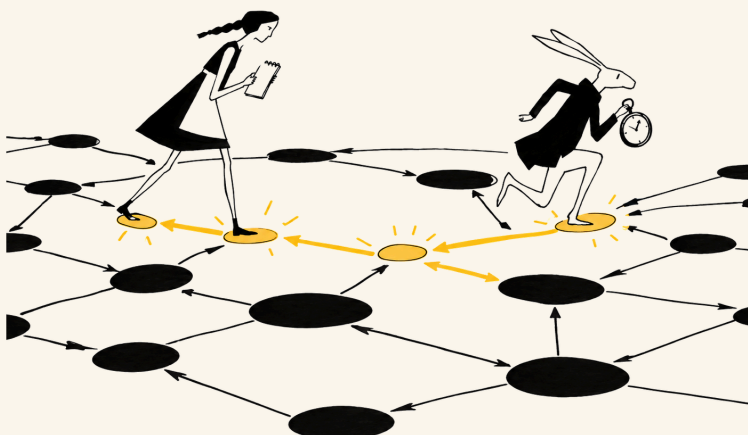


Alice's Adventures in a Differentiable Wonderland



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 Pieces, Endlessly Scaled

Old Ideas, New Power

Boxes of Numbers and Their Shapes

Arrows That Point the Same Way

Templates, Batches and Machines Made of Numbers

How Much Work Is a Multiplication

A Recipe in Letters, and the Slope of a Curve

Slopes in Every Direction

Walking Downhill in the Fog

Where the Walk Can End

A Ball With Momentum

From Practice Code to Real Data

Learning Without Labels, and Reusing What Was Learned

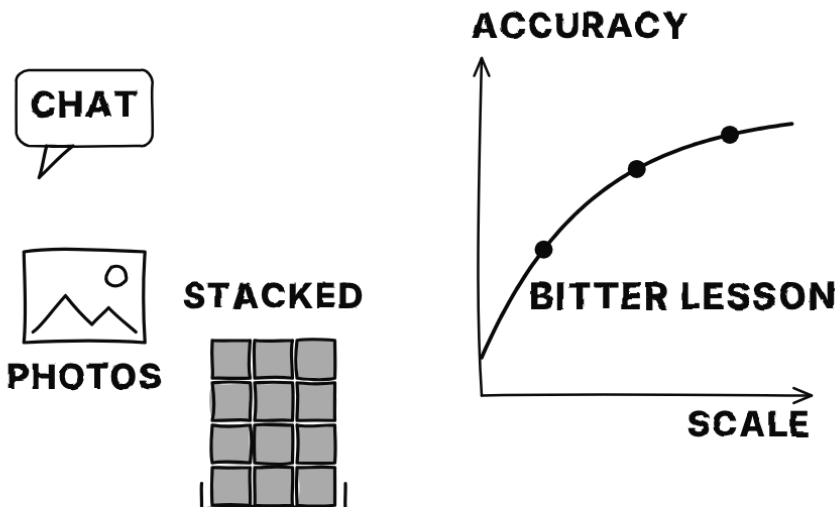
Scoring a Guess

Memorising Is Not Learning

The Most Believable Model

A Few Pieces, Endlessly Scaled

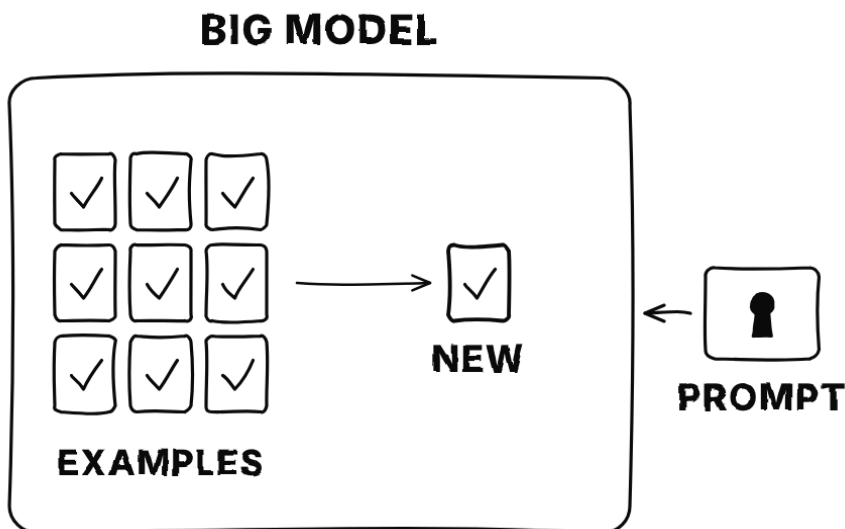
Open the chat assistant on your phone and ask it to write a birthday poem for your aunt, then open the photo app and search for "dog on a beach". Two very different jobs, one with words and one with pictures, and both answered in a second. You might expect two completely different machines behind them, each built from its own secret recipe. The surprise this book starts from is the opposite: both are neural networks, and both are built from a very small set of the same simple pieces. What changed in recent years is not so much the pieces as the scale at which they are stacked and trained.



Same Pieces, Bigger Stacks

The book points to an idea called scaling laws. Roughly, for almost any task, if you give a model more data, more computing power and more size, all at once, its accuracy goes up in a way you can predict in advance. Said the other way round, the computing power needed to reach a given accuracy keeps falling by a steady factor as time passes. This is why companies pour enormous sums into artificial intelligence: the curve has been reliable enough to bet on. The researcher Richard Sutton gave the lesson a name, the bitter lesson: simple, general tools combined with ever more computing power tend to beat clever hand-made tricks.

Taken at face value, that sounds almost like magic, and the book is careful about where the magic could fail. A network is trained to imitate the hidden pattern behind the examples it is shown, the way a forger learns a signature from many samples. Modern networks are so large that they could simply memorise every example, like a student who learns the answer sheet by heart and understands nothing. Yet in practice trained models handle well many tasks that never appeared in their training at all. As the training collections grow, the line between familiar and unfamiliar questions blurs, and large models seem to lean surprisingly little on pure memorising.



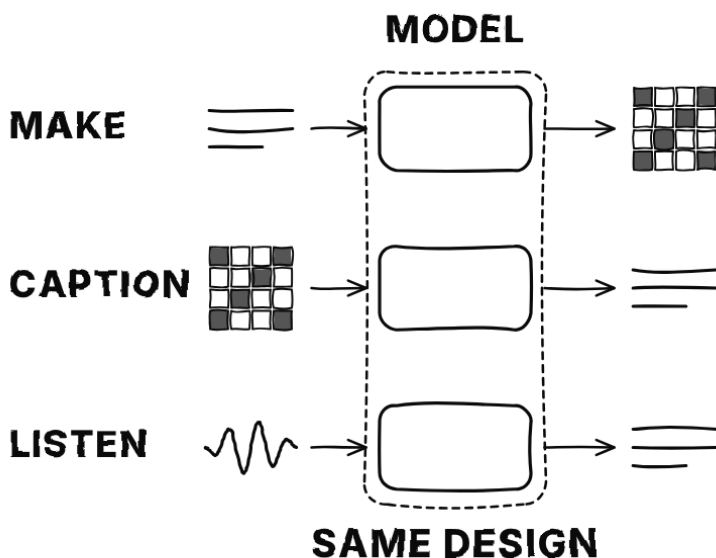
Memorise or Generalise

These very large models, which can be adapted to many later jobs, are sometimes called foundation models, and they have changed how people use networks. Very often you no longer build anything: you type an instruction, a prompt, into a model someone else trained, and its insides stay a closed box. The book compares this with ordinary software. Instead of writing every program yourself, you install a ready-made one whose source you never see; and just as only a few firms can build huge business software, only a few teams in the world have the computers and the know-how to train the largest language models, such as the Llama family.

But the comparison has a second half, and it is the reason for the book. Just as open-source software lets anyone change or rebuild a

program, ordinary hardware and some ingenuity let anyone experiment with these models. You can fine-tune one for your own task, merge two models, shrink one so it runs on a small device, test how easily it breaks, or invent a new design. For any of that you need to look under the hood: how the model holds and transforms data inside, with all the tricks that were learned from experience and debugging. The author offers the book as that entry point, and hopes that a reader who is curious, like Alice, will enjoy the journey.

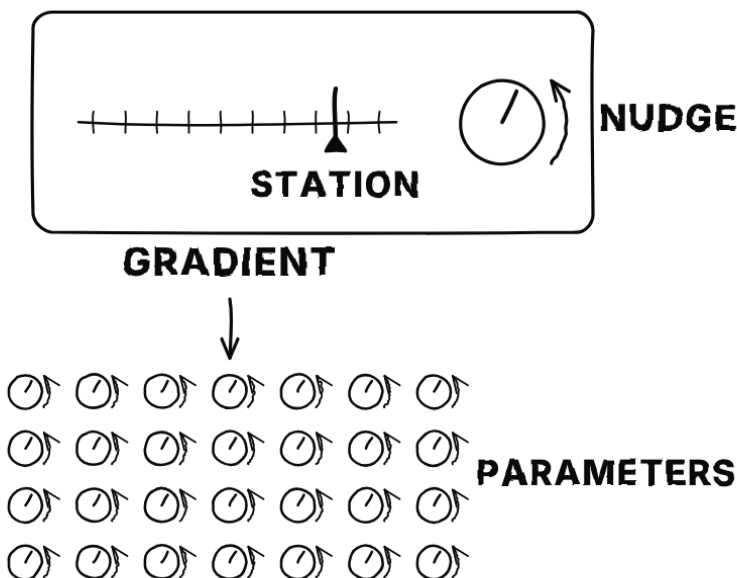
The book assumes you already know the basic idea of supervised learning: collect examples of the behaviour you want, then train a system until it imitates them. That simple idea goes surprisingly far. Making pictures from words becomes a matter of gathering a huge pile of images with their captions; imitating English becomes learning to guess the next sentence from the ones before, which is how a chat assistant is trained; reading an X-ray becomes a big collection of scans with the doctors' verdicts. The book's figure sorts tasks by what goes in and what comes out: picture generation turns text into a grid of pixels, captioning goes the other way, and answering a spoken question turns sound into text. What fascinates the author is that the models for all of these are designed along very similar lines.



What Goes In, What Comes Out

Try it yourself

Take a radio with a single tuning knob and an unfamiliar station you want to find. Turn the knob a little and listen: is the music clearer or noisier than a moment ago? That small test tells you which way to keep turning, and after a few nudges you are on the station without ever having known where it was. Now imagine not one knob but billions of them, all nudged at once by the same rule; that is training.



Nudge the Knob, Listen

The book calls learning a search. You start with a program that has an enormous number of adjustable knobs, called parameters, and you keep adjusting them until the program behaves well enough. With millions or even trillions of knobs, blind trial and error is hopeless, so the search needs a guide. Differentiable models get that guide by being built only from pieces whose response to a tiny nudge can be worked out exactly. Knowing, for every knob at once, what a tiny turn would do is what makes automatic tuning possible, and the two tools behind it, automatic differentiation and gradient descent, fill the book's first part, from chapter two to chapter six.

Seeing a network as a chain of such pieces raises two questions: what kinds of data can go in and come out, and what kinds of pieces can we

use? The nudging trick needs smooth quantities, so letters or whole numbers, which jump from one value to the next, cannot be used directly. Large arrays of numbers, which the book calls tensors, are perfect: a photo is exactly such an array, and it can be reshaped by simple combinations of straight-line and bending steps. The second part, chapters seven to nine, studies one famous piece, the convolution, which works on anything laid out in order, such as sound, pictures, text and video. Along the way it introduces how to build deep models, made of many steps one after another, and the ideas of cutting text into tokens and generating it piece by piece, the basis of today's language models. The third part goes further down the rabbit hole: attention and transformers for sets of items in chapters ten and eleven, graphs in chapter twelve, and recurrent layers for sequences in time in chapter thirteen.

Remember

The chat assistant and the photo app on your phone are both neural networks built from the same small set of simple pieces. Scaling laws say that more data, more computing and bigger models predictably bring better accuracy, which is why so much is invested, and Sutton called this the bitter lesson. Huge models could memorise their examples, yet they generalise well, and most people now use them through prompts as closed boxes. To change or build such models you must look inside, and the book's view is that learning is a search over countless knobs, guided by how each piece responds to a tiny nudge. The book then moves from these tools, to convolutions for ordered data, to attention, graphs and recurrent layers.

Questions

1. Why do companies invest such enormous sums in making models bigger?
 - a. Because bigger models need no training data at all
 - b. Because more data, computing and size together have predictably brought better accuracy
 - c. Because every new task needs a completely new kind of network
 - d. Because large models are the only ones that can memorise answers
2. A huge network could simply memorise every training example. What does the book report actually happens?
 - a. Trained models handle well many tasks that never appeared in their training
 - b. Large models only repeat their training examples word for word
 - c. Memorising is impossible, because networks are too small to store examples
 - d. Models work only on questions identical to their training data
3. Why does the book insist that, to fine-tune, merge or shrink a model, you need to look under the hood?
 - a. Because prompting a model is forbidden for such changes
 - b. Because only the largest companies are allowed to change models

- c. Because changing a model means understanding how it holds and transforms data inside
 - d. Because open-source models have no parameters to adjust
4. Why can letters or whole numbers not be fed to a differentiable model directly?
- a. Because they take too much memory to store
 - b. Because the nudging trick needs smooth quantities, and these jump from one value to the next
 - c. Because tensors can only hold photos
 - d. Because convolutions refuse anything that is not sound

Answers

1. **b)** Scaling laws say that growing data, computing power and model size together raises accuracy in a predictable way, so the bet is reliable.
2. **a)** Although they are big enough to learn the answer sheet by heart, trained models generalise well and lean surprisingly little on pure memorising.
3. **c)** Using a model as a closed box is enough for prompting, but customising or building one needs to know how it processes data inside, with its learned tricks.
4. **b)** Training relies on knowing what a tiny nudge does, which needs smooth quantities; letters and whole numbers are discrete, while large arrays of numbers such as photos work well.

Practice

1. Explain in your own words why the book calls learning a search, and why that search needs differentiable pieces when a model has

billions of parameters.

Hint: Think of the radio knob: what does a small turn tell you, and what happens when there are billions of knobs?

Answer: A model is a program with an enormous number of adjustable knobs, and learning means adjusting them until the program behaves well. With billions of knobs, random trial and error would never finish. If every piece responds to a tiny nudge in a way that can be worked out exactly, we know for all knobs at once which way to turn, and automatic tools such as gradient descent can do the tuning.

2. The book says that many very different tasks become the same kind of problem under supervised learning. Take making pictures from words and explain what the 'examples' are and what the model learns.

Hint: Ask what goes in, what should come out, and what collection of pairs you would gather.

Answer: The input is a text and the output is an image, so the examples are a very large pile of images with their captions. The model is trained until, given a caption, it produces a picture that fits, imitating the behaviour shown in the collected pairs.

3. Using the book's comparison with software, explain both sides: why most people now use large models as closed boxes, and why ordinary people can still experiment with them.

Hint: One side is about who can build the largest models; the other is about open source.

Answer: Like large business software, the biggest models can be built only by a few teams with huge computers and know-how, so most people simply prompt a model someone else trained. But just as open-source software can be changed or rebuilt by anyone, ordinary hardware and some ingenuity let people fine-tune, merge, shrink or test models, as long as they understand what happens inside.

Old Ideas, New Power

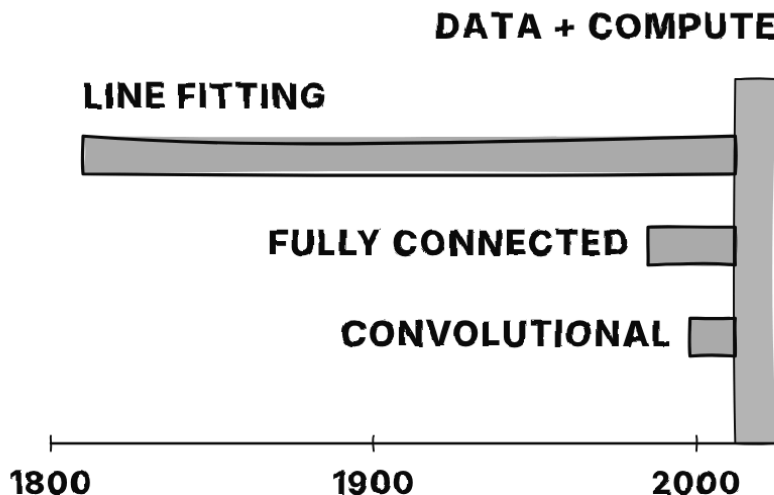
Picture a newspaper photo from 1958: a serious man in a white shirt beside a cabinet of wires, under a headline promising a machine that will learn like a brain. The man is the American psychologist Frank Rosenblatt, and the machine, the perceptron, was one of the first working neural networks. The book shows this scene on purpose, with a wink: artificial intelligence hype, except it is 1958. The excitement you see today around chat assistants is not the first wave, and the ideas under it are older than most people think.

The very name neural network is a leftover from early attempts to imitate the neurons of living brains in the twentieth century. Words from that time stayed with us, such as neurons, layers and activations, and the book keeps them to match the software everyone uses. Interest came in waves. Between 2012 and 2017, large public challenges, above all the ImageNet competition on recognising objects in photos, pushed networks to grow fast in size and complexity. A second wave began in 2017 with transformers, which took over the field of language in a very short time, just as one family of models had taken over computer vision a few years before. Progress on video, graphs and sound followed, leading to today's excitement about large language models, models that mix text and pictures, and models that generate.

The vocabulary changed with each wave, from connectionism in the 1980s to deep learning, which set modern networks apart from the smaller, shallower models of the past. The author finds all these labels vague, because modern artificial networks have almost nothing to do

with real brains. What they truly have in common is that they are built from blocks that can be differentiated, blocks whose response to a tiny nudge can be worked out. So the book prefers the plainer name differentiable models. Seen that way, neural networks belong to a wider, young discipline called differentiable programming, which mixes computer science and optimisation to study any computer program that can be tuned this way.

Walking through this land is also walking through history. Fitting simple straight-line models by repeated small corrections, the method of the book's fourth chapter, was known at least since the nineteenth century. The fully connected networks used later in the book go back to the 1980s, and the convolutional models used for images were already in use by the end of the 1990s. What took decades was not the ideas but enough data and enough computing power to show how well these models work when they are big enough. The phone that recognises your face runs on ideas older than you might guess, made practical only recently.



Old Ideas, Waiting for Power

The book cannot cover everything, partly because research moves so quickly, but it aims to give you enough to find your way through the newest papers on your own. Beyond the printed chapters, the author promises a website with extra chapters on topics that do not depend on one kind of data, such as generative models, models that switch parts of themselves on and off, reusing what one model has learned for another task, and explaining what models do. Those chapters lean towards research and can be read in any order. There are also guided practical sessions in notebook form, which follow much of the book and go further, for example into learning by comparing pairs of examples, and into merging models.

Remember

The hype around learning machines is not new: in 1958 Rosenblatt's perceptron already made headlines. Names like neuron and layer are historical leftovers, and the waves of interest came with ImageNet from 2012 and with transformers from 2017. Because modern networks resemble brains very little, the book calls them differentiable models, part of the wider field of differentiable programming. Most of the key ideas are decades old; what was missing was enough data and computing power to show their strength.

Questions

1. Why does the book prefer the name differentiable models to neural networks?
 - a. Because modern networks copy real neurons more closely than before
 - b. Because what modern networks truly share is being built from blocks that can be differentiated
 - c. Because the word neural is forbidden in today's software
 - d. Because differentiable models are a completely new invention of 2017
2. If the key ideas are decades old, what held these models back for so long?

- a. Nobody had yet invented the idea of correcting a model step by step
 - b. The models were banned after the 1958 hype
 - c. There was not enough data and computing power to show how well big models work
 - d. Fully connected networks were only invented in 2012
3. Why does the book still use words such as neurons, layers and activations?
- a. To match the terms used in existing software, even though they are historical leftovers
 - b. Because modern networks are exact models of the brain
 - c. Because differentiable programming has no words of its own
 - d. Because the ImageNet competition required those names

Answers

- 1. b)** Modern artificial networks have almost nothing to do with brains; their real common trait is that they are built from differentiable blocks.
- 2. c)** Straight-line fitting dates from the nineteenth century, fully connected networks from the 1980s and convolutional models from the 1990s; what was missing was data and computing power.
- 3. a)** The names come from early attempts to imitate brain neurons and stayed in use, so the book keeps them to be consistent with the frameworks everyone uses.

Practice

- 1.** In your own words, explain what the book means by the joke 'AI hype, except it is 1958'.

Hint: Think about who Rosenblatt was and what the perceptron was.

Answer: In 1958 Frank Rosenblatt's perceptron, one of the first working neural networks, drew a lot of media attention, much like today's excitement about AI. The joke reminds us that enthusiasm for learning machines comes in waves and that the basic ideas are much older than the current boom.

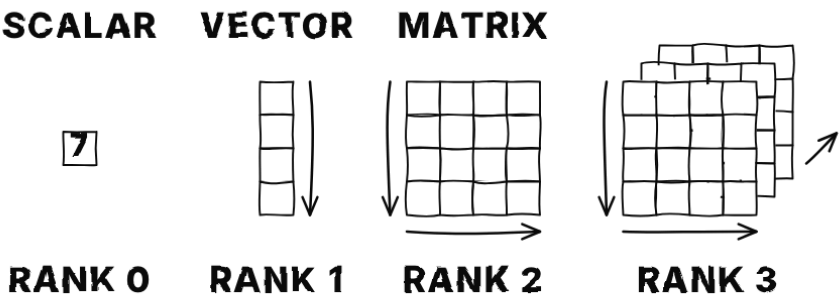
2. Describe the two big waves of interest in neural networks that the book names after 2012, and what each one changed.

Hint: One wave was driven by a competition on photos, the other by a new kind of model for language.

Answer: From 2012 to 2017, large challenges such as ImageNet pushed networks for recognising images to grow quickly in size and complexity. From 2017, transformers rapidly took over language processing, just as convolutional models had taken over vision earlier, and further progress on video, graphs and sound led to today's excitement about large language models and generative models.

Boxes of Numbers and Their Shapes

Take a sheet of squared paper and write a single number in one square. Then write a column of numbers going down, then fill a whole rectangle, and finally stack several filled sheets one on top of another. You have just built the four kinds of objects the whole book is made of. A single number, a column, a grid and a stack of grids differ in only one thing: how many directions you need to find a square.

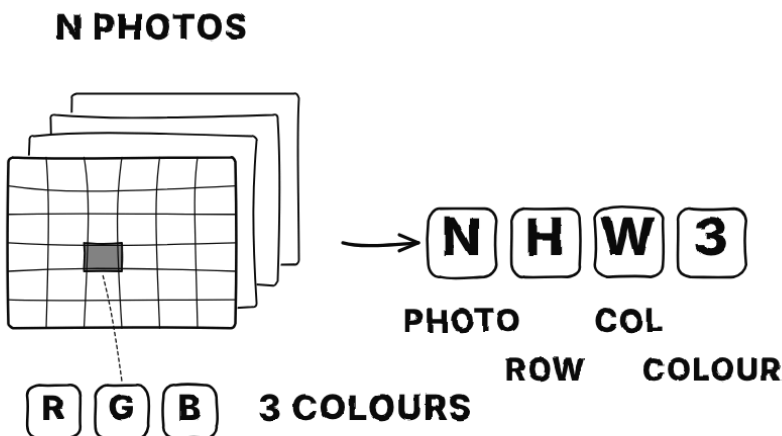


Tensors of Rank 0 to 3

The book calls all of them tensors, and the number of directions it calls the rank of the tensor. A lone number has rank zero and is called

a scalar; a column of numbers has rank one and is called a vector; a rectangle of numbers has rank two and is called a matrix; anything with more directions has no special name. The author admits the word tensor means something stricter to mathematicians, and apologises to them, but the loose meaning is so common in software that the book keeps it. Each kind gets its own style of letter in the book, so that a glance tells you which you are looking at, and fancy curly letters are kept for collections and for probabilities.

Now think of the photos on your phone, loaded into a model in one go. Each photo is a grid of pixels, each pixel holds three numbers for red, green and blue, and there are many photos at once. So the whole batch is a box with four directions: which photo, which row, which column, and which of the three colours. The book writes such a box by listing its sizes, the shape, and leaves some of them as letters, because the number of photos or their height may change while the three colours never do. The author's key advice is that understanding an operation very often comes down to understanding the shape of every box going in and coming out.



A Batch of Photos as a Box

The same short sign that announces a shape is also used, rarely, for drawing a random number from a probability distribution; the author promises the meaning will always be clear from the context. Two more labels tell you what may sit inside a box. One says the box holds only zeros and ones, like a row of light switches that are either off or on. The other says the numbers are never negative and add up to one, like the slices of a pie chart, or like the confidence a photo app spreads over the labels cat, dog and car. For bigger boxes of that second kind, each row on its own adds up to one, so every photo gets its own pie.

1	0	1	1	0
---	---	---	---	---

0 OR 1

CAT DOG CAR

0.7	0.2	0.1
-----	-----	-----


SUMS TO 1

0.1	0.8	0.1
-----	-----	-----


1 PHOTO = H × W × 3

Switches and Pies

Try it yourself

Open the information panel of any photo on your phone and look for its size, for example a width and a height in pixels. Multiply them in your head and add that every pixel holds three colour values: that is the size of the box the model sees for one photo. Now imagine a hundred such photos sent together, and you have a box with four directions, exactly the kind the book describes.

The book grew out of a course on neural networks for data science that the author has taught for many years at Sapienza University in

Rome, with material added from other courses, tutorials and summer schools. It starts on the same ground as other good recent books, and then follows the author's own research interests, offering a complementary point of view. Its central conviction is that these models are understood through both theory and code at once, and automatic differentiation is the best example of that constant back and forth. The book credits much of the comeback of neural networks since 2012 to powerful software libraries, from early ones such as Theano, Caffe and Chainer to today's TensorFlow, PyTorch and JAX. It connects its ideas to PyTorch and JAX wherever it can, but it is not a programming manual, and it sends you to each library's own documentation for that.

The author is also honest about what the book is not. It picks ideas that are common today and general enough to last, but it cannot see the future, and some parts may be outdated by the time you read them. For each idea it shows a few variants found in research papers, such as two different ways of keeping numbers in a network well scaled, yet hundreds more exist, and a large public website that collects papers together with their code is the place to explore them. Finally, making these models work at large scale takes engineering skill and some intuition: the book says little about hardware, and for intuition nothing beats experience and the frank blog posts of practitioners.

Remember

Everything in the book is a tensor, a box of numbers whose rank is the number of directions you need to find one entry: a number, a column, a grid, or more. A batch of colour photos is a box with four directions, and understanding an operation usually means knowing the shape of every box involved. Some boxes hold only zeros and ones, others hold non-negative numbers adding up to one, like a pie chart of confidence. The book comes from a long-running university course, joins theory and code, leans on PyTorch and JAX, and admits that engineering and experience must fill in the rest.

Questions

1. What makes a scalar, a vector and a matrix different kinds of tensor?
 - a. The size of the numbers they contain
 - b. How many directions you need to find one entry
 - c. Whether the numbers are whole or have decimals
 - d. Whether they come from photos or from text
2. A model receives a batch of colour photos in one go. Why is that batch a tensor with four directions?
 - a. Because each photo has four corners
 - b. Because the model always needs exactly four photos

- c. Because you must say which photo, which row, which column and which of the three colours
 - d. Because colour photos are stored as four separate files
3. A photo app reports how confident it is that a picture shows a cat, a dog or a car. Which label from the book fits these numbers?
- a. Numbers that are never negative and add up to one, like slices of a pie
 - b. A box that holds only zeros and ones
 - c. A tensor of rank three with no restrictions
 - d. A single scalar with no shape
4. According to the author, what most often unlocks the understanding of an operation in a model?
- a. Memorising the names of every software library
 - b. Knowing the shape of every tensor going in and coming out
 - c. Reading the hardware specifications
 - d. Counting the number of footnotes in the chapter

Answers

- 1. b)** A number, a column, a grid and a stack of grids differ only in how many directions you need to locate a square, and that number is the rank.
- 2. c)** Each photo is a grid of pixels with three colour values per pixel, and many photos come together, so finding one number needs four choices.
- 3. a)** Confidences spread over labels are non-negative and sum to one, like the slices of a pie chart; for many photos, each row gets its own pie.

4. b) The book's key advice is that understanding an operation usually comes down to understanding the shape of each tensor involved.

Practice

1. A model reads a batch of short sound clips. Each clip is a long row of loudness values, and every clip in the batch has the same length. How many directions does the batch have, and what does each direction stand for?

Hint: Ask what you must choose to point at one single loudness value.

Answer: Two directions: which clip, and which moment within the clip. You need both choices to find one value, so the batch is a grid of numbers, a tensor of rank two, a matrix.

2. Explain why the book leaves some sizes in a shape as letters but writes others as fixed numbers, using the batch of colour photos as your example.

Hint: Which sizes can change from one batch to the next, and which can never change?

Answer: The number of photos and their height and width can differ from one use to another, so they are left as letters; every pixel always has exactly three colour values, so that size is written as a fixed number.

3. In your own words, say what the book admits it is not, and where it sends the reader for those things.

Hint: Think of programming details, completeness, and making models work at scale.

Answer: It is not a programming manual, so it points to the documentation of PyTorch and JAX. It is not complete or future-proof, so for the many variants of each idea it points to a public website collecting papers with their code. And it says little about hardware and the craft of

making models work at scale, for which experience and practitioners' blog posts are the best teachers.

Arrows That Point the Same Way

Hold two pencils in front of you, both starting from the same fingertip. Point them in exactly the same direction, then open them into a right angle, then turn one until they point in opposite directions. Without any numbers, you can already say how much the two pencils agree: fully, not at all, or completely against each other. By the end of this part you will see that one simple operation measures exactly that agreement, and that it sits at the heart of almost every model in the book.



AGREE



NONE



OPPOSE

Agree, Ignore, Oppose

First, a word about the book itself, from its acknowledgments. The coloured equations come from a free typesetting package by another author, the colour pictures of Alice and the little symbols in the margin are licensed from a stock photo agency, and the Alice drawings inside the chapters are John Tenniel's original illustrations, taken from Wikimedia. The author thanks the colleagues and readers who sent corrections and encouraged the author to publish.

The license matters for what you are listening to now. The book is released under a Creative Commons license that lets anyone share, remix, adapt and build on it, in any form and even commercially, as long as the author is credited. There is one condition: whatever is built on it must be released under the same terms. This course is exactly such a remix, so it carries the same license, and it does not reuse the licensed stock pictures, which the book's license does not cover.

Chapter two then gathers the mathematics the rest of the book needs. The author assumes you have met these topics before, and uses the chapter mainly to agree on notation and to give a connected overview, pointing out where an idea meets real code. It moves in three steps, each building on the last: linear algebra, then gradients for boxes of numbers of any rank, then how gradients are used to improve a function. Probability gets its own appendix, built around the principle of maximum likelihood, and the author warns, kindly, that the chapter is dense with definitions.

Formally, a tensor is a box of elements that all have the same type, and it is described by that type and by its shape, the list of its sizes we met before. Tensors matter so much because they suit massively parallel hardware: the graphics chips in gaming computers, and even

more specialised chips, can work on thousands of entries at the same time. When a chat assistant answers you within seconds, it is pushing huge boxes of numbers through such chips. Most boxes in the book hold numbers with decimals, but they can hold whole numbers, for example class labels, or even pieces of text. You can cut slices out of a box, a row, a block, a single entry, with the conventions of the popular library NumPy, except that the book counts from one rather than from zero.

	+	10	20	30	40	
1		11	21	31	41	
2		12	22	32	42	
3		13	23	33	43	
4		14	24	34	44	16, NOT 4

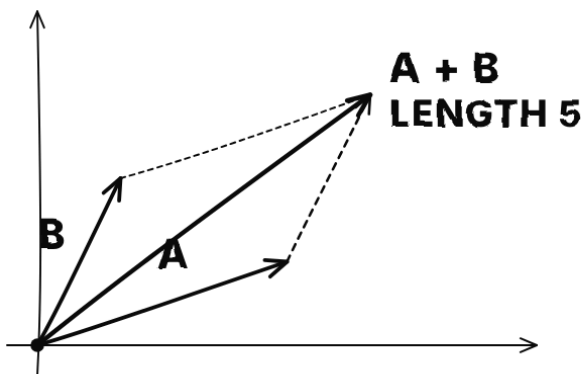
Four Plus Four Makes Sixteen

Almost every model in the book is a chain of very ordinary operations: sums, multiplications, and a few bending functions such as the exponential, sine, cosine and square root. Textbooks distinguish carefully between a vector written as a column and the same vector

written as a row. In code the distinction is trickier, because a column, a row and a plain list of numbers are three different shapes. Libraries follow a rule called broadcasting: when two boxes do not match, the smaller one is quietly repeated until they do. The book shows a classic accident: you add a column of four numbers to a plain list of four numbers, expecting four results, and you get a grid of sixteen instead, with no error message at all.

Try it yourself

Draw a small cross on a sheet of paper for the origin, and draw two arrows starting from it. Now slide a copy of the second arrow so that it starts at the tip of the first, and draw the arrow from the origin to where you end up. You have just added two vectors: the result is the diagonal of the parallelogram the two arrows make. Measure the result with a ruler: that straight-line length from the origin is what the book calls the norm.

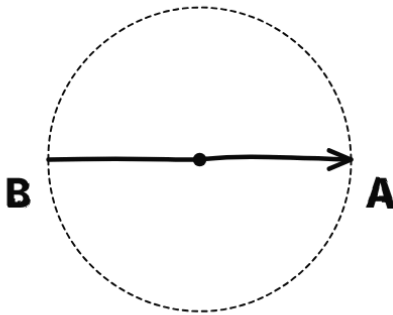


Adding Arrows

That is the geometric picture behind vectors. Any two vectors of the same shape can be stretched and added to give a third one, entry by entry, and this freedom is what mathematicians call a vector space. Seen as a point in space, a vector has a length, its distance from the origin, and the book often uses the length squared, which is simply all entries squared and added up.

The most important vector operation in the whole book is the dot product, also called the inner product. You multiply the two vectors entry by entry and add up everything you got, so the result is a single number, not a vector. The book's small example multiplies two short lists of three numbers and ends with one slightly negative number. Its meaning is geometric: the dot product is the two lengths multiplied

together, times a measure of how well the two arrows are aligned. When both arrows have length one, all that is left is the alignment itself, called the cosine similarity. It is at its highest when the arrows point the same way, at its lowest when they point in opposite directions, and exactly zero when they stand at a right angle, which the book calls orthogonal. Search features that find photos or messages similar in meaning to your query typically turn each item into such an arrow and compare directions this way.



$$\mathbf{A} \cdot \mathbf{B} = \sum A_i B_i$$

$$\text{COS} = -1.00$$

Cosine Similarity

Remember

The book's acknowledgments credit the equation package, the licensed stock images and Tenniel's original Alice drawings. Its license allows anyone to remix it, even commercially, with credit and under the same terms, which is why this course exists and why it avoids the stock images. Chapter two reviews linear algebra, gradients and optimisation; tensors are boxes of one type with a shape, ideal for parallel chips, and broadcasting can silently turn four results into sixteen. The dot product multiplies two vectors entry by entry and adds up, giving one number that measures lengths and alignment together. For arrows of length one it is the cosine similarity: highest for the same direction, lowest for opposite directions, zero for a right angle.

Questions

1. Two arrows of length one stand at a right angle to each other. What is their cosine similarity?
 - a. Its highest value, because both arrows have length one
 - b. Its lowest value, because they point in different directions
 - c. Exactly zero, because they are orthogonal
 - d. It cannot be said without knowing every entry
2. You add a column of four numbers to a plain list of four numbers and get a grid of sixteen numbers. What happened?
 - a. The library made an arithmetic error

- b. Broadcasting quietly repeated the boxes to make their shapes match
 - c. The dot product was computed instead of the sum
 - d. The numbers were too large and overflowed
3. Why are tensors so central in deep learning?
- a. Because they suit massively parallel chips that work on thousands of entries at once
 - b. Because they can hold only numbers with decimals
 - c. Because they make every vector a column
 - d. Because they remove the need for gradients
4. What does the dot product of two vectors measure, in the book's geometric reading?
- a. Only the difference between their lengths
 - b. The number of entries they have in common
 - c. Their two lengths multiplied together, times how well the arrows are aligned
 - d. The area of the parallelogram they form

Answers

- 1. c)** The cosine similarity is highest for the same direction, lowest for opposite directions, and exactly zero for a right angle, which the book calls orthogonal.
- 2. b)** A column and a plain list are different shapes, and broadcasting silently repeats the smaller box until the shapes match, giving a grid instead of four results.
- 3. a)** Boxes of numbers of one type are ideal for graphics chips and other specialised hardware that process many entries at the same time.

4. c) The dot product equals the product of the lengths times a measure of alignment; for arrows of length one only the alignment, the cosine similarity, is left.

Worked examples

The book's dot product The book takes two short vectors, the first with entries 0.1, 0 and minus 0.3, the second with entries minus 4.0, 0.05 and 0.1. What is their dot product?

Multiply the first entries: 0.1 times minus 4.0 gives minus 0.4.

Multiply the second entries: 0 times 0.05 gives 0.

Multiply the third entries: minus 0.3 times 0.1 gives minus 0.03.

Add the three results, because the dot product is the sum of the entry-by-entry products: minus 0.4 plus 0 minus 0.03.

Answer: Minus 0.43, a single number, slightly negative, so the two arrows point somewhat against each other.

Practice

1. Without computing anything, say whether the dot product of two vectors is positive, negative or zero when the two arrows point in exactly opposite directions, and why.

Hint: Recall what the alignment measure does for opposite directions.

Answer: Negative. The dot product is the two lengths, which are positive, times the alignment, and for opposite directions the alignment is at its lowest, a negative value.

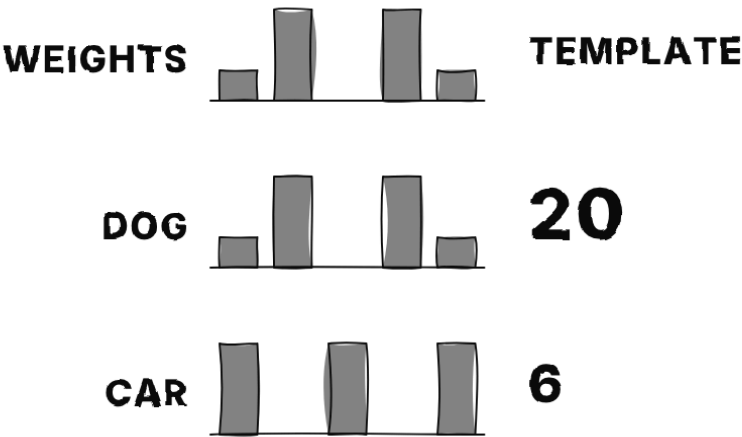
2. Explain in your own words what the book's license allows, and the one condition attached to it.

Hint: Think of sharing, changing, selling, and what must happen to the new work.

Answer: Anyone may share, remix, adapt and build on the book, in any form and even commercially, provided the author is credited. Whatever is built on it must be released under the same license, which is why this course carries it too; the licensed stock images are not part of that permission and are not reused.

Templates, Batches and Machines Made of Numbers

Take a stencil, a sheet of card with a shape cut out of it, and slide it over a page of printed shapes. Where the shape on the page matches the hole exactly, it shows through perfectly; where it only half matches, part of it is hidden. The dot product behaves just like that stencil. Fix one arrow of length one and try every other arrow of length one against it: the dot product is highest exactly when the second arrow is the same as the first.



Weights as a Stencil

This simple fact carries much of the book. In the coming chapters one arrow will be the input, say a photo, and the other will be the model's adjustable parameters, which the book calls weights. The dot product is largest when the input resonates with the weights, so the weights act like a stencil the input is held against, and the book calls this template matching. You can picture a photo app holding a stencil for dogs: a photo that lines up well with it gives a big number, and a photo of a car gives a small one.

Two small observations come along. Adding up all the entries of a vector is the same as taking its dot product with a vector made entirely of ones. And the distance between two vectors can be written using dot products alone; in particular, the dot product of a vector with itself is its length squared. Both tricks turn up again and again, on paper and in code.

Next come matrices, grids with some number of rows and columns. The book asks you to see a matrix as a stack of vectors, one per row, and calls such a stack a batch of data. Think of a spreadsheet where each row is one photo, described by the same list of numbers: the whole sheet is one matrix, and models are written to work on whole sheets like this at once.

1	2	0
0	1	3
2	2	1

×

1
0
2

=

1
6
4

$1 + 0 + 0 = 1$

$0 + 0 + 6 = 6$

$2 + 0 + 2 = 4$

ALL AT ONCE

One Row Against One Column

The key operation on matrices is matrix multiplication, and it is built from the dot product. Each entry of the result is the dot product of one row of the first matrix with one column of the second, which is why the rows of the first must be exactly as long as the columns of the second. When the second matrix is a single column, you get the special case of a matrix times a vector. Here is the practical payoff: if the first matrix is a batch of inputs and the second holds the weights, one single multiplication computes the dot product of every input with the weights, all at once. And a batch multiplied by its own flipped version gives, in one go, the dot product of every pair of rows, a complete table of how similar every item is to every other.

Try it yourself

Write a small spreadsheet with three rows, each a short list of three numbers, and a fourth list of three numbers in a separate column as your weights. Next to each row, ask the spreadsheet for the sum of the products of matching entries with the weights. You have computed three dot products by hand, one per row; a matrix multiplication does exactly this, for thousands of rows and many weight columns in a single step.

Matrices can also be multiplied entry by entry, when they have the same shape, a rule named after the mathematician Hadamard. It lacks the elegant properties of true matrix multiplication, but it is extremely handy: multiplying by a grid of zeros and ones switches entries off, like a mask, and multiplying by other numbers scales them. Some recent model families also lean on such entry-by-entry products, as later chapters will show.

5	2	7
1	8	3
4	6	9

 $\odot$

1	0	1
0	1	0
1	1	0

 $=$

5	0	7
0	8	0
4	6	0

MASK

A Mask, Entry by Entry

In the same spirit, when the book applies a function such as the exponential to a whole matrix, it means applying it to every entry separately. That is not the same as the true matrix exponential of the textbooks, which works only on square matrices and is built from repeated matrix multiplications. The entry-by-entry version works on boxes of any shape, which is why libraries such as PyTorch keep the genuinely matrix-only operations, such as inverses and determinants, in a separate corner of their toolbox. The book's code box shows the two side by side, one ordinary call for the entry-by-entry exponential and a specialised one for the matrix exponential. Finally, a reduction squeezes a box along one direction, for example summing or averaging each row, and in code it is chosen by naming the direction.



MACHINES, PLUGGED TOGETHER

Matrices Are Machines

A closing box asks why matrix multiplication is defined in this row-against-column way rather than entry by entry. The answer is that it captures linear functions, the functions that respect stretching and adding: stretch or add the inputs first, or the outputs afterwards, and you get the same result. Every such function can be written as a matrix. Multiplying a matrix by a vector is then simply applying the function, and multiplying two matrices is doing one function after the other. So a matrix is not just a table of numbers; it is a small machine, and multiplying matrices plugs machines together.

Remember

Among arrows of length one, the dot product is largest when both are the same, so a model's weights work like a stencil that the input resonates with: template matching. A matrix can be seen as a batch, a stack of vectors one per row. Matrix multiplication fills each entry with the dot product of a row and a column, so one multiplication computes all the dot products of a batch at once. Entry-by-entry multiplication masks and scales, functions like the exponential act entry by entry, and reductions squeeze a box along one direction. Matrix multiplication is defined this way because a matrix is a linear function, so multiplying by a vector applies it and multiplying matrices chains functions.

Questions

1. Why does the book call the dot product between an input and the weights 'template matching'?
 - a. Because it copies the input into the weights
 - b. Because it is largest when the input lines up with the weights, like a shape fitting a stencil
 - c. Because it counts how many entries the two vectors have
 - d. Because it always gives zero for photos of cars
2. A batch of inputs, one per row, is multiplied by a matrix holding the weights. What does that single multiplication achieve?

- a. It averages all the inputs into one row
 - b. It computes the dot product of every input with the weights, all at once
 - c. It multiplies the inputs entry by entry with the weights
 - d. It squares every entry of the batch
3. You want to switch off some entries of a matrix and keep the others unchanged. Which operation fits?
- a. Entry-by-entry multiplication by a grid of zeros and ones of the same shape
 - b. The true matrix exponential
 - c. Multiplying the matrix by its flipped version
 - d. A reduction that sums each row
4. Why is matrix multiplication defined row against column rather than entry by entry?
- a. Because it is faster to compute on graphics chips
 - b. Because only square matrices can be multiplied entry by entry
 - c. Because a matrix stands for a linear function, and this rule makes multiplying matrices chain the functions
 - d. Because entry-by-entry multiplication is not allowed in PyTorch

Answers

- 1. b)** Among arrows of length one the dot product is highest when both are the same, so the weights act like a stencil the input resonates with.
- 2. b)** Each entry of a matrix product is the dot product of a row with a column, so one multiplication gives all the dot products of the batch

together.

3. a) Hadamard, entry-by-entry, multiplication by zeros and ones acts as a mask, switching entries off where the mask holds zeros.

4. c) Every linear function can be written as a matrix; multiplying by a vector applies it and multiplying two matrices does one function after the other.

Worked examples

Which shapes can be multiplied A batch has 3 rows of 4 numbers each, and a weight matrix has 4 rows and 2 columns. Can they be multiplied, and what shape is the result?

Each entry of the product is the dot product of a row of the first matrix with a column of the second, so these must have the same length.

A row of the batch has 4 numbers and a column of the weights has 4 numbers, so the multiplication is possible.

There is one result entry for every pair of a batch row and a weight column: 3 rows paired with 2 columns.

Answer: Yes; the result has 3 rows and 2 columns, one row per input and one column per set of weights.

Practice

1. Without the formula, explain why taking the dot product of a vector with a vector made entirely of ones gives the sum of its entries.

Hint: Think about what multiplying each entry by one does before everything is added up.

Answer: The dot product multiplies matching entries and adds the results. Multiplying each entry by one leaves it unchanged, so what is added up is just the entries themselves, which is their sum.

2. A batch of photos, one per row, is multiplied by its own flipped version. Describe the table you get and what one of its entries tells you.

Hint: Each entry pairs one row with one row.

Answer: You get a square table with one row and one column per photo. Each entry is the dot product of two photos' rows, a measure of how similar those two items are, so the whole table compares every item with every other in one step.

3. Explain the difference between applying the exponential to a matrix entry by entry and the true matrix exponential, and why libraries keep them apart.

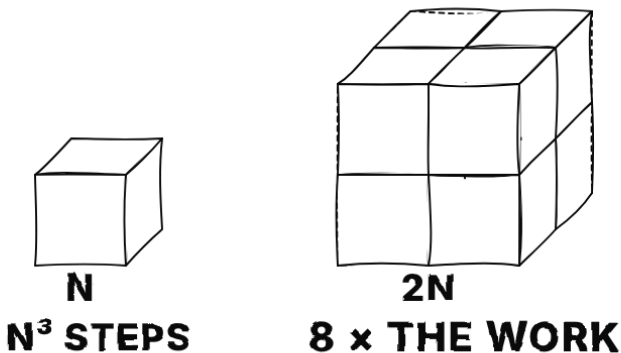
Hint: Ask which one works for boxes of any shape.

Answer: Entry by entry, the exponential is applied to each number separately and works for boxes of any shape. The true matrix exponential is built from repeated matrix multiplications and works only for square matrices. Because matrix-only operations such as this one, inverses and determinants are special, libraries like PyTorch keep them in a separate module.

How Much Work Is a Multiplication

Imagine filling in a big multiplication table by hand, where every square needs a full row of a matrix matched against a full column of another. Now make both matrices twice as wide and twice as tall, and fill the new table. You would expect twice the work, but you will find yourself working roughly eight times longer. That surprise is what computer scientists call complexity, and the book uses matrix multiplication to introduce it.

$$O(N^3)$$



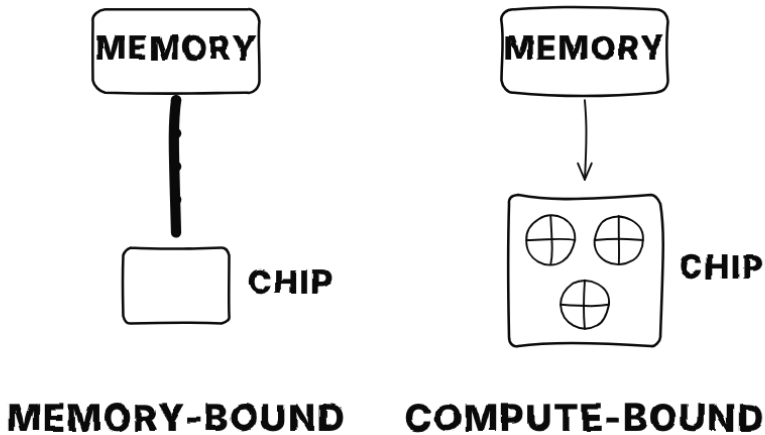
Twice the Size, Eight Times the Work

To multiply two matrices directly by the definition, you compute one dot product for every square of the result, and each of those dot products is as long as the shared side of the two matrices. The total number of steps is called the time complexity. The memory needed, if you compute the squares one after another, is just the size of the result, and that is called the space complexity.

To talk about cost without tying it to a particular computer, computer science uses what is called big O notation, from the German word for order. It describes how the cost grows as the problem grows, and deliberately ignores fixed factors that stop mattering once the problem is large enough; this kind of reasoning is called asymptotic analysis. In those terms, the plain way of multiplying matrices grows in proportion to each of the three sizes involved. For two square matrices it grows with the cube of their size, which is exactly why doubling the size meant eight times the work.

The book warns that this elegant view does not tell the whole story. Cleverer methods for multiplying square matrices are known that grow noticeably more slowly than the cube, yet they are hard to spread across the thousands of small workers inside a graphics chip, so in practice almost nobody uses them. What actually happens when you ask a library such as PyTorch to multiply two matrices also depends on the machine: the library picks a low-level routine written for that hardware, one for the main processor, another for a graphics chip, another for specialised chips. Compilers can go further and merge several operations into one, tuned to a specific chip. The same chat assistant can run on a phone or in a data centre, and quite different routines do the multiplying underneath.

Two more limits of the theory are worth remembering. Having many processors working in parallel only divides the work by a fixed factor, so it does not change how the cost grows with size. And the theory ignores the time spent moving data from one place in memory to another, which in real programs is often the true bottleneck. When moving data is what holds a program back, it is called memory-bound; when the arithmetic itself is the limit, it is called compute-bound. The only reliable way to tell which is which is to measure the running program with a profiler, and the book promises that the interplay between theory and measurement will keep coming back.



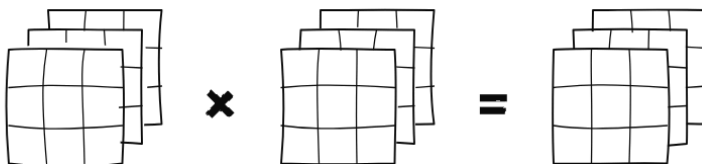
Memory-Bound or Compute-Bound

Try it yourself

Think of a kitchen with one cook and a fridge in the next room. If every dish needs one quick chop but a walk to the fridge for each ingredient, hiring more cooks hardly helps: the walking is the bottleneck, and the kitchen is memory-bound. If every dish instead needs hours of chopping with everything already on the counter, more hands help directly: the kitchen is compute-bound.

Finally, the book looks beyond vectors and matrices. Many special operations, such as determinants or the matrix exponential, make sense only for matrices, so most operations on bigger boxes are either many matrix operations done side by side or combinations of matrix operations with reductions. The first kind is the batched matrix multiplication: take a stack of matrices and a second stack of the same length, and multiply them pair by pair, getting a stack of results. Libraries treat the first directions of a box as batch directions without being asked, so in PyTorch the batched version is written exactly like an ordinary matrix multiplication, as the book's code box shows. When a phone app processes many photos at once, it relies on exactly this habit of handling whole stacks in one call.

BATCH OF 3



FLATTEN, THEN DOT

Stacks Multiplied Pair by Pair

The second kind is a generalised dot product: two boxes of the same shape, with any number of directions, multiplied entry by entry and then all summed up. It is nothing more than flattening each box into one long row of numbers and taking an ordinary dot product. With these operations, the book says, it has covered most of the tensor operations needed for the rest of the book.

Remember

Multiplying matrices directly costs one dot product per entry of the result, the time complexity, and needs memory for the result, the space complexity. Big O notation describes how cost grows with size, and for square matrices the plain method grows with the cube of the size, so doubling the size means about eight times the work. Faster methods exist on paper but suit parallel chips poorly, libraries pick hardware-specific routines, and many processors only divide the cost by a fixed factor. Moving data is often the real bottleneck, the difference between memory-bound and compute-bound programs, and only measuring tells which applies. Bigger boxes are handled as stacks of matrix operations side by side, or as generalised dot products of flattened boxes.

Questions

1. Two square matrices are made twice as large on each side. Roughly how does the work of the plain multiplication change?
 - a. It doubles
 - b. It grows about four times
 - c. It grows about eight times
 - d. It stays the same on a graphics chip
2. Why are the cleverer, faster-growing-slower methods for multiplying matrices rarely used in practice?
 - a. They give wrong answers for large matrices

- b. They are hard to spread across the many small workers of a graphics chip
 - c. They need matrices that are not square
 - d. They are forbidden by big O notation
3. A program barely gets faster when you add more processors, and a measurement shows it spends most of its time fetching data. What is it?
- a. Compute-bound
 - b. Memory-bound
 - c. Asymptotically cubic
 - d. Space-complexity free
4. What does big O notation deliberately leave out?
- a. How the cost grows as the problem grows
 - b. Fixed factors that stop mattering once the problem is large enough
 - c. The size of the inputs
 - d. Whether an operation takes time at all

Answers

- 1. c)** For square matrices the plain method grows with the cube of the size, so doubling the size means about eight times the work.
- 2. b)** Although they grow more slowly than the cube on paper, they parallelise poorly on highly parallel hardware such as graphics chips.
- 3. b)** When moving data, not arithmetic, holds a program back it is memory-bound, and only a profiler can show this reliably.

4. b) Big O describes how cost grows with size and ignores fixed factors, which is why many parallel processors do not change it.

Worked examples

The book's batched multiplication In the book's code box, the first stack holds 4 matrices of 5 rows and 2 columns, and the second holds 4 matrices of 2 rows and 3 columns. What shape comes out of the batched multiplication?

The first direction is the batch: there are 4 pairs, multiplied pair by pair, so the result is a stack of 4 matrices.

In each pair, a 5-by-2 matrix meets a 2-by-3 matrix; the shared side of 2 matches, so they can be multiplied.

Each product has one row per row of the first matrix and one column per column of the second: 5 rows and 3 columns.

Answer: A stack of 4 matrices, each with 5 rows and 3 columns.

Practice

1. Explain why having many processors does not change the big O complexity of an operation, even if it makes the program faster.

Hint: What kind of speed-up do many processors give at best?

Answer: Many processors can at best divide the work by a fixed factor, the number of processors. Big O ignores fixed factors and only describes how cost grows with size, so the growth stays the same even though each run finishes sooner.

2. Describe the generalised dot product of two boxes with three directions in your own words, and say how it relates to the ordinary dot product.

Hint: Think of unrolling each box into one long row.

Answer: Two boxes of the same shape are multiplied entry by entry and every product is added up into one number. It is exactly the ordinary dot product of the two boxes after each has been flattened into one long row of numbers.

A Recipe in Letters, and the Slope of a Curve

Imagine giving every direction of a box of numbers a letter, like labelling the drawers of a cabinet: one letter for which photo, one for which row, one for which colour. Now describe an operation just by listing the letters of what goes in and the letters of what should come out. Any letter that disappears on the way has been added up. That tiny idea, borrowed from a shorthand Albert Einstein used for physics, is called `einsum`, and the book presents it as an optional but very handy section.

Take the two operations from the last part. The batched matrix multiplication becomes a short string saying: two stacks come in, with a shared letter in the middle, and a stack comes out without that letter; the shared letter is the one summed away, exactly the dot products of rows and columns. The generalised dot product becomes an even shorter string: two boxes with the same three letters come in, and nothing comes out, so everything is summed into one number. There is a direct match between these strings and the full written definitions, and most libraries offer a function called `einsum` that reads them.

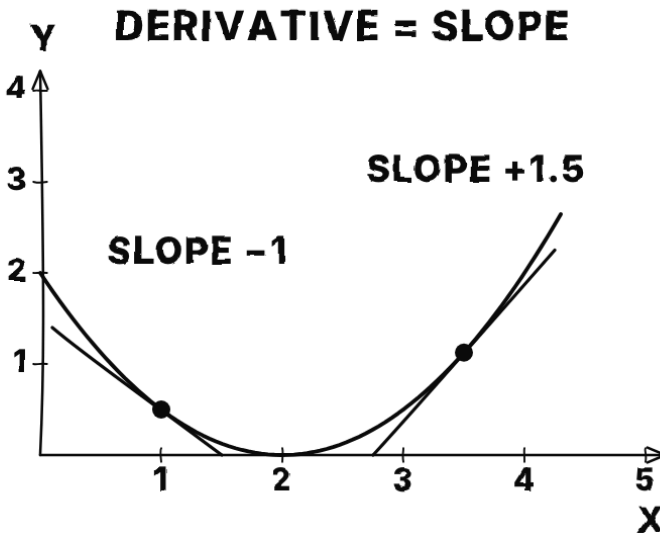
The advantage is practical. You no longer need to remember which of a library's several multiplication functions has which shape rules, because one string covers them all. The same string works unchanged in PyTorch and in JAX, so moving code from one library to the other becomes transparent. Flipping a matrix inside an operation is as simple as swapping two letters in the string. For these reasons

einsum, and tools that extend it such as a popular package called einops, have become widely used.

The book now turns to the heart of the word differentiable: gradients. They are what lets models improve themselves almost automatically, through a method called gradient descent, and the book recalls the essentials, preferring usefulness over full mathematical rigour. It begins with the simplest case, a function that takes one number and gives one number.

Try it yourself

Draw a smooth curve that dips down and then climbs again, like a shallow valley, and choose a point on its slope. Lay a ruler so that it just touches the curve at that point without cutting through it. Now move along the curve a tiny step to the right: if the ruler tilts upward, the curve rises there; if it tilts downward, the curve falls. The tilt of the ruler is the derivative at that point.



The Slope of a Touching Line

That is exactly the book's picture. The derivative is defined by asking how much the output changes for a tiny change of the input, as that tiny change shrinks towards nothing. Geometrically it is the slope of the line touching the curve at a point, and also the best straight-line imitation of the function close to that point. The book's figure shows a gentle valley-shaped curve with two such touching lines at two different points, one sloping down and one sloping up. A positive slope means that, in a small neighbourhood, the function goes up to the right and down to the left, and a negative slope means the opposite. It is like the tilt sensor in your phone: it tells you which way the ground slopes right where you stand, not what lies a mile away. The book stresses that this local view carries over unchanged to functions of many numbers.

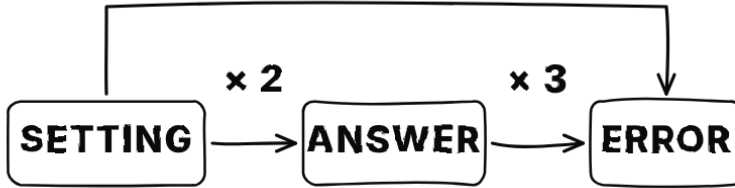
The familiar rules from school, for powers, logarithms and sines, all come straight from that definition. The book does not worry about whether a derivative exists, which can fail even for an unbroken curve, for example at the sharp corner of the absolute value; it returns to such corners in chapter six.

Three rules carry over to the many-dimensional case and will be used constantly. The sum rule says that the slope of a sum is the sum of the slopes. The product rule says that when two quantities are multiplied, the change of the product comes from each one changing while the other stands still, added together. The chain rule says that when one function feeds into another, the slopes along the chain multiply. If a training run changes a model's setting, which changes its answer, which changes its error, the overall effect of the setting on the error is those effects multiplied along the chain, and this is the rule the whole book will lean on.

SUM: SLOPES ADD

PRODUCT: EACH IN TURN

$\times 6$



$$2 \times 3 = 6$$

Slopes Multiply Along a Chain

Remember

Einsum describes almost any tensor operation as a short string of letters, one per direction, where letters missing from the result are summed away. One string replaces many library functions, works the same in PyTorch and JAX, and turns flipping into a swap of letters. The derivative is the slope of the line touching a curve, the best straight-line imitation of the function near a point, telling whether it rises or falls nearby. The sum, product and chain rules extend to many dimensions, and the chain rule, multiplying slopes along a chain of functions, is the one the book will use most.

Questions

1. In an einsum string, a letter appears in the inputs but not in the output. What happens along that direction?
 - a. It is copied unchanged into the result
 - b. It is added up, summed away
 - c. It is flipped, like a transpose
 - d. It causes an error, because every letter must appear in the output
2. At a point where the slope of a function is negative, what does the function do close to that point?
 - a. It goes down to the right and up to the left
 - b. It goes up to the right and down to the left
 - c. It stays flat in both directions
 - d. It must have a sharp corner
3. Why does the book find einsum so useful in practice?
 - a. Because it is faster than every other multiplication
 - b. Because it only works in PyTorch
 - c. Because one string replaces many library functions and works the same in PyTorch and JAX
 - d. Because it computes derivatives automatically
4. A model's setting affects its answer, and the answer affects its error. How does the chain rule combine the two effects?
 - a. It adds them
 - b. It keeps only the larger one

- c. It multiplies them along the chain
- d. It ignores the first one

Answers

1. **b)** By the einsum convention, any letter that disappears on the way from inputs to output is summed over.
2. **a)** A positive slope means rising to the right and falling to the left in a small neighbourhood; a negative slope means the opposite.
3. **c)** One string covers the different multiplication functions with their shape rules and moves unchanged between libraries.
4. **c)** When one function feeds into another, their slopes multiply, so the effect of the setting on the error is the product of the effects along the chain.

Practice

1. In your own words, explain why the derivative at a point is called the best straight-line imitation of the function there, using the ruler touching a curve.

Hint: Think about what the ruler and the curve do very close to the touching point.

Answer: The ruler touches the curve at the point and tilts the same way the curve is heading there, so for a tiny step to either side the ruler and the curve almost coincide. No other straight line follows the curve as closely near that point, and the tilt of that line is the derivative.

2. A setting changes a model's answer three times as fast as the setting itself moves, and the answer changes the error twice as fast as the answer moves. By the chain rule, how fast does the error change with the setting?

Hint: The chain rule multiplies the effects along the chain.

Answer: Six times as fast: the chain rule multiplies the two effects, three times two.

3. Describe, without symbols, what the einsum string for the generalised dot product says, and why nothing appears after the arrow.

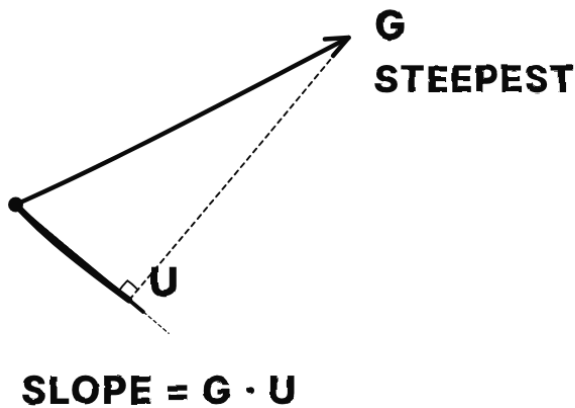
Hint: Which letters appear in the inputs, and which in the output?

Answer: Two boxes with the same three letters go in, so they are multiplied entry by entry. Since no letter appears in the output, every direction is summed away, and the result is a single number.

Slopes in Every Direction

Stand on a grassy hillside and ask a simple question: is the ground going up or down here? On a hill there is no single answer. Step east and you might climb, step north and you might descend, and step north-east and you might stay almost level. With one number as input, a slope only had left and right; with many numbers as input there are endless directions, and the book shows how to handle them all.

The simplest move is along one axis only, keeping every other coordinate fixed, like walking strictly east. The slope you feel then is called a partial derivative. Do this once for every axis, east, north, and so on for as many directions as the input has, and stack all those slopes into one list. That list is the gradient of the function, and it is so important that the book gives it a special symbol of its own. You can picture it as an arrow lying on the hillside.



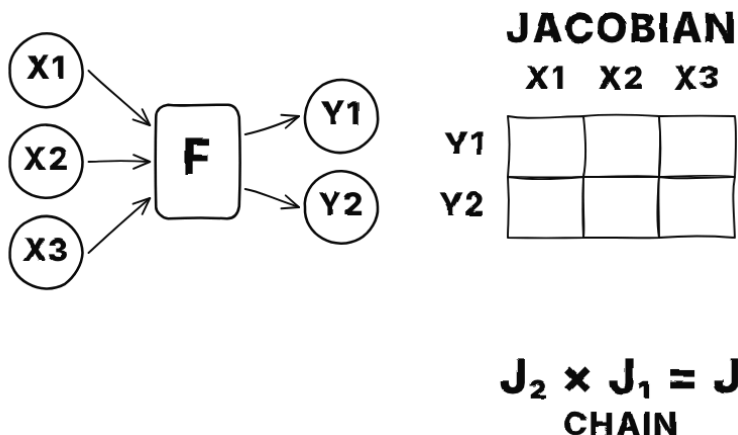
Slope in Any Direction

What about walking in any other direction, say a little east and a lot north? The slope along an arbitrary direction is called the directional derivative. Because any movement can be broken into pieces along each axis, it turns out to be simply the dot product of the gradient with the direction you walk in. So the gradient alone is enough: from that one arrow you can read off the slope in every possible direction. When a photo app is being trained, its millions of settings are the axes of an enormous hillside, and the gradient tells, for every setting at once, which way makes its mistakes grow.

Try it yourself

Tilt a tray on a table and roll a marble onto it from different starting directions. Whichever way you push it, it ends up running down along one particular line, the steepest one. That line is fixed by the tilt of the tray alone, and every other direction's slope is just a share of it, which is exactly what it means that the gradient determines all directional slopes.

Now let the function give back several numbers instead of one. Each output has its own gradient, and stacking these gradients, one per row, gives a grid called the Jacobian, with one row per output and one column per input. With a single output the Jacobian is just the gradient, and with a single input and a single output it is the ordinary derivative. The book says this is the most general case it will ever need.

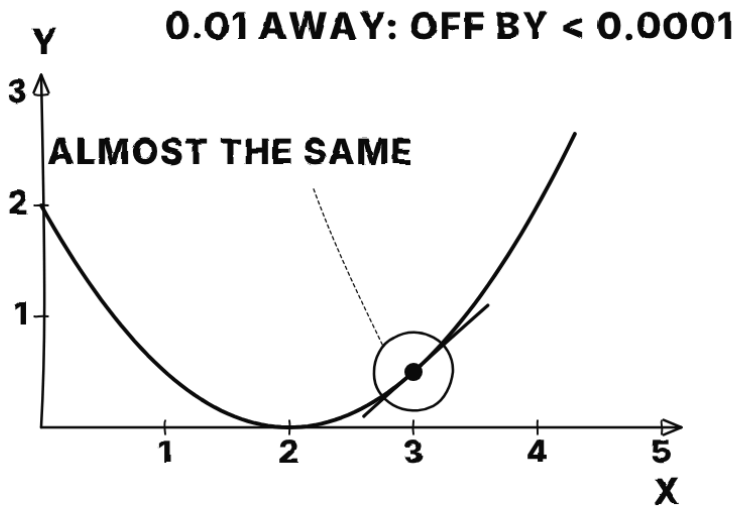


Gradients Stacked: the Jacobian

Jacobians keep all the good habits of derivatives. Most importantly, when two functions are chained, the Jacobian of the chain is the product of their Jacobians, a matrix multiplication, which is the chain rule grown up. A model is a long chain of steps, so this rule lets the effect of every setting on the final error be assembled by multiplying the grids of all the steps in between.

Gradients and Jacobians also have a geometric meaning, like the touching ruler from before. Near a chosen point, the function behaves almost exactly like a flat, straight-line version of itself: its value at that point, plus the gradient matched against how far you have moved. This is the best first-order approximation, a result known as Taylor's theorem. The book checks it in code on the same valley-shaped curve

as before: one hundredth of a step away from the chosen point, the straight-line guess and the true value agree to within a ten-thousandth.



The Straight-Line Guess

Remember

With many inputs, a slope depends on direction; moving along one axis gives a partial derivative, and stacking them all gives the gradient, an arrow. The slope in any direction is the dot product of the gradient with that direction, so the gradient tells everything about local slopes. With several outputs, the gradients stack into the Jacobian, one row per output, and chained functions multiply their Jacobians. Near any point a function behaves like its straight-line approximation built from the gradient, which is Taylor's theorem.

Questions

1. Why is knowing the gradient enough to know the slope in every direction?
 - a. Because a function with many inputs has only one direction
 - b. Because any movement splits into pieces along the axes, so the slope is the dot product of the gradient with the direction
 - c. Because the gradient is always zero
 - d. Because the directional derivative ignores the direction
2. A function takes 5 numbers and returns 3 numbers. What does its Jacobian look like?
 - a. A grid with 3 rows and 5 columns, one row per output
 - b. A single list of 5 slopes
 - c. A grid with 5 rows and 3 columns, one row per input

- d. A single number
3. Two functions are chained. How is the Jacobian of the chain obtained?
- a. By adding the two Jacobians
 - b. By multiplying the two Jacobians, a matrix multiplication
 - c. By keeping only the Jacobian of the last function
 - d. By multiplying them entry by entry
4. What does Taylor's theorem say about a function close to a chosen point?
- a. That it equals its value at the point everywhere
 - b. That it behaves almost exactly like a straight-line version built from its value and its gradient
 - c. That it must be a polynomial
 - d. That its gradient is the same everywhere

Answers

1. **b)** The directional derivative equals the dot product of the gradient with the direction of movement, so the gradient alone gives every directional slope.
2. **a)** Each output has its own gradient, one per row, so the Jacobian has one row per output and one column per input.
3. **b)** The chain rule grows up into a matrix multiplication of the Jacobians of the chained functions.
4. **b)** Near the point the function is best imitated by its value plus the gradient matched against the displacement: the first-order approximation.

Worked examples

The book's straight-line check The book takes the valley-shaped curve $f(x) = x^2 - 1.5x$, whose slope is $2x - 1.5$, builds its straight-line approximation at $x = 0.5$, and compares it with the true curve at $x = 0.51$. What values does it get?

Value at the chosen point: 0.5 squared is 0.25, minus 1.5 times 0.5, which is 0.75, gives minus 0.5.

Slope at the chosen point: 2 times 0.5 minus 1.5 gives minus 0.5.

Straight-line guess one hundredth to the right: minus 0.5 plus the slope times 0.01, so minus 0.5 minus 0.005, which is minus 0.505.

True value: 0.51 squared is 0.2601, minus 1.5 times 0.51, which is 0.765, gives minus 0.5049.

Answer: The guess is minus 0.5050 and the true value minus 0.5049: they differ by only one ten-thousandth.

Practice

1. Explain in your own words the difference between a partial derivative and a directional derivative, using the hillside.

Hint: One is about walking along a single axis, the other about any chosen direction.

Answer: A partial derivative is the slope you feel when you walk strictly along one axis, say due east, keeping all other coordinates fixed. A directional derivative is the slope along any direction you choose, for example a little east and a lot north; it combines the partial derivatives according to how much you move along each axis.

2. Why does the chain rule for Jacobians matter so much for models, according to this part?

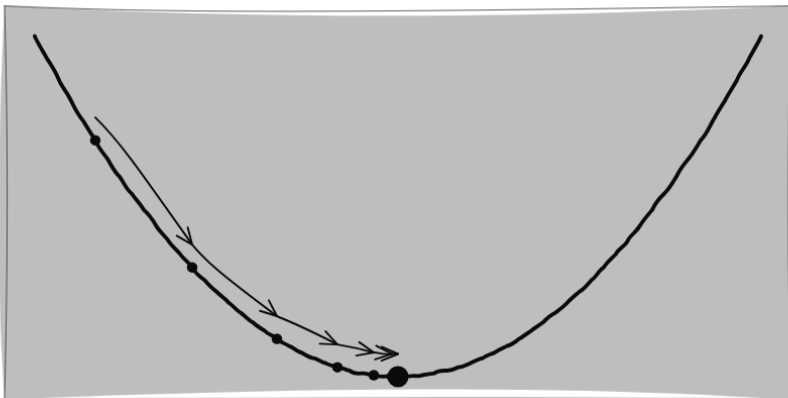
Hint: Think of a model as a long chain of steps.

Answer: A model is a long chain of steps, and the effect of each setting on the final error passes through every step in between. Because the

Jacobian of a chain is the product of the Jacobians of its steps, that effect can be assembled by multiplying the grids of all the steps.

Walking Downhill in the Fog

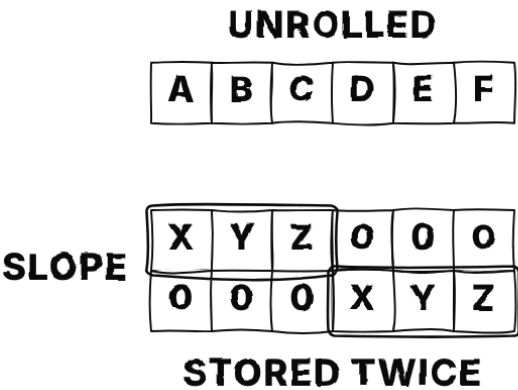
Imagine you are on a mountain in thick fog and want to reach the bottom of the valley. You cannot see the valley, only the ground right under your feet. So you feel which way the ground drops most steeply, take a small step that way, and feel again. Step after step, without ever seeing the whole landscape, you work your way down. That is gradient descent, the method at the centre of this part, but first the book adds a careful note about the sizes of Jacobians.



MINIMUM

Feeling the Way Down

Take the simplest model step: a grid of weights multiplied by an input vector. Seen as a function of the input, its Jacobian is simply the weight grid itself, which is reassuring: the slope of a linear machine is the machine. Seen as a function of the weights, the input to the function is itself a grid, and the slope becomes a box with three directions, one for each output and two for the position of each weight. The book's trick is to imagine the weight grid unrolled into one long row of numbers; then the slope is an ordinary grid again. This is why the book can write everything with vectors without losing generality.

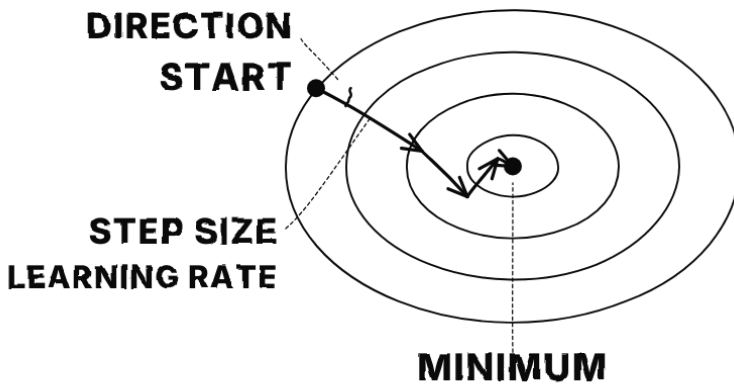


Unrolling the Weight Grid

A small box works this out entry by entry. Each output is a sum of weights times input entries, so the slope of an output with respect to

one of its weights is just the matching entry of the input. Writing the whole slope box into memory would therefore mean storing the same input numbers over and over. The book promises that in chapter six this waste disappears, because in practice we never need the box itself, only what it does when applied to another box. It also warns that the neat grids written on paper can hide huge sizes inside, above all the batch size, so the real memory cost may be far larger than the notation suggests. A training batch of thousands of photos makes such hidden boxes enormous, which is why real training software never builds them.

Now the main event: finding the lowest point of a function, its minimum. The book assumes the function gives a single number and that the input may wander anywhere. In the rest of the book the input will be the model's parameters and the function will measure how badly the model does on the data, the supervised learning setup of the next chapter. Minimising is no less general than maximising: turn the curve upside down and its highest peaks become its deepest valleys.

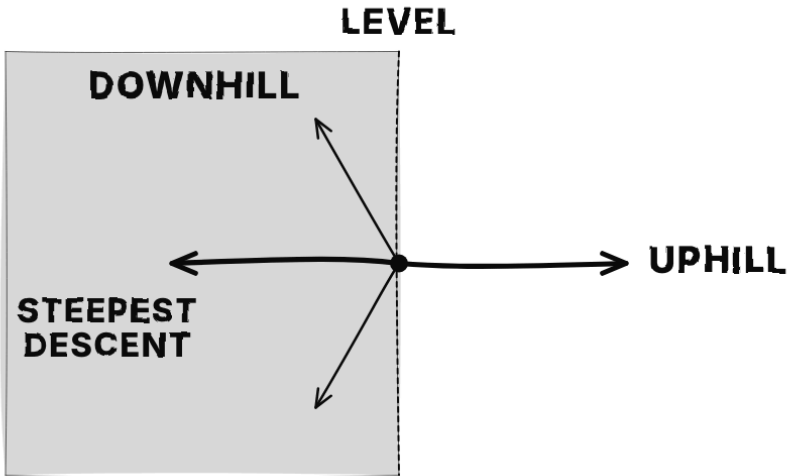


Step by Step to the Bottom

Only in rare cases can the lowest point be written down directly with a formula; the book will show one such case later, for fitting straight lines. Usually we must search step by step. We start from a random guess and repeatedly take a step, which has two ingredients: how long it is, called the step size, and which way it points, its direction. In machine learning the step size is called the learning rate, for reasons the next chapter makes clear. A direction that can lower the function, with a small enough step, is called a descent direction, and if every step uses one and the step sizes are chosen with care, the search settles on a minimum.

Try it yourself

Put a ball on a slanted board and give it small nudges in different directions. Push it straight up the slope and it climbs; push it sideways along the slope and it stays at the same height; push it in any direction that points even slightly downhill and it descends. Push it straight down the slope and it descends fastest of all.



Which Directions Go Downhill

That is exactly what the gradient tells us, through the directional slope from the previous part. The slope in a direction is the dot product of the gradient with that direction, which means the length of the

gradient times how well the direction is aligned with it. The length of the gradient does not depend on the direction we choose, and we can take every direction to have length one, so only the alignment decides. Any direction pointing more than a right angle away from the gradient goes downhill, and the one pointing exactly opposite to the gradient goes down most steeply of all: the steepest descent direction.

Put the two ideas together and you have an algorithm to minimise any differentiable function, steepest gradient descent. From the current guess, step a little against the gradient, then compute the gradient again at the new spot, and repeat. The book does not deal with choosing the step size; it simply assumes the steps are small enough that each one lowers the function. Training the model behind a chat assistant is, at its core, this same walk repeated a vast number of times on a landscape with billions of directions. Gradient descent is only as fast as our way of computing gradients, and a general method for that is the subject of chapter six.

Remember

Seen from the input, the Jacobian of a weight grid times a vector is the weight grid itself; seen from the weights, it is a bigger box that can be flattened into an ordinary grid. Such boxes repeat many values and hide batch sizes, so practical methods only apply them, never store them. To minimise a function we usually search step by step, each step having a length, the learning rate, and a direction. Any direction more than a right angle away from the gradient goes downhill, and the exact opposite of the gradient goes down fastest. Gradient descent repeats small steps against the gradient, and its speed depends on how fast gradients can be computed.

Questions

1. Among all directions of length one, which one makes a function drop fastest?
 - a. The direction of the gradient
 - b. Any direction at a right angle to the gradient
 - c. The direction exactly opposite to the gradient
 - d. Every direction drops equally fast
2. Why does the book say we can study minimising without losing anything about maximising?
 - a. Because functions never have maximum points

- b. Because turning a function upside down makes its peaks into valleys
 - c. Because gradient descent can only find maximum points
 - d. Because maximising needs no gradient
3. Why do practical methods avoid storing the full slope box of a weight grid times an input?
- a. Because it would repeat the same input values many times and hide large batch sizes
 - b. Because it is always zero
 - c. Because it cannot be computed at all
 - d. Because it has fewer entries than the input
4. What decides how fast gradient descent can run in practice, according to the book?
- a. The colour of the data
 - b. How fast the gradients can be computed
 - c. Whether the function is written with letters
 - d. How many times the curve is turned upside down

Answers

- 1. c)** Only the alignment with the gradient decides the slope, and pointing exactly opposite to it gives the most negative slope: the steepest descent direction.
- 2. b)** Minimising a function is the same as maximising its upside-down version, so one method covers both.
- 3. a)** Each slope with respect to a weight is just a matching input entry, so the box is full of repeats, and batch sizes can make it enormous; only its effect on another box is needed.

4. b) Gradient descent is only as efficient as the procedure for computing gradients, the subject of chapter six.

Practice

1. A walker on a slope tries a direction that makes an angle of less than a right angle with the gradient. Does the walker go downhill or uphill, and why?

Hint: Recall what alignment with the gradient means for the slope.

Answer: Uphill. A direction less than a right angle away from the gradient is positively aligned with it, so the slope in that direction is positive; only directions more than a right angle away go downhill.

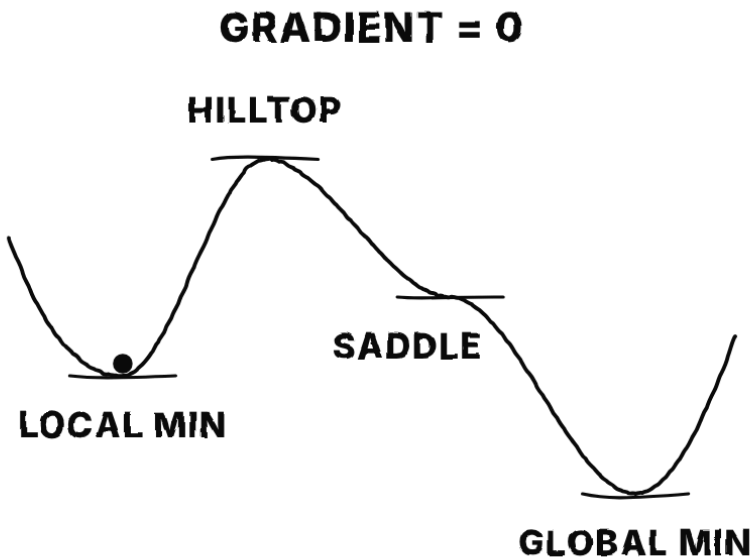
2. Describe gradient descent in your own words, naming the two ingredients of every step and what happens between steps.

Hint: Think of the walker in the fog.

Answer: Start from a guess. Each step has a length, the step size or learning rate, and a direction, which for steepest descent is exactly opposite to the gradient. After the step, the gradient is computed again at the new spot and the process repeats, with steps small enough that each one lowers the function.

Where the Walk Can End

Pour a little water onto a crumpled sheet of aluminium foil and watch where it settles. It does not all run to the single lowest spot of the sheet; it collects in whichever small dent happens to be nearest. Gradient descent behaves the same way, and the book uses this to ask a precise question: when the walk downhill stops, where exactly has it stopped? The author adds that if you simply trust gradient descent, you may skip ahead; this part is for those who want to know what it guarantees.



Flat Places on a Crumpled Curve

A local minimum is a point that is lower than everything in a small neighbourhood around it, the bottom of one dent in the foil. At such a point the touching line is flat, and all around it the ground rises. The flatness is captured by a broader idea, the stationary point: any point where the gradient is zero, where there is no slope at all.

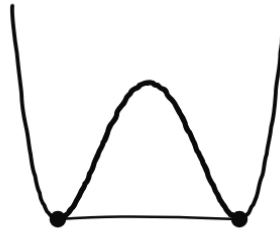
Not every flat point is a dent. A hilltop is flat too, and so is a saddle point, a place where the ground curves up in one direction and down in another, like the middle of a horse's saddle or a mountain pass. The book's figure shows such a saddle, and invites you to picture the flat touching line there. Without any further knowledge of the function, the only thing one can prove is that gradient descent ends at some stationary point, and which one depends on where it started.

Some landscapes are friendlier. Think of a simple bowl, like a valley carved in one smooth curve: it has no saddles and only one bottom, and that bottom is the lowest point of the whole landscape, a global minimum. From any starting point, every slope in a bowl leads to that bottom, so gradient descent cannot miss it. The book generalises the bowl with the idea of a convex function.

CONVEX



NOT CONVEX



The Thread Test

Try it yourself

Draw any valley-shaped curve, pick two points on it, and stretch a thread straight between them. If the curve between the two points always stays below the thread, whichever two points you choose, the curve is convex. Now draw a curve with two dips and a bump between them and try again: with the two points on either side of the bump, the curve rises above the thread, and it is not convex.

Convexity tells you how easy the search is. On a non-convex landscape, gradient descent ends at some stationary point, and

nothing more can be said without looking at how the slopes themselves change. On a convex landscape, it reaches a global minimum from any starting point. And if the curve stays strictly below every thread, the global minimum is also unique. For non-convex problems the bad news is severe: to be sure of the global minimum you would have to restart the search from every possible starting point, which makes the task hopelessly hard in general.

This has deep historical weight. As chapter five will show, any model worth using is non-convex, so its training landscape may be full of dents and saddles. Other learning methods, such as support vector machines, were attractive precisely because they keep a convex, safe search while still handling non-straight patterns. Yet complex differentiable models work remarkably well in practice: started from a sensible point, training tends to end at spots that perform well on real data. The chat assistant you use was trained on such a crumpled landscape, and it works even though nobody can promise its training found the very lowest point.

Remember

A local minimum is lower than its close surroundings, and it is a stationary point, a place with zero gradient, like hilltops and saddle points. In general, gradient descent is only guaranteed to stop at some stationary point, depending on where it starts. A convex function stays below the straight line between any two of its points, like a bowl, and there gradient descent reaches a global minimum from anywhere, a unique one if the convexity is strict. Useful models are non-convex, unlike support vector machines, yet in practice their training lands on points that work well.

Questions

1. Gradient descent stops where the gradient is zero. Why does that not guarantee it found a minimum?
 - a. Because the gradient is never exactly zero
 - b. Because a zero gradient also happens at hilltops and saddle points
 - c. Because minima always have a large gradient
 - d. Because gradient descent only works on bowls
2. On a convex function, what can be said about gradient descent?
 - a. It reaches a global minimum from any starting point
 - b. It always gets stuck on a saddle point
 - c. It must be restarted from every possible point

- d. It only works if started at the minimum
3. Why were support vector machines attractive compared with neural networks?
- a. They need no data
 - b. They keep a convex, safe search while still handling non-straight patterns
 - c. They always find saddle points
 - d. They are trained with larger steps

Answers

- 1. **b)** Any point with zero gradient is a stationary point, and stationary points include maxima and saddle points as well as minima.
- 2. **a)** Like a bowl, a convex function leads every slope to its lowest point, so gradient descent reaches a global minimum from anywhere.
- 3. **b)** Useful neural models are non-convex, whereas support vector machines handle non-linear patterns while keeping the optimisation convex.

Practice

1. Explain in your own words the difference between a local minimum and a global minimum, using the crumpled foil.

Hint: One is about a small neighbourhood, the other about the whole sheet.

Answer: A local minimum is the bottom of one dent: lower than everything close to it, but not necessarily the lowest point of the sheet. A global minimum is the lowest point of the whole sheet. Water poured on the foil usually settles in the nearest dent, a local minimum, not necessarily the global one.

2. Why is finding the global minimum of a non-convex function with gradient descent so hard in general?

Hint: Think about how the starting point decides where the walk ends.

Answer: On a non-convex landscape, where the walk ends depends on where it starts, and it may stop in any dent or flat spot. To be sure of the very lowest point you would have to start from every possible point, which is hopelessly expensive in general.

3. Despite being non-convex, why do neural networks still work well, according to the book?

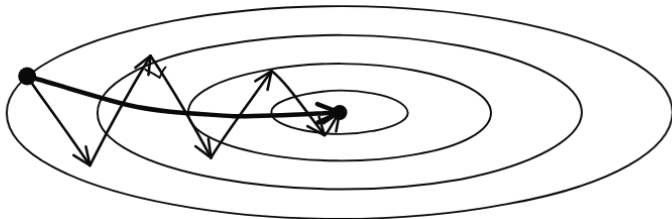
Hint: Think about what happens when training starts from a sensible point.

Answer: In practice, when training starts from a reasonable initialisation, gradient descent tends to end at points that perform well on real data, even if nobody can guarantee they are the global minimum.

A Ball With Momentum

Roll a marble across a rumpled bedsheet. A marble that obeyed only the slope under it at each instant would twitch from side to side at every wrinkle. A real marble does not: it keeps some of its speed and direction from a moment ago, and rolls smoothly over small bumps. That memory of past motion, momentum, is the first trick the book shows for speeding up gradient descent.

NO MEMORY



MOMENTUM

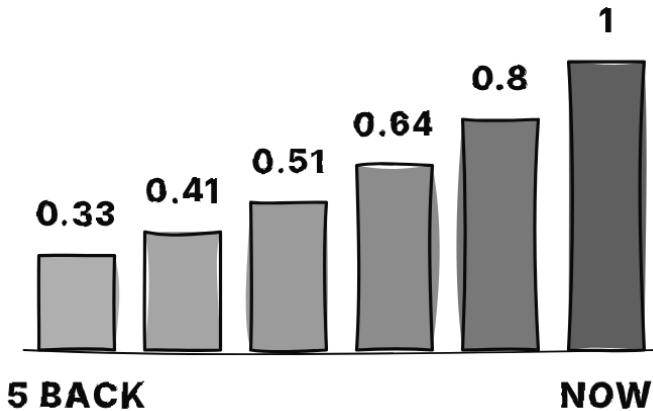
Zigzag Versus Momentum

The opposite of the gradient is the steepest way down, but only in a tiny neighbourhood of the current point. The book warns that, as

chapter five will show, these directions can be very noisy, especially in large models. Many methods have been invented to choose better directions and reach the bottom faster. The book is most interested in those that need neither slopes of slopes, which are expensive, nor many extra evaluations of the function.

Picture plain gradient descent as a ball rolling downhill that has no memory: its path can be erratic, because each new gradient may point somewhere quite different. The book even notes that, with an ideal step size on a bowl-shaped function, two consecutive gradients stand at a right angle to each other, so the path zigzags. Momentum smooths the zigzag by keeping a share of the previous step and adding it to the new one. A single setting decides how much of the past is kept. Unrolling the rule shows that older steps still count, but each one a little less than the one after it, fading away the further back it lies.

KEEP 0.8 OF THE PAST



Older Steps Count Less

The book's figure compares the two on a wavy curve, plain gradient descent and the version with momentum, both started from the same point. Momentum is known to speed up training by smoothing the path. Another common trick gives every parameter its own step size, adapted to how large its gradients have been. A widely used method called Adam combines several of these ideas.

Try it yourself

Fill a bowl with water and stir it in tiny strokes that keep changing direction: the water barely moves. Now stir steadily in one direction for a few seconds: the water starts turning and keeps turning even after you pause. Small pushes that agree with each other add up, and pushes that keep cancelling do not: that is what momentum does to the steps of gradient descent.

Adam's great practical virtue is that it works well without much fiddling with its settings. Settings that the user chooses, rather than ones learned by gradient descent, are called hyper-parameters, and for Adam the library defaults are usually a good start. Many models you meet every day, from photo apps to chat assistants, have been trained with Adam or a close relative of it. Building a new optimiser that can replace Adam, or its variant AdamW, as the default choice remains an open research problem. The price of these accelerations is memory: momentum, for instance, must remember the previous step, which doubles the memory the optimiser itself needs, although computing the gradient usually needs far more memory still.

The chapter ends with a section called From theory to practice. Instead of classic exercises, the book proposes a self-guided path through two programming libraries, JAX and PyTorch, with solutions on the book's website, and warns that web links may have moved by the time you read them. The starting block is NumPy, the standard library for working with boxes of numbers: creating them, cutting slices out of them and transforming them. The author recommends getting comfortable with it, above all with slicing, and points to a public collection of one hundred gentle exercises to test yourself.

Remember

The steepest direction holds only very locally and can be noisy, so faster methods choose better directions without expensive extra derivatives. Momentum keeps a fading share of past steps, smoothing the zigzag path of plain gradient descent and speeding up training. Adam combines momentum with a step size for each parameter, and it works well with its default hyper-parameters, so it remains the standard choice. These tricks cost memory, and the practical path starts with learning NumPy well.

Questions

1. What does momentum add to plain gradient descent?
 - a. A share of the previous step, added to the new one, so the path is smoothed
 - b. A second derivative computed at every step
 - c. A random jump to a new starting point
 - d. A larger step size for every parameter
2. With momentum, how much does a step taken long ago still count?
 - a. Exactly as much as the newest step
 - b. Not at all after one step
 - c. A little less with every step further back, fading away
 - d. More than the newest step

3. Why is Adam so popular as the default optimiser?
 - a. It needs no gradients at all
 - b. It works well without much fiddling, with the library defaults as a good start
 - c. It uses no memory beyond the parameters
 - d. It always finds the global minimum
4. What is the main cost of using momentum?
 - a. It makes the path more erratic
 - b. It must remember the previous step, doubling the optimiser's memory
 - c. It requires the function to be convex
 - d. It stops working on large models

Answers

1. **a)** Momentum keeps a share of the previous step and adds it to the new one, smoothing the zigzag of plain gradient descent.
2. **c)** Unrolling the rule shows that older steps still count, each a little less than the one after it, fading the further back it lies.
3. **b)** Adam is relatively robust to the choice of its hyper-parameters, and the defaults of most libraries are a good starting point.
4. **b)** Momentum stores the previous step, which doubles the optimiser's memory, although computing the gradient usually needs much more memory.

Practice

1. Using the stirred bowl of water, explain why momentum helps when successive gradients keep changing direction.

Hint: Think about what happens to pushes that agree and pushes that cancel.

Answer: Pushes in changing directions largely cancel out, so with momentum the zigzag parts of the steps shrink, while the parts that agree from step to step add up. The path becomes smoother and makes steadier progress downhill.

2. What is a hyper-parameter, and how does it differ from the parameters that gradient descent learns? Give the momentum setting as an example.

Hint: Who chooses it: the user or the training?

Answer: A hyper-parameter is a setting chosen by the user before training, while parameters are adjusted by gradient descent. The share of past steps that momentum keeps is a hyper-parameter: the user picks it, and it is not learned.

From Practice Code to Real Data

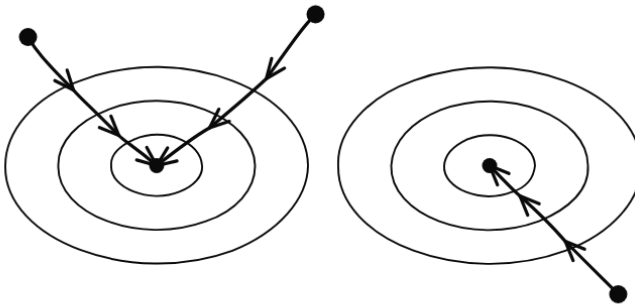
Imagine trying to write, by hand, a list of rules that tells a car from a bicycle in any photo: the colour, the outline, the number of wheels, the shadows, the angle. You would never finish. Yet collecting ten thousand photos labelled car or bicycle is easy. That contrast, rules impossible but examples easy, is the starting point of chapter three, but first the book finishes its practical advice.

NumPy, for all its influence, has two weaknesses for our purposes: on its own it makes little use of graphics chips, and it cannot compute gradients automatically. JAX copies NumPy's way of working and adds support for such hardware, automatic gradients, and extra transformations, such as one that turns a function written for a single input into one that handles a whole batch. PyTorch also has a NumPy-like core, with a few changes of names and behaviour, plus high-level tools for building models. The author suggests skimming the basic array pages of both libraries to see how much they share with NumPy, and ignoring the high-level model-building parts for now.

Then comes a hands-on exercise, to be done three times, once in each library, which should take only minutes once you know the syntax. It uses a function of two numbers with several hills and valleys, and the author adds a candid note: this one function was generated by ChatGPT, and nothing else in the book was written by a language model, a disclaimer the author feels is becoming important. You first write the function so it accepts a whole batch of points at once, then write its gradient by hand, since automatic gradients come only later. Next you run gradient descent from several starting points and draw

the paths, and finally you add momentum and watch the length of the gradient shrink towards zero as the search reaches a flat spot.

START DECIDES THE VALLEY

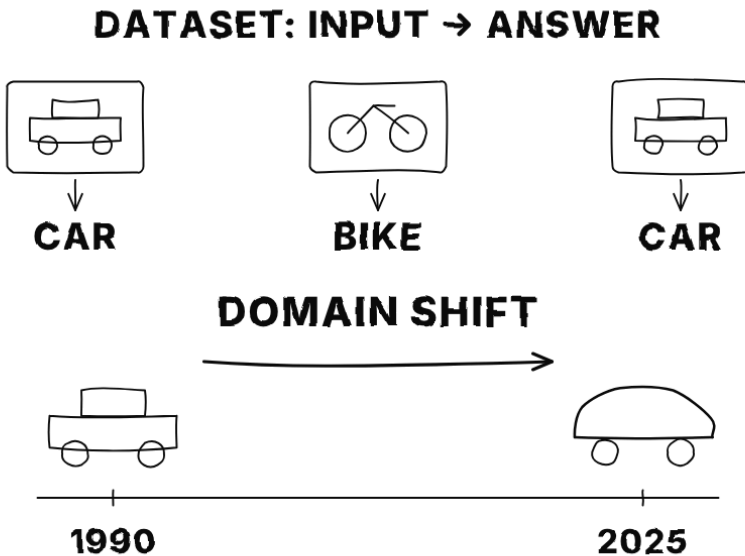


Different Starts, Different Valleys

Try it yourself

If you have a computer at hand, try the exercise's spirit on paper first. Draw a landscape of a few hills and valleys as contour lines, like a hiking map, and mark three starting points. From each one, draw short arrows always pointing straight downhill across the contour lines: you will see different starting points end in different valleys, just as the exercise's plots will show.

Chapter three now makes supervised learning precise: datasets, losses, the idea of minimising the average error on the data, the assumptions behind all this, and a version built on probability. The author calls this short chapter the backbone of the rest of the book. It begins exactly where the car-and-bicycle problem left us: tasks that cannot be programmed by hand but for which examples are easy to gather, from recognising speech to guiding a robot. A dataset is then simply a collection of pairs, each one an input together with the output we want for it.



Pairs From a Changing World

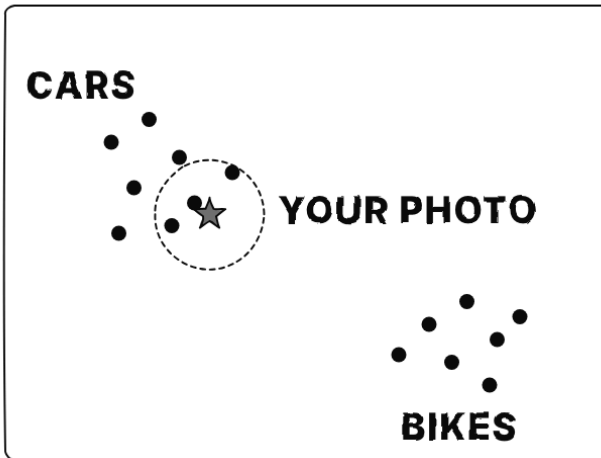
One assumption about how the pairs were collected matters enormously, and the book explains it with cars. The examples should be drawn the same way from a stable, unchanging source. If you

collect car photos over a few months, that holds; if you collect them over several decades, car designs change, and a model trained on them will fail on new models or work poorly. They should also be gathered without bias, so the collection fairly represents everything the model will meet. Photos taken only near one brand's car dealership would be full of that brand and short of all the others. And the answer depends on context: car photos from Italy may suit a model used in Rome or Milan, but not one used in Tokyo.

The book insists that this is not checked once and forgotten; it must be watched for the whole life of a model. If the cars on the road slowly change and nobody notices, the model's performance quietly decays, a problem called domain shift. A recommendation system, like the one choosing what your apps suggest next, even changes the very behaviour it learns from, because people react to its suggestions, creating feedback loops that require constant re-checking. For today's huge language models, trained on such vast and varied data, it becomes hard even to say which tasks count as familiar and which as new.

The chapter's figure adds a picture worth keeping. A trained model transforms its input step by step, and the inner versions of the data it produces often have a meaningful geometry: images with similar meaning land at nearby points in an inner space. Turning raw data, such as photos, into points where distance means similarity is called embedding the data. It is the reason your photo app can find pictures similar to one you pick: similar photos sit close together in its inner space.

EMBEDDING



Similar Things Sit Close

Remember

JAX and PyTorch keep NumPy's way of working but add graphics-chip support and automatic gradients, and the book's exercise implements gradient descent on a hilly function in all three libraries. Supervised learning is for tasks we cannot program by hand but can show by example, and a dataset is a collection of input-output pairs. The examples must come from a stable source and be collected without bias, which depends on context and must be checked throughout a model's life to catch domain shift and feedback loops. Trained models often embed data so that similar items sit close together in an inner space.

Questions

1. A car-recognition model is trained on photos collected over the last forty years. What problem does the book predict?
 - a. The model will be too small to store so many photos
 - b. Car designs changed over time, so the source was not stable and the model may fail on new models
 - c. The photos will be too similar to each other
 - d. The model will only recognise bicycles
2. Why would photos taken only near one brand's dealership make a poor dataset?
 - a. Because they would be collected with bias and would not represent all the cars the model will meet
 - b. Because dealership photos are always blurry
 - c. Because the photos would come from too long a period
 - d. Because a dataset must contain bicycles too
3. What does it mean that a trained model embeds its data?
 - a. It hides the data so nobody can read it
 - b. It turns raw data into points where closeness means similarity
 - c. It stores every photo twice
 - d. It removes all the numbers from the data
4. Why must the assumptions about a dataset be checked throughout a model's life, not just once?
 - a. Because libraries change their names every year

- b. Because the world the model meets can slowly change, and some systems even change the behaviour they learn from
- c. Because datasets lose examples over time
- d. Because gradients grow larger after training

Answers

1. **b)** The examples should come from a stable, unchanging source; over decades car designs change, so the model may fail on new models or perform poorly.
2. **a)** Examples should be gathered without bias; near one dealership, that brand would be over-represented and other makes and models would be missing.
3. **b)** Inside a trained model, items with similar meaning land at nearby points, which is called embedding the data.
4. **b)** Slow changes cause domain shift, and systems like recommenders create feedback loops by changing how people behave.

Practice

1. The book asks you to implement gradient descent on a hilly function of two numbers in NumPy, JAX and PyTorch. List the four steps of the exercise and say what you expect to see in the last one.

Hint: Think of the batch-friendly function, the hand-written gradient, the paths, and momentum.

Answer: First write the function so it handles a whole batch of points at once; second, write its gradient by hand; third, run gradient descent from several starting points and draw the paths; fourth, add momentum and plot the length of the gradient. In the last step the gradient's length should shrink towards zero as the search approaches a flat spot.

2. A car dataset collected in Italy is used for a model deployed in Milan and another deployed in Tokyo. Explain why the assumptions may hold for one and not for the other.

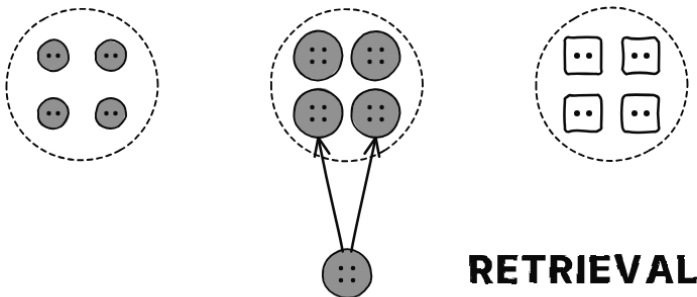
Hint: Ask whether the cars the model meets look like the cars in the dataset.

Answer: Cars on the roads of Milan resemble those photographed elsewhere in Italy, so the dataset represents what that model will meet. In Tokyo the mix of makes and models is different, so the dataset no longer represents the cars the model sees and its performance may suffer.

Learning Without Labels, and Reusing What Was Learned

Tip a box of mixed buttons onto a table and sort them into piles, without anyone telling you what the piles should be. You will probably group them by colour or size or number of holes, and the piles will make sense even though nobody gave you a single label. Supervised learning always had an answer for each example; this part looks at what happens when there are no answers, and at how a model trained once can be reused many times.

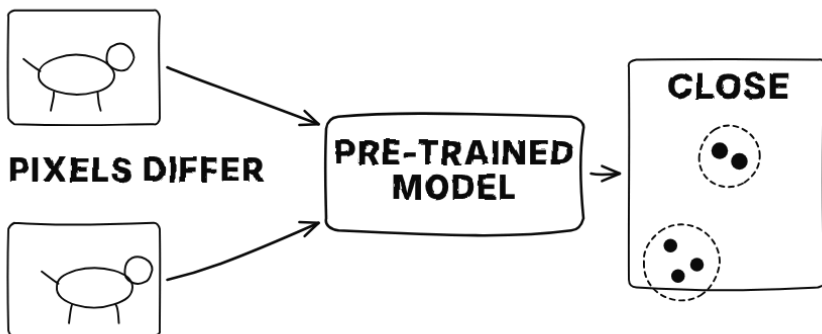
CLUSTERING



Piles Without Labels

When a dataset has inputs but no target answers, the book speaks of unsupervised learning. A classic job is clustering: grouping the data so that items within a pile are similar and items in different piles are not, like the buttons. Another is retrieval: searching a huge collection for the few items most similar to what someone asked for. Asking your photo app to find more pictures like one you selected is exactly such a retrieval.

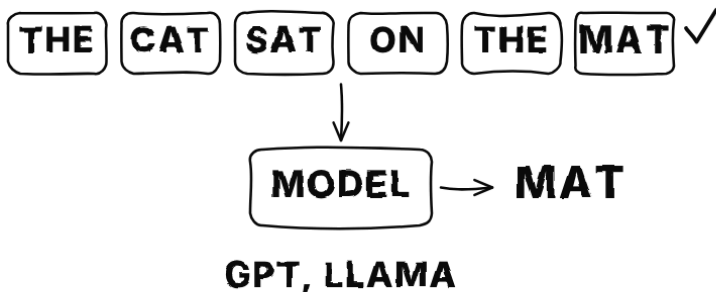
With images, though, similarity is surprisingly hard. Comparing photos pixel by pixel fails, because a tiny shift of the camera changes millions of pixels while the picture means the same thing. The book's way out is a pre-trained model, one already trained for a fairly general task such as naming what is in a photo. Its inner states can be read as points in a space with very many directions, and often similar things land close together there, as we saw with embeddings. So you can take those inner points and run an ordinary clustering method on them, and the piles now follow meaning rather than pixels.



Pixels Disagree, Meaning Agrees

And if no pre-trained model is available? Then a popular variant called self-supervised learning invents a supervised task out of unlabelled data. With a large collection of text, for instance, you can always train a program to guess how a short piece of text continues, because the true continuation is right there in the text. Discovering that networks trained this way also learn excellent embeddings of text had a profound impact on the field. The book adds a warning: web-scale text is full of biases and crude content, models absorb those biases, and this is one of the major criticisms of closed foundation models.

THE ANSWER WAS IN THE TEXT



A Task Hidden in the Text

Today's large language models, such as GPT or Llama, are the modern form of this idea: their training always begins with guessing the next piece of text, and they are among the foundation models met earlier. The book's figure shows three ways to use such a model. Zero-shot use simply asks the model a new question directly. Few-shot prompting puts a handful of worked examples of the new task into the prompt first, and both work only if the model generalises well. Fine-tuning goes further and adjusts the model's parameters by gradient descent on a small dataset for the new task, much like training from scratch, only shorter. When you paste two sample replies into a chat assistant and ask for a third in the same style, you are doing few-shot prompting.

Fine-tuning is easy today thanks to large open collections of shared models. It can adjust all the parameters, or only a small part, or a few added ones, a family of techniques called parameter-efficient fine-tuning, which the author leaves for the next volume. When a model learns a task from examples in its prompt alone, without any fine-tuning, the book calls it in-context learning. Many more variants exist: datasets labelled only in part, which chapter twelve touches on; related datasets or the same data shifting over time; and problems such as adapting to new domains, learning to learn, learning continually, learning distances, and even making a model forget. The book only hints at these and leaves several for its second volume.

Remember

Without target answers, unsupervised learning groups data into clusters or retrieves the most similar items. For images, pixel distances fail, but the inner representations of a pre-trained model place similar things close together, so clustering can work on them. Self-supervised learning invents its own task, such as guessing how text continues, and it is how language models like GPT and Llama begin their training, biases included. A foundation model can be used zero-shot, with a few examples in the prompt, or fine-tuned by gradient descent, in full or with parameter-efficient methods.

Questions

1. Why is comparing two photos pixel by pixel a poor way to judge whether they are similar?
 - a. Because photos have too few pixels
 - b. Because a tiny camera shift changes millions of pixels while the meaning stays the same
 - c. Because pixels cannot be stored as numbers
 - d. Because similar photos always have identical pixels
2. How can self-supervised learning train on a pile of text with no labels at all?
 - a. By asking people to label every sentence
 - b. By inventing a task whose answer is already in the text, such as guessing how a piece of text continues
 - c. By clustering the text into piles at random
 - d. By fine-tuning a model that already exists
3. You give a chat assistant two examples of a task in your prompt and then ask for a third, without changing the model. What is this called?
 - a. Fine-tuning
 - b. Clustering
 - c. Few-shot prompting
 - d. Parameter-efficient fine-tuning
4. What distinguishes fine-tuning from zero-shot and few-shot use?
 - a. Fine-tuning changes the model's parameters by gradient descent on a small dataset
 - b. Fine-tuning needs no data at all

- c. Fine-tuning only works on unsupervised data
- d. Fine-tuning is the same as asking the question directly

Answers

- 1. **b)** Small perturbations change huge numbers of pixels, so pixel distances say little about meaning; a pre-trained model's inner representations work better.
- 2. **b)** The true continuation is right there in the text, so predicting it gives a supervised task for free.
- 3. **c)** Putting a handful of examples into the prompt is few-shot prompting; no parameters are changed, which the book calls in-context learning.
- 4. **a)** Zero-shot and few-shot use leave the model as it is, while fine-tuning adjusts its parameters on examples of the new task.

Practice

- 1. You have thousands of holiday photos and no labels, and you want them grouped by what they show. Describe how the book's ideas let you do it.

Hint: Pixels are not enough; think about where meaningful similarity lives.

Answer: Pass each photo through a pre-trained model, for instance one trained to name objects, and take its inner representation, a point in a high-dimensional space where similar things land close together. Then run an ordinary clustering method on those points, so the groups follow meaning rather than pixels.

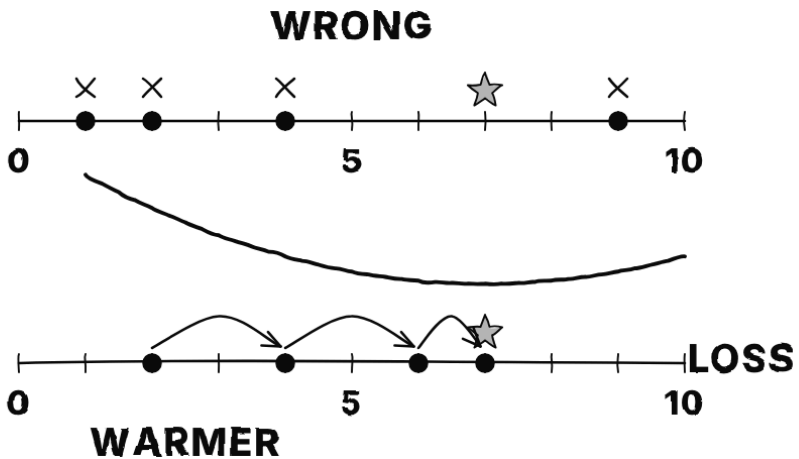
- 2. Explain the book's warning about training on web-scale text in your own words.

Hint: Think about what the text contains and what the model learns from it.

Answer: Text from the web contains biases and crude content. A model trained to imitate that text absorbs those biases, which is one of the main criticisms of closed foundation models whose training data cannot be inspected.

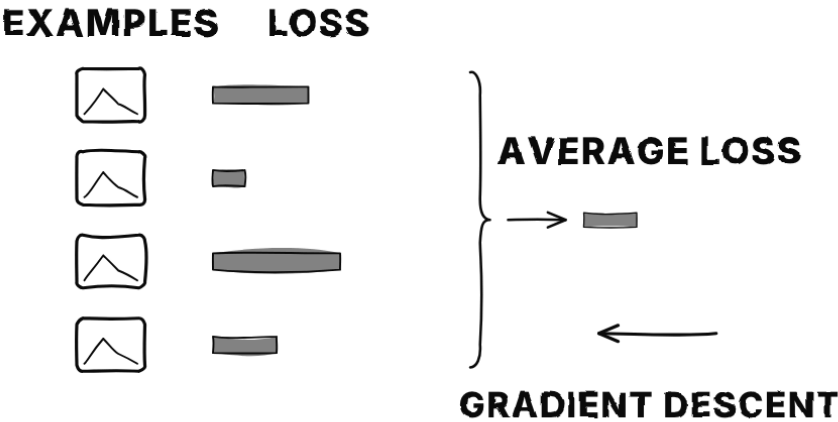
Scoring a Guess

Play a guessing game with a friend: they think of a number, you guess, and they only ever answer "right" or "wrong". Guess after guess you hear "wrong", and you have no idea whether you are getting closer. Now change the rule: they tell you "warmer" or "colder". Suddenly every guess teaches you which way to move, and you home in quickly. That difference is the whole story of this part: a model can only learn if its mistakes are scored in a way that says which way to move.



Wrong, or Warmer?

The score is called a loss function. It compares the model's prediction with the target answer and gives a single number, lower meaning better, so a lower loss always means a better prediction. And it must be differentiable, so that we can tell how it changes when the prediction is nudged. The book stresses that a loss function packs our understanding of the task and our preferences among possible answers into one scale that an optimiser can work with.



Average Loss, Pushed Down

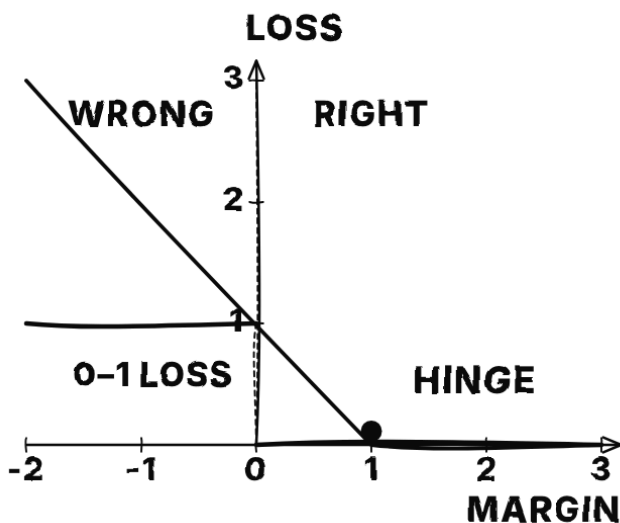
Training then becomes a clear mathematical task: find the model whose loss, averaged over all the examples in the dataset, is as small as possible. For historical reasons this is called empirical risk minimisation, where risk is simply another word for loss, and empirical means measured on the data we actually have. In practice

we do not search through all imaginable functions; the model has a fixed form with adjustable parameters, and gradient descent tunes those parameters to bring the average loss down. When a photo app is trained, this is literally what happens: its parameters are nudged again and again to lower its average mistake score over millions of labelled photos.

Try it yourself

Hide a coin under one of a row of cups and ask someone to find it, first answering only "yes" or "no" to each guess, then answering how many cups away they are. With yes or no, they must check cup after cup at random. With "how far", they can head straight for it. The first is a loss that never says which way to go; the second is a loss with a slope.

The book shows why differentiability is not a technicality. Take a task with two possible answers, say yes or no, and a model that answers by the sign of a number it computes. The most natural score is simply right or wrong: zero for a right answer, one for a wrong one. It matches our sense of being correct, but it is useless for training: almost everywhere its slope is zero, so gradient descent receives no hint and stays stuck where it started.



A Step Has No Slope

A less intuitive but smoother quantity is the margin, which compares the model's number with the target: positive when the sign agrees with the right answer, negative when it does not, and growing more positive the more confident and correct the model is. Because the margin changes smoothly, a loss built on it gives useful slopes. One such loss, the hinge loss, used to train support vector machines, is zero once the margin is comfortably large and grows steadily as the margin shrinks or turns wrong. The example shows a lasting tension: a loss must capture what we mean by good performance, and at the same time be convenient for numerical optimisation.

A small box traces these ideas to the Russian computer scientist Vladimir Vapnik and the field of statistical learning theory. That field

studies how well the average loss on a limited dataset stands in for the average over all possible data, for well-behaved families of models. Modern neural networks have puzzled it: they keep generalising well long after theory said they should start to memorise, and this has opened new lines of research.

Remember

A loss function scores a prediction against its target with a single number, lower for better, and must be differentiable so it tells which way to adjust. Training is empirical risk minimisation: tuning the model's parameters by gradient descent to make the average loss on the dataset as small as possible. A plain right-or-wrong score is useless for training, because its slope is almost always zero. Smooth quantities such as the margin, and losses built on them like the hinge loss, give slopes to follow, which shows the tension between what we mean by good and what can be optimised.

Questions

1. Why is a plain right-or-wrong score useless for training with gradient descent?
 - a. Because it gives too large numbers
 - b. Because its slope is zero almost everywhere, so gradient descent gets no hint and stays stuck
 - c. Because it cannot be averaged over a dataset

- d. Because it rewards wrong answers
2. What does empirical risk minimisation ask for?
- a. The model with the smallest average loss over the examples in the dataset
 - b. The model with the most parameters
 - c. The model that never makes a single mistake
 - d. The model with the largest margin on one example
3. A model answers yes or no by the sign of a number it computes. When is its margin negative?
- a. When the model is right and very confident
 - b. When the sign of the model's number disagrees with the correct answer
 - c. When the model's number is exactly zero
 - d. Never; margins are always positive
4. What puzzle about modern neural networks does the book mention in connection with statistical learning theory?
- a. They cannot be trained with gradient descent
 - b. They keep generalising well long after theory expected them to start memorising
 - c. They only work with the right-or-wrong score
 - d. They never use loss functions

Answers

1. **b)** The right-or-wrong score only changes when the answer flips, so almost everywhere it has no slope to follow.

- 2. a) Training becomes finding the parameters that make the loss, averaged over the dataset, as small as possible.
- 3. b) The margin is positive when the sign agrees with the target and negative when it does not, and it changes smoothly.
- 4. b) Networks show strong generalisation well beyond the point where overfitting was expected, opening new research directions.

Practice

- 1. Using the coin-under-the-cups game, explain why a loss function must be differentiable for gradient descent to work.

Hint: Compare the yes-or-no answers with the how-far answers.

Answer: With only yes or no, a wrong guess says nothing about which way to move, so the searcher must try cups at random. With how far, each guess tells which direction reduces the distance. A differentiable loss is like the how-far answer: its slope tells gradient descent which way to adjust the parameters.

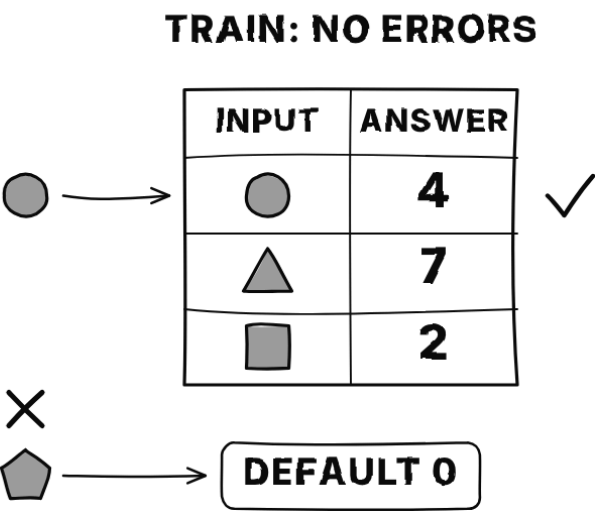
- 2. Describe the behaviour of the hinge loss in words, and explain why it is more useful than the right-or-wrong score.

Hint: What happens when the margin is comfortably large, and when it shrinks or becomes wrong?

Answer: The hinge loss is zero once the margin is comfortably large, meaning the model is clearly right, and grows steadily as the margin shrinks or turns negative. Because it changes smoothly with the margin, it has slopes that tell gradient descent how to improve, unlike the right-or-wrong score.

Memorising Is Not Learning

Picture a student who gets last year's exam paper, memorises every answer, and scores full marks on a practice run with that same paper. Then the real exam arrives with new questions, and the student has nothing to say. The practice score was perfect and meant nothing. The book uses exactly this trap to show why a low average loss on the training data is not the real goal.

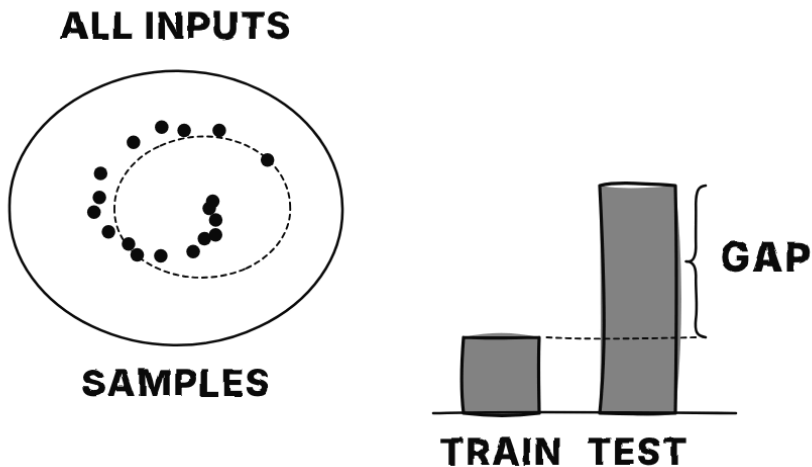


A Table That Only Remembers

In fact, the average loss on the training data is always trivially easy to drive to its minimum. Build a look-up table: if the input is one of the

training examples, return its stored answer; otherwise return some fixed default, such as zero. This table never makes a mistake on the training data, so it reaches the lowest possible average loss, and yet it is completely useless in practice. That is the difference between memorising and learning.

What we truly want is a low loss on future inputs that nobody has seen yet; the training examples are only a stand-in for them. The book names this the expected risk: the average loss over all possible input-output pairs the model could ever meet, weighted by how likely each one is, for example over all possible emails. A model with low expected risk would be guaranteed to work well, but it cannot be computed, because nobody can list and label every possible input. The average loss on the dataset, the empirical risk from before, is an estimate of it, built from the samples we happen to have.



The Generalisation Gap

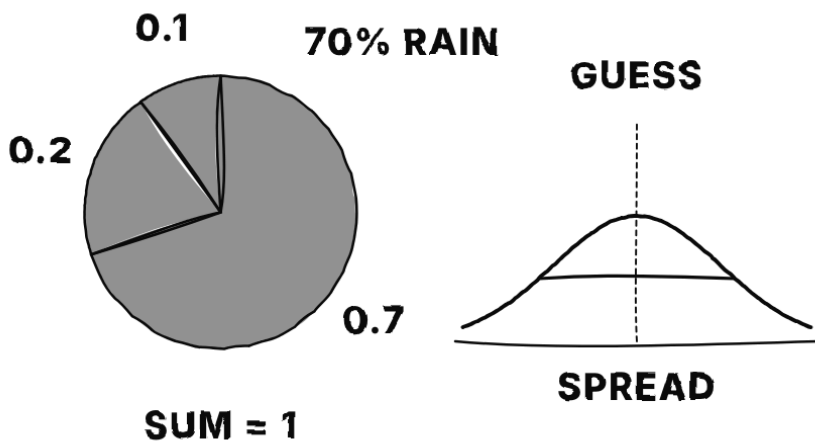
The difference between the two is called the generalisation gap. A memorising model like the look-up table has a huge gap: it overfits, fitting the specific training data instead of the pattern behind it. The practical test is to keep a separate test set, examples never used during training, and compare the loss on the training data with the loss on the test set. A large difference is a sign of overfitting. It is why a photo app is judged on photos that were never shown to it during training: only those reveal whether it learned dogs or merely memorised certain pictures.

Try it yourself

Ask a friend to memorise the answers to ten trivia questions and then quiz them on those same ten: they will score perfectly. Now ask ten new questions on the same topics. Their second score, not the first, tells you how much they have actually learned, and the drop between the two scores is your friend's own generalisation gap.

How should one choose a loss function? The book suggests a quick review of its probability appendix first, then explains that the loss carries our prior knowledge of the task and strongly affects performance. Sometimes common sense is enough to design one, as with the hinge loss. But there is a more principled route: to describe the whole training process in the language of probability, a viewpoint many other books prefer.

The starting idea is that the examples come from an unknown source of input-output pairs, which can be split into two pieces: how likely each input is, and how likely each output is once the input is known. Methods that model how likely the inputs are are called generative, those that model only the output given the input are called discriminative, and apart from language models this book focuses on the second kind. Predicting one single answer makes sense when, for each input, almost all of the probability sits on one value.



Answers as Distributions

The book relaxes that: instead of one answer, the model's outputs set the knobs of a whole probability distribution over the possible answers. For a choice among three classes, the model gives three probabilities that add up to one, a pie chart over the classes. For predicting a single number, the model can give two outputs: a best guess and a measure of spread, the second one squared so that it can never be negative. This setup includes everything said so far, and it is often easier to choose a sensible distribution than to invent a loss by hand. It also expresses uncertainty directly, through the spread. A weather app that says "70 percent chance of rain" rather than simply "rain" is giving you exactly this kind of answer.

Remember

A look-up table can reach the minimum average loss on training data while being useless, which shows that memorising is not learning. What matters is the expected risk, the average loss over all possible future data; it cannot be computed, and the empirical risk only estimates it. The difference between them is the generalisation gap; a large one means overfitting, and a separate test set measures it. In the probabilistic view, the model outputs the settings of a distribution over answers, such as class probabilities or a guess with its spread, which also expresses uncertainty.

Questions

1. A look-up table reaches the lowest possible average loss on the training data. Why is it still useless?
 - a. Because it is too slow to run
 - b. Because it only memorises the training pairs and gives a fixed default for anything new
 - c. Because its loss is not differentiable
 - d. Because it needs a test set to work
2. Why can't we simply train a model by minimising the expected risk directly?
 - a. Because the expected risk is always zero

- b. Because it would need every possible input listed and labelled, which is impossible
 - c. Because it ignores the loss function
 - d. Because it only works for emails
3. A model has a very low loss on its training data and a much higher loss on a separate test set. What does this indicate?
- a. The model generalises perfectly
 - b. The test set must contain errors
 - c. The model is overfitting, with a large generalisation gap
 - d. The loss function is not differentiable
4. In the probabilistic view, what does a model for predicting a single number output instead of one answer?
- a. A best guess together with a measure of spread
 - b. Three probabilities that add up to one
 - c. Only the loss value
 - d. A look-up table of past answers

Answers

1. **b)** The table returns stored answers for training inputs and a default otherwise, so it never learned the pattern: memorising is not learning.
2. **b)** The expected risk averages over all possible input-output pairs, which nobody can enumerate; the empirical risk only estimates it.
3. **c)** A large difference between training and test loss is the practical sign of a large generalisation gap, that is, overfitting.
4. **a)** For a single number the model gives a best guess and a spread, kept positive by squaring, which also expresses its uncertainty.

Practice

1. You train a spam filter and report its average loss on the emails it was trained on. Explain why this number alone cannot tell you whether the filter works, and what you should do instead.

Hint: Think of the student and last year's exam paper.

Answer: The training loss can be made tiny even by pure memorisation, so it says little about new emails. Keep a separate test set of emails never used in training and measure the loss there; the difference from the training loss estimates the generalisation gap and reveals overfitting.

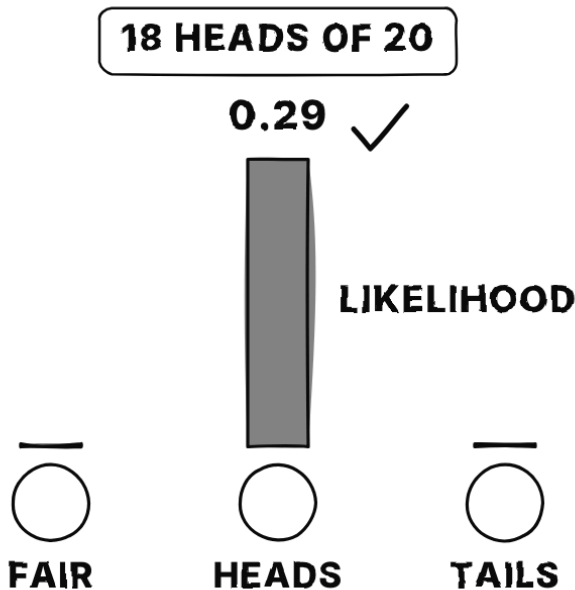
2. Explain in your own words why the probabilistic view can be more useful than a model that gives one single answer.

Hint: Think of the weather app and of choosing a loss.

Answer: Giving a distribution over answers, such as class probabilities or a guess with its spread, tells how certain the model is, like a chance of rain rather than a flat forecast. It also covers the earlier setup as a special case and makes it easier to design training, because choosing a sensible distribution is often simpler than inventing a loss by hand.

The Most Believable Model

Imagine three friends each claim to have flipped a coin twenty times, and one of them hands you a sheet with eighteen heads. One friend says they used an ordinary coin, another a coin weighted towards heads, the third a coin weighted towards tails. Without any formula you already know whose story to believe: the weighted-towards-heads coin makes that sheet far more likely than the others do. Choosing the explanation that makes the observed data most likely is the idea behind maximum likelihood, and it is how the book trains probabilistic models.

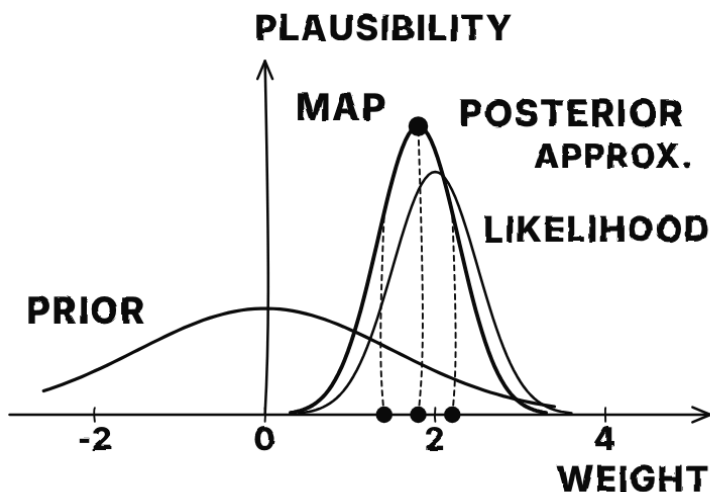


Which Coin Explains 18 Heads?

Recall that the examples in a dataset are assumed to be drawn independently from the same unknown source. So a model that gives probabilities to answers also gives a probability to the entire dataset: the chances of all the examples, multiplied together. That number is called the likelihood of the dataset. A model picked at random spreads its probabilities almost blindly, so it finds our particular dataset very unlikely. The sensible choice is the model under which our dataset is as likely as possible: the maximum likelihood solution.

This is still an optimisation problem, but with a difference. Before, the loss function was a design decision; here, once the probability distributions are chosen, the objective follows directly from the laws of probability. And yet the two views meet. Taking logarithms turns the long product into a sum, and flipping the sign turns maximising into minimising, so maximum likelihood becomes minimising an average of minus-log-probabilities, a kind of pseudo-loss. The book adds that every loss function used in practice can be obtained this way, for a suitable choice of distribution, and urges the reader to keep both views in mind. The training of a chat assistant, which learns to give high probability to the actual next word in its training text, is maximum likelihood at a gigantic scale.

The book then goes one step further, to Bayesian learning and Bayesian neural networks, giving only the main idea. Even a probabilistic model, once trained, is still one single function, yet many different settings of a model might explain the data equally well, and keeping all of them could make predictions more faithful. The recipe is again to let probability guide us. First, state a prior: how plausible each possible function seems before seeing any data. In practice that means a prior over the weights, and a common choice favours smaller weights, because such functions tend to be more stable.



Prior Times Likelihood

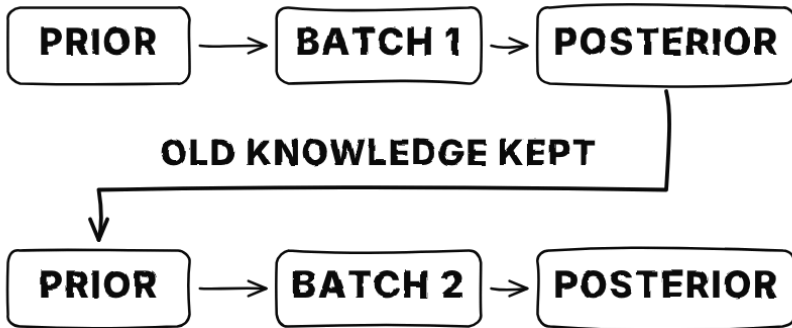
When the data arrives, Bayes' theorem combines the prior with the likelihood to give the posterior, the updated plausibility of every function after seeing the data; a normalising term called the evidence makes it a proper distribution. To predict, one averages the predictions of all the functions, each weighted by its posterior plausibility, and in practice one approximates this by sampling a handful of functions and averaging them. The catch is that the posterior can almost never be computed exactly, so approximations are needed, with methods such as Markov chain Monte Carlo or variational inference. Chapter nine will show a simple practical example, a technique called Monte Carlo dropout.

Try it yourself

Fill a jar with marbles of two colours in a proportion you do not reveal, and ask someone to guess the proportion before seeing any marble: that guess is their prior. Now let them draw one marble at a time and update their guess after each draw. Notice that each updated guess becomes the starting point for the next draw: yesterday's conclusion is today's prior.

Two remarks close the section. First, if we only want the single most plausible function, we can ignore the evidence and maximise the likelihood and the prior together, which is called the maximum a posteriori solution. If the prior treats all functions equally, it is just maximum likelihood again. Otherwise the prior penalises functions that stray too far from what we expected, and this penalty, called a regulariser, is a useful weapon against overfitting. The book notes that the difference loosely mirrors two ways of reading probability: as the frequency of events, treating parameters as fixed and data as random, or as a measure of uncertainty, treating the data as fixed and the parameters as uncertain.

Second, the Bayesian view makes it natural to learn from new data as it arrives. The posterior computed on the first batch of data simply becomes the prior for the next batch, which then updates it again. This can ease a notorious problem of models trained continually, called catastrophic forgetting, where learning new information wipes out the old. An app that keeps learning from your behaviour faces exactly this risk of forgetting what it learned before, and this way of updating beliefs is one route to softening it.



Today's Posterior, Tomorrow's Prior

Remember

The likelihood is the probability a model gives to the whole dataset, and maximum likelihood picks the model that makes the observed data most probable. In log form it becomes minimising a sum of minus-log-probabilities, and every practical loss function can be derived this way. Bayesian learning keeps a whole distribution over functions: a prior before the data, updated by Bayes' theorem into a posterior, with predictions averaged over plausible functions, though usually only approximately. The most plausible single function, the maximum a posteriori solution, adds the prior as a regulariser against overfitting, and the posterior can serve as the prior for new data to fight catastrophic forgetting.

Questions

1. Why does a randomly chosen model give a very low likelihood to our dataset?
 - a. Because it spreads its probabilities almost blindly, so our particular examples get little probability
 - b. Because likelihoods are always zero before training
 - c. Because random models cannot multiply probabilities
 - d. Because the dataset was not collected independently
2. How does maximum likelihood connect to the loss functions seen before?

- a. It replaces gradient descent with counting
 - b. Taking logarithms and flipping the sign turns it into minimising a sum of minus-log-probabilities, a pseudo-loss
 - c. It only works with the hinge loss
 - d. It has no connection to losses at all
3. In the maximum a posteriori solution, what happens if the prior treats all functions as equally plausible?
- a. It becomes the plain maximum likelihood solution
 - b. It always picks the smallest weights
 - c. It cannot be computed
 - d. It ignores the data completely
4. How does the Bayesian view help a model keep learning from new data without forgetting the old?
- a. It deletes the old data after each batch
 - b. The posterior from earlier data becomes the prior for the new data
 - c. It trains a brand-new model for every batch
 - d. It sets all weights to zero before each update

Answers

- 1. a)** The likelihood multiplies the chances the model gives to all the examples, and a random model gives our specific examples little probability.
- 2. b)** The log turns the product into a sum and the sign flip turns maximising into minimising, so it becomes an ordinary loss minimisation.
- 3. a)** With a uniform prior the prior term is constant, so only the likelihood matters and the solution is maximum likelihood.

4. b) Yesterday's conclusion becomes today's prior, so the update builds on what was learned, easing catastrophic forgetting.

Practice

1. Explain in your own words the roles of the prior, the likelihood and the posterior in Bayesian learning, using the jar of marbles.

Hint: Which belief comes before drawing, which comes from the draws, and which after?

Answer: The prior is the guess about the proportion of colours before any marble is drawn. The likelihood says how probable the drawn marbles are under each possible proportion. The posterior combines the two into an updated belief after the draws, and it can serve as the prior for the next draws.

2. Why does the book call the prior term in the maximum a posteriori solution a regulariser, and why does that help against overfitting?

Hint: Think about what the prior does to functions that stray from what we expected.

Answer: The prior adds a penalty for functions that stray far from what we expected, for example functions with very large weights. This pushes the solution towards simpler, more stable functions instead of ones that fit every quirk of the training data, which reduces overfitting.

3. Why is full Bayesian learning hard to use exactly in practice, and what is done instead?

Hint: Think about computing the posterior.

Answer: The posterior over all functions can almost never be computed exactly, except for special choices of prior and likelihood. So approximations are used, such as Markov chain Monte Carlo or variational inference, and predictions are averaged over a handful of sampled functions.