides the time by k , a constant factor, so a cubic method stays cubic. Second, big-O ignores the time spent moving data from one place to another, and that can become the main bottleneck. A chat assistant answering you is often limited less by arithmetic than by how fast its billions of numbers can be pulled out of memory for every word it writes. A program limited by data movement is called memory-bound, as opposed to compute-bound, and the only reliable way to tell which you have is to run a profiler, a tool that measures where the time actually goes. The book concludes that judging the cost of an algorithm is far from trivial, because the theoretical growth and the observed speed interact.



Remember

Complexity describes how work grows with size: multiplying an a by b table by a b by c table takes a times c dot products of length b , its time complexity, and needs memory for the result, its space complexity. Big-O keeps only the shape of the growth for large sizes, ignoring constants: naive matrix multiplication is of order a times b times c , cubic for square tables, so doubling the size makes eight times the work. Real speed also depends on hardware, the kernel the library dispatches and compilers; methods with better growth, below the power two point four, are rarely used because they parallelise badly. Parallel processors give only a constant speed-up, and moving data can dominate, making a program memory-bound rather than compute-bound, which only a profiler can reveal.

Questions

1. Naive multiplication of two square tables is cubic. What happens to the work when the size of the tables doubles?
 - a. It doubles
 - b. It quadruples
 - c. It grows eightfold
 - d. It stays the same on a graphics chip
2. Why are the matrix multiplication methods with growth below the power two point four rarely used in practice?

- a. They are hard to spread efficiently over highly parallel hardware such as graphics chips
 - b. They give wrong answers for large tables
 - c. They need more memory than the result table
 - d. They only work for tables with fewer than a thousand rows
3. A program runs on 8 processors instead of 1. What does big-O say about its order of growth?
- a. It falls from cubic to quadratic
 - b. It becomes eight times smaller in order
 - c. It cannot be computed any more
 - d. It does not change: parallelism gives at best a constant speed-up
4. What does it mean that a program is memory-bound?
- a. It uses more memory than the computer has
 - b. Its speed is limited mainly by moving data rather than by arithmetic
 - c. It stores every intermediate result on disk
 - d. It has a space complexity of order zero

Answers

- 1. c)** Cubic growth means the work goes as the size times itself three times, so doubling the size multiplies the work by two times two times two, eight.
- 2. a)** Their better asymptotic growth does not translate into speed, because they parallelise badly on GPUs.
- 3. d)** With k processors the time is divided by at most k , a constant factor, which big-O ignores.

4. b) Big-O ignores data movement, which can become the main bottleneck; then the program is memory-bound rather than compute-bound, as a profiler can show.

Worked examples

Counting multiplications How many multiplications does the naive method need to multiply a table of 10 rows by 20 columns with a table of 20 rows by 5 columns, and what is the shape of the result?

The inner sizes are both 20, so the product exists and has 10 rows and 5 columns.

That is 10 times 5, which is 50 entries.

Each entry is a dot product of length 20, so it needs 20 multiplications.

In all, 50 times 20 gives 1,000 multiplications, which is a times b times c with a 10, b 20 and c 5.

Answer: 1,000 multiplications; the result is 10 by 5.

Practice

1. A naive multiplication of two square tables takes one second. Roughly how long would tables three times as large take?

Hint: The work is cubic in the size.

Answer: About 27 seconds: three times three times three is twenty-seven, so the work, and roughly the time, grows twenty-seven-fold.

2. Show that a program needing five times the size plus one hundred steps is of order the size.

Hint: Find a multiple of the size that is never exceeded beyond some point.

Answer: Once the size reaches one hundred, five times the size plus one hundred is never more than six times the size, so beyond that point the

work is at most a fixed multiple of the size: it is of order the size.

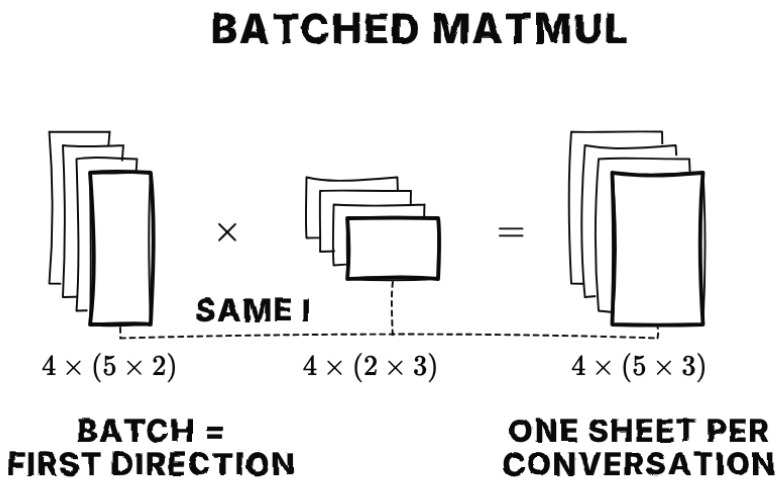
3. For square tables of a thousand rows, compare the work of the naive method with that of a method whose work grows as the size to the power two point four.

Hint: A thousand cubed, and a thousand to the power two point four, which is about sixteen million.

Answer: The naive method needs about a billion operations, the clever one about sixteen million, roughly sixty times fewer; yet the clever one is rarely used because it does not parallelise well on graphics chips.

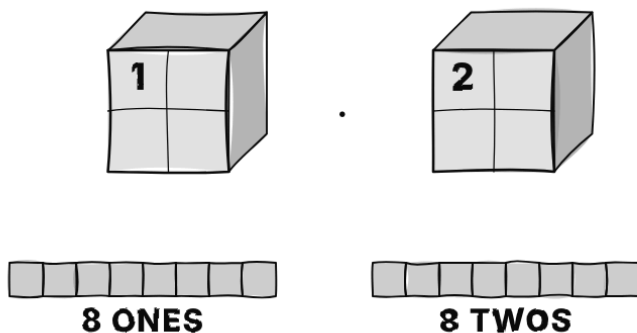
Stacks of Tables and a Shorthand by Letters

Imagine four pupils, each with a homework sheet asking them to multiply a five by two table by a two by three table. Each pupil works alone on their own pair and hands in a five by three result. Collected together, the four answers form a pile: four results of five by three. Nothing new happened inside, the same multiplication was simply done four times side by side. This is how most operations on tensors with more than two directions work, and the book calls this one batched matrix multiplication.



Vectors and matrices are special because many operations make sense for them and become awkward or undefined beyond: matrix multiplication, the true matrix exponential, determinants. So when tensors have more directions, the operations we need are mostly of two kinds: batched versions of matrix operations, or combinations of matrix operations with reductions such as sums. For batched multiplication, take a first pile of n tables, each a by b , and a second pile of n tables, each b by c . The result is a pile of n tables, each a by c , where table number i is just table i of the first pile times table i of the second. Libraries treat the leading direction, the first one, as the batch and handle it quietly: in PyTorch, the ordinary multiplication command applied to a pile of four five by two tables and a pile of four two by three tables returns a pile of four five by three tables, exactly as the book's code box shows. This is how a chat assistant answers many people's messages at once: each conversation is one sheet in the pile, and the same multiplications run on all the sheets together.

For the second kind, the book gives a generalised dot product. Take two tensors of the same shape, with three directions, multiply them position by position, and add up every single product. It is nothing but the ordinary dot product of the two tensors flattened into long lists. For example, two little cubes of two by two by two, the first filled with ones and the second with twos, have eight positions, each product is two, and the generalised dot product is sixteen. The book adds that this short survey covers most of the tensor operations used in the rest of the book, and that anything new will be introduced when needed.



$$8 \times (1 \cdot 2) = 16$$

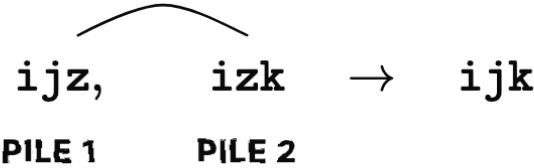
EINSUM

A Dot Product of Whole Tensors

Then comes an optional section on a clever shorthand called einsum, after Einstein's summation convention, which can describe practically any tensor operation with a short string of letters. The idea is to give every direction of every tensor a letter. For batched multiplication, the first pile has directions i, j, z , the second has i, z, k , and the result has i, j, k . For the generalised dot product, both tensors have directions i, j, k , and the result has no letters at all, since it is a single number. The rule, in the simplified form the book uses, is this: any letter that appears in the inputs but not in the result is summed over. In batched multiplication the letter z appears in both inputs but not in the result, so it is the direction that gets multiplied and added up, which is exactly the dot product inside each table multiplication. In the generalised dot product all three letters vanish from the result, so

everything is summed. The whole operation is then written as one string: the input letters separated by a comma, an arrow, and the result letters.

**IN THE INPUTS, NOT IN THE RESULT → SUMMED
SUMMED**



ijk, ijk → **EINOPS**
ONE NUMBER

ijz, ikz → ijk TURNED OVER

A BATCH OF 2, BY HAND

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

Einsum: Letters for Directions

Most frameworks provide an einsum operation that takes exactly such a string. Its first advantage is that you need not remember each library's own commands: PyTorch alone has several multiplication functions with different shape rules and broadcasting behaviour, while einsum describes them all the same way. Its second advantage is portability: the string for batched multiplication is identical in PyTorch and in JAX, so code moves from one library to the other unchanged. Turning a direction around is easy too: if the second pile has its tables lying the other way, with directions i, k, z, you simply

write those letters, and einsum multiplies by the turned-over tables without any extra step. For these reasons einsum, and packages built on the same idea such as einops, have become very popular.

Try it yourself

Take two sheets of paper and on each draw a pile of two small tables, two by two. On the first sheet write the tables one, two over three, four and one, zero over zero, one; on the second write two, zero over zero, two and five, six over seven, eight. Multiply table one by table one and table two by table two: you should get two, four over six, eight, and then five, six over seven, eight. That is a batched multiplication with a batch of two, done by hand.

Remember

Beyond two directions, most operations are batched matrix operations or matrix operations combined with reductions. Batched matrix multiplication multiplies table i of one pile by table i of another, giving a pile of results, and libraries handle the leading batch direction automatically. The generalised dot product multiplies two same-shaped tensors position by position and adds everything, the dot product of their flattened forms. Einsum gives each direction a letter and sums over letters that appear in the inputs but not the result, so one string describes the operation in the same way in PyTorch and JAX, which made it, and einops, popular.

Questions

1. A pile of 4 tables of 5 by 2 is multiplied, batch-wise, by a pile of 4 tables of 2 by 3. What comes out?
 - a. One table of 20 by 3
 - b. A pile of 4 tables of 5 by 3
 - c. A pile of 4 tables of 2 by 2
 - d. Nothing, because the shapes do not match
2. In the einsum rule used by the book, which directions are summed over?
 - a. Every letter that appears in the result
 - b. Only the first letter of each input
 - c. Letters that appear in the inputs but not in the result
 - d. Letters that appear in only one input
3. What is the generalised dot product of two same-shaped tensors?
 - a. The sum of all their position-by-position products, the dot product of their flattened forms
 - b. A pile of matrix products
 - c. The product of their largest entries
 - d. A tensor with the same shape as the inputs
4. Why is einsum useful when moving code from PyTorch to JAX?
 - a. Because JAX cannot multiply matrices otherwise
 - b. Because einsum runs faster than any other command

- c. Because einsum removes the need for batches
- d. Because the same letter string describes the operation identically in both libraries

Answers

- 1. **b)** Batched multiplication multiplies table i by table i ; each 5 by 2 times 2 by 3 gives 5 by 3, and there are four of them.
- 2. **c)** A letter that is in the inputs but missing from the result is added up; in batched multiplication that is z , the direction of the inner dot products.
- 3. **a)** Multiplying position by position and adding everything is the same as an ordinary dot product of the two tensors written out as long lists.
- 4. **d)** The einsum string for batched multiplication is identical in PyTorch and JAX, so code carries over unchanged, without remembering each library's commands.

Worked examples

Shapes in a batched multiplication In the book's code box, a pile shaped 4 by 5 by 2 is multiplied by a pile shaped 4 by 2 by 3. Find the shape of the result.

The leading direction, 4, is the batch: there are four pairs of tables.

Each pair is a 5 by 2 table times a 2 by 3 table; the inner sizes, 2 and 2, agree.

Each product is therefore 5 by 3.

Stacking the four products gives a pile shaped 4 by 5 by 3.

Answer: A pile shaped 4 by 5 by 3.

Practice

1. Two tensors of shape 2 by 2, flattened, read one, two, three, four and two, zero, one, one. What is their generalised dot product?

Hint: Multiply position by position, then add everything.

Answer: Nine: one times two is two, two times zero is zero, three times one is three, four times one is four, and two plus zero plus three plus four is nine.

2. Using the letter rule, describe ordinary matrix multiplication of an i by z table with a z by k table.

Hint: Which letter must disappear in the result?

Answer: The inputs carry the letters i , z and z , k , and the result keeps i , k . The letter z appears in the inputs but not in the result, so it is summed over: each entry is the dot product along z , which is matrix multiplication.

3. A pile holds 32 conversations, each a table of 10 by 64, and every one must be multiplied by its own 64 by 64 table. What is the shape of the result, and how is it computed?

Hint: Think of each conversation as one sheet in a batch.

Answer: A pile of 32 tables of 10 by 64: each conversation's 10 by 64 table times its 64 by 64 table gives 10 by 64, and the 32 products are done side by side as one batched multiplication.

The Slope at a Point

Take a calculator and square the number three: nine. Now square three point zero one: nine point zero six zero one. The input moved by one hundredth and the output moved by six hundredths and a tiny bit, so near three the square changes about six times as fast as its input. Try three point zero zero one and the ratio comes even closer to six. That limiting ratio is the derivative, and it is the first tool the book needs, because gradients, its many-dimensional cousins, are what drive the training of every model in the book.

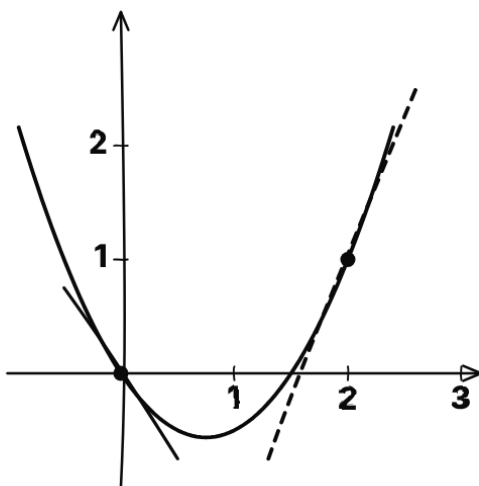
x	x^2	RATIO
3	9	
3.01	9.0601	6.01
3.001	9.006001	6.001 $\longrightarrow$ 6

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

written $\frac{\partial f}{\partial x}$ or $f'(x)$

For a function with one number in and one number out, the derivative at a point is defined exactly as you just did it: nudge the input by a small amount, see how much the output changes, divide the change of the output by the nudge, and let the nudge shrink towards zero. The book uses several ways to write it: a curly d for derivatives and gradients of any size, the same sign with the input written small beside it to say which input is being varied, and for one-number functions a prime mark after the function's name, known as Lagrange's notation. It does not worry about whether a derivative always exists, which is not guaranteed even for a smooth-looking continuous curve; it simply assumes so, and returns to awkward cases, such as the sharp corner of the absolute value at zero, in chapter six.

For familiar functions the definition gives familiar rules. A power of the input has a derivative equal to that power times the input raised to one less: the cube has derivative three times the square, so at two it is three times four, twelve. The logarithm has derivative one divided by the input, so at four its slope is one quarter. The sine has derivative equal to the cosine, so at zero its slope is one.



$$(x^3)' = 3x^2 \rightarrow 12$$

$$(\log x)' = \frac{1}{x} \rightarrow \frac{1}{4}$$

$$(\sin x)' = \cos x \rightarrow 1$$

$$f'(x) = 2x - 1.5$$

**SLOPE AT 0:
-1.5, FALLING**

**SLOPE AT 2:
+2.5, RISING**

Tangent Slopes in a Valley

Geometrically, the derivative is the slope of the tangent, the straight line that just touches the curve at a point, and it is the best straight-line approximation of the function there. The book's figure plots the curve $x^2 - 1.5x$, a shallow valley, with its tangent drawn at two points. Its derivative is $2x - 1.5$. At zero the slope is minus one and a half: the curve goes down to the right, so it is decreasing there. At two the slope is plus two and a half: the curve rises to the right. That is the lasting lesson: the sign of the slope tells you, in a small neighbourhood, which way the function is heading, and the same insight will carry over to functions of many inputs. When the photo app is trained, it is exactly this kind of slope, computed for millions of dials at once, that tells each dial which way to turn.

Three rules let us differentiate functions built from simpler ones, and all three extend to many dimensions. Linearity: the derivative of a sum is the sum of the derivatives; for x squared plus three x at one, that is two plus three, five. The product rule: the derivative of a product is the derivative of the first times the second, plus the first times the derivative of the second; for x squared times x cubed this gives two x to the fourth plus three x to the fourth, five x to the fourth, which is indeed the derivative of x to the fifth. The chain rule: the derivative of one function applied to the result of another is the outer derivative, taken at the inner value, times the inner derivative. For two x plus one, all squared, at one: the inner value is three, the outer derivative there is six, the inner derivative is two, so the slope is twelve; expanding the square first and differentiating gives the same twelve. The chain rule is the one to remember: a slope through a chain of steps is the product of the slopes of the steps, and that idea will carry the whole book.

SUM $(f + g)' = f' + g' \quad (x^2 + 3x)'|_1 = 2 + 3 = 5$

PRODUCT $(fg)' = f'g + fg' \quad (x^2x^3)' = 2x^4 + 3x^4 = 5x^4$

CHAIN

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

$$((2x + 1)^2)'|_1 = \underbrace{6}_{\text{outer}} \cdot \underbrace{2}_{\text{inner}} = 12$$

Sum, Product and Chain Rules

Remember

The derivative of a one-number function is the limit of the change of the output divided by a shrinking nudge of the input; squaring near three gives six. Powers, logarithms and sines have familiar derivatives: the cube gives three times the square, the logarithm one over the input, the sine the cosine. The derivative is the slope of the tangent and the best straight-line approximation; its sign says whether the function rises or falls nearby, as the figure's valley shows. Three rules build derivatives of combined functions: sums add, products follow the product rule, and chains multiply their slopes, the chain rule.

Questions

1. The slope of a function at a point is negative. What does that say about the function near that point?
 - a. It falls as the input increases, and rises to the left
 - b. It rises as the input increases
 - c. It has reached its lowest value
 - d. It is not differentiable there
2. Which rule gives the slope of one function applied to the result of another?
 - a. Linearity
 - b. The product rule
 - c. The chain rule: outer slope at the inner value times inner slope
 - d. The power rule
3. Why does the book care so much about derivatives?
 - a. Because every model must be a polynomial
 - b. Because derivatives replace the need for data
 - c. Because they only matter for sines and logarithms
 - d. Because gradients, their many-input form, drive the training of every model in the book
4. What is the slope of the logarithm at the input four?
 - a. Four
 - b. One quarter

- c. Zero
- d. Minus one quarter

Answers

1. **a)** A negative slope means the curve goes down to the right in a small neighbourhood, as the figure's valley does at zero.
2. **c)** The chain rule multiplies the slopes of the steps; for two x plus one, squared, at one it gives six times two, twelve.
3. **d)** Gradients, the many-dimensional cousins of the derivative, tell each adjustable number which way to turn during training.
4. **b)** The logarithm's derivative is one divided by the input, so at four it is one quarter.

Worked examples

Reading the slope of the book's curve The book's figure shows the curve x squared minus one and a half x . Find its slope at one, and the point where the curve is flat.

The power rule gives two x for x squared, and linearity gives minus one and a half for the second term, so the derivative is two x minus one and a half.

At one the slope is two minus one and a half, which is one half: the curve rises gently there.

The curve is flat where the slope is zero: two x minus one and a half equals zero.

That happens at x equal to three quarters, the bottom of the valley.

Answer: Slope one half at one; flat at three quarters, the lowest point.

Practice

1. Estimate the slope of the square at five by nudging the input by one hundredth, and compare it with the power rule.

Hint: Square five point zero one, subtract twenty-five, divide by one hundredth.

Answer: Five point zero one squared is twenty-five point one zero zero one, so the change is about one tenth, divided by one hundredth gives ten point zero one. The power rule gives two times five, ten, the value the ratio approaches as the nudge shrinks.

2. Use the chain rule to find the slope of three x minus one, cubed, at x equal to one.

Hint: Find the inner value first, then the outer slope there, then multiply by the inner slope.

Answer: The inner value is two. The outer function is the cube, with slope three times the square, so at two it is twelve. The inner slope is three, so the slope is twelve times three, thirty-six.

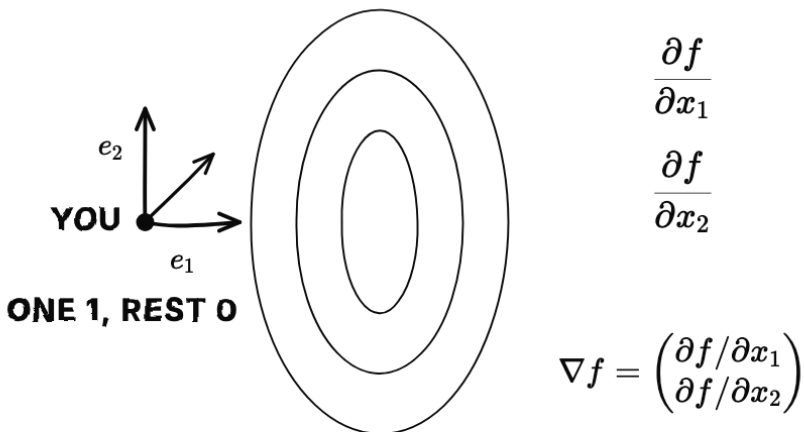
3. Use the product rule for x times the logarithm of x at x equal to one.

Hint: Derivative of the first times the second, plus the first times the derivative of the second; the logarithm of one is zero.

Answer: The first gives one times the logarithm of one, which is zero; the second gives one times one over one, which is one. The slope is zero plus one, one.

Slopes in Every Direction

Stand on a hillside with a map in your hand. Ask how steep the ground is, and the honest answer is: it depends which way you walk. Step due east and you might climb sharply, step north and you might barely rise, step north-east and you get something in between. On a line there were only two ways to move, left and right, but on a map, and even more in many dimensions, there are endless directions, so a slope means nothing until you say which way. This part shows how one list of numbers, the gradient, answers the question for every direction at once.



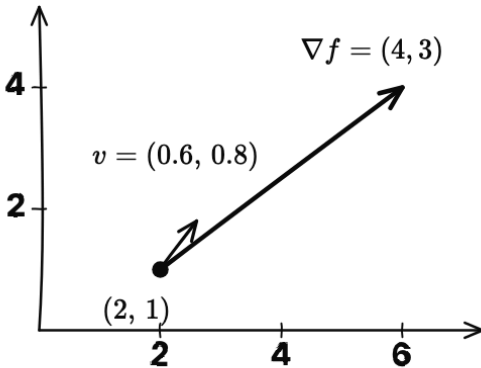
A Slope Needs a Direction

Take a function that turns a vector of d numbers into one number, like the height of the ground at a point of the map. The simplest directions are the axes: move along one axis only, keeping every other coordinate fixed, and measure the slope exactly as in the one-number case. Moving along axis i means adding a small multiple of a special vector that has a one in position i and zeros elsewhere, one row of the identity table, called a basis vector. The slope obtained this way is called the partial derivative with respect to that coordinate. Stack the d partial derivatives into one vector and you have the gradient, so important that the book gives it its own sign, an upside-down triangle called nabla.

Here is an example. Let the height be the first coordinate squared plus three times the second, and stand at the point two, one. Moving along the first axis only, the slope is twice the first coordinate, four; moving along the second only, the slope is three. So the gradient there is the vector four, three.

$$f(x) = x_1^2 + 3x_2, \quad x = (2, 1)$$

$$\partial_1 f = 2x_1 = 4, \quad \partial_2 f = 3$$



slope along $v = \nabla f \cdot v$

$$(4, 3) \cdot (1, 1) = 7$$

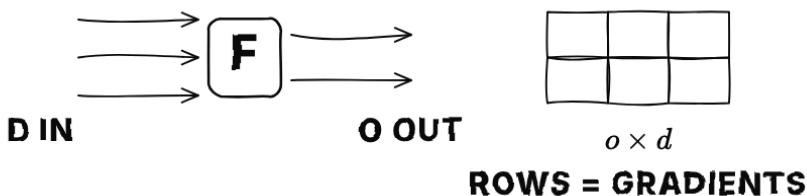
$$(4, 3) \cdot (0.6, 0.8) = 4.8$$

**MILLIONS OF DIALS,
ONE LIST**

Gradient (4, 3) and Directional Slopes

What about a direction that is not an axis, say any arrow v ? The slope along it is called the directional derivative, defined again by a small step, this time along v . Because any movement can be split into small movements along each axis, the directional derivative turns out to be simply the dot product of the gradient with the direction. In our example, walking along one, one gives four plus three, seven; walking along the arrow six tenths, eight tenths, which has length one, gives four times six tenths plus three times eight tenths, two point four plus two point four, four point eight. So knowing the gradient is enough to know the slope in every possible direction. When the photo app is trained, its error depends on millions of dials, and its gradient is one list of millions of slopes that tells how the error changes whichever way the dials are moved together.

Now let the function return several numbers instead of one, say o outputs from d inputs. Each output has its own gradient, and stacking these gradients as rows gives a table of o rows and d columns, called the Jacobian. With one output the Jacobian is just the gradient, and with one input and one output it is the ordinary derivative. For instance, with two inputs, the first output their product and the second their sum, at the point two, one, the first row is one, two and the second row is one, one. The Jacobian inherits the properties of the derivative, and the most important one is the chain rule: the Jacobian of one function applied after another is the product of their two Jacobians, the outer one taken at the point produced by the inner one. This is the chain of slopes from the previous part, now as a matrix multiplication, and it is the mathematical heart of how networks are trained.



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

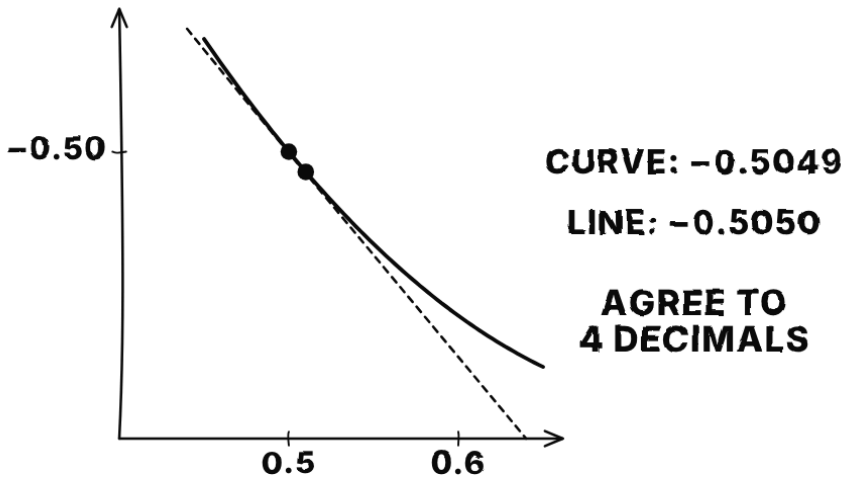
$$J_{f \circ g}(x) = J_f(g(x)) J_g(x)$$

CHAIN RULE = MATRIX PRODUCT

The Jacobian and the Chain Rule

Like the derivative, the gradient also gives the best straight approximation of a function near a point, a statement known as Taylor's theorem: near a chosen point, the function is roughly its value there plus the dot product of the gradient with the displacement from that point. The book checks this on its earlier curve, $x^2 - 1$, at the point $(1, 0)$. There the value is -1 and the slope is 2 . Moving one hundredth to the right, the true value is -0.99 , while the straight approximation gives $-1 + 2 \times 0.01 = -0.98$. The two agree to the fourth decimal place, which is what a good first-order approximation means.

$$f(x + h) \approx f(x) + \nabla f(x) \cdot h$$



First-Order Approximation, Checked

Remember

For a function of many inputs, a slope needs a direction; along one axis it is a partial derivative, and the stack of all partial derivatives is the gradient. The slope in any direction, the directional derivative, is the dot product of the gradient with that direction, so the gradient alone gives every directional slope: four, three along one, one gives seven. For a function with several outputs, the gradients of the outputs stacked as rows form the Jacobian, a table of outputs by inputs. The Jacobian of a chain of functions is the product of their Jacobians, the chain rule in matrix form. Near a point, the value there plus the gradient's dot product with the displacement is the best straight approximation, as the check at one half shows to four decimals.

Questions

1. The gradient of a function at a point is four, three. What is its slope along the direction one, one?
 - a. Seven, the dot product of the gradient with the direction
 - b. Twelve, the product of four and three
 - c. Five, the length of the gradient
 - d. One, because the direction has two ones
2. A function takes 5 numbers in and gives 3 numbers out. What is the shape of its Jacobian?
 - a. 5 rows by 3 columns

- b. 5 rows by 5 columns
 - c. 3 rows by 5 columns, one row per output
 - d. A single vector of 15 numbers
3. How is the Jacobian of one function applied after another obtained?
- a. By adding the two Jacobians
 - b. By taking the larger of the two
 - c. By multiplying them position by position
 - d. By multiplying the two Jacobians, the outer one taken at the inner function's result
4. Why is knowing the gradient enough to know the slope in every direction?
- a. Because every function has the same slope in all directions
 - b. Because any movement splits into movements along the axes, so the slope is the dot product of the gradient with the direction
 - c. Because the gradient already lists every possible direction
 - d. Because only axis directions matter in practice

Answers

- 1. a)** The directional derivative is the dot product of the gradient with the direction: four times one plus three times one, seven.
- 2. c)** Each output has a gradient over the 5 inputs, and the 3 gradients stacked as rows give a 3 by 5 table.
- 3. d)** The chain rule in matrix form: the Jacobian of a composition is the matrix product of the individual Jacobians.

4. b) Splitting a movement into its axis parts turns the directional derivative into a dot product of the gradient with the direction.

Worked examples

The book's straight-line check For the curve $x^2 - 1.5x$, use the straight approximation at one half to predict the value at one half plus one hundredth, and compare it with the true value.

At one half the value is one quarter minus three quarters, which is minus one half.

The slope is two times one half minus one and a half, which is minus one half.

The approximation adds the slope times the step: minus one half plus minus one half times one hundredth gives minus zero point five zero five.

The true value at zero point five one is zero point two six zero one minus zero point seven six five, which is minus zero point five zero four nine, agreeing to four decimals.

Answer: Approximation minus zero point five zero five zero; true value minus zero point five zero four nine.

Practice

1. For the function that multiplies two inputs, find the gradient at the point three, two and the slope along the direction one, minus one.

Hint: Each partial derivative is the other input.

Answer: The partial derivative with respect to the first input is the second, two; with respect to the second it is the first, three. The gradient is two, three, and the slope along one, minus one is two minus three, minus one.

2. A function has two outputs: twice the first input plus the second, and the first input minus the second. Write its Jacobian.

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

Answer: The first row is two, one, and the second row is one, minus one: a table of two rows by two columns, the same at every point because the function is linear.

3. Use the straight approximation of the square at three to estimate three point one squared, and compare with the true value.

Hint: The value at three is nine and the slope there is six.

Answer: Nine plus six times one tenth gives nine point six. The true value is nine point six one, so the approximation is off by only one hundredth for this step.

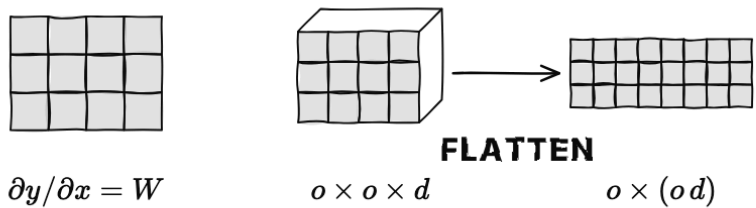
Where the Ground Falls Fastest

Back on the hillside from the last part, now in thick fog. You want to reach the valley floor but can see only the ground at your feet. A sensible plan: feel which way the ground drops most steeply, take a step that way, and repeat. That plan, made precise, is gradient descent, and this part prepares it in two moves: first a careful look at the sizes of Jacobians, then the reason why the steepest way down is exactly opposite to the gradient.



$$y = W x$$

D IN O OUT

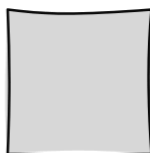


The book begins with what it calls a pedantic note on sizes. Take the basic operation of a network layer: a table of weights multiplies an input vector to give an output vector, with d inputs and o outputs. Seen as a function of the input, its Jacobian is simply the weight table itself, o rows by d columns, which makes sense: each weight says how strongly one output responds to one input. Seen as a function of the weights, the picture changes, because now the thing being varied is itself a table. The "Jacobian" then has three directions: one for the output, and two for the row and column of the weight being nudged, o by o by d . One can always flatten the weight table into one long vector of o times d numbers, and then the Jacobian is an ordinary table of o rows and o times d columns.

The book works this out in a box. Each output is the dot product of one row of weights with the input, so nudging a weight changes only the output of its own row, and at a rate equal to the input value that the weight multiplies. Let us try it with two outputs and three inputs, the input being one, two, three. The first output responds to the three weights of the first row with rates one, two and three, and does not respond to the second row's weights at all. The second output responds to the second row's weights with the same rates, one, two, three, and not to the first row's. So the flattened Jacobian is a table of two rows by six columns: one, two, three, then three zeros on the first row; three zeros, then one, two, three on the second. Half of it is zeros, and the rest is the input copied over and over. Storing such a table explicitly, materialising it, wastes memory on repeated values, and chapter six will show how to avoid it, because in practice we only ever need the table applied to another tensor.

$$x = (1, 2, 3)$$

1	2	3	0	0	0
0	0	0	1	2	3



$$3200 \times 3200$$

> 10 MILLION ENTRIES

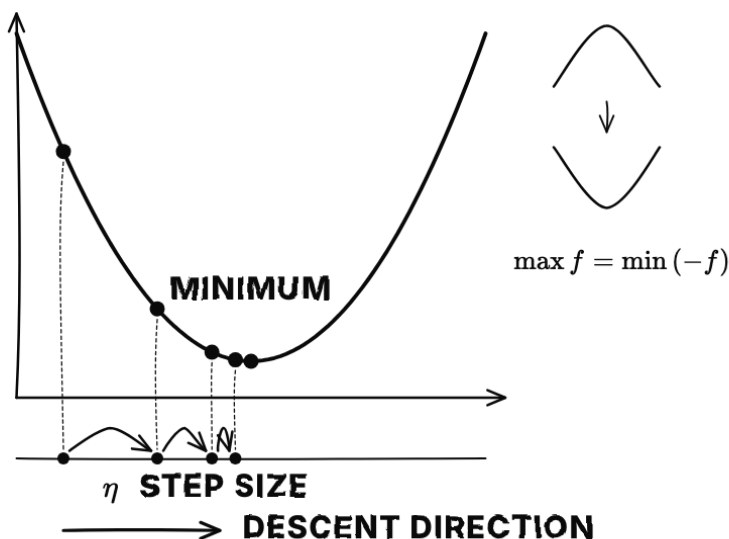
$$32 \times 100 = 3200$$

VECTOR-JACOBIAN PRODUCTS

Mostly Zeros and Copies

That is the warning to keep in mind. The book will often write Jacobians as simple tables to keep the notation clean, but their true sizes hide inside the shapes of the inputs and outputs, batch sizes included. For a batch of thirty-two examples with a hundred numbers each, one layer's input already holds three thousand two hundred numbers, and a Jacobian from such an input to a same-sized output would be a table of three thousand two hundred by three thousand two hundred, more than ten million entries, for one layer and one batch. Chapter six avoids this with what it calls vector-Jacobian products, and the book points to a more formal treatment that views Jacobians as abstract linear maps.

Now to gradient descent. Suppose we want the input that makes a function of many numbers as small as possible, its minimum. Later the input will be the dials of a model, the parameters, and the function will measure how badly the model does on its data; for the photo app, how many pictures it labels wrongly. Minimising loses nothing compared with maximising: making a function as large as possible is the same as making its negative as small as possible, just as turning a hill upside down turns its peak into a valley's lowest point. Only rarely can the best point be written down directly by a formula, as will happen for least squares in chapter four. Usually we must search step by step: start from a random guess, and at each step move by a certain length in a certain direction. The length is called the step size, or, in machine learning, the learning rate. A direction is called a descent direction if some step along it lowers the function. Pick a descent direction every time, choose the step size with care, and the procedure approaches a minimum, in a sense the next part makes precise.

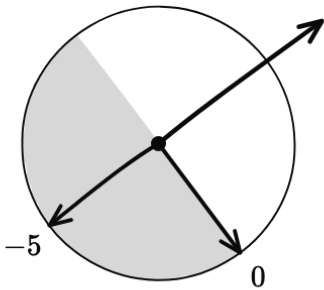


Step by Step Downhill

Which directions go downhill? Exactly those whose directional derivative is negative, and from the last part the directional derivative is the dot product of the gradient with the direction. Writing that dot product as the length of the gradient, times the length of the direction, times the cosine of the angle between them, and taking directions of length one, only the cosine is left to choose. The cosine is negative when the angle lies between a quarter turn and three quarters of a turn, so all those directions lead down. The most negative cosine, minus one, comes at half a turn, pointing exactly opposite to the gradient: that is the steepest descent direction. With the gradient four, three from before, whose length is five, stepping in the opposite direction, minus eight tenths, minus six tenths, gives a slope of minus five, the steepest possible; stepping at a right angle to it, along six

tenths, minus eight tenths, gives a slope of zero, a level path along the hillside.

$\nabla f = (4, 3), \quad \|\nabla f\| = 5$



DOWNHILL: COS < 0

$(-0.8, -0.6) : \text{slope } -5$

$(0.6, -0.8) : \text{slope } 0$

Opposite the Gradient Is Steepest

Remember

For a layer that multiplies an input by a weight table, the Jacobian with respect to the input is the weight table itself, while with respect to the weights it has three directions, or, flattened, o rows by o times d columns. That weight Jacobian is mostly zeros and copies of the input, and real sizes, batches included, make explicit Jacobians enormous, which chapter six avoids with vector-Jacobian products. Gradient descent searches for a minimum step by step, moving by a step size, the learning rate, along a descent direction, one whose directional derivative is negative. Directions at more than a quarter turn from the gradient go downhill, and the direction exactly opposite to the gradient is the steepest descent direction.

Questions

1. For a layer where a weight table multiplies an input, what is the Jacobian with respect to the input?
 - a. The weight table itself
 - b. The input copied o times
 - c. A table of zeros
 - d. A tensor with three directions
2. Why does the book warn against materialising the Jacobian with respect to the weights?
 - a. Because it cannot be computed at all

- b. Because it is mostly zeros and copies of the input, wasting memory, and real batch sizes make it enormous
 - c. Because it changes every time the input changes
 - d. Because it is the same as the gradient
- 3. A direction makes an angle of a quarter turn, a right angle, with the gradient. What happens if you step along it?
 - a. The function falls fastest
 - b. The function rises fastest
 - c. To first order the function does not change: the path is level
 - d. The step size becomes zero
- 4. Why can we always talk about minimising rather than maximising?
 - a. Because every function has a minimum but not a maximum
 - b. Because maximising requires a closed formula
 - c. Because gradients only point downwards
 - d. Because making a function as large as possible is the same as making its negative as small as possible

Answers

- 1. **a)** Each weight says how strongly one output responds to one input, so the Jacobian with respect to the input is exactly the weight table, o rows by d columns.
- 2. **b)** Its entries are the input values and zeros; with batches its size explodes, and chapter six avoids building it with vector-Jacobian products.
- 3. **c)** The cosine of a right angle is zero, so the directional derivative is zero; only angles beyond a quarter turn go downhill.

4. d) Turning a function upside down turns its peaks into lowest points, so maximising f is minimising minus f .

Worked examples

The size of a weight Jacobian A layer has 4 inputs and 3 outputs. How many entries does its Jacobian with respect to the weight table have, and what shape does it have when the weights are flattened?

The weight table has 3 rows by 4 columns, 12 weights.

With respect to the weights, the Jacobian has one direction for the 3 outputs and two for the 3 by 4 weights: 3 by 3 by 4.

That is 3 times 3 times 4, which is 36 entries.

Flattening the 12 weights into one vector gives a table of 3 rows by 12 columns, still 36 entries, most of them zeros.

Answer: 36 entries; flattened, 3 rows by 12 columns.

Practice

1. The gradient at your current guess is six, eight. What is the steepest descent direction of length one, and what slope does it give?

Hint: The length of six, eight is ten; point exactly the other way.

Answer: The direction is minus six tenths, minus eight tenths, opposite to the gradient. Its slope is the dot product: minus three point six minus six point four, which is minus ten, the length of the gradient with a minus sign.

2. With the same gradient six, eight, is the direction zero, one a descent direction?

Hint: Compute the dot product of the gradient with the direction and look at its sign.

Answer: No. The dot product is eight, which is positive, so the function rises along that direction; it is less than a quarter turn from the gradient.

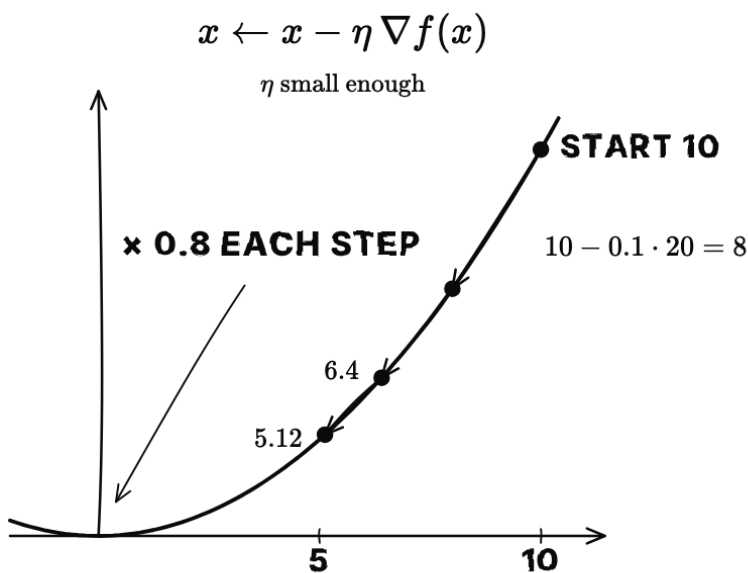
3. With weights in a table of 2 rows and the input one, two, three, what is the rate at which the second output responds to the weight in row two, column three, and to the weight in row one, column three?

Hint: Each output depends only on its own row of weights, at a rate equal to the input value the weight multiplies.

Answer: To the weight in row two, column three it responds at a rate of three, the third input value. To the weight in row one it does not respond at all: the rate is zero.

Walking Downhill, and Where You End Up

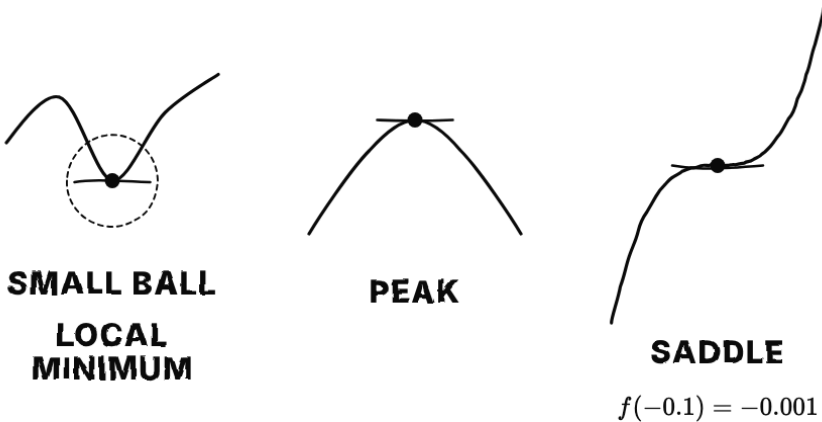
Let us run the foggy-hillside plan on the simplest valley there is: the height is the square of your position, and you start at ten. The slope there is twice the position, twenty, so you step against it, by a step size of one tenth: ten minus one tenth of twenty brings you to eight. At eight the slope is sixteen, and the same rule brings you to six point four; then to five point one two, and so on. Each step keeps four fifths of the distance to the bottom, and you glide towards zero, the lowest point. That simple loop, moving against the gradient over and over, is steepest gradient descent.



Descent on x^2

In words, the rule is: the new guess is the old guess minus the step size times the gradient at the old guess. The book does not dwell on how to pick the step size; it assumes it is small enough that each step lowers the function. It also notes that gradient descent is only as fast as the method used to compute the gradient, and chapter six introduces a general method for that. The question of this part is different: where does the walk end?

$$\nabla f = 0 : \text{stationary}$$



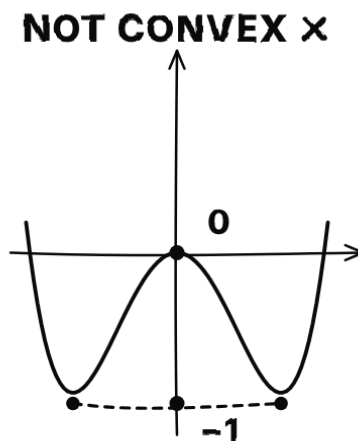
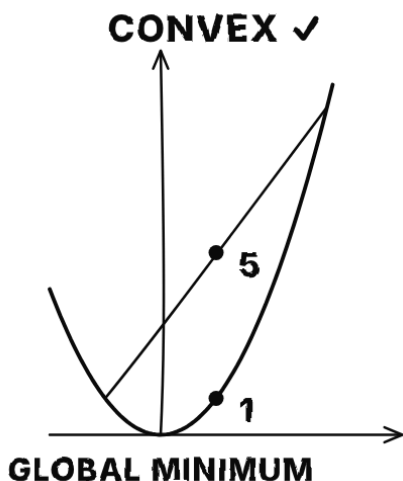
WHICH ONE DEPENDS ON THE START

Flat Spots: Minimum, Peak, Saddle

The book first defines its goals carefully, noting that a trusting reader may skip ahead. A local minimum is a point whose value is no larger than the values at all nearby points, within some small ball around it, like the bottom of one dip in a bumpy landscape. At such a point the tangent is flat, and around it the ground rises. The flatness is captured by a stationary point: a point where the gradient is zero, so there is no slope in any direction. But a flat spot is not always a bottom. It can be a peak, which is just the bottom of the upside-down function, or a saddle point, where the curve changes the way it bends. The book's figure shows such a point; a simple case is the cube of x at zero, where the slope is zero yet the curve keeps going down to the left, with minus one thousandth at minus one tenth. In general, with no assumption on the function, the most one can prove is that gradient

descent reaches some stationary point, and which one depends on where it started.

Can we do better? Picture a bowl-shaped parabola: it has no saddle points and only one minimum, and that minimum is the lowest value over the entire range of inputs, called a global minimum. On a parabola, from any starting point, every gradient points back towards that bottom, so gradient descent always finds it, as our walk from ten did. The property that generalises the parabola is called convexity. A function is convex if, whenever you pick two points and stretch a straight string between their values on the graph, the graph between them never rises above the string. Check it on the square: take the points minus one and three, whose values are one and nine; halfway between, at one, the string is at the average, five, while the graph is at only one, below the string. Now try a landscape with two valleys, the fourth power minus twice the square: its two bottoms are at minus one and plus one, both with value minus one, and halfway between, at zero, the graph sits at zero, above the string at minus one, so this function is not convex.



Convex or Not: the String Test

Convexity sorts optimisation problems into easy and hard. For a non-convex function, gradient descent reaches a stationary point and nothing more can be said without looking at derivatives of derivatives. For a convex function, it reaches a global minimum whatever the starting point. And if the string is always strictly above the graph, called strict convexity, that global minimum is the only one. Finding the global minimum of a non-convex function with gradient descent alone would mean restarting from every possible starting point, endlessly, which places the task among the hardest problems computer science knows, called NP-hard.

This has real historical weight. Chapter five will show that every non-trivial network is non-convex, so its training may have many

stationary points. Older methods for supervised learning, such as support vector machines, kept non-linear behaviour while still enjoying a convex problem, and for years that looked like a decisive advantage. Yet the networks behind your photo app and chat assistant are non-convex, and they work. Interestingly, the book observes, when such models are started from a reasonable initial guess, their training tends to settle at points that perform well in practice.

Remember

Gradient descent repeats one rule: the new guess is the old one minus the step size times the gradient; on the square, from ten with step one tenth, it goes to eight, then six point four, then five point one two. A local minimum is lowest within a small neighbourhood; a stationary point has zero gradient and may be a minimum, a maximum or a saddle point, and in general gradient descent only reaches some stationary point. A global minimum is lowest everywhere; for a convex function, whose graph never rises above the string between two of its points, gradient descent reaches it from any start, and strict convexity makes it unique. Networks are non-convex, unlike support vector machines, yet from a reasonable start their training reaches points that work well.

Questions

1. For a convex function, where does gradient descent end, with a small enough step?

- a. At a global minimum, whatever the starting point
 - b. At whichever saddle point is nearest
 - c. At a different point for every start, with no guarantee
 - d. Only at a maximum
2. Why is reaching a stationary point not the same as reaching a minimum?
- a. Because stationary points always have a large gradient
 - b. Because a flat point can also be a peak or a saddle point
 - c. Because stationary points only exist for convex functions
 - d. Because a minimum must have a negative slope
3. Midway between the points minus one and plus one, the function with two valleys has the value zero, while the string between its values sits at minus one. What does this show?
- a. The function is convex
 - b. The function has a single global minimum
 - c. The function is not convex, since its graph rises above the string
 - d. The function has no stationary points
4. Why does it matter historically that networks are non-convex?
- a. Because it makes them impossible to train
 - b. Because it means their gradients are always zero
 - c. Because convex methods are always slower
 - d. Because older methods like support vector machines kept a convex problem, yet networks still train well from a reasonable start

Answers

1. **a)** On a convex function every downhill path leads to the global minimum, as on a bowl-shaped parabola.
2. **b)** A zero gradient means a flat spot, which may be a minimum, a maximum or a saddle point such as the cube at zero.
3. **c)** A convex graph never rises above the string between two of its points; here it does, so the function is not convex.
4. **d)** Non-convexity means many stationary points, a supposed weakness against convex methods, yet networks started reasonably reach points that work well.

Worked examples

Three steps of gradient descent Minimise the square of x with gradient descent, starting at five, with a step size of one quarter.

Where are you after three steps?

The gradient of the square is twice x , so each step subtracts one quarter of twice x , which is half of x .

Step one: from five, subtract two and a half, reaching two and a half.

Step two: from two and a half, subtract one and a quarter, reaching one and a quarter.

Step three: subtract five eighths, reaching five eighths, halving the distance to the bottom each time.

Answer: Five eighths, approaching the global minimum at zero.

Practice

1. Find the stationary point of the function x squared minus four x , and say whether it is a global minimum.

Hint: Set the slope, two x minus four, to zero; the function is a bowl-shaped parabola.

Answer: The slope two x minus four is zero at x equal to two, where the value is four minus eight, minus four. The function is a convex parabola, so this stationary point is its unique global minimum.

2. Check the convexity condition for the square between the points zero and four, at the point one quarter of the way from four towards zero, that is at three.

Hint: Compare the graph's value at three with the string's value there, which is one quarter of the value at zero plus three quarters of the value at four.

Answer: The graph at three is nine. The string there is one quarter of zero plus three quarters of sixteen, which is twelve. Nine is below twelve, as convexity requires.

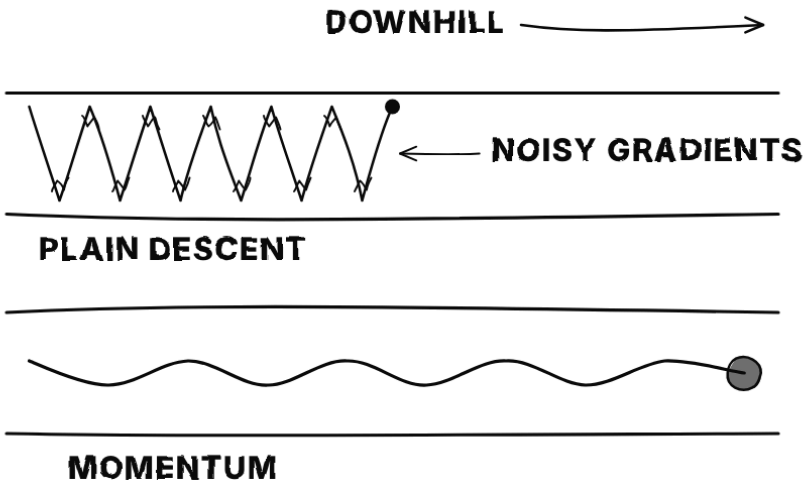
3. Run gradient descent on the square from five with a step size of one. What happens, and what does it suggest about the step size?

Hint: Each step subtracts one times twice x .

Answer: Each step sends x to x minus two x , that is minus x : five, minus five, five, and so on, never approaching zero. The step is too large; the book assumes a step size small enough that each step lowers the function.

A Ball That Remembers

Roll a marble down a long, narrow gutter that is slightly tilted along its length. It does not slide straight down the middle; it rattles from one side wall to the other, while slowly drifting along the gutter. Now roll a heavy ball down the same gutter. Its weight carries it forward: the side-to-side wobbles partly cancel, and the steady downhill pull adds up, so it gets down much faster. That difference is the idea behind momentum.



Marble and Heavy Ball in a Gutter

The negative gradient gives the steepest way down, but only in a tiny neighbourhood of the current point. Plain gradient descent behaves like the light marble, an erratic path, because each new gradient may point somewhere quite different; the book notes that, with a perfectly chosen step on a convex function, two successive gradients are even at right angles to each other. Chapter five will show that in large models these directions are also very noisy. When a chat assistant is trained, each step's gradient comes from a small handful of examples, so it jitters a great deal from one step to the next. Many techniques try to speed things up by choosing better directions, and the book prefers those that need no derivatives of derivatives, such as the table called the Hessian, and no extra evaluations of the function, since both are expensive.

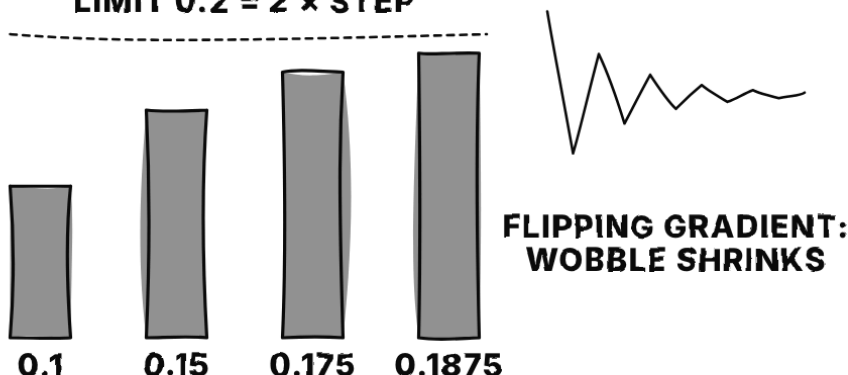
Momentum is one such technique. Instead of moving by the plain downhill step, each move is the plain downhill step plus a fraction of the previous move. The fraction, a number between zero and one, is the momentum coefficient, and the very first previous move is taken to be zero. Here is a small calculation with one dial, a step size of one tenth, a gradient that stays at one, and a coefficient of one half. The first move is minus one tenth. The second is minus one tenth plus half of minus one tenth, minus fifteen hundredths. The third is minus one tenth plus half of that, minus one hundred and seventy-five thousandths, and the moves keep growing towards minus two tenths, twice the plain step, because a steady direction keeps adding up. When the gradient instead flips back and forth, the remembered part works against the new one and the wobble shrinks.

$$\text{move} = -\eta \nabla f + \beta \cdot \text{previous move}$$

$$\beta \in (0, 1), \quad \text{first previous move} = 0$$

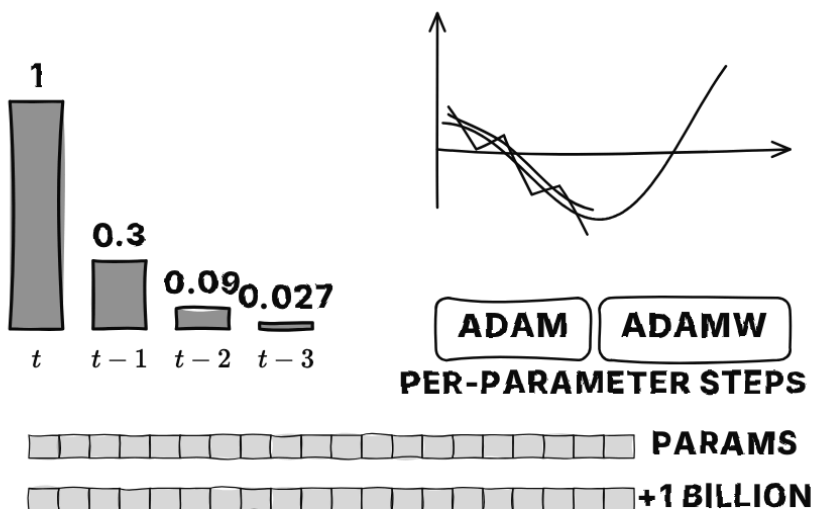
$$\eta = 0.1, \quad \nabla f = 1, \quad \beta = 0.5$$

LIMIT 0.2 = 2 × STEP



Moves That Build Up

Unrolling the rule shows how memory fades. The newest gradient counts in full; the one before is scaled by the coefficient; the one before that by the coefficient squared, and so on, so a gradient from n steps back is scaled by the coefficient raised to n minus one. With the coefficient three tenths, used in the book's figure, the gradient from two steps back counts three tenths and the one from three steps back only nine hundredths. The figure compares plain gradient descent and gradient descent with momentum on a wavy curve, x times the sine of twice x , starting just past one, and the momentum path is the smoother of the two. Momentum can indeed be shown to speed up training by smoothing the path.



Fading Memory, Adam, and Its Cost

Another common trick gives each parameter its own step size, adapted to how large its gradients have been. A popular method called Adam combines several of these ideas, and it is valued for being fairly insensitive to its own settings, the hyper-parameters, meaning numbers chosen by the user rather than learned by gradient descent; the library defaults are a good start in most cases. Adam and its variants, such as AdamW, remain the standard choice in deep learning, and designing an optimiser that clearly beats them is still an open research problem.

The price of acceleration is memory. Momentum must keep the previous move, one extra number for every parameter, which doubles the memory the optimiser itself needs: for a model with a billion

parameters, a billion more numbers. In most cases, though, the book adds, the memory needed to compute the gradient matters far more, as chapter six will show.

Remember

The negative gradient is the best direction only very locally, and in large models it is noisy, so plain gradient descent follows an erratic path. Momentum adds to each step a fraction of the previous move, starting from zero, so steady directions build up and back-and-forth wobbles cancel; older gradients fade by powers of the coefficient. With step one tenth, a constant gradient of one and a coefficient of one half, the moves grow from one tenth towards two tenths. Adam combines momentum-like ideas with per-parameter step sizes, is robust to its hyper-parameters, and is the default optimiser, with AdamW among its variants. Momentum stores the previous move for every parameter, doubling the optimiser's memory, though computing the gradient usually needs more.

Questions

1. What does momentum add to each gradient descent step?
 - a. A fraction of the previous move
 - b. A second gradient computed at a random point
 - c. The derivative of the gradient, the Hessian
 - d. A fixed amount in the direction of the first step

2. Why does momentum reduce the wobbling of plain gradient descent?
 - a. Because it uses a larger step size
 - b. Because it ignores the gradient after the first step
 - c. Because when the gradient flips back and forth, the remembered move works against the new one
 - d. Because it computes the gradient on more examples
3. With a momentum coefficient of three tenths, how much does the gradient from three steps back count?
 - a. Three tenths
 - b. Nothing at all
 - c. Nine tenths
 - d. Nine hundredths
4. What is the memory cost of momentum?
 - a. It stores the previous move for every parameter, doubling the optimiser's memory
 - b. It needs no memory at all
 - c. It stores every gradient since the start of training
 - d. It stores the Hessian table

Answers

1. **a)** Each move is the plain downhill step plus the momentum coefficient times the previous move, starting from a previous move of zero.
2. **c)** Steady directions add up, while alternating ones partly cancel against the remembered part, smoothing the path.

- 3. d)** A gradient n steps back is scaled by the coefficient raised to n minus one; for three steps back that is three tenths squared, nine hundredths.
- 4. a)** Keeping one previous move per parameter doubles what the optimiser needs, though computing the gradient usually needs more memory still.

Worked examples

Momentum against a flipping gradient One dial, step size one tenth, momentum coefficient one half. The gradient is one, then minus one, then one. Compute the three moves and compare them with plain gradient descent.

The first move is minus one tenth times one, which is minus one tenth, since the previous move is zero.

The second move is minus one tenth times minus one, plus one tenth, plus half of minus one tenth: one tenth minus five hundredths, which is plus five hundredths.

The third move is minus one tenth plus half of five hundredths: minus seventy-five thousandths.

Plain gradient descent would move minus one tenth, plus one tenth, minus one tenth; with momentum the back-and-forth moves are smaller.

Answer: Moves of minus one tenth, plus five hundredths and minus seventy-five thousandths: the wobble shrinks.

Practice

1. With a step size of one tenth, a constant gradient of one and a momentum coefficient of nine tenths, what size do the moves approach?

Hint: The moves stop growing when a move equals one tenth plus nine tenths of itself.

Answer: One: if a move of size m satisfies m equals one tenth plus nine tenths of m , then one tenth of m equals one tenth, so m is one. The moves approach ten times the plain step.

2. A model has seven billion parameters. How many extra numbers does momentum have to store?

Hint: One previous move per parameter.

Answer: Seven billion: one remembered move for each parameter, doubling the optimiser's memory, though computing the gradient usually takes even more memory.

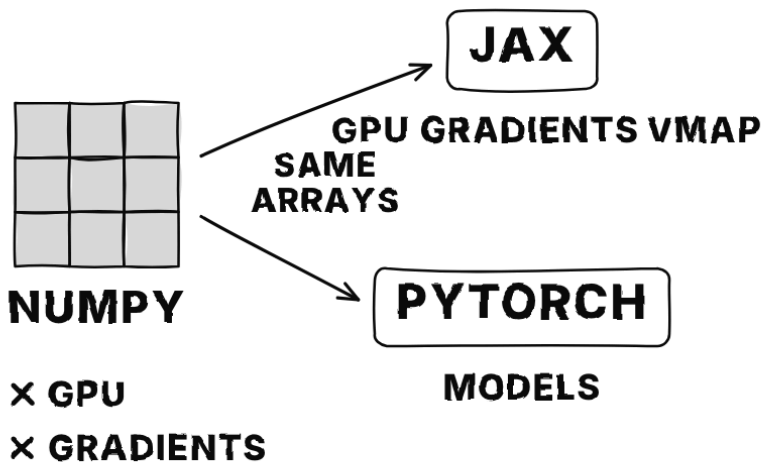
3. Explain in your own words why the book prefers acceleration methods without second derivatives or extra function calls.

Hint: Think about the size of a table of second derivatives for a model with millions of parameters.

Answer: For a model with millions of parameters, a table of second derivatives would have millions times millions of entries, and every extra evaluation of the function costs a full run over the data. Methods like momentum improve the direction using only gradients already computed.

From Paper to Code

Reading about a bicycle never taught anyone to ride one. The book knows this, and at the end of the chapter it hands you not a list of textbook exercises but a short self-learning path through two real programming tools, JAX and PyTorch, to follow alongside the reading. Solutions are to be published on the book's website, and the book warns that its many web links may have moved or expired by the time you look for them.



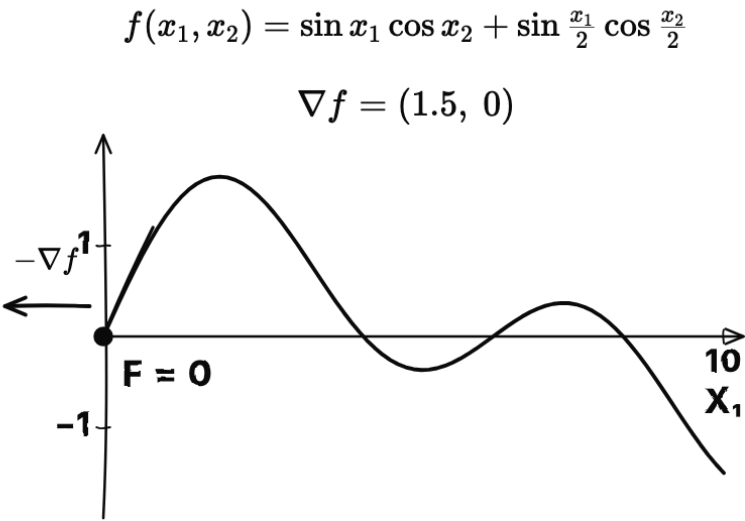
NumPy, Then JAX and PyTorch

The path starts with NumPy, the classic library for working with multidimensional arrays, the tensors of this book, and for indexing and transforming their contents. The book suggests reading NumPy's quick-start guide until handling arrays feels natural, especially slicing them, and practising with a well-known, gently paced collection of one hundred NumPy exercises.

NumPy, influential as it is, has two limits. On its own it makes little use of parallel hardware such as graphics chips, and it cannot compute gradients automatically, the topic of chapter six. JAX copies NumPy's way of writing things and adds broad hardware support, automatic gradients, and extra transformations, such as one that takes a function written for a single example and runs it over a whole batch, called the vectorised map. PyTorch is also built around a NumPy-like core, with small differences in names and behaviour, plus higher-level tools for building differentiable models. The book suggests skimming how each library creates arrays, to see how much they share with NumPy, and ignoring the higher-level model-building parts for now; chapter six explains how these frameworks are designed. The chat assistant and the photo app you use were very likely trained with exactly these tools.

Then comes the real exercise, to be done three times, once in NumPy, once in JAX and once in PyTorch, each a matter of minutes once the syntax is familiar. The function to explore takes two numbers, each between zero and ten, and returns the sine of the first times the cosine of the second, plus the sine of half the first times the cosine of half the second. It is a wavy landscape with several hills and dips. The book adds a remarkable footnote: this function was generated by asking ChatGPT for a nice function with several minima and maxima, and

nothing else in the book was generated by a language model, a disclaimer the author feels is becoming important to make.



A Wavy Function and Its Slope

At the starting corner, both inputs zero, every sine is zero and the function is zero. Its slope along the first input there is the cosine of zero times the cosine of zero, one, plus one half times the same kind of product, one half: one and a half in total. Along the second input the slope involves sines of zero, so it is zero. The gradient at the corner is therefore one and a half, zero, and gradient descent would step from there in the opposite direction, the way the function falls.

The four steps of the exercise are these. First, write the function so that it takes a whole table of n points, one per row, and returns n

values at once, which is called vectorised. Second, write by hand a second function that returns its gradient, since automatic differentiation has not yet been introduced. Third, run gradient descent from several starting points and draw the paths each one follows over the landscape; in JAX or PyTorch, the vectorised map is a good tool to try here. Fourth, add a momentum term and plot the length of the gradient as the steps go by: it should shrink towards zero as the walk approaches a stationary point.

Try it yourself

Even without programming, you can try the spirit of the exercise with a spreadsheet. Put the two inputs in two cells, compute the function in a third, and the two slopes in two more cells. Then make each new row equal to the previous inputs minus one tenth times the slopes, copy the row down a few dozen times, and watch the value fall and the slopes shrink towards zero.

Remember

The chapter ends with a self-learning path instead of classic exercises, with solutions planned on the book's website. It starts with NumPy arrays and indexing, then moves to JAX, which adds hardware support, automatic gradients and the vectorised map, and PyTorch, which adds tools for building models. The exercise uses a wavy two-input function, generated with ChatGPT, the only language-model output in the book. In each library: compute the function on many points at once, code its gradient by hand, draw gradient descent paths from several starts, then add momentum and watch the gradient's length shrink to zero.

Questions

1. What does JAX add to NumPy's way of working with arrays?
 - a. Broad hardware support, automatic gradients and transformations such as the vectorised map
 - b. A new way of writing arrays that has nothing in common with NumPy
 - c. Only faster printing of results
 - d. High-level model building, but no gradients
2. In the exercise, why must the gradient be written by hand for now?
 - a. Because the function has no gradient

- b. Because hand-written gradients are always faster
 - c. Because automatic differentiation has not been introduced yet; it comes in chapter six
 - d. Because NumPy forbids automatic gradients
3. What should happen to the length of the gradient as gradient descent with momentum runs on the exercise's function?
- a. It should grow steadily
 - b. It should stay constant
 - c. It should jump between zero and one
 - d. It should shrink towards zero as the walk approaches a stationary point
4. What does it mean to write the function in a vectorised way?
- a. To draw it as arrows
 - b. To make it take a whole table of points, one per row, and return all their values at once
 - c. To store it as a single long vector
 - d. To compute only its gradient

Answers

- 1. a)** JAX copies NumPy's interface and adds hardware support, automatic computation of gradients and extra transformations like the vectorised map.
- 2. c)** The exercise comes before automatic differentiation, so the slope functions are hard-coded.
- 3. d)** At a stationary point the gradient is zero, so its length falls towards zero as the walk settles.

4. b) A vectorised function takes n points at once and returns n values, instead of handling one point at a time.

Worked examples

One step on the exercise landscape At the point where the first input is π , about three point one four, and the second is zero, the function is zero plus one, which is one, and its gradient is minus one, zero. Take one gradient descent step with step size one tenth and check that the function falls.

Gradient descent subtracts the step size times the gradient: the first input becomes π minus one tenth times minus one, π plus one tenth, about three point two four.

The second input stays zero, since its slope is zero.

At the new point the first term is the sine of about three point two four, roughly minus one tenth, and the second term is the sine of about one point six two, roughly nought point nine nine nine.

The new value is about nought point nine, lower than the starting value of one, as a descent step should be.

Answer: The step moves to about three point two four, zero, and the value falls from one to about nought point nine.

Practice

1. Evaluate the exercise's function at the starting corner, both inputs zero, and give its gradient there.

Hint: Every sine of zero is zero and every cosine of zero is one.

Answer: The function is zero, since both terms contain the sine of zero. The slope along the first input is one plus one half, one and a half; along the second it is zero. The gradient is one and a half, zero.

2. Why does the book ask for the same exercise in three different libraries?

Hint: Think of what changes and what stays the same between NumPy, JAX and PyTorch.

Answer: The three libraries share a NumPy-like core, so repeating a short exercise in each makes you fluent in all of them and shows how little differs between them, while revealing what JAX and PyTorch add.

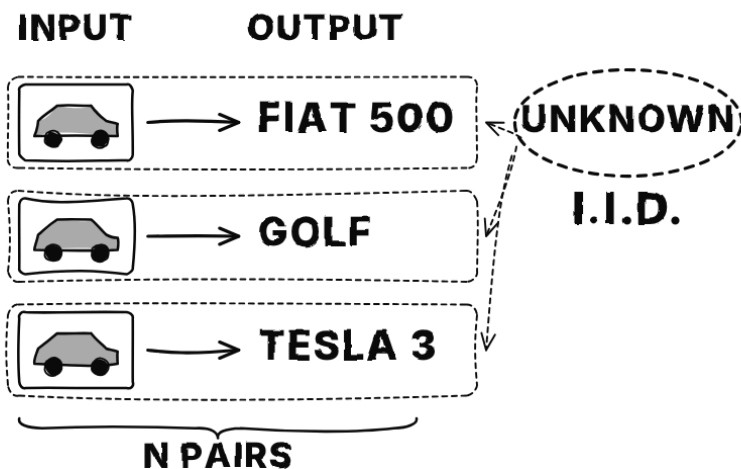
3. Describe how you would follow gradient descent in a spreadsheet for this function.

Hint: Each new row is computed from the previous one.

Answer: Put the two inputs in a row with the function and its two hand-written slopes. Make each next row's inputs equal to the previous inputs minus one tenth times the slopes, recompute value and slopes, and copy the row downwards; the value should fall and the slopes shrink towards zero.

What Makes a Good Pile of Examples

Suppose you want an app that names the model of a car from a photo. Writing the rules by hand is hopeless: no one can describe in code every curve of every car from every angle. But collecting examples is easy: walk the streets with a camera and write the model under each picture. This is the situation the whole of supervised learning starts from, and chapter three, short but described by the book as the backbone of what follows, makes it precise. It covers datasets, losses, what it means to minimise the average error on the examples, the assumptions behind it all, and finally a version in the language of probability built on maximum likelihood.

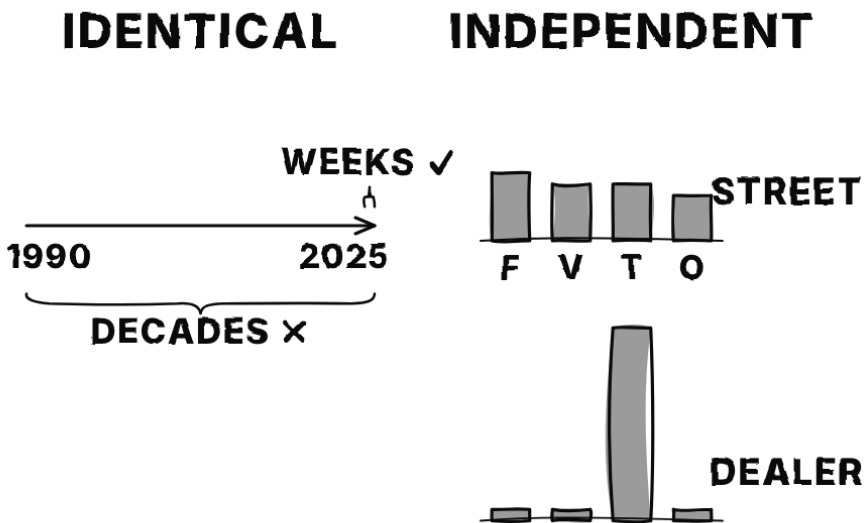


A Dataset of Pairs

A supervised dataset is simply a collection of n pairs, each pair one input together with the output we want for it: a photo and the car model in it. Such problems are everywhere, from turning speech into text to steering a robot. The photo app on your phone was trained on exactly such a collection: millions of pictures, each paired with a description of what is in it. The definition comes with one extra assumption that sounds technical: every pair is drawn independently from the same unknown probability distribution, and that distribution can never be known exactly. This is called the i.i.d. assumption, short for independent and identically distributed.

Identically distributed means that what we are trying to imitate stays stable over time. For the car app, photos collected over a few weeks

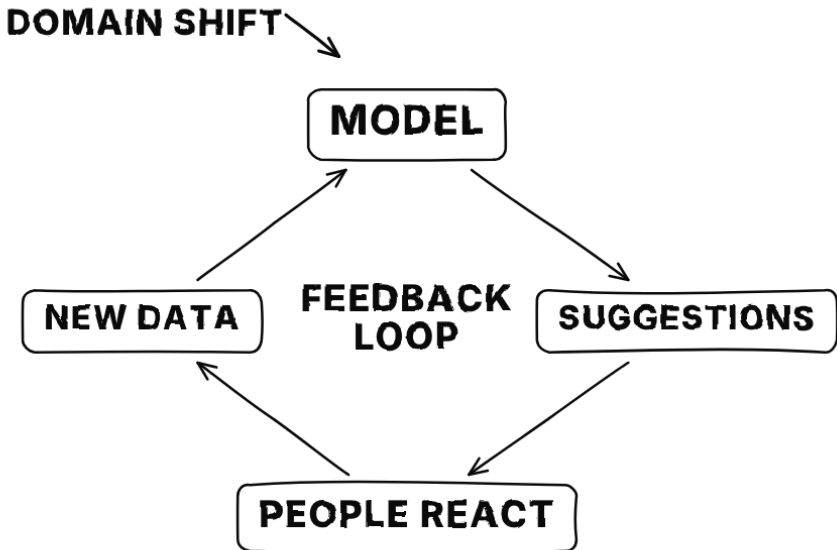
satisfy this. Photos collected over the last few decades do not, because the cars on the road have changed: a model trained on them would fail to recognise new models, or would perform worse than it should.



Stable in Time, Fair in Sample

Independent means the collection has no bias and fairly represents the whole range of cases. If you took all your photos outside a Tesla dealership, you would have far too many Teslas and far too few cars of other makes. Whether the assumption holds also depends on where the model is used. A car collection gathered in Italy may serve perfectly well in Rome or Milan, and poorly in Tokyo or Taiwan, where different cars fill the streets. So the assumption must always be checked before trusting the tools of supervised learning. The book adds that today's large language models are trained on such vast and

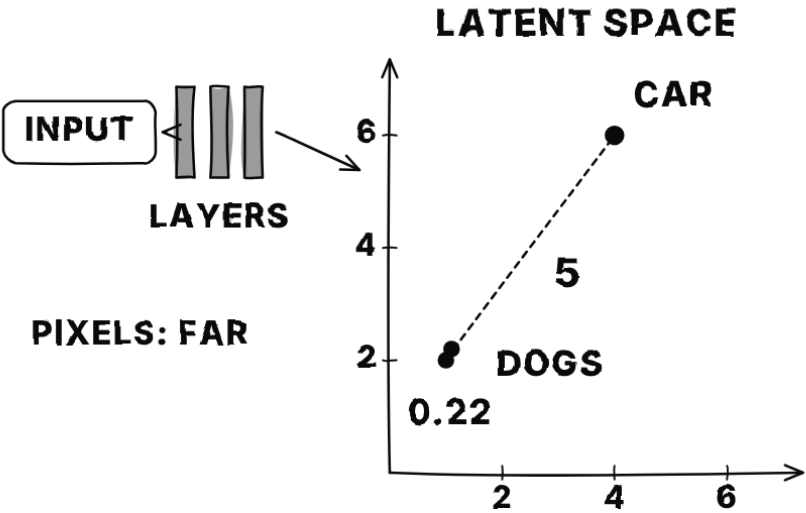
varied collections that it becomes hard to say which tasks are inside their training distribution and which are outside it, and how far they truly generalise.



A Recommender Feeds Itself

Checking this is not done once and forgotten, as a box in the book insists; it has to continue for the whole life of a model. If nobody watches, the mix of cars on the road slowly changes and the car app quietly gets worse, a problem called domain shift. A subtler case is a recommender system, the part of an app that suggests what to watch or buy. Once it starts suggesting, people react to its suggestions, so the data it learns from next is partly shaped by its own earlier choices. That is a feedback loop, and it forces constant re-checking of how well the system and the app perform.

The chapter's opening figure looks ahead to something remarkable. A differentiable model transforms its input step by step with operations from linear algebra. After training, the intermediate versions of the data inside such a model, in what is then called a pre-trained model, often have a useful geometry: pictures with similar meaning end up as points close to each other in this hidden, or latent, space. Raw images have no sensible notion of distance; two photos of the same dog, taken a step apart, can differ in nearly every pixel. Moving data from such a space into a space with meaningful distances, a metric space, is called embedding the data. Let us measure it on a toy example: if one dog photo lands at the point one, two and another at one point one, two point two, while a car photo lands at four, six, then the two dogs are about two tenths apart and the car is about five away.



Close Meanings, Close Points

Remember

A supervised dataset is a collection of input-output pairs showing the behaviour we want, used when gathering examples is easy but writing the rules is not. The pairs are assumed to be drawn independently from one unchanging, unknown distribution, the i.i.d. assumption. Identically distributed means the target stays stable over time; independent means the collection is unbiased and representative, which also depends on where the model will be used. The assumption must be watched for a model's whole life, because of domain shift and of feedback loops such as a recommender reacting to its own effects. Trained models often embed data into a latent space where similar meanings sit close together, a space with meaningful distances.

Questions

1. A car-recognition dataset is built from photos taken over the last forty years. Which part of the i.i.d. assumption does it break?
 - a. Identically distributed: the cars on the road changed over time
 - b. Independent: the photos are too far apart
 - c. Neither: older photos are always better
 - d. Both, because photos cannot be independent

2. All photos are taken outside one Tesla dealership. What is wrong with the dataset?
 - a. It changes too quickly over time
 - b. It is biased and not representative, breaking the independence assumption
 - c. It contains too many different car makes
 - d. It is too small to embed
3. Why must the i.i.d. assumption be checked during the whole life of a model?
 - a. Because datasets grow automatically
 - b. Because the model forgets its training
 - c. Because the data can drift, as in domain shift, or be reshaped by the system itself, as in feedback loops
 - d. Because the assumption is only needed after training
4. What is embedding data?
 - a. Storing data inside a model's weights
 - b. Compressing a photo into fewer pixels
 - c. Adding labels to raw examples
 - d. Mapping data into a space with meaningful distances, where similar items sit close together

Answers

1. **a)** Identically distributed means the target stays stable over time; decades of changing car models violate it, so the model would fail on new cars.
2. **b)** Independent sampling means no bias in collection; a dealership gives far too many Teslas and too few other cars.

3. c) Slow changes in the world degrade performance, and systems like recommenders influence the data they later learn from.

4. d) Raw images have no sensible distance; embedding moves them into a metric space where semantically similar items are near each other.

Worked examples

Distances in an embedding In a latent space, one dog photo lands at the point one, two, another dog photo at one point one, two point two, and a car photo at four, six. Which photos are close?

Between the two dogs, the differences are one tenth and two tenths.

Their squares are one hundredth and four hundredths, which add up to five hundredths, and the square root is about two tenths.

Between the first dog and the car, the differences are three and four, with squares nine and sixteen, adding up to twenty-five.

The square root of twenty-five is five, so the car is far from the dogs, while the dogs are close together.

Answer: The two dogs are about two tenths apart; the car is five away from the first dog.

Practice

1. A car dataset collected in Italy is used for an app in Milan and another in Tokyo. Will the i.i.d. assumption hold in each case? Explain.

Hint: Ask whether the cars seen in use resemble the cars in the collection.

Answer: In Milan it plausibly holds, since the cars on the road resemble those in the Italian collection. In Tokyo it may fail, because the mix of makes and models differs, so the collection is not representative of what the app will see.

2. Explain in your own words how a recommender system can create a feedback loop.

Hint: Where does the data it learns from next come from?

Answer: Once the recommender suggests items, people click partly because of those suggestions, so the next data it learns from reflects its own earlier choices rather than people's independent tastes. Its performance and the app's must therefore be re-checked constantly.

3. Why is a raw image a poor space for measuring similarity, and what does an embedding offer instead?

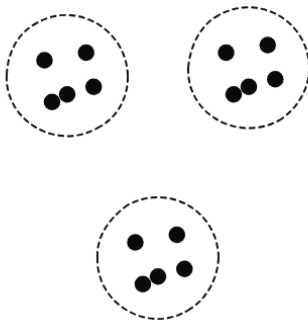
Hint: Think of two photos of the same dog taken a step apart.

Answer: Two photos of the same dog can differ in nearly every pixel, so pixel-by-pixel distance says little about meaning. An embedding from a trained model places similar images close together, so distance in that space reflects similarity of content.

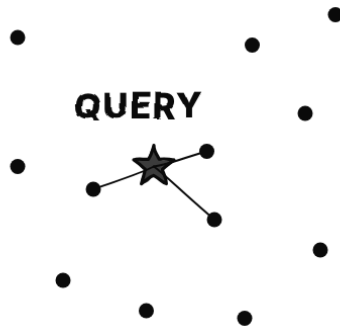
Learning Without Labels, and Reusing What Was Learned

Open the photo app and look at the album that gathers pictures of the same person together, although you never told it who anyone is. Nobody labelled those faces; the app simply put similar pictures in the same pile. That is not supervised learning in the strict sense, because there were no target outputs to imitate. Most successful applications use supervised learning in one form or another, but the book now surveys its many relatives.

NO LABELS

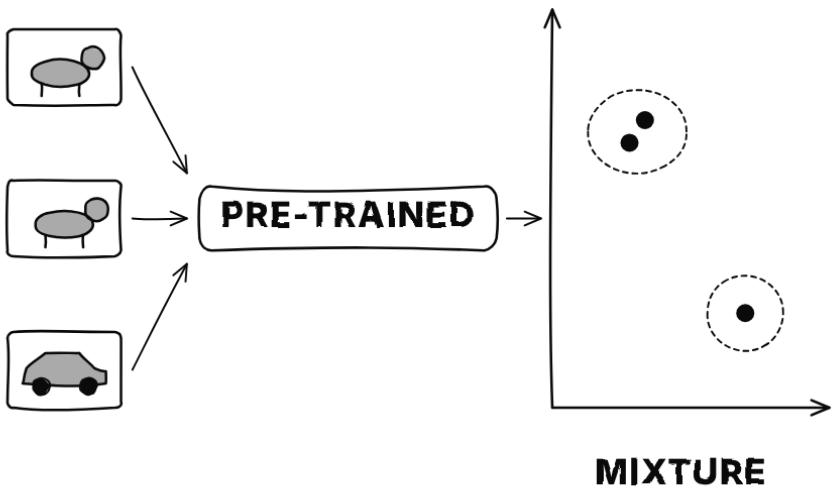


CLUSTERS



K NEAREST

When a dataset has inputs but no targets, the setting is called unsupervised learning. A typical task is clustering: grouping the data into piles so that items within a pile are similar and items in different piles are not. Another is retrieval: given a query from a user, find in a large collection the k items most similar to it, say the ten closest. With images both tasks are tricky, because distance between raw pictures is meaningless: a tiny shift can change millions of pixels.

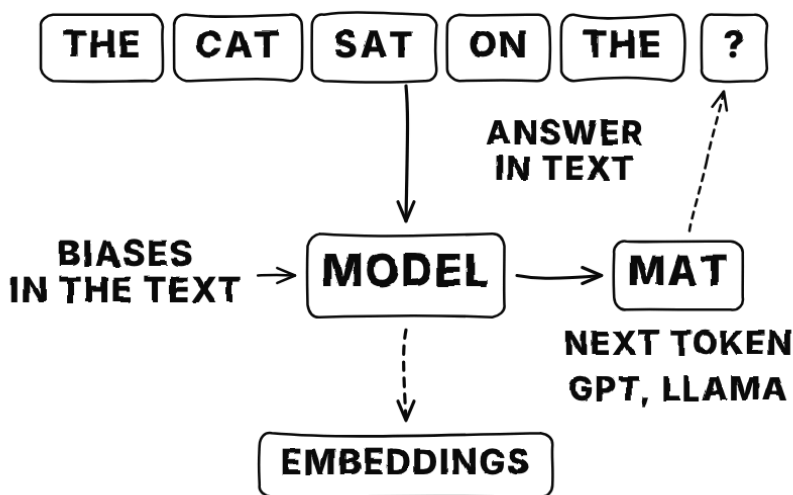


Embed, Then Cluster

The trick is to borrow a model already trained on some broad task, such as naming objects in photos, called a pre-trained model. Its internal states can be read as vectors in a space with many dimensions, and in many cases these vectors have a helpful geometry: things that mean similar things are embedded at nearby points. So

one can feed these vectors to an ordinary clustering method, such as a Gaussian mixture model, which describes the data as a blend of a few bell-shaped clouds, one per pile. The book's figure shows exactly this: images go through the pre-trained model, and their vectors gather into groups by meaning.

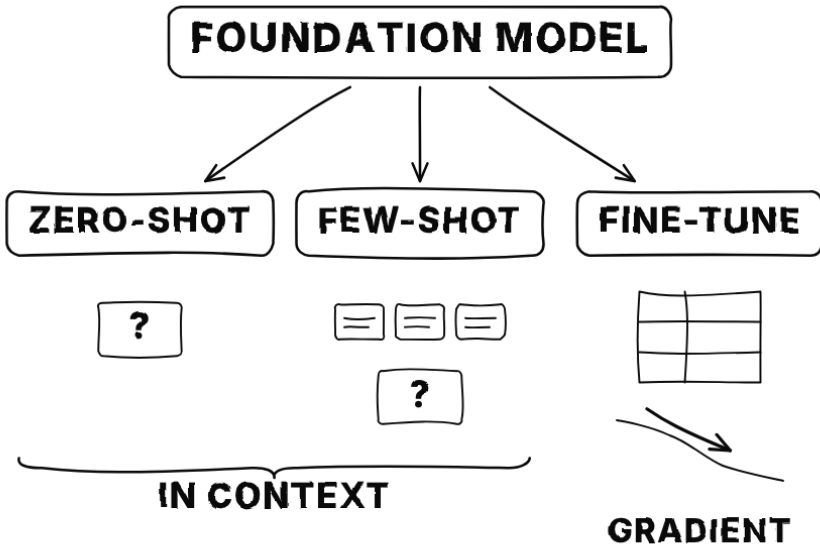
What if no pre-trained model is available? Then a very popular variant called self-supervised learning invents a supervised task from unlabelled data, in order to train a model that can later serve many other tasks. With a large body of text, for instance, one can always train a program to predict how a short piece of text continues, because the true continuation is right there in the text. The discovery that networks trained this way also produce excellent embeddings of text had, in the book's words, a profound impact on the community. A footnote adds a sober point: large datasets scraped from the web are full of biases, profanity and vulgar content, models trained on them absorb these biases, and this is one of the main criticisms of closed-source foundation models.



Predict What Comes Next

Large language models such as GPT and Llama are modern versions of this idea: their training always starts with self-supervised next-token prediction, guessing the next piece of text over and over. Such models are sometimes called foundation models, and the book's second figure shows three ways to use them. In zero-shot use, you simply ask the model a question about a new task. Asking a chat assistant to summarise an e-mail, without any example, is zero-shot use. In few-shot prompting, you include a few examples of the new task in the prompt itself, for instance three e-mails with their summaries, followed by the one you want summarised. Both work only if the model generalises very well. When no training is needed in this way, the model is said to be learning in context. In fine-tuning, finally, you take the pre-trained model and continue training its

parameters with gradient descent on a small dataset for the new task, in the same way as training from scratch. The book focuses on building models from scratch, but notes that fine-tuning uses the same means.



Three Ways to Use a Model

Fine-tuning is made easy by large open-source collections of models online, and it need not touch every parameter. Training only a small subset of the parameters, or a small number of added ones, is called parameter-efficient fine-tuning; the book leaves it to its second volume. To see why it matters, imagine adjusting only one percent of a model with seven billion parameters: that is seventy million numbers to train instead of seven billion.

The list of variants goes on, and the book only hints at them. If only part of a dataset carries labels, the setting is semi-supervised, with examples in chapter twelve. With several datasets from similar distributions, or from the same distribution at different times, a whole family of problems appears, depending on the order in which tasks and data arrive: adapting to a new domain, learning to learn, learning continually, learning distances, and even unlearning; some of these are promised for the next volume.

Remember

Without targets, learning is unsupervised: clustering groups similar items and retrieval finds the closest items to a query, both hard on raw images. A pre-trained model embeds data so that similar meanings sit close together, which lets ordinary clustering such as Gaussian mixtures work on its vectors. Self-supervised learning invents a task from unlabelled data, such as predicting how text continues, and is how large language models like GPT and Llama begin; web data also carries biases into them. Foundation models can be used zero-shot, with a few examples in the prompt, or fine-tuned by gradient descent, fully or parameter-efficiently. Semi-supervised learning and many other settings, such as continual learning or unlearning, extend the picture.

Questions

1. Why is clustering raw images difficult without a pre-trained model?
 - a. Because images have no pixels
 - b. Because distance between raw pictures is meaningless: a tiny shift changes millions of pixels
 - c. Because clustering only works on text
 - d. Because images cannot be stored as vectors
2. How does self-supervised learning get a training task from unlabelled text?
 - a. By asking people to label each sentence
 - b. By clustering the words into piles
 - c. By predicting how a short piece of text continues, since the true continuation is in the text
 - d. By fine-tuning a model that already exists
3. You give a chat assistant three e-mails with their summaries and then a fourth e-mail to summarise, without any training. What is this called?
 - a. Fine-tuning
 - b. Zero-shot use
 - c. Semi-supervised learning
 - d. Few-shot prompting, a form of in-context learning
4. What distinguishes fine-tuning from zero-shot and few-shot use?
 - a. The model's parameters are trained further by gradient descent on a small dataset
 - b. The model receives no input at all

- c. The model is trained from scratch on the web
- d. The model is used only for clustering

Answers

1. **b)** Pixel distances do not reflect meaning; a pre-trained model's embedding places similar images close together, so clustering can work on those vectors.
2. **c)** The data supplies its own targets: the actual continuation of each piece of text is the answer to predict.
3. **d)** Including a few examples of the task in the prompt is few-shot prompting; no parameters change, so the model learns in context.
4. **a)** Fine-tuning continues training the pre-trained parameters with gradient descent, while zero-shot and few-shot use leave them unchanged.

Worked examples

How much parameter-efficient fine-tuning saves A model has seven billion parameters. If fine-tuning adjusts only one percent of them, how many numbers are trained?

One percent means one hundredth.

Seven billion divided by one hundred is seventy million.

So seventy million numbers are adjusted, while the other six billion nine hundred and thirty million stay as they were.

That is a hundred times fewer numbers to train than full fine-tuning.

Answer: Seventy million parameters instead of seven billion.

Practice

1. Classify each situation: grouping customers by buying habits with no labels; a dataset where only a tenth of the photos are labelled;

training a model to fill in hidden words of sentences.

Hint: Unsupervised, semi-supervised or self-supervised.

Answer: Grouping without labels is unsupervised learning, specifically clustering. A partly labelled dataset is semi-supervised. Filling in hidden words invents targets from the data itself, which is self-supervised learning.

2. Why does the book connect self-supervised learning with large language models such as GPT and Llama?

Hint: How does their training begin?

Answer: Their training always starts with self-supervised next-token prediction on huge amounts of text, which gives them general abilities and good embeddings; they are then used zero-shot, few-shot or after fine-tuning.

3. What criticism does the book raise about models trained on large web datasets?

Hint: Think about what such datasets contain.

Answer: Web data is full of biases, profanity and vulgar content, and models trained on it absorb those biases; this is one of the major criticisms of closed-source foundation models.

Scoring a Prediction

Ask three friends to guess the price of a flat that actually sold for three hundred thousand. One says three hundred and ten, one says two hundred and fifty, one says three hundred exactly. Who did best is obvious, but to let a computer improve its guesses automatically, "best" has to become a number that gets smaller as guesses get better. That number is the loss, and once data has been gathered, choosing it is how we say precisely what we mean by imitating the desired behaviour.



Three Guesses, Three Losses

A loss function takes the target and the model's prediction and returns a single number, and it must be differentiable. Its value tracks quality: a smaller loss for one prediction than for another means the first prediction is better, measured against that target. One common choice, for instance, is the squared difference: the guesses above score ten squared, a hundred; fifty squared, two thousand five hundred; and zero. The loss carries our understanding of the task and our preferences among solutions, written on a scale of real numbers that an optimiser can work with. Because it is differentiable, learning becomes a mathematical optimisation problem that gradient descent can solve.

The natural goal is the smallest possible average loss over the whole dataset. Take each example, compare the model's prediction with the target, compute the loss, add up all n losses and divide by n ; then look for the model that makes this average as small as it can be. With our three guesses as three predictions on three examples, the average squared loss would be one hundred plus two thousand five hundred plus zero, over three, about eight hundred and sixty-seven. For historical reasons this goal is called empirical risk minimisation, where risk is simply another word for loss, and empirical means measured on the examples we actually have.

$$\frac{100 + 2500 + 0}{3} \approx 867$$

EMPIRICAL RISK

$$L(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(y_i, f(x_i, \theta))$$

θ **PARAMETERS**

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

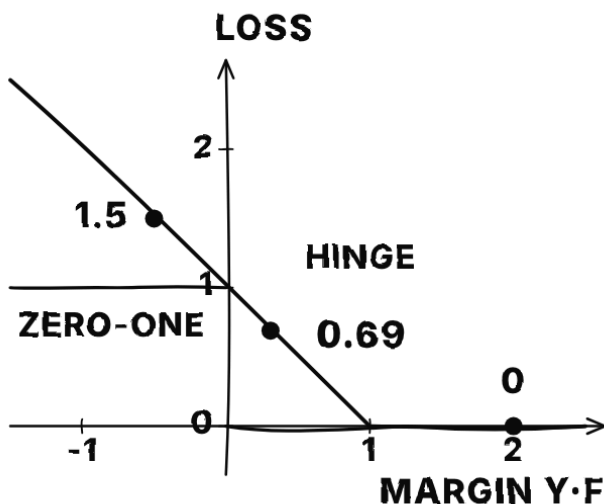
GRADIENT DESCENT

Average the Losses

Written that way, the search seems to range over every possible function of the input. In practice, the book explains, a model is always described by a set of tensors, its parameters, so the model becomes a function of both the input and the parameters. Training then means searching for the parameter values that minimise the average loss, and the search is done with gradient descent on the parameters. Training the photo app is exactly this: millions of labelled pictures, a loss that is small when the app names a picture correctly, and gradient descent turning the parameters to shrink the average.

The book then stresses why the loss must be differentiable. Consider telling cats from dogs, with targets plus one for cat and minus one for dog, and a model that outputs any real number whose sign is its

decision. The most natural score is the zero-one loss: zero when the sign is right, one when it is wrong. It matches our idea of being right perfectly, yet it is useless for training, because its slope is zero almost everywhere: nudging the output from three tenths to thirty-one hundredths changes nothing, so gradient descent never moves from where it starts. A less intuitive but better quantity is the margin, the target times the prediction, positive when the sign is right and negative when it is wrong, and changing smoothly. A loss built on it is the hinge loss: zero once the margin reaches one, otherwise one minus the margin. For a cat, a prediction of two has margin two and costs nothing; a prediction of three tenths is right but timid and costs seven tenths; a prediction of minus one half is wrong and costs one and a half. And now a small nudge from three tenths to thirty-one hundredths lowers the cost to sixty-nine hundredths, which gradient descent can feel. The hinge loss is the one used to train support vector machines, and the example shows the built-in tension between losses that express what we want and losses that optimisers can use.



Zero-One Loss Against Hinge

A box gives the background: empirical and expected risk minimisation are associated with the Russian computer scientist Vladimir Vapnik, whose work founded statistical learning theory. That theory studies how well the average loss on a finite set of examples stands in for the true average over all possible data, for restricted kinds of models. Modern networks, which keep generalising well long after one would expect them to overfit, have opened many new questions for it.

Remember

A loss function compares a prediction with its target and returns one differentiable number that is smaller for better predictions; it encodes what we want from the model. Training minimises the average loss over the dataset, called empirical risk minimisation, by searching over the model's parameters with gradient descent. The zero-one loss matches "being right" but has zero slope almost everywhere, so gradient descent cannot use it. The margin, target times prediction, varies smoothly, and the hinge loss built on it, used for support vector machines, gives gradient descent something to follow. Risk minimisation comes from Vapnik's statistical learning theory, which modern networks' surprising generalisation keeps challenging.

Questions

1. Why is the zero-one loss useless for gradient descent?
 - a. Its slope is zero almost everywhere, so small nudges never change it and the parameters never move
 - b. It is always equal to one
 - c. It cannot be averaged over a dataset
 - d. It gives larger values to better predictions
2. What does empirical risk minimisation minimise?
 - a. The largest loss on any single example
 - b. The loss on the one hardest example

- c. The average loss over the examples in the dataset
 - d. The number of parameters of the model
3. For a cat (target plus one), the model outputs three tenths.
What is its hinge loss?
- a. Zero, because the sign is right
 - b. One and three tenths
 - c. Three tenths
 - d. Seven tenths, because the margin is below one
4. What does the margin, the target times the prediction, tell you?
- a. The size of the dataset
 - b. Whether the sign is right (positive) or wrong (negative),
changing smoothly
 - c. The number of parameters needed
 - d. The exact probability of being right

Answers

- 1. a)** The zero-one loss only jumps when the sign of the prediction switches; elsewhere a small change has no effect, so gradient descent stays stuck at the start.
- 2. c)** It averages the loss over all n examples and searches for the parameters that make that average smallest.
- 3. d)** The margin is one times three tenths; the hinge loss is one minus the margin while the margin is below one, so seven tenths.
- 4. b)** The margin is positive when the prediction's sign matches the target and negative otherwise, and unlike the zero-one loss it varies continuously.

Worked examples

Average hinge loss for three dog photos Three dog photos have target minus one. The model outputs minus two, minus four tenths and one half. Compute the hinge losses, their average, and compare with the zero-one loss.

The margins are the target times the output: two, four tenths, and minus one half.

Hinge losses: the first margin is at least one, so zero; the second is one minus four tenths, six tenths; the third is one minus minus one half, one and a half.

The average hinge loss is zero plus six tenths plus one and a half, two point one, divided by three, seven tenths.

The zero-one losses are zero, zero and one, average one third; it ignores that the second prediction was right but timid.

Answer: Hinge losses zero, six tenths and one and a half, average seven tenths; zero-one average one third.

Practice

1. The flat sold for three hundred thousand. Using the squared difference measured in thousands, which guess has the lower loss, two hundred and ninety or three hundred and twenty, and by how much?

Hint: Square the differences ten and twenty.

Answer: Two hundred and ninety costs ten squared, one hundred; three hundred and twenty costs twenty squared, four hundred. The first guess is better, with a loss three hundred lower.

2. For a cat, the model's output moves from one half to six tenths. How do the zero-one loss and the hinge loss change?

Hint: Both outputs have the right sign; compute one minus the margin for the hinge loss.

Answer: The zero-one loss stays at zero, since the sign was already right. The hinge loss falls from one half to four tenths, so gradient descent gets a signal to push further.

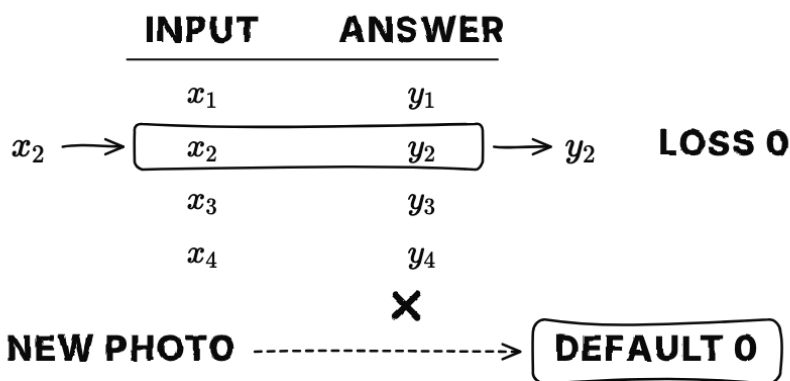
3. Explain in your own words the tension the book describes when choosing a loss.

Hint: Compare what we want to measure with what an optimiser can use.

Answer: The loss should express what we really care about, such as being right, but it must also be differentiable and informative for gradient descent. The most natural measure, the zero-one loss, gives no slope, so we use substitutes such as the hinge loss that track performance while changing smoothly.

Memorising Is Not Learning

Here is a model that never makes a mistake on its training data. Keep the whole dataset in a table; when an input arrives that is in the table, return its stored answer; when anything else arrives, return some fixed default, say zero. On the training examples its predictions are exactly the targets, so, as long as the loss is lowest when prediction equals target, its average loss on the dataset is the lowest possible. And it is worthless: show it one new photo and it answers zero. The book uses this look-up table to make a point that runs through all of machine learning: minimising the loss on the examples is not the real goal.

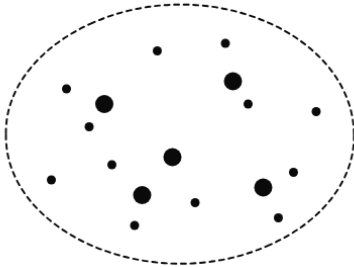


MEMORISING $\neq$ LEARNING

A Look-Up Table Learns Nothing

What we truly want is a small loss on future inputs we have not seen yet; the training examples are only a stand-in for them. The book formalises this as the expected risk: the average loss over all the input-output pairs the model could ever meet, weighted by how likely each one is, for instance over every e-mail that could ever be written. A model with a low expected risk would be guaranteed to work. But it cannot be computed: we can never list and label every possible input. The empirical risk, the average over our dataset, is an estimate of it, what mathematicians call a Monte Carlo approximation, an average over random samples standing in for an average over everything. It works like rolling a die many times: the average of your rolls comes close to three and a half, the true average, without listing every possible outcome.

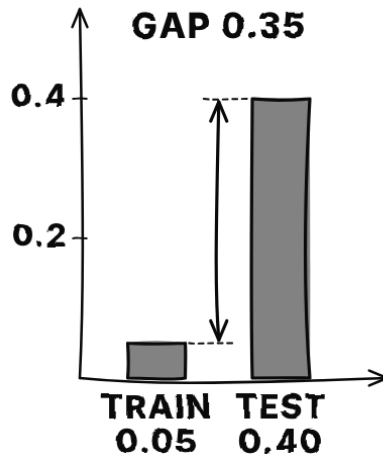
ALL POSSIBLE DATA



OUR DATASET

$$\begin{array}{|c|} \hline \cdot \\ \cdot \\ \cdot \\ \hline \end{array} \approx 3.5$$

OVERFITTING

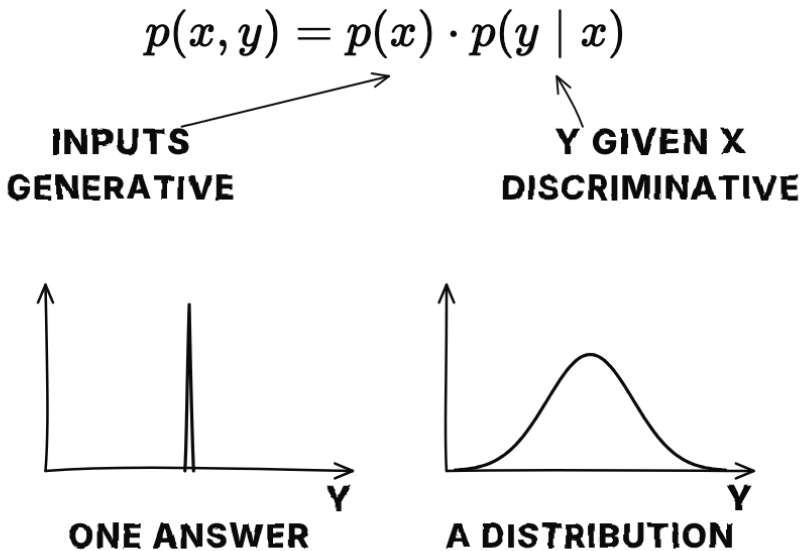


The Generalisation Gap

The difference between the expected risk and the empirical risk is called the generalisation gap. A pure memoriser, like the look-up table, has a huge gap: it fits its particular training data and fails beyond it, which is what overfitting means. In practice the gap is measured with a separate test set, examples kept aside and never used during training. If the average loss is five hundredths on the training set and forty hundredths on the test set, the gap of thirty-five hundredths is a rough measure of how much the model overfits. Before a new version of the photo app is released, its makers check it on pictures it never saw during training, for exactly this reason.

How should the loss itself be chosen? The book suggests this as a good moment to read its appendix on probability, especially distributions,

sufficient statistics and maximum likelihood. The loss encodes what we know in advance about the task and strongly affects performance. Sometimes common sense is enough, as with the hinge loss in the last part. But there is a more principled route that recasts the whole of training in the language of probability, a viewpoint many books prefer, and the book now lays out its basic idea.

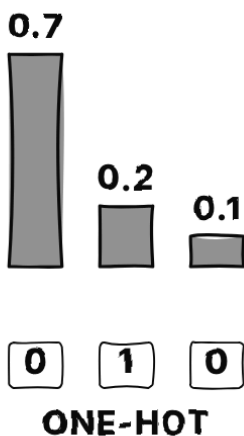


Splitting the Joint Distribution

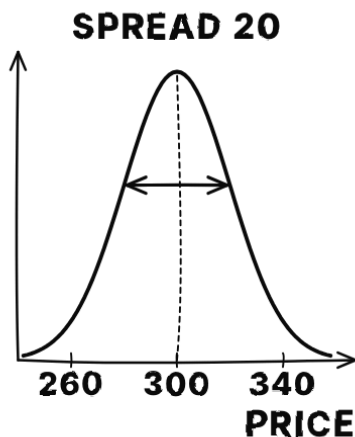
We assumed the examples come from a joint distribution over inputs and outputs. By the product rule of probability, it splits into two pieces: how likely each input is, and how likely each output is once the input is known, the conditional distribution of the output given the input. A footnote adds that it could be split the other way too; methods that model the inputs are called generative, those that model

only the output given the input are called discriminative, and apart from language models, the book concentrates on the second kind. Using a model that returns a single prediction for each input makes sense when, for each input, almost all the probability sits on one output, a spike that mathematicians call a Dirac delta. That simplifies everything, but it can be relaxed.

Instead of producing the prediction itself, the model can produce the numbers that describe a whole distribution over possible outputs, its sufficient statistics. Take a classification with three classes. The model returns three probabilities that add up to one, a vector in the simplex met earlier, say seven tenths, two tenths and one tenth. The true class is written as a one-hot vector: all zeros except a one at the position of the class. The probability the model gives to the true class is then the product of each predicted probability raised to the matching entry of the one-hot vector, which simply picks out the predicted probability of the true class. If the true class is the second, the one-hot vector is zero, one, zero, and the product is two tenths to the power one, times the others to the power zero, which is two tenths.



$$0.7^0 \cdot 0.2^1 \cdot 0.1^0 = 0.2$$



Predicting a Distribution

For predicting a single number, called regression, the model can return two numbers instead: the centre and the spread of a bell curve, a Gaussian, over the possible outputs. For the flat-price example, the model might say: around three hundred thousand, give or take twenty thousand. That way the model expresses not just a guess but how sure it is, and the next part shows how such probabilistic predictions lead to losses in a principled way.

Remember

A look-up table of the training data reaches the lowest possible empirical risk and is useless, which separates memorising from learning. The real target is the expected risk, the average loss over all possible data, which cannot be computed; the empirical risk is a Monte Carlo estimate of it. The difference between them is the generalisation gap; overfitting means a large gap, measured in practice with a separate test set never used in training. In the probabilistic view, the joint distribution splits into a distribution over inputs and a conditional distribution of outputs given inputs, which discriminative models learn. A model can output the description of a distribution, such as class probabilities scored against a one-hot target, or the centre and spread of a Gaussian for regression.

Questions

1. Why is the look-up table model worthless even though its empirical risk is the lowest possible?
 - a. Because it only reproduces stored training answers and returns a default for every new input
 - b. Because its loss on the training data is very high
 - c. Because it cannot store the whole dataset
 - d. Because its gradient is too large
2. Why can the expected risk not be computed directly?

- a. Because the loss function is not differentiable
 - b. Because it would require listing and labelling every possible input
 - c. Because it is always zero
 - d. Because the test set is too small
3. A model's average loss is five hundredths on training data and forty hundredths on a held-out test set. What does this suggest?
- a. The model generalises perfectly
 - b. The test set was used for training
 - c. The model overfits: the generalisation gap is about thirty-five hundredths
 - d. The loss function is wrong
4. A model gives the probabilities seven tenths, two tenths and one tenth to three classes, and the true class is the third. What probability does it give the true class?
- a. Seven tenths
 - b. Two tenths
 - c. One
 - d. One tenth

Answers

1. **a)** It memorises the training pairs perfectly but has learned nothing about new inputs, so its expected risk, the real goal, is poor.
2. **b)** The expected risk averages over all possible input-output pairs, which can never be enumerated and labelled; the empirical risk estimates it instead.

3. **c)** The difference between test and training loss approximates the generalisation gap; a large gap signals overfitting.
4. **d)** With the one-hot vector zero, zero, one, the product picks out the predicted probability of the third class, one tenth.

Worked examples

Picking out the true class A model predicts the class probabilities one fifth, three fifths and one fifth for a photo whose true class is the second. Write the one-hot vector and compute the probability the model assigns to the true class.

The true class is the second, so the one-hot vector is zero, one, zero.

Raise each predicted probability to the matching entry: one fifth to the power zero is one, three fifths to the power one is three fifths, one fifth to the power zero is one.

Multiply the three results: one times three fifths times one.

The product is three fifths, simply the predicted probability of the true class.

Answer: Three fifths.

Practice

1. Explain the difference between memorising and learning, using the photo app as an example.

Hint: Think about what happens with a photo the app has never seen.

Answer: A memorising app would recall the labels of its training photos and fail on any new photo. A learning app captures the patterns behind the photos and labels new ones correctly; this is why its makers test it on photos never used in training.

2. Why is an average over a dataset a reasonable estimate of the expected risk, and when would it fail?

Hint: Recall the die-rolling comparison and the i.i.d. assumption.

Answer: Like averaging many die rolls to estimate three and a half, averaging the loss over examples drawn from the true distribution estimates the average over all data. It fails when the examples are not representative, for instance when the i.i.d. assumption is broken, or when the model has simply memorised them.

3. For predicting a flat's price, the model outputs a centre and a spread. What does 'three hundred thousand, give or take twenty thousand' express that a single number does not?

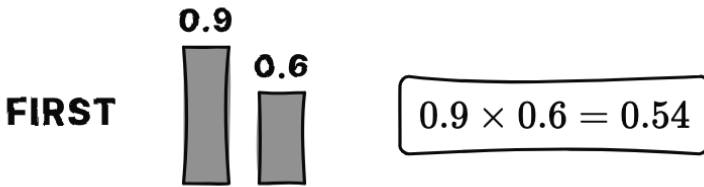
Hint: Think of the bell curve.

Answer: It describes a Gaussian distribution of possible prices: the most likely value and how uncertain the model is. A single number gives the guess but not the model's confidence in it.

The Most Likely Explanation, and Many Explanations at Once

Two weather forecasters have been predicting rain for a week. On the days it rained, the first had given rain a chance of nine in ten and six in ten; the second had said one half both times. Which forecaster do you trust more? Multiply the chances each gave to what actually happened: nine tenths times six tenths is fifty-four hundredths for the first, one half times one half is a quarter for the second. The first forecaster made what happened look much more likely, and that simple idea, called maximum likelihood, is how probabilistic models are trained.

$$\text{spread} = s^2$$



MAXIMUM LIKELIHOOD

Two Forecasters, Two Products

First, a detail left from the last part: when a model predicts a bell curve, its second output is squared to give the spread, so that the spread can never become negative. This probabilistic set-up is very general and includes everything said about losses so far. It gives the designer more freedom, since it can be easier to choose a shape for the distribution of outputs given an input than to invent a loss function directly, and it expresses uncertainty naturally, through the spread.

How is such a model trained? Because the examples are assumed independent and drawn from the same distribution, the probability a model assigns to the whole dataset is the product of the probabilities it assigns to each example's true output. That product is called the likelihood of the dataset. A model chosen at random spreads its

probabilities more or less at random and gives our particular dataset a small likelihood. So the sensible choice is the model that makes the likelihood as large as possible: the maximum likelihood solution. The difference from before is that the training goal now follows from the rules of probability once the distributions are chosen, instead of being designed by hand. For the photo app, this means preferring the settings under which the true labels of its training pictures look as probable as possible.

$$\text{likelihood} = p_1 p_2 \cdots p_n$$

MAKE IT LARGEST

$$0.9^{1000} \approx 1.7 \times 10^{-46}$$

↓

$$-\log \text{likelihood} = \sum_{i=1}^n -\log p_i$$

MINIMISE = A LOSS

$$-\log 0.9 \approx 0.105, \quad -\log 0.6 \approx 0.511$$

$$0.105 + 0.511 \approx 0.616 = -\log 0.54$$

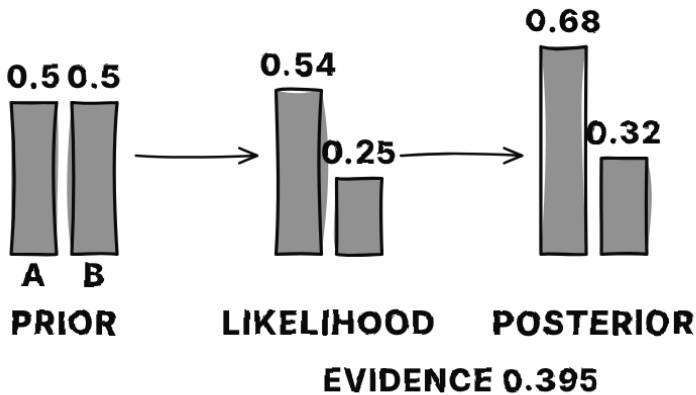
From Product to Sum of Logs

The two viewpoints are closely linked. Products of many probabilities become absurdly small: a thousand examples each given nine chances in ten have a joint likelihood of about two in a hundred thousand billion billion billion billion, a decimal point followed by forty-five

zeros. Taking the logarithm turns the product into a sum, and flipping the sign turns maximising into minimising. So maximising the likelihood is the same as minimising the sum, over the examples, of minus the logarithm of the probability given to each true output. With the first forecaster, minus the logarithm of nine tenths is about one tenth and minus the logarithm of six tenths is about half, about six tenths in total, which is exactly minus the logarithm of fifty-four hundredths. That minus-logarithm term plays the role of a loss, and the book announces that every loss used in practice can be derived this way from a particular choice of distribution. It urges the reader to keep both viewpoints in mind.

Section three point three takes one more step, to Bayesian learning and Bayesian neural networks, which the book only sketches, pointing to tutorials for depth. A probabilistic model can already allow several plausible outputs for one input. But training still picks one single function, while several quite different settings of the model might explain the data equally well, and keeping all of them could give more faithful predictions. So we put a probability distribution over the functions themselves. We start with a prior, a distribution expressing which functions we favour before seeing any data; since a model is described by its parameters, the prior is really a distribution over the weights. For example, functions with a smaller norm are often preferred because they are more stable, so the prior can give them more probability.

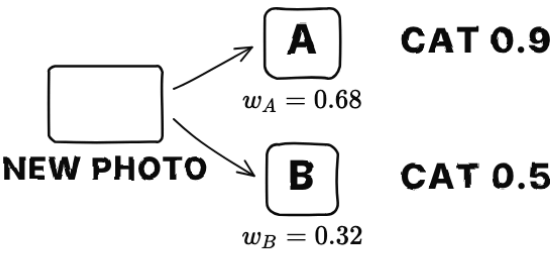
$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}}$$



Prior Times Likelihood

Once the dataset is observed, Bayes' theorem updates the prior: the posterior, the distribution after seeing the data, is the likelihood times the prior, divided by a normalising number called the evidence, which makes everything add up to one. Here is a tiny case with only two candidate models, each with a prior of one half, and data whose likelihood is fifty-four hundredths under the first and a quarter under the second. Likelihood times prior gives twenty-seven hundredths and twelve and a half hundredths; their sum, about forty hundredths, is the evidence; dividing, the posterior gives the first model about sixty-eight in a hundred and the second about thirty-two. To predict, we average the predictions of all models, each weighted by its posterior. If the first model gives a new photo a nine-tenths chance of being a cat and the second one half, the averaged prediction is about seventy-

seven in a hundred. With real models there are infinitely many candidates, and the average is approximated by drawing a handful of models from the posterior, again a Monte Carlo average.



$$0.68 \cdot 0.9 + 0.32 \cdot 0.5 \approx 0.77$$



Average Over Models

The catch, the book admits, is that the posterior can almost never be computed exactly, except for very special choices of prior and likelihood. One must approximate it, either by Markov chain Monte Carlo, which draws samples by a long random walk, or by variational inference, which fits a simpler distribution in its place; section nine point three will show one practical example called Monte Carlo dropout.

Remember

A probabilistic model is trained by maximum likelihood: choose the model that gives the observed dataset the highest probability, the product of the probabilities of each true output. Taking logarithms turns the product into a sum, so maximising likelihood equals minimising the sum of minus the log probabilities, which acts as a loss; all practical losses arise this way. Bayesian learning keeps many models at once: a prior over models, updated by Bayes' theorem into a posterior, likelihood times prior divided by the evidence. Predictions average all models weighted by their posterior, in practice by sampling a few; with two models weighted sixty-eight and thirty-two in a hundred, cat chances of nine tenths and one half average to about seventy-seven in a hundred. The posterior is rarely computable exactly, so it is approximated by Markov chain Monte Carlo or variational inference.

Questions

1. What does maximum likelihood choose?
 - a. The model under which the observed dataset is most probable
 - b. The model with the fewest parameters
 - c. The model with the largest prior
 - d. The model that predicts one half for everything

2. Why are logarithms taken before optimising the likelihood?
 - a. Because probabilities can be negative
 - b. Because logarithms make the model Bayesian
 - c. Because the product of many probabilities becomes tiny, and the logarithm turns it into a manageable sum
 - d. Because logarithms remove the need for data
3. In Bayes' theorem, what is the posterior?
 - a. The distribution over models before seeing data
 - b. The probability of the dataset under one model
 - c. A fixed loss function
 - d. The distribution over models after seeing the data, likelihood times prior divided by the evidence
4. Why is Bayesian learning hard in practice?
 - a. Because priors cannot be defined
 - b. Because the posterior can almost never be computed exactly and must be approximated
 - c. Because it uses only one model
 - d. Because it cannot make predictions

Answers

1. a) The likelihood is the product of the probabilities given to each true output; maximum likelihood picks the model that makes this product largest.

2. c) A product over thousands of examples is vanishingly small; its logarithm is a sum, and minimising minus that sum is the same as maximising the likelihood.

- 3. d)** The posterior updates the prior with the likelihood of the observed data, normalised by the evidence.
- 4. b)** Except for special choices of prior and likelihood, the posterior has no closed form, so Markov chain Monte Carlo or variational inference are used.

Worked examples

Updating two models with Bayes' theorem Two candidate models have priors of one fifth and four fifths. The data has likelihood fifty-four hundredths under the first and one quarter under the second. Find the posterior of each.

Multiply likelihood by prior: fifty-four hundredths times one fifth is about eleven hundredths, precisely one hundred and eight thousandths; one quarter times four fifths is two tenths.

Add them to get the evidence: one hundred and eight thousandths plus two hundred thousandths is three hundred and eight thousandths.

Divide each product by the evidence: one hundred and eight over three hundred and eight is about thirty-five in a hundred.

The second gets two hundred over three hundred and eight, about sixty-five in a hundred.

Answer: Posterior about thirty-five in a hundred for the first model and sixty-five for the second.

Practice

1. A model gives the true outputs of three examples the probabilities eight tenths, one half and nine tenths. Compute the likelihood and the sum of minus the natural logarithms.

Hint: Multiply for the likelihood; minus the natural logarithms are about twenty-two hundredths, sixty-nine hundredths and eleven hundredths.

Answer: The likelihood is eight tenths times one half times nine tenths, thirty-six hundredths. The sum of minus the logarithms is about twenty-two plus sixty-nine plus eleven hundredths, about one point zero two, which equals minus the logarithm of thirty-six hundredths.

2. Two models have posteriors of seven tenths and three tenths. For a new photo, the first says cat with probability eight tenths and the second with probability four tenths. What is the averaged prediction?

Hint: Weight each prediction by its posterior and add.

Answer: Seven tenths times eight tenths is fifty-six hundredths, and three tenths times four tenths is twelve hundredths; together sixty-eight hundredths.

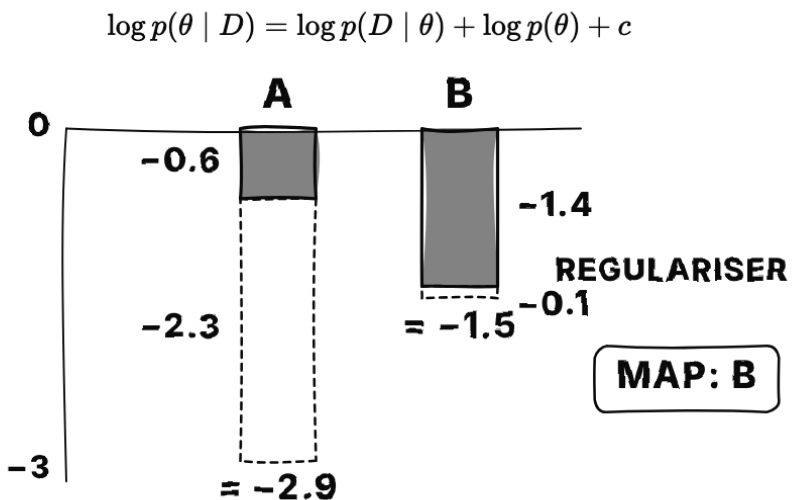
3. Why might a prior favour functions with a smaller norm?

Hint: Think of how strongly such a function reacts to small changes.

Answer: Functions with smaller norm are generally more stable, reacting less wildly to small changes of the input, so before seeing any data it is reasonable to consider them more plausible.

A Penalty from the Prior, and the Simplest Block

Suppose you must pick just one model, not a whole crowd. Bayesian thinking still helps: pick the single model with the highest posterior. Since the evidence is the same for every model, it can be ignored, and taking logarithms splits the goal into two parts added together: the log-likelihood, how well the model explains the data, and the log of the prior, how plausible the model seemed beforehand. This is called the maximum a posteriori solution.



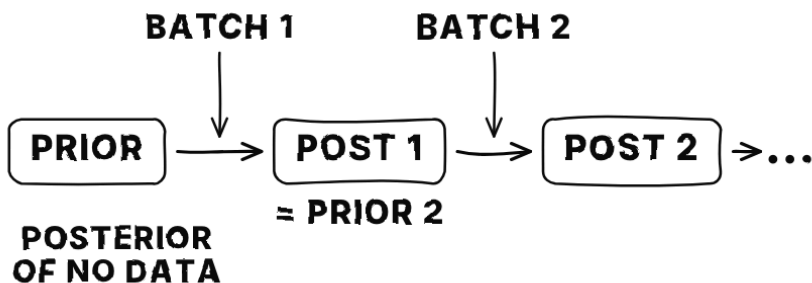
The Prior Can Flip the Choice

Here is how the prior can change the verdict. Say model A explains the data with likelihood fifty-four hundredths, a log of about minus six tenths, and model B with a quarter, a log of about minus one point four. Maximum likelihood picks A. But if the prior gave A only one chance in ten, a log of about minus two point three, and B nine in ten, a log of about minus one tenth, the totals become about minus two point nine for A and minus one point five for B, and the maximum a posteriori choice is B. If every model had the same prior, the second part would be a constant and we would be back to plain maximum likelihood. In general, though, the prior penalises models that stray too far from what we expected, which the book calls a regulariser: it pulls the solution towards the region the prior favours, a useful weapon against overfitting and a way to impose constraints.

A footnote links this loosely to an old debate about what probability means. For frequentists, probability is how often events occur, and maximum likelihood treats the parameters as fixed unknowns and the data as random. For Bayesians, probability measures uncertainty, and the data are fixed while the parameters are random.

FREQUENTIST
 θ FIXED

BAYESIAN
 θ RANDOM



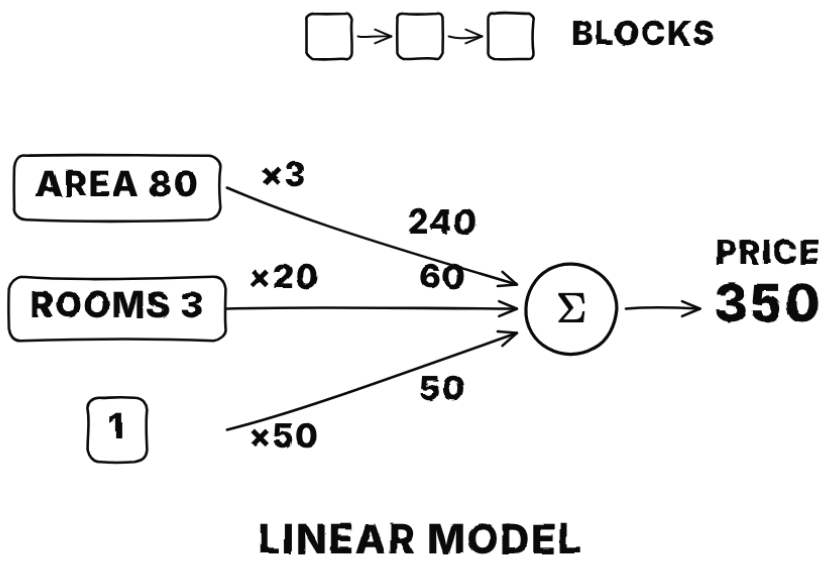
ONLINE TRAINING

NO FORGETTING

Today's Posterior, Tomorrow's Prior

The full Bayesian treatment has a second advantage: new data can be folded in easily. When a new batch of data arrives from the same distribution, the posterior computed on the first batch simply becomes the prior for the second. The book's footnote puts it neatly: the original prior is just the posterior after observing nothing. Imagine the photo app learning from your new photos every week: carrying the posterior forward like this helps it learn new faces without forgetting old ones. Training as data arrives is called online training, and this approach can ease its best-known problem, catastrophic forgetting, where a model loses old knowledge while learning new things.

With this, chapter four begins, on linear models. Programming, the book reminds us, is choosing a good sequence of basic operations; building a model is choosing a good sequence of differentiable blocks. The simplest block is the linear model, where each input adds its own weighted contribution to the output. For a flat, a linear model might say: price equals three times the area, plus twenty times the number of rooms, plus fifty; a flat of eighty square metres with three rooms then gets two hundred and forty plus sixty plus fifty, three hundred and fifty. In a sense, the book says, every differentiable model is a clever variation or combination of such linear blocks.



The Simplest Block: Linear

Remember

The maximum a posteriori solution picks the single model with the highest posterior: the log-likelihood plus the log-prior, the evidence being irrelevant. With a uniform prior it reduces to maximum likelihood; otherwise the log-prior acts as a regulariser, penalising unlikely models and fighting overfitting, and it can overturn the likelihood's choice. Maximum likelihood fits the frequentist view, with fixed parameters and random data; the Bayesian view treats parameters as random. In a Bayesian treatment the posterior becomes the next prior when new data arrive, which helps online training against catastrophic forgetting. Chapter four starts with linear models, where inputs add weighted contributions, the basic block of all differentiable models.

Questions

1. When does the maximum a posteriori solution coincide with maximum likelihood?
 - a. When every model has the same prior, so the log-prior is a constant
 - b. When the dataset is empty
 - c. When the likelihood is one
 - d. Never
2. Why does the book call the log-prior a regulariser?

- a. Because it makes the likelihood larger
 - b. Because it penalises models that stray from what the prior favours, pulling the solution towards them
 - c. Because it removes the need for gradient descent
 - d. Because it normalises the posterior
3. How does a Bayesian treatment absorb a new batch of data?
- a. By discarding the old posterior
 - b. By retraining from the original prior on the new batch only
 - c. By using the posterior from the first batch as the prior for the second
 - d. By averaging the two datasets' likelihoods
4. A linear model prices a flat as three times the area plus twenty times the rooms plus fifty. What does it give for sixty square metres and two rooms?
- a. Three hundred and fifty
 - b. Two hundred and thirty
 - c. Two hundred and seventy
 - d. Two hundred and seventy, minus fifty

Answers

- 1. a)** With a uniform prior the second term adds the same amount to every model, so only the likelihood decides.
- 2. b)** Adding the log-prior penalises unlikely models, which helps against overfitting and imposes constraints.
- 3. c)** The posterior after the first batch becomes the starting belief, updated by the new data, which helps against catastrophic forgetting.

4. c) Three times sixty is one hundred and eighty, twenty times two is forty, plus fifty gives two hundred and seventy.

Worked examples

When the prior overturns the likelihood Model A has log-likelihood about minus six tenths and model B about minus one point four. Their priors are one in ten and nine in ten, with logs about minus two point three and minus one tenth. Which model does maximum likelihood choose, and which does maximum a posteriori choose?

Maximum likelihood compares only the log-likelihoods: minus six tenths beats minus one point four, so it chooses A.

Maximum a posteriori adds the log-prior: for A, minus six tenths plus minus two point three is about minus two point nine.

For B, minus one point four plus minus one tenth is about minus one point five.

The larger total is minus one point five, so maximum a posteriori chooses B.

Answer: Maximum likelihood chooses A; maximum a posteriori chooses B.

Practice

1. Explain in your own words the difference between the frequentist and the Bayesian view as the book sketches it.

Hint: Which is treated as random: the parameters or the data?

Answer: In the frequentist view, probability is the frequency of events; the parameters are fixed but unknown and the data are a random sample. In the Bayesian view, probability measures uncertainty; the data are fixed and the parameters are random variables with a distribution.

2. What is catastrophic forgetting, and why can carrying the posterior forward help?

Hint: Think of a model trained week after week on new data.

Answer: It is when a model trained on new data loses what it learned from old data. Using the previous posterior as the new prior keeps the old knowledge as a starting belief that new data only adjust, rather than overwrite.

3. Why does the book say all differentiable models are, in a sense, variations of linear blocks?

Hint: Compare building a model with writing a program.

Answer: Just as programs are sequences of basic operations, models are sequences of differentiable blocks, and the most basic block is the linear one, where inputs add weighted contributions; more powerful models arise from clever variations and compositions of such blocks.