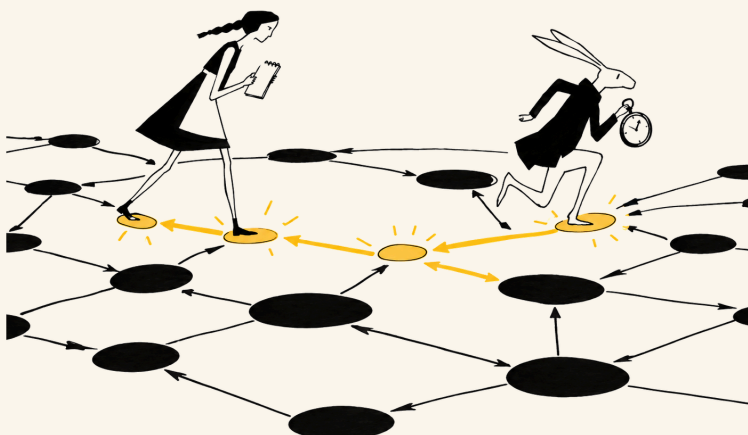


Alice's Adventures in a Differentiable Wonderland



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

Introduction

About this book

In the land of differentiability

Notation and symbols

Final thoughts before departing

License, and the mathematical preliminaries

Common matrix operations

Computational complexity

Einstein's notation, and derivatives

Gradients, directional derivatives and Jacobians

The dimensions of Jacobians, and gradient descent

Convergence of gradient descent

Accelerating gradient descent

From theory to practice, and what a dataset is

Variants of supervised learning

Loss functions

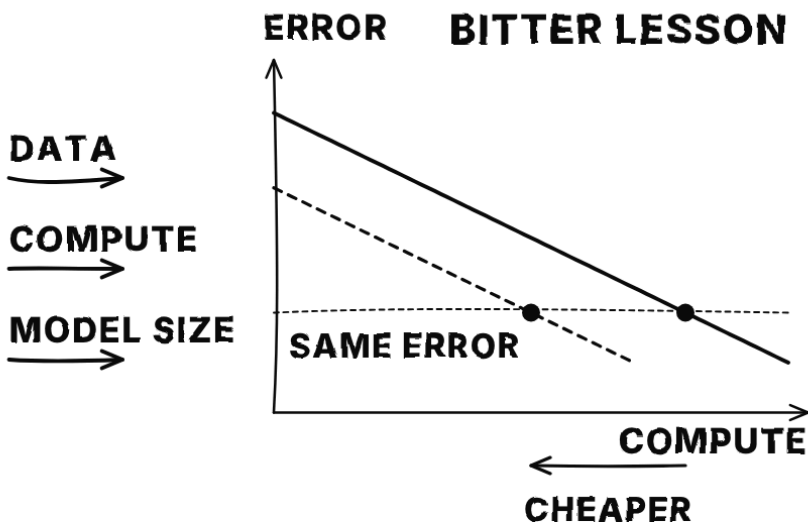
Expected risk, and choosing a loss

Maximum likelihood, and Bayesian learning

Linear models, and the squared loss

Introduction

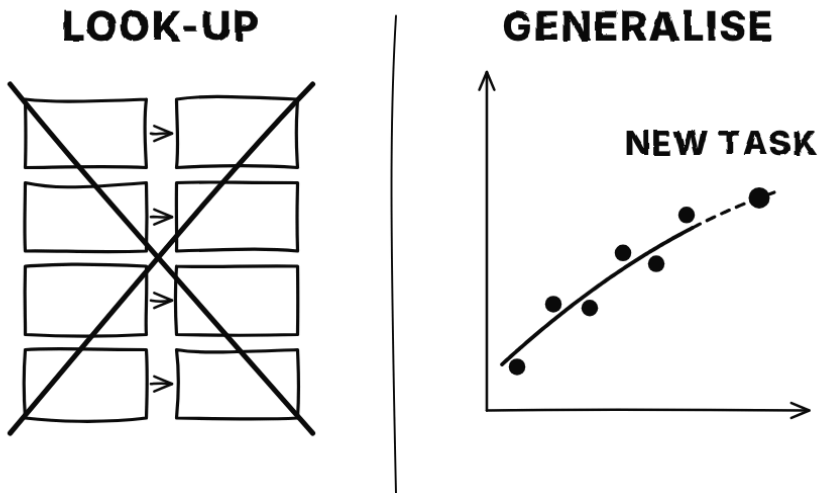
Neural networks have become an integral component of our everyday world, either openly, in the guise of large language models (LLMs), or hidden from view, by powering countless technologies and scientific discoveries including drones, cars, search engines, molecular design, and recommender systems. As we will see, all of this has been done by relying on a very small set of guiding principles and components, forming the core of this book, while the research focus has shifted to scaling them up to the limits of what is physically possible.



More Data, Compute, Size

The power of scaling is embodied in the relatively recent concept of neural scaling laws, which in turn has been instrumental in driving massive investments in artificial intelligence (AI). Informally, for practically any task, simultaneously increasing data, compute power, and the size of the models almost always results in a predictable increase in accuracy. Stated in another way, the compute power required to achieve a given accuracy for a task is decreasing by a constant factor per period of time. The tremendous power of combining simple, general-purpose tools with exponentially increased computational power in AI was called the bitter lesson by Richard Sutton.

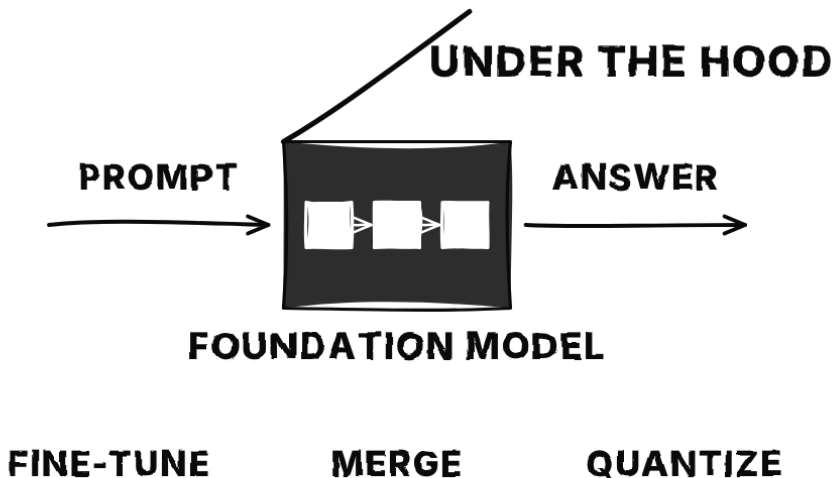
If we take scaling laws as given, we are left with an almost magical tool. In a nutshell, neural networks are optimized to approximate some probability distribution given data drawn from it. In principle, this approximation may fail: for example, modern neural networks are so large that they can easily memorize all the data they are shown and transform into a trivial look-up table. Instead, trained models are shown to generalize well even to tasks that are not explicitly considered in the training data. In fact, as the size of the datasets increases, the concept of what is in-distribution and what is out-of-distribution blurs, and large-scale models show hints of strong generalization capabilities and a fascinating low dependency on pure memorization, that is, overfitting.



Memorise or Generalise

The emergence of extremely large models that can be leveraged for a variety of downstream tasks, sometimes called foundation models, coupled with a vibrant open-source community, has also shifted how we interact with these models. Many tasks can now be solved by simply prompting, that is, interacting with text or visual instructions, a pre-trained model found on the web, with the internals of the model remaining a complete black box. From a high-level perspective, this is similar to a shift from having to program your libraries in, for example, C++, towards relying on open-source or commercial software whose source code is not accessible. The metaphor is not as far-fetched as it may seem: nowadays, few teams worldwide have the compute and the technical expertise to design and release truly large-

scale models such as the Llama LLMs, just like few companies have the resources to build enterprise CRM software.



A Black Box, Opened

And in the same way, just like open-source software provides endless possibilities for customizing or designing your programs from scratch, consumer-grade hardware and a bit of ingenuity give you a vast array of options to experiment with differentiable models, from fine-tuning them for your tasks to merging models, quantizing them for low-power hardware, testing their robustness, or even designing completely new variants and ideas. For all of this, you need to look “under the hood” and understand how these models process and manipulate data internally, with all their tricks and idiosyncrasies that are born from experience and debugging. This book is an entry point

into this world: if, like Alice, you are naturally curious, I hope you will appreciate the journey.

Questions

1. What do neural scaling laws say, informally?
 - a. Increasing data, compute and model size together almost always gives a predictable rise in accuracy
 - b. Only a better learning algorithm can raise accuracy once a model is large
 - c. Larger models always memorize their data and therefore get worse on new tasks
 - d. Accuracy grows with model size alone, whatever the amount of data
2. Why could one expect very large networks to fail at approximating a distribution, and what is observed instead?
 - a. They are too slow to train, but in practice they train quickly
 - b. They could simply memorize the data like a look-up table, yet trained models generalize well even to tasks not in the training data
 - c. They need labelled data, yet they work without any data
 - d. They forget earlier data, yet they keep everything
3. What does the comparison with C++ libraries and closed commercial software illustrate?
 - a. That neural networks are written in C++

- b. That prompting is slower than programming
 - c. That using a pre-trained model through prompts is like relying on software whose internals you cannot see, built by the few who have the resources
 - d. That open-source models cannot be modified
4. Why does the author say you need to look under the hood of these models?
- a. Because prompting no longer works
 - b. Because foundation models are illegal to use without inspection
 - c. Because only then can you train them on a phone
 - d. Because fine-tuning, merging, quantizing, testing or designing new models requires understanding how they process data internally

Answers

1. **a)** The author describes scaling laws as a predictable increase in accuracy when data, compute power and model size are increased together, for practically any task.
2. **b)** Modern networks are big enough to memorize everything they see, but the book notes that trained models are shown to generalize well instead.
3. **c)** Prompting a pre-trained model treats it as a black box, much as one uses commercial software without its source code; few teams can build truly large models, as few companies can build enterprise CRM software.
4. **d)** All the options open to an experimenter, from fine-tuning to new designs, require understanding how the models manipulate data internally, with their tricks and idiosyncrasies.

Practice

1. In your own words, what did Richard Sutton call the bitter lesson, and how does it relate to scaling laws?

Hint: Think of what is combined with what.

Answer: The bitter lesson is that simple, general-purpose tools combined with exponentially growing computational power have been tremendously powerful in AI. Scaling laws say the same in measurable form: giving such models more data, compute and size predictably improves accuracy.

2. Explain why, as datasets grow, the line between in-distribution and out-of-distribution blurs.

Hint: Ask what fraction of possible tasks a huge dataset already touches.

Answer: With very large datasets, so many kinds of examples are present that it becomes hard to say what lies outside the training distribution. Large models then show signs of strong generalization and depend little on pure memorization, that is, overfitting.

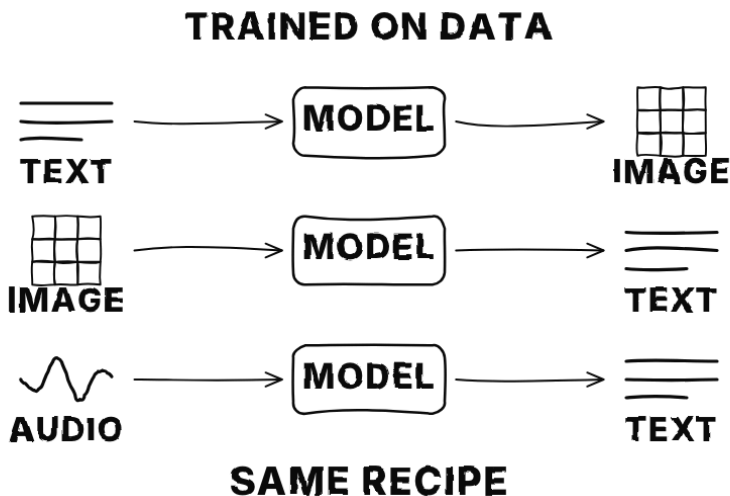
3. List four things the author says you can do with differentiable models on consumer-grade hardware, and say what they have in common.

Hint: Look at the list after 'from fine-tuning them'.

Answer: Fine-tuning them for your tasks, merging models, quantizing them for low-power hardware, and testing their robustness (or designing new variants). All of them require looking under the hood and understanding how the models process data internally.

About this book

We assume our readers are familiar with the basics of machine learning (ML), and more specifically supervised learning (SL). Supervised learning can be used to solve complex tasks by gathering data on a desired behavior, and “training”, that is, optimizing, systems to approximate that behavior. This deceptively simple idea is extremely powerful. For example, image generation can be turned into the problem of collecting a sufficiently large collection of images with their captions; simulating the English language becomes the task of gathering a large collection of text and learning to predict a sentence from the preceding ones; and diagnosing an X-ray becomes equivalent to having a large database of scans with the associated doctors’ decisions, as Figure F.1.2 shows.

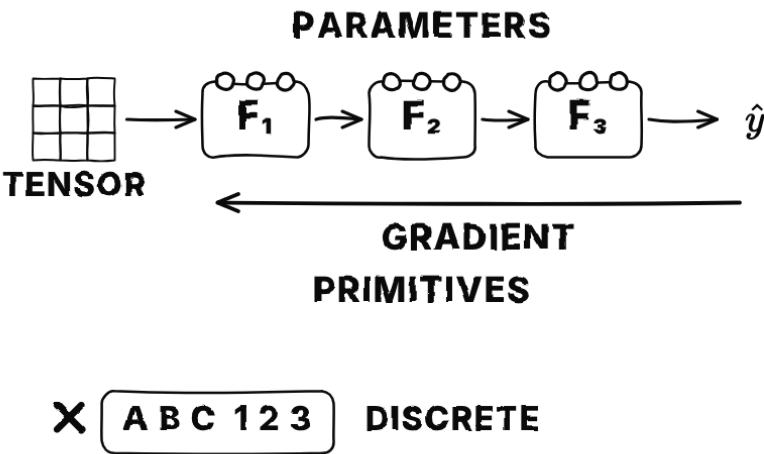


Tasks as Input and Output

The figure sorts most tasks by the input and output they need. Image generation wants an image, an ordered grid of pixels, from a text, a sequence of characters, while the inverse, image captioning, is the problem of generating a caption from an image. As another example, audio query answering requires a text from an audio, another ordered sequence, this time numerical. Fascinatingly, the design of the models follows similar specifications in all cases.

In general, learning is a search problem. We start by defining a program with a large number of degrees of freedom, which we call parameters, and we manipulate the parameters until the model's performance is satisfying. To make this idea practical, we need efficient ways of searching for the optimal configuration even in the

presence of millions, or billions, or trillions of parameters. As the name implies, differentiable models do this by restricting the selection of the model to differentiable components, that is, mathematical functions that we can differentiate. Being able to compute a derivative of a high-dimensional function, a gradient, means knowing what happens if we slightly perturb its parameters, which in turn leads to automatic routines for their optimization, most notably automatic differentiation and gradient descent. Describing this setup is the topic of the first part of the book, Compass and Needle, going from Chapter 2 to Chapter 6.

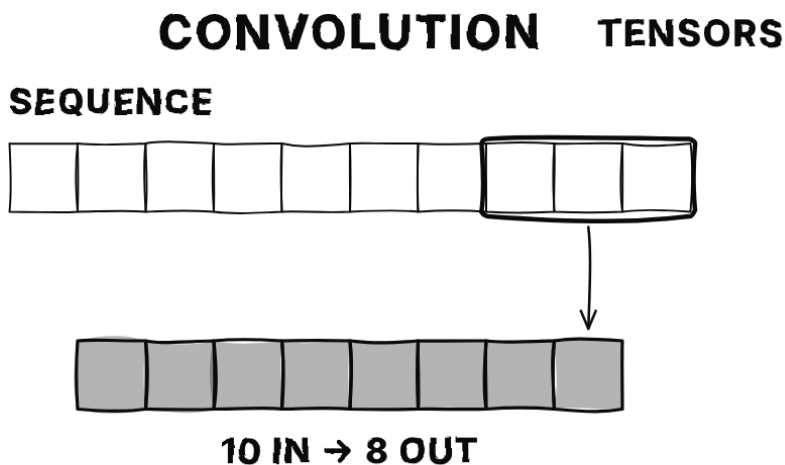


A Chain of Differentiable Blocks

By viewing neural networks as simply compositions of differentiable primitives, we can ask two basic questions, shown in Figure F.1.3.

First, what data types can we handle as inputs or outputs? And second, what sort of primitives can we use? Differentiability is a strong requirement that does not allow us to work directly with many standard data types, such as characters or integers, which are fundamentally discrete and hence discontinuous. By contrast, we will see that differentiable models can work easily with more complex data represented as large arrays of numbers, which we will call tensors, such as images, which can be manipulated algebraically by basic compositions of linear and nonlinear transformations.

In the second part of the book we focus on a prototypical example of a differentiable component, the convolutional operator, from Chapter 7 until Chapter 9. Convolutions can be applied whenever our data can be represented by an ordered sequence of elements: these include, among others, audio, images, text, and video. Along the way we also introduce a number of useful techniques to design deep models, that is, models composed of many steps in sequence, as well as several important ideas such as text tokenization, autoregressive generation of sequences, and causal modeling, which form the basis for state-of-the-art LLMs.



Convolution Slides Along a Sequence

Figure F.1.3 shows neural networks as sequences of differentiable primitives which operate on structured arrays, the tensors. Each primitive can be categorized based on its input and output signature, which in turn defines the rules for composing them.

The third part of the book, *Down the Rabbit Hole*, continues our exploration of differentiable models by considering alternative designs for sets, most importantly attention layers and transformer models in Chapters 10 and 11, graphs in Chapter 12, and finally recurrent layers for temporal sequences in Chapter 13.

PART I

CH. 2-6

PART II

CH. 7-9

PART III

ATTENTION

WEBSITE

LABS

The Book's Three Parts

The book is complemented by a website where I will, hopefully, collect additional chapters and material on topics of interest that do not focus on a specific type of data, including generative modeling, conditional computation, transfer learning, and explainability. These chapters are more research-oriented in nature and can be read in any order. In addition, I provide a series of guided lab sessions in notebook form, which cover a large part of the material from the book as well as advanced topics such as contrastive learning and model merging.

Questions

1. Why does the book call learning a search problem?
 - a. Because models look up answers on the web
 - b. Because we define a program with many parameters and adjust them until performance is satisfying
 - c. Because training data must be searched for errors
 - d. Because each task has exactly one correct program
2. What does knowing the gradient of a high-dimensional function give us?
 - a. The exact optimal parameters in one step
 - b. A list of all the training data
 - c. Knowledge of what happens when the parameters are slightly perturbed, which leads to automatic optimization
 - d. A way to work directly with characters and integers
3. Why can differentiable models not work directly with characters or integers?
 - a. Those data types are discrete and hence discontinuous, while differentiability needs continuity
 - b. Those data types take too much memory
 - c. Characters and integers cannot be stored in tensors at all
 - d. They can, but only with convolutions
4. When can convolutions be applied?
 - a. Only to images
 - b. Only to data with no order
 - c. Whenever the data can be represented as an ordered sequence of elements, such as audio, images, text and video
 - d. Only after the data is converted to integers

Answers

1. **b)** Learning starts from a program with a large number of degrees of freedom, the parameters, which are manipulated until the model performs well.
2. **c)** A gradient tells what happens under a slight perturbation of the parameters, and that leads to routines such as automatic differentiation and gradient descent.
3. **a)** Differentiability is a strong requirement; characters and integers are fundamentally discrete and hence discontinuous.
4. **c)** The book says convolutions apply whenever the data is an ordered sequence of elements, and lists audio, images, text and video.

Practice

1. Pick one of the book's examples (image generation, language simulation, X-ray diagnosis) and explain how it becomes a supervised learning problem.

Hint: Say what data you would gather and what behavior you would train for.

Answer: For X-ray diagnosis, one gathers a large database of scans with the doctors' decisions attached and trains a system to approximate that behavior: given a scan, produce the decision. The task becomes collecting examples of the desired input-output behavior and optimizing a model to imitate it.

2. Name the two basic questions the book asks once neural networks are seen as compositions of differentiable primitives, and say which parts of the book answer them for which data.

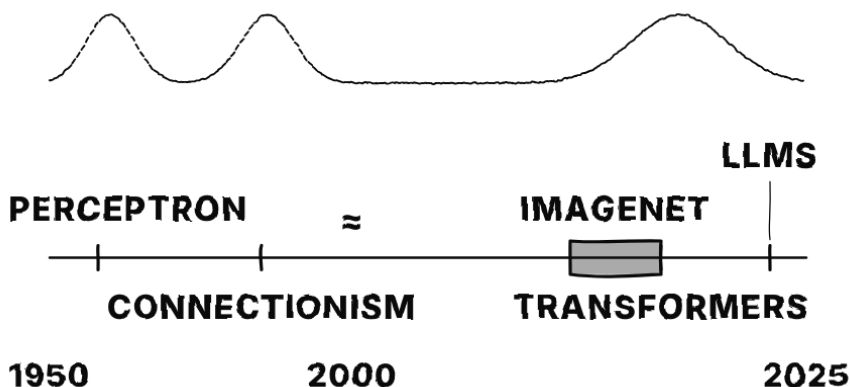
Hint: One question is about data types, the other about building blocks.

Answer: What data types can we handle as inputs or outputs, and what primitives can we use? The first part sets up the differentiable framework, the second studies convolutions for ordered sequences such as audio, images, text and video, and the third covers sets (attention and transformers), graphs and temporal sequences (recurrent layers).

In the land of differentiability

Neural networks have a long and rich history. The name itself is a throwback to early attempts at modeling biological neurons in the twentieth century, and similar terminology has remained pervasive: to be consistent with existing frameworks, in the upcoming chapters we may refer to neurons, layers, or, for example, activations. After multiple waves of interest, the period between 2012 and 2017 saw an unprecedented rise in the complexity of the networks, spurred by large-scale benchmarks and competitions, most notably the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) that we cover in Chapter 9. A second major wave of interest came from the introduction of transformers, the subject of Chapter 10, in 2017: just as computer vision was overtaken by convolutional models a few years before, natural language processing was overtaken by transformers in a very short period. Further improvements in these years were made for videos, graphs, which are the subject of Chapter 12, and audio, culminating in the current excitement around LLMs, multimodal networks, and generative models. This is not the place for a complete historical overview of modern neural networks.

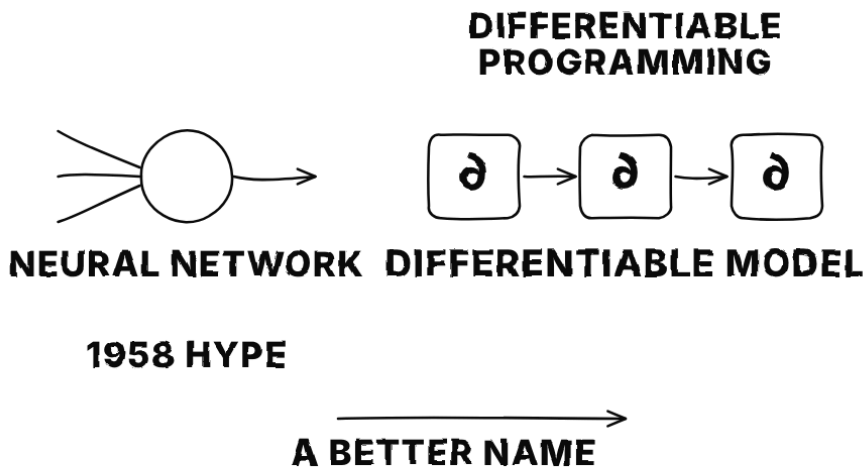
WAVES OF INTEREST



Waves of Neural Networks

This period paralleled a quick evolution in terminology, from the connectionism of the eighties to the use of deep learning for referring to modern networks, in opposition to the smaller, shallower models of the past. Despite this, all these terms remain inexorably vague, because modern artificial networks retain almost no resemblance to biological neural networks and neurology. Looking at modern neural networks, their essential characteristic is being composed of differentiable blocks: for this reason, in this book I prefer the term differentiable models when feasible. Viewing neural networks as differentiable models leads directly to the wider topic of differentiable programming, an emerging discipline that blends computer science and optimization to study differentiable computer programs more broadly. Like many, I was inspired by a “manifesto” published by

Yann LeCun on Facebook in 2018, and for the connection between neural networks and open-source programming and development I am also thankful to a second manifesto, published by Colin Raffel in 2021.

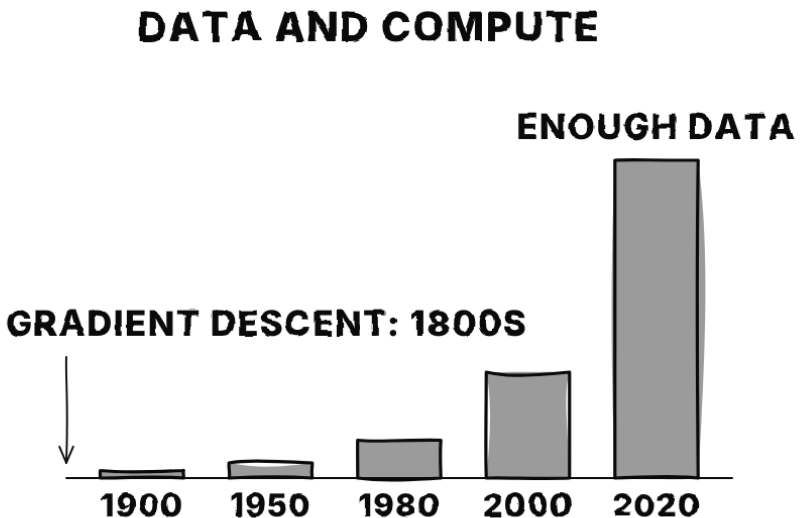


Not Neurons: Differentiable Blocks

A figure in the book shows AI hype, except that it is 1958, and the American psychologist Frank Rosenblatt has gathered significant media attention with his studies on “perceptrons”, one of the first working prototypes of neural networks.

As we travel through this land of differentiable models, we are also traveling through history. The basic concepts of numerical optimization of linear models by gradient descent, covered in Chapter

4, were known since at least the nineteenth century; so-called “fully-connected networks” in the form we use later on can be dated back to the 1980s; convolutional models were known and used already at the end of the nineties. For an opinionated history at length, there is also an annotated history of neural networks by Jürgen Schmidhuber. However, it took many decades to have sufficient data and power to realize how well these models can perform given enough data and enough parameters.



Old Ideas, New Scale

While we do not have space to go in depth on all possible topics, also due to how quickly the research is progressing, I hope the book provides enough material to allow the reader to easily navigate the most recent literature.

Questions

1. Why does the author prefer the term differentiable models to neural networks?
 - a. Because modern networks are biologically accurate
 - b. Because their essential characteristic is being composed of differentiable blocks, and they retain almost no resemblance to biological neurons
 - c. Because neural networks is a trademark
 - d. Because differentiable models are smaller than neural networks
2. What happened to natural language processing after transformers appeared in 2017?
 - a. It was overtaken by transformers in a very short period, as computer vision had been by convolutional models
 - b. It returned to rule-based systems
 - c. It stopped using neural networks
 - d. It was merged with ImageNet
3. Why did ideas known since the nineteenth century or the 1980s take so long to show their power?
 - a. They were mathematically wrong at first
 - b. Their inventors kept them secret
 - c. It took decades to have sufficient data and computing power to see how well they perform at scale
 - d. They only work with transformers

Answers

1. b) The terms around neurons are vague since modern networks barely resemble biology; what defines them is that they are built from differentiable blocks.

2. a) The book draws the parallel: computer vision was overtaken by convolutional models, and a few years later language processing by transformers.

3. c) Gradient descent, fully-connected networks and convolutions were all old ideas; what was missing was enough data and power.

Practice

1. Place these on a timeline using only what the passage says: gradient descent for linear models, fully-connected networks, convolutional models, Rosenblatt's perceptron hype, the ImageNet-driven rise, transformers.

Hint: The passage gives a century, a decade or a year for each.

Answer: Gradient descent for linear models: nineteenth century at least. Perceptron hype: 1958. Fully-connected networks: 1980s. Convolutional models: end of the nineties. ImageNet-driven rise: 2012 to 2017. Transformers: 2017.

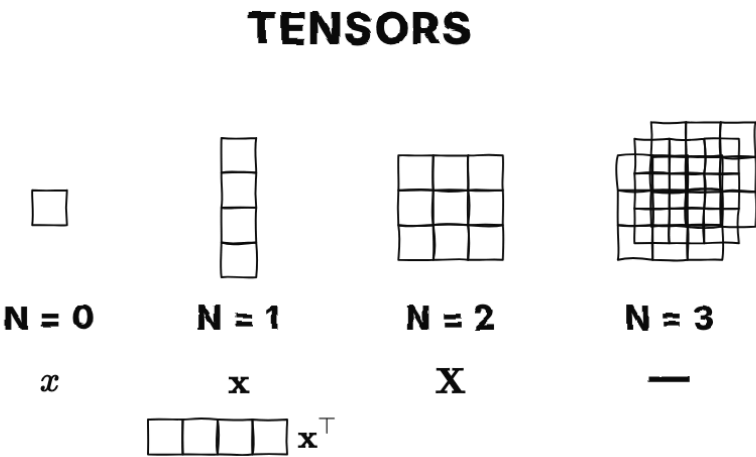
2. In your own words, what is differentiable programming, according to the passage?

Hint: It blends two fields.

Answer: It is an emerging discipline that blends computer science and optimization to study differentiable computer programs in general; seeing neural networks as differentiable models leads straight into it.

Notation and symbols

A figure in the book shows the fundamental data types: scalars, vectors, matrices, and generic n-dimensional arrays. We use the name tensors to refer to them, and n is called the rank of the tensor. The figure shows the vector as a row for readability, but in the text we assume all vectors are column vectors.



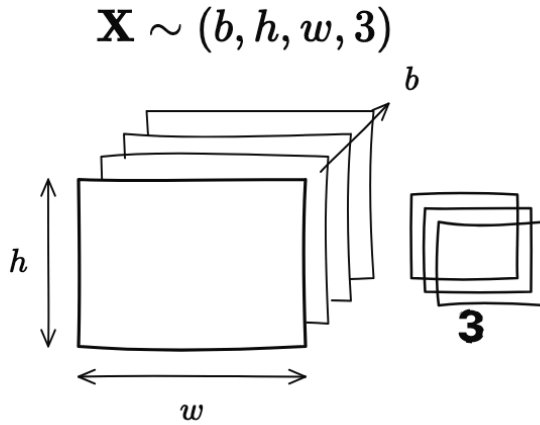
Scalar, Vector, Matrix, Tensor

The fundamental data type when dealing with differentiable models is a tensor, which we define as an n-dimensional array of objects, typically real-valued numbers. In the scientific literature, tensors have

a more precise definition as multilinear operators, while the objects we use in the book are simpler multidimensional arrays; although it is a misnomer, the use of tensor is so widespread that we keep this convention here. With apologies to any mathematician reading us, we call n the rank of the tensor. The notation in the book varies depending on n .

A single-item tensor, with $n = 0$, is just a single value, a scalar. For scalars, we use lowercase letters, such as x or y . If you are wondering, scalars are named like this because they can be written as scalar multiples of one; also, I promise to reduce the number of footnotes from now on. Columns of values, with $n = 1$, are vectors. For vectors we use a lowercase bold font, such as $\mathbf{x}$. The corresponding row vector is denoted by $\mathbf{x}^T$ when we need to distinguish them, and we can also ignore the transpose for readability, if it is clear from context. Rectangular arrays of values, with $n = 2$, are matrices. We use an uppercase bold font, such as $\mathbf{X}$ or $\mathbf{Y}$. No specific notation is used for $n > 2$. We avoid calligraphic symbols such as $\mathcal{X}$, which we reserve for sets or probability distributions.

For working with tensors, we use a variety of indexing strategies, described better in Section 2.1. In most cases, understanding an algorithm or an operation boils down to understanding the shape of each tensor involved. To denote the shape concisely, we use the following notation: $\mathbf{X} \sim (b, h, w, 3)$. This is a rank-4 tensor with shape $(b, h, w, 3)$. Some dimensions can be pre-specified, such as the 3, while other dimensions can be denoted by variables. We use the same symbol to denote drawing from a probability distribution, for example $\varepsilon \sim \mathcal{N}(0, 1)$, but we do this rarely, and the meaning of the symbol should always be clear from context. Hence, $\mathbf{x} \sim (d)$ will substitute the more common $\mathbf{x} \in \mathbb{R}^d$, and similarly $\mathbf{X} \sim (n, d)$ instead of $\mathbf{X} \in \mathbb{R}^{n \times d}$.



$\mathbf{x} \sim (d)$
instead of $\mathbf{x} \in \mathbb{R}^d$

Shapes: $\mathbf{X} \sim (b, h, w, 3)$

Finally, we may want to constrain the elements of a tensor, for which we use a special notation. $\mathbf{x} \sim \text{Binary}(c)$ denotes a tensor with only binary values, that is, elements that are either 0 or 1. $\mathbf{x} \sim \Delta(a)$ denotes a vector belonging to the so-called simplex, that is, $x_i \geq 0$ and $\sum_i x_i = 1$. For tensors with higher rank, for example $\mathbf{X} \sim \Delta(n, c)$, we assume the normalization is applied with respect to the last dimension; in this case, each row $\mathbf{X}_i$ belongs to the simplex.

Binary(5)

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

$\mathbf{X} \sim \Delta(2, 3)$

ROWS	→	0.5	0.3	0.2	= 1
	→	0.1	0.6	0.3	= 1

Binary and the Simplex

Additional notation is introduced in each chapter when necessary. We also have a few symbols on the side.

Questions

1. What is the rank of a matrix in the book's notation?
 - a. 0
 - b. 1
 - c. 2
 - d. It depends on the number of rows

2. What does $X \sim (b, h, w, 3)$ tell you?
- a. That X is drawn from a distribution with four parameters
 - b. That X is a rank-4 tensor whose shape is b by h by w by 3
 - c. That X has four elements
 - d. That X is a matrix with b rows
3. Which vector belongs to the simplex $\Delta(3)$?
- a. $(0.6, 0.6, -0.2)$
 - b. $(0.5, 0.5, 0.5)$
 - c. $(1, 1, 1)$
 - d. $(0.2, 0.5, 0.3)$
4. How does the book write a row vector, given that its vectors are columns?
- a. With a transpose, $\mathbf{x}^T$
 - b. With an uppercase bold letter
 - c. With a calligraphic letter
 - d. With a tilde

Answers

1. **c)** Rectangular arrays of values, matrices, have $n = 2$, and n is what the book calls the rank.
2. **b)** The tilde followed by a shape denotes the shape of the tensor; here four dimensions, so rank 4, with the last fixed to 3.
3. **d)** A simplex vector has non-negative entries summing to one; only $(0.2, 0.5, 0.3)$ satisfies both.
4. **a)** Vectors are columns, and the corresponding row vector is denoted $\mathbf{x}^T$ when the two must be distinguished.

Worked examples

Shape of a batch of photos A batch holds 32 colour photos, each 64 pixels high and 64 wide, with 3 colour channels. Write its shape in the book's notation, give its rank and the number of values it holds.

The book writes shapes with the tilde: list the dimensions in order, batch first, then height, width and channels.

So $X \sim (32, 64, 64, 3)$.

The rank is the number of dimensions, here four.

The number of values is the product of the dimensions: $32 \times 64 \times 64 \times 3 = 32 \times 4096 \times 3 = 393,216$.

Answer: $X \sim (32, 64, 64, 3)$, a rank-4 tensor holding 393,216 values.

Practice

1. Is the vector $(0.6, 0.6, -0.2)$ in the simplex $\Delta(3)$? Explain.

Hint: Check both conditions of the simplex, not only the sum.

Answer: No. Its entries sum to 1, but one entry is negative, and the simplex requires every $x_i \geq 0$.

2. A table of 100 customers with 8 features each is written $X \sim (100, 8)$. How would the same statement look in the more common notation, and what is its rank?

Hint: The book says which notation its tilde replaces.

Answer: $X \in \mathbb{R}^{100 \times 8}$, a matrix, so a tensor of rank 2.

3. If $Y \sim \Delta(5, 4)$, what must be true of each row of Y ?

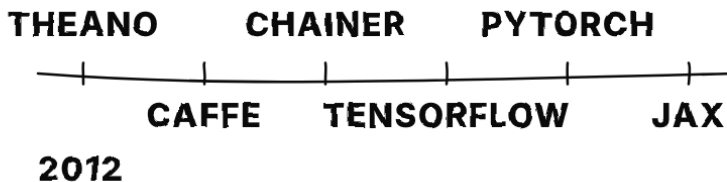
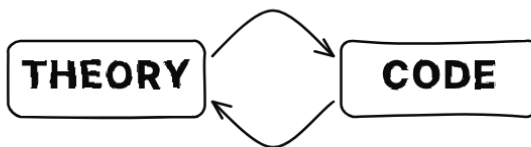
Hint: For higher ranks the normalization applies to the last dimension.

Answer: Each of the 5 rows has 4 non-negative entries that sum to 1, so each row belongs to the simplex $\Delta(4)$.

Final thoughts before departing

The book stems from my desire to give a coherent form to my lectures for Neural Networks for Data Science Applications, a course I have taught for many years in the Master's Degree in Data Science at Sapienza University of Rome. The core chapters of the book constitute the main part of the course, while the remaining chapters are topics that I cover on and off depending on the year. Some parts have been supplemented by additional courses I have taught, or intend to teach, including parts of Neural Networks for Computer Engineering, an introduction to machine learning for Telecommunication Engineering, plus a few tutorials, PhD courses, and summer schools over the years.

SAPIENZA COURSE



Theory and Code, Hand in Hand

There are already a number of excellent and recent books on the topic of modern, deep neural networks. This book covers content similar to all of them in the beginning, while the exposition and some additional parts, or a few sections in the advanced chapters, intersect less, and depend mostly on my research interests. I hope I can provide an additional and complementary viewpoint on existing material.

As my choice of name suggests, understanding differentiable programs comes from both theory and coding: there is a constant interplay between how we design models and how we implement them, with topics like automatic differentiation being the best example. The current resurgence of neural networks, roughly from 2012 onwards, can be traced in large part to the availability of

powerful software libraries, going from Theano to Caffe, Chainer, and then directly to the modern iterations of TensorFlow, PyTorch, and JAX, among others. I try whenever possible to connect the discussion to concepts from existing programming frameworks, with a focus on PyTorch and JAX. The book is not a programming manual, however, and I refer to the documentation of the libraries for a complete introduction to each of them.

Before moving on, I would like to list a few additional things this book is not. First, I have tried to pick up a few concepts that are both common today and general enough to be of use in the near future. However, I cannot foresee the future and I do not strive for completeness, and several parts of these chapters may be incomplete or outdated by the time you read them. Second, for each concept I try to provide a few examples of variations that exist in the literature, for example from batch normalization to layer normalization. However, keep in mind that hundreds more exist: for this, I invite you to explore the many pages of Papers With Code. Finally, this is a book on the fundamental components of differentiable models, but implementing them at scale, and making them work, requires both engineering sophistication and a bit of intuition. I cover little on the hardware side, and for the latter nothing beats experience and opinionated blog posts, such as the training recipe by Andrej Karpathy or his Zero to Hero video series.

Acknowledgments. The coloring of the equations is thanks to a beautiful LaTeX package by S. T. John. Color images of Alice in Wonderland and the black and white symbols in the margin are all licensed from Shutterstock.com. The images of Alice in Wonderland in the figures from the main text are reproductions of the original John Tenniel illustrations, thanks to Wikimedia. I thank Roberto

Alma for feedback on a previous draft of the book and for encouraging me to publish it. I also thank Corrado Zoccolo, Emanuele Rodolà, Marcin Slaby, Konstantin Burlachenko, and Diego Sandoval for providing extensive corrections and suggestions to the current version, and everyone who sent me feedback via email.

Questions

1. What does the author give as a main reason for the resurgence of neural networks from about 2012?
 - a. The discovery of the perceptron
 - b. The availability of powerful software libraries, from Theano and Caffe to TensorFlow, PyTorch and JAX
 - c. The end of the ImageNet challenge
 - d. A new theory of biological neurons
2. Why does the book keep connecting ideas to PyTorch and JAX while not being a programming manual?
 - a. Because there is a constant interplay between how models are designed and how they are implemented
 - b. Because the reader is expected to know no mathematics
 - c. Because the libraries replace the theory
 - d. Because the author wrote those libraries
3. Which of these does the author say the book is not?
 - a. A book on the fundamental components of differentiable models

- b. A complete and future-proof catalogue of every variant and of the hardware side
- c. A book based on a university course
- d. A book that connects to existing frameworks

Answers

1. **b)** The resurgence can be traced in large part to powerful software libraries, which is why the book connects theory to code.
2. **a)** Understanding differentiable programs comes from theory and coding together; automatic differentiation is the best example of that interplay.
3. **b)** The author does not strive for completeness, expects parts to age, points to hundreds of variants elsewhere, and covers little on hardware.

Practice

1. List the three things the author says this book is not, in your own words.

Hint: They start with First, Second and Finally.

Answer: It is not complete or guaranteed current, since the field moves and parts may be outdated; it is not a catalogue of every variant, only a few examples among hundreds; and it is not a guide to implementing models at scale or to hardware, which need engineering skill, intuition and experience.

2. Why is automatic differentiation the author's best example of the interplay between design and implementation?

Hint: Think about what has to exist in software for a model to be trained.

Answer: Designing differentiable models only pays off if gradients can be computed efficiently by the software; automatic differentiation is the

algorithm that frameworks implement to do it, so model design and its implementation are tied together there most clearly.

License, and the mathematical preliminaries

The book is released under the CC BY-SA license. This license enables “reusers to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the creator. The license allows for commercial use. If you remix, adapt, or build upon the material, you must license the modified material under identical terms.”

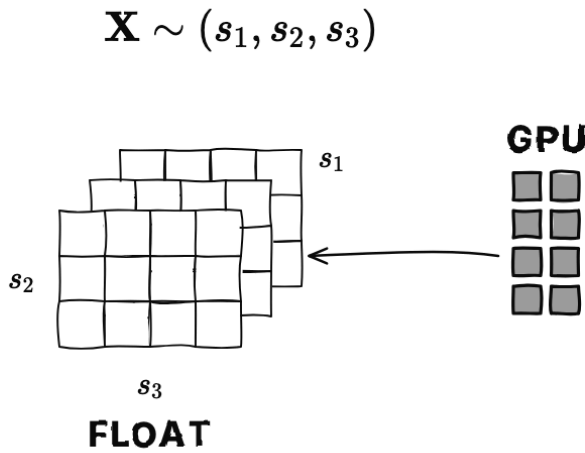
Chapter 2, Mathematical preliminaries. In short: we compress here the mathematical concepts required to follow the book. We assume prior knowledge of all these topics, focusing more on describing specific notation and giving a cohesive overview. When possible, we stress the relation between some of this material, for example tensors, and its implementation in practice.

The chapter is composed of three parts that follow sequentially from each other, starting from linear algebra, moving to the definition of gradients for n -dimensional objects, and finally how we can optimize functions by exploiting such gradients. A self-contained overview of probability theory is given in an appendix, with a focus on the maximum likelihood principle.

This chapter is full of content and definitions: bear with me for a while!

Section 2.1, Linear algebra. We recall here some basic concepts from linear algebra that will be useful in the following, and that let us agree

on a shared notation. Most of the book revolves around the idea of a tensor.



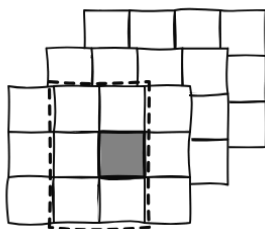
A Tensor and Its Shape

Definition, tensors: a tensor $\mathbf{X}$ is an n -dimensional array of elements of the same type. In the book we use $\mathbf{X} \sim (s_1, s_2, \dots, s_n)$ to quickly denote the shape of the tensor.

For $n = 0$ we obtain scalars, single values, while we have vectors for $n = 1$, matrices for $n = 2$, and higher-dimensional arrays otherwise. Recall that we use lowercase x for scalars, lowercase bold $\mathbf{x}$ for vectors, and uppercase bold $\mathbf{X}$ for matrices. Tensors in the sense described here are fundamental in deep learning because they are well

suited to a massively parallel implementation, such as using GPUs or more specialized hardware, for example TPUs and IPUs.

A tensor is described by the type of its elements and its shape. Most of our discussion will be centered around tensors of floating-point values, whose specific format we will consider later on, but they can also be defined for integers, for example in classification, or for strings, for example for text. Tensors can be indexed to get slices, that is, subsets of their values, and most conventions from NumPy indexing apply; for readability, in the book we index from 1, not from 0. For simple equations we use subscripts: for example, for a three-dimensional tensor $X \sim (a, b, c)$ we can write X_i to denote a slice of size (b, c) , or $X_{i j k}$ for a single scalar. We use commas for more complex expressions, such as $X_{i,:j:k}$ to denote a slice of size $(b, k - j)$. When necessary to avoid clutter, we use a light-gray notation, $[X]_{i j k}$, to visually split the indexing part from the rest, where the argument inside the brackets can also be an expression.



$$\mathbf{X}_i \sim (b, c)$$

$$X_{ijk}$$

$$\mathbf{X}_{i,:j:k}$$

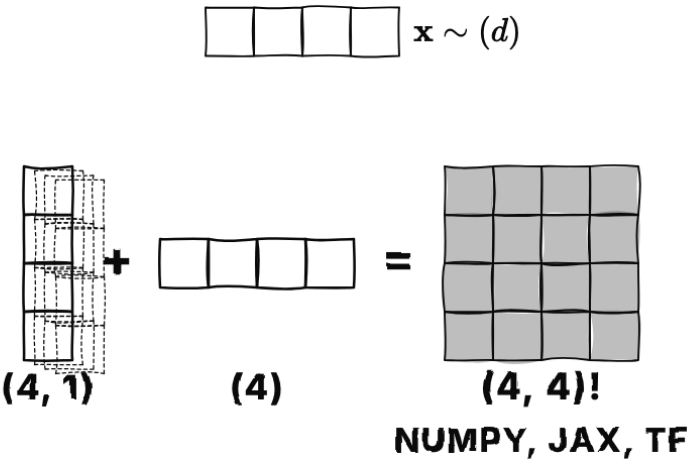
+ **×** **EXP** **SIN** **√**

Slicing a Tensor

Common vector operations. We are mostly concerned with models that can be written as compositions of differentiable operations. In fact, the majority of our models will consist of basic compositions of sums, multiplications, and some additional non-linearities such as the exponential $\exp(x)$, sines and cosines, and square roots.

Vectors $\mathbf{x} \sim (d)$ are examples of one-dimensional tensors. Linear algebra books are concerned with distinguishing between column vectors $\mathbf{x}$ and row vectors $\mathbf{x}^T$, and we will try to adhere to this convention as much as possible. In code this is trickier, because row and column vectors correspond to two-dimensional tensors of shape $(1, d)$ or $(d, 1)$, which are different from one-dimensional tensors of shape (d) . This is important to keep in mind because most

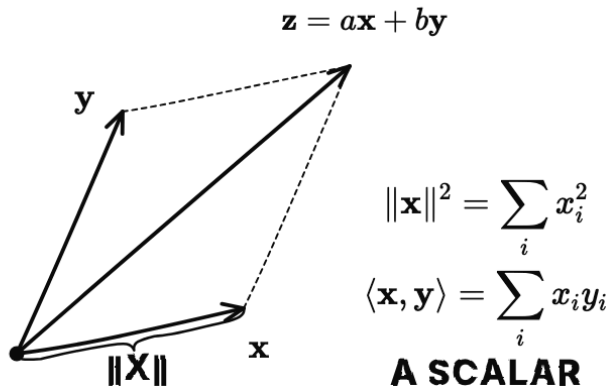
frameworks implement broadcasting rules inspired by NumPy, giving rise to non-intuitive behaviors; in a nutshell, broadcasting aligns the tensors' shapes from the right, and repeats a tensor whenever possible to match the two shapes. Box C.2.1 shows an example of a very common error arising from implicit broadcasting of tensors' shapes.



A Broadcasting Trap

In that box, which is probably incorrect broadcasting, a “column” of shape 4 by 1 is added to a one-dimensional tensor of shape 4, and the result has shape 4 by 4: an elementwise operation on two vectors produces a matrix because of their shapes. The same result can be obtained in practically any framework, such as NumPy, TensorFlow or JAX.

Vectors possess their own algebra, which we call a vector space, in the sense that any two vectors $\mathbf{x}$ and $\mathbf{y}$ of the same shape can be linearly combined, $\mathbf{z} = a\mathbf{x} + b\mathbf{y}$, to provide a third vector, with $z_i = ax_i + by_i$.



Adding Vectors, Measuring Length

If we understand a vector as a point in d -dimensional Euclidean space, the sum is interpreted by forming a parallelogram, while the distance of a vector from the origin is given by the Euclidean, or ℓ_2 , norm: $\|\mathbf{x}\| = \sqrt{(\sum_i x_i^2)}$.

The squared norm $\|\mathbf{x}\|^2$ is of particular interest, as it corresponds to the sum of the elements squared. The fundamental vector operation we are interested in is the inner product, or dot product, which is

given by multiplying the two vectors element-wise and summing the resulting values.

Definition, inner product: the inner product between two vectors $\mathbf{x}$, $\mathbf{y}$ $\sim$ (d) is given by the expression $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y} = \sum_i x_i y_i$.

The angle-bracket notation is common in physics, and we use it sometimes for clarity. Importantly, the dot product between two vectors is a scalar. For example, if $\mathbf{x} = [0.1, 0, -0.3]$ and $\mathbf{y} = [-4.0, 0.05, 0.1]$, then $\langle \mathbf{x}, \mathbf{y} \rangle = -0.4 + 0 - 0.03 = -0.43$.

$$\begin{array}{c} \mathbf{x} \quad \boxed{\begin{array}{|c|c|c|} \hline 0.1 & 0 & -0.3 \\ \hline \end{array}} \\ \mathbf{y} \quad \boxed{\begin{array}{|c|c|c|} \hline -4 & 0.05 & 0.1 \\ \hline \end{array}} \\ -0.4 \quad 0 \quad -0.03 = -0.43 \end{array}$$

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

COSINE SIMILARITY



-1

A Dot Product, Worked

A simple geometric interpretation of the dot product is given by its relation with the angle α between the two vectors: $\mathbf{x}^T \mathbf{y} = \|\mathbf{x}\| \|\mathbf{y}\| \cos(\alpha)$.

Hence, for two normalized vectors, whose norm equals 1, the dot product is equivalent to the cosine of their angle, in which case we call the dot product the cosine similarity. The cosine similarity $\cos(\alpha)$ oscillates between 1, for two vectors pointing in the same direction, and -1 , for two vectors pointing in opposite directions, with the special case of $\langle \mathbf{x}, \mathbf{y} \rangle = 0$ giving rise to orthogonal vectors pointing in perpendicular directions. Looking at this from another direction, for two normalized vectors, having unit norm, once we fix the first argument $\mathbf{x}$ we get what follows.

Questions

1. Why does adding a tensor of shape (4, 1) to one of shape (4) give a (4, 4) result?
 - a. Because addition of vectors always produces a matrix
 - b. Because broadcasting aligns shapes from the right and repeats each tensor to match, turning the elementwise sum into a 4 by 4 table
 - c. Because the framework transposes the column automatically
 - d. Because one of the tensors holds integers
2. What is the inner product of two vectors?
 - a. A scalar: the sum of the elementwise products
 - b. A vector of the elementwise products
 - c. A matrix built from the two vectors
 - d. The angle between the two vectors

3. For two normalized vectors, what does an inner product of 0 mean?
 - a. They point in the same direction
 - b. They point in opposite directions
 - c. They are orthogonal, pointing in perpendicular directions
 - d. One of them is the zero vector
4. What does the book mean by the rank n of a tensor?
 - a. The number of its elements
 - b. The number of its dimensions, 0 for scalars, 1 for vectors, 2 for matrices
 - c. Its largest value
 - d. The number of tensors it is composed of

Answers

1. **b)** Broadcasting aligns the shapes from the right and repeats tensors to match them, so a column plus a one-dimensional tensor becomes a full matrix: probably not what was intended.
2. **a)** The inner product multiplies the vectors element-wise and sums the results, so it is a single number.
3. **c)** For unit vectors the dot product is the cosine of their angle, and a cosine of 0 means perpendicular, orthogonal vectors.
4. **b)** A tensor is an n -dimensional array; $n = 0$ gives scalars, $n = 1$ vectors, $n = 2$ matrices.

Worked examples

A dot product by hand Compute the inner product of $x = [0.1, 0, -0.3]$ and $y = [-4.0, 0.05, 0.1]$.

Multiply the vectors element by element: $0.1 \times (-4.0) = -0.4$, $0 \times 0.05 = 0$, and $-0.3 \times 0.1 = -0.03$.

Sum the three products, since the inner product is the sum of the elementwise products.

$-0.4 + 0 - 0.03 = -0.43$; the result is a scalar, as every dot product is.

Answer: -0.43

Practice

1. Find the Euclidean norm of $x = [3, 4]$ and its squared norm.

Hint: Square the elements, sum them, then take the square root.

Answer: The squared norm is $3^2 + 4^2 = 9 + 16 = 25$, and the norm is $\sqrt{25} = 5$.

2. The unit vectors $x = [1, 0]$ and $y = [0, 1]$ are given. What is their cosine similarity, and what does it say about their directions?

Hint: For unit vectors the dot product equals the cosine of the angle.

Answer: Their dot product is $1 \times 0 + 0 \times 1 = 0$, so the cosine similarity is 0: the vectors are orthogonal, pointing in perpendicular directions.

3. You add a tensor of shape (3, 1) to a tensor of shape (3). What shape does broadcasting give, and how would you get an elementwise sum of shape (3) instead?

Hint: Broadcasting aligns shapes from the right and repeats to match.

Answer: The result has shape (3, 3), because the column is repeated across 3 columns and the one-dimensional tensor across 3 rows. To get a plain elementwise sum, give both tensors the same shape, for example by reshaping the (3, 1) tensor to (3) first.

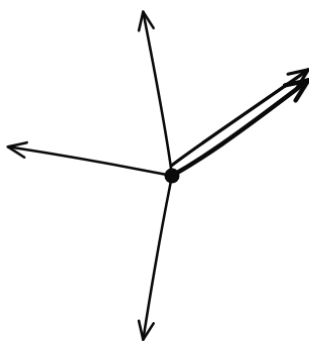
Common matrix operations

$\mathbf{y}^* = \operatorname{argmax} \langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}$, where argmax denotes the operation of finding the value of $\mathbf{y}$ corresponding to the highest possible output value.

From equation E.2.3 we see that, to maximize the dot product, the second vector must equal the first one. This is important, because in the following chapters $\mathbf{x}$ will represent an input, while $\mathbf{w}$ will represent adaptable parameters, so that the dot product is maximized whenever $\mathbf{x}$ “resonates” with $\mathbf{w}$, which is called template matching.

$$\langle \mathbf{x}, \mathbf{1} \rangle = \sum_i x_i$$

RESONATES
 $\mathbf{W} = \mathbf{X}$: MAX

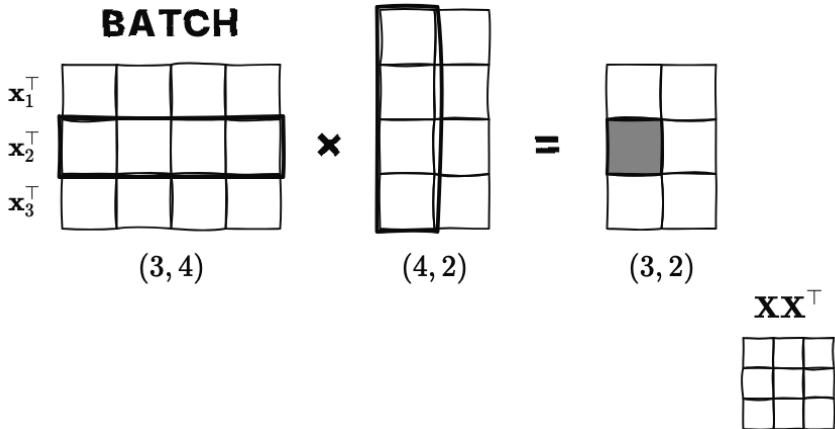

$$\|\mathbf{x} - \mathbf{y}\|^2 = \langle \mathbf{x}, \mathbf{x} \rangle + \langle \mathbf{y}, \mathbf{y} \rangle - 2\langle \mathbf{x}, \mathbf{y} \rangle$$
$$\|\mathbf{x}\|^2 = \langle \mathbf{x}, \mathbf{x} \rangle$$

The Best Match Is Itself

We close with two additional observations that will be useful. First, we can write the sum of the elements of a vector as its dot product with a vector $\mathbf{1}$ composed entirely of ones, $\mathbf{1} = [1, 1, \dots, 1]^T$: $\langle \mathbf{x}, \mathbf{1} \rangle = \sum_{i=1}^d x_i$. Second, the distance between two vectors can also be written in terms of their dot products: $\|\mathbf{x} - \mathbf{y}\|^2 = \langle \mathbf{x}, \mathbf{x} \rangle + \langle \mathbf{y}, \mathbf{y} \rangle - 2\langle \mathbf{x}, \mathbf{y} \rangle$. The case $\mathbf{y} = \mathbf{0}$ gives us $\|\mathbf{x}\|^2 = \langle \mathbf{x}, \mathbf{x} \rangle$. Both equations can be useful when writing equations or in the code.

Common matrix operations. In the two-dimensional case we have matrices: $\mathbf{X} \sim (n, d)$, a grid of elements from X_{11} in the top-left corner to X_{nd} in the bottom-right. In this case we can talk about a matrix with n rows and d columns. Of particular importance for the following, a matrix can be understood as a stack of n vectors $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$, where the stack is organized in a row-wise fashion: the rows of $\mathbf{X}$ are $\mathbf{x}_1^T$ down to $\mathbf{x}_n^T$.

DOT PRODUCT



Rows Times Columns

We say that $\mathbf{X}$ represents a batch of data vectors. As we will see, it is customary to define models, both mathematically and in code, to work on batched data of this kind. A fundamental operation for matrices is multiplication.

Definition, matrix multiplication: for any two matrices $\mathbf{X} \sim (a, b)$ and $\mathbf{Y} \sim (b, c)$ of compatible shape, matrix multiplication $\mathbf{Z} = \mathbf{XY}$, with $\mathbf{Z} \sim (a, c)$, is defined element-wise as $Z_{ij} = \langle \mathbf{X}_i, \mathbf{Y}_j^\top \rangle$. That is, the element (i, j) of the product is the dot product between the i -th row of $\mathbf{X}$ and the j -th column of $\mathbf{Y}$.

As a special case, if the second term is a vector we have a matrix-vector product: $\mathbf{z} = \mathbf{W}\mathbf{x}$.

If we interpret $\mathbf{X}$ as a batch of vectors, the matrix multiplication $\mathbf{XW}^T$ is a simple vectorized way of computing n dot products as in equation E.2.5, one for each row of $\mathbf{X}$, with a single linear algebra operation. As another example, the multiplication of a matrix by its transpose, $\mathbf{XX}^T \sim (n, n)$, is a vectorized way to compute all possible dot products of pairs of rows of $\mathbf{X}$ simultaneously.

We close by mentioning a few additional operations on matrices that will be important.

1	2
3	4

⊙

5	6
7	8

=

5	12
21	32

exp($\mathbf{X}$)

e^1	e^2
e^3	e^4

mat-exp($\mathbf{X}$) = $\sum_k \mathbf{X}^k / k!$

SQUARE ONLY

LINALG MODULE

Element by Element

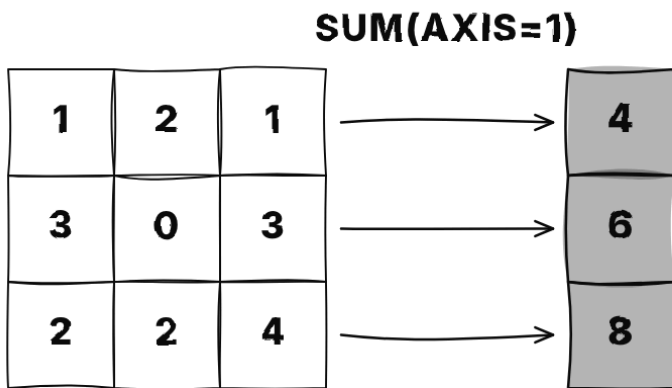
Definition, Hadamard multiplication: for two matrices of the same shape, the Hadamard multiplication is defined element-wise as $[\mathbf{X} \odot \mathbf{Y}]_{i,j} = X_{i,j} Y_{i,j}$.

While Hadamard multiplication does not have all the interesting algebraic properties of standard matrix multiplication, it is commonly used in differentiable models for performing masking operations, for example setting some elements to zero, or scaling operations. Multiplicative interactions have also become popular in some recent families of models, as we will see next.

Sometimes we write expressions such as $\exp(\mathbf{X})$, which are to be interpreted as element-wise applications of the operation: $[\exp(\mathbf{X})]_{i,j} = \exp(X_{i,j})$.

By comparison, “true” matrix exponentiation is defined for a square matrix as $\text{mat-exp}(\mathbf{X}) = \sum_{k=0}^{\infty} \mathbf{X}^k / k!$.

Importantly, equation E.2.6 can be defined for tensors of any shape, while E.2.7 is only valid for square matrices. This is why all frameworks, like PyTorch, have specialized modules that collect all matrix-specific operations, such as inverses and determinants. Box C.2.2 gives an example, contrasting the element-wise exponential of a matrix with the matrix exponential of linear algebra textbooks: in PyTorch the first is the ordinary `exp` function, the second lives in the linear algebra sub-package as `matrix exp`. Specialized linear algebra operations are generally encapsulated in their own sub-package.



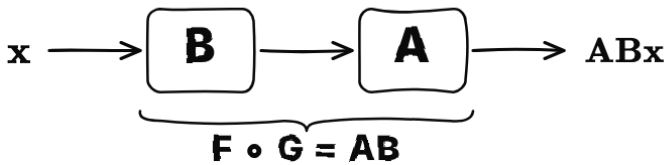
Reducing Along an Axis

Finally, we can write reduction operations, such as sum or mean, across axes without specifying lower and upper indices, in which case we assume that the summation runs along the full axis: $\sum_i \mathbf{X}_i = \sum_{i=1}^n \mathbf{X}_i$. In PyTorch and other frameworks, reduction operations correspond to methods having an axis argument, for example a sum along axis 1.

A box in the book asks: on the definition of matrix multiplication, why is it defined as in equation E.2.4 and not as Hadamard multiplication? Consider a vector $\mathbf{x}$ and some generic function f defined on it. The function is said to be linear if $f(\alpha \mathbf{x}_1 + \beta \mathbf{x}_2) = \alpha f(\mathbf{x}_1) + \beta f(\mathbf{x}_2)$. Any such function can be represented as a matrix $\mathbf{A}$; this can be seen by extending the two vectors in a basis representation. Then the matrix-

vector product $\mathbf{Ax}$ corresponds to function application, $f(\mathbf{x}) = \mathbf{Ax}$, and the matrix multiplication $\mathbf{AB}$ corresponds to function composition $f \circ g$, where $(f \circ g)(\mathbf{x}) = f(g(\mathbf{x}))$ and $g(\mathbf{x}) = \mathbf{Bx}$.

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



Matrices Are Linear Functions

Questions

1. For unit vectors, which $\mathbf{y}$ maximizes the dot product $\langle \mathbf{x}, \mathbf{y} \rangle$?
 - a. $\mathbf{y} = -\mathbf{x}$
 - b. $\mathbf{y} = \mathbf{x}$
 - c. Any $\mathbf{y}$ orthogonal to $\mathbf{x}$

- d. The zero vector
2. What is the element (i, j) of the matrix product $\mathbf{XY}$?
- a. $X_{i,j}$ times $Y_{i,j}$
 - b. The sum of row i of $\mathbf{X}$
 - c. The dot product of the i -th row of $\mathbf{X}$ with the j -th column of $\mathbf{Y}$
 - d. The dot product of the i -th column of $\mathbf{X}$ with the j -th row of $\mathbf{Y}$
3. What does the product $\mathbf{XX}^T$ compute for a batch $\mathbf{X}$ of n row vectors?
- a. All the dot products between pairs of rows, in an n by n matrix
 - b. The sum of all the rows
 - c. The element-wise squares of $\mathbf{X}$
 - d. The norm of each column
4. Why is matrix multiplication defined row-times-column rather than element by element?
- a. Because it is faster on GPUs
 - b. Because a linear function is a matrix, so the matrix product corresponds to composing two linear functions
 - c. Because the Hadamard product only works for vectors
 - d. Because it avoids broadcasting errors

Answers

1. **b)** To maximize the dot product, the second vector must equal the first; this is why an input 'resonates' with parameters that match it, template matching.

- 2. c)** Matrix multiplication is defined element-wise as the dot product between row i of the first matrix and column j of the second.
- 3. a)** Multiplying a matrix by its transpose is a vectorized way to compute every pairwise dot product of its rows at once.
- 4. b)** Any linear function can be represented as a matrix; the matrix-vector product is applying it, and the matrix product is function composition.

Worked examples

Matrix product versus Hadamard product Let $X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $Y = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$. Compute the matrix product XY and the Hadamard product $X \odot Y$.

For the matrix product, each element is a row of X dotted with a column of Y .

Row 1 with column 1: $1 \times 5 + 2 \times 7 = 19$; row 1 with column 2: $1 \times 6 + 2 \times 8 = 22$.

Row 2 with column 1: $3 \times 5 + 4 \times 7 = 43$; row 2 with column 2: $3 \times 6 + 4 \times 8 = 50$.

For the Hadamard product, multiply matching elements: $1 \times 5 = 5$, $2 \times 6 = 12$, $3 \times 7 = 21$, $4 \times 8 = 32$.

Answer: $XY = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$, while $X \odot Y = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$.

Practice

- 1.** Use the vector of ones to find the sum of the elements of $x = [2, -1, 4]$.

Hint: The sum of the elements equals $\langle x, 1 \rangle$.

Answer: $\langle x, 1 \rangle = 2 \times 1 + (-1) \times 1 + 4 \times 1 = 5$.

- 2.** For $x = [1, 2]$ and $y = [3, 1]$, compute the squared distance $\|x - y\|^2$ using only dot products, then check it directly.

Hint: Use $\|x - y\|^2 = \langle x, x \rangle + \langle y, y \rangle - 2\langle x, y \rangle$.

Answer: $\langle x, x \rangle = 5$, $\langle y, y \rangle = 10$, $\langle x, y \rangle = 3 + 2 = 5$, so $5 + 10 - 10 = 5$.

Directly, $x - y = [-2, 1]$ and $(-2)^2 + 1^2 = 5$.

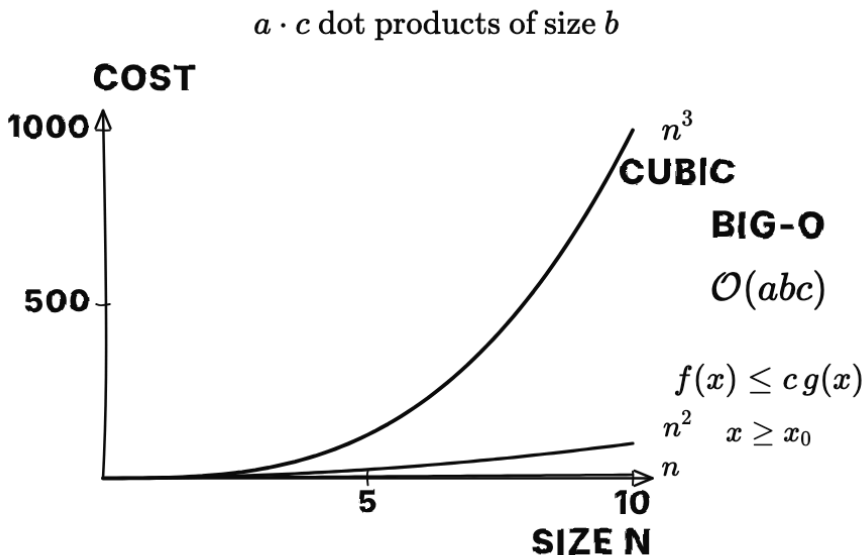
3. A matrix X has shape $(10, 3)$ and W has shape $(4, 3)$. What is the shape of XW^T , and what does each element hold?

Hint: Transpose W first, then check the inner dimensions.

Answer: W^T has shape $(3, 4)$, so XW^T has shape $(10, 4)$. Each element (i, j) is the dot product of row i of X with row j of W .

Computational complexity

I will use matrix multiplication to introduce the topic of the complexity of an operation. Looking at equation E.2.4, we see that computing the matrix $\mathbf{Z} \sim (a, c)$ from the input arguments $\mathbf{X} \sim (a, b)$ and $\mathbf{Y} \sim (b, c)$ requires ac inner products of dimension b if we directly apply the definition, which is what we call the time complexity, while the memory requirement for a sequential implementation is simply the size of the output matrix, which we call instead the space complexity.



How Cost Grows with Size

To abstract away from the specific hardware details, computer science focuses on the so-called big- $\mathcal{O}$ notation, from the German Ordnung, which stands for order of approximation. A function $f(x)$ is said to be $\mathcal{O}(g(x))$, where we assume both inputs and outputs are non-negative, if we can find a constant c and a value x_0 such that $f(x) \leq c \cdot g(x)$ for any $x \geq x_0$. This means that as soon as x grows sufficiently large, we can ignore all factors in our analysis outside of $g(x)$. This is called an asymptotic analysis. Hence, we can say that a naive implementation of matrix multiplication is $\mathcal{O}(abc)$, growing linearly with respect to all three input parameters. For two square matrices of size (n, n) , we say matrix multiplication is cubic in the input dimension.

Reasoning in terms of asymptotic complexity is important, and elegant, but choosing an algorithm only in terms of big- $\mathcal{O}$ complexity does not necessarily translate into practical performance gains, which depend on many details such as what hardware is used, what parallelism is supported, and so on. When you call a specific primitive in a linear algebra framework, such as matrix multiplication in PyTorch, the low-level implementation that is executed, the kernel, depends on the run-time hardware, through a process known as dispatching; hence the same code can run via a GPU kernel, a CPU kernel, a TPU kernel, and so on. This is made even more complex by compilers such as XLA, which can optimize code by fusing operations with a specific target hardware in mind. As an example, it is known that the best asymptotic algorithm for multiplying two square matrices of size (n, n) scales as $\mathcal{O}(n^c)$ for a constant $c < 2.4$, which is much better than the cubic $\mathcal{O}(n^3)$ requirement of a naive implementation. However, these algorithms are much harder to parallelize efficiently on highly parallel hardware such as GPUs, making them uncommon in practice.

$\mathcal{O}(n^{2.37})$ is hard to run on GPUs

MEMORY-BOUND



COMPUTE-BOUND



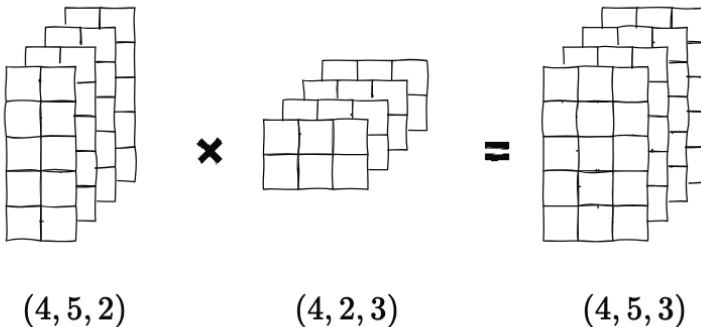
MEASURE: PROFILER

Memory-Bound or Compute-Bound

Note that, from the point of view of asymptotic complexity, having access to a parallel environment with k processors has no impact, since it can only provide, at best, a constant $1/k$ speedup over a non-parallel implementation. In addition, asymptotic complexity does not take into consideration the time it takes to move data from one location to another, which can become the major bottleneck in many situations. In these cases, we say the implementation is memory-bound, as opposed to compute-bound. In practice, this can only be checked by running a profiler over the code. We will see that analyzing the complexity of an algorithm is far from trivial due to the interplay of asymptotic complexity and observed complexity.

Higher-order tensor operations. Vectors and matrices are interesting because they allow us to define a large number of operations which are undefined or complex in higher dimensions, for example matrix exponentials, matrix multiplication, and determinants. When moving to higher dimensions, most of the operations we are interested in are either batched variants of matrix operations, or specific combinations of matrix operations and reduction operations.

$$[\text{BMM}(\mathbf{X}, \mathbf{Y})]_i = \mathbf{X}_i \mathbf{Y}_i$$



BATCH FIRST

Batched Matrix Multiplication

As an example of the former, consider two tensors $\mathbf{X} \sim (n, a, b)$ and $\mathbf{Y} \sim (n, b, c)$. Batched matrix multiplication, BMM, is defined as $[\text{BMM}(\mathbf{X}, \mathbf{Y})]_i = \mathbf{X}_i \mathbf{Y}_i \sim (n, a, c)$.

Operations in most frameworks operate transparently on batched versions of their arguments, which are assumed, as in this case, to be the leading dimensions, the first ones. For example, batched matrix multiplication in PyTorch is the same as standard matrix multiplication, as Box C.2.3 shows: multiplying a tensor of shape 4, 5, 2 by one of shape 4, 2, 3 gives a result of shape 4, 5, 3. Most operations can work on possibly batched inputs.

As an example of a reduction operation, consider two tensors $X, Y \sim (a, b, c)$. A generalized version of the dot product, GDT, can be written as $\text{GDT}(X, Y) = \sum_{i,j,k} [X \odot Y]_{i,j,k}$, which is simply a dot product over the “flattened” versions of its inputs. This brief overview covers most of the tensor operations we will use in the rest of the book, with additional material introduced when necessary.

Questions

1. Why is a naive matrix multiplication of $X \sim (a, b)$ by $Y \sim (b, c)$ said to be $\mathcal{O}(abc)$?
 - a. Because it needs $a \cdot c$ inner products, each of dimension b
 - b. Because the output has $a \cdot b \cdot c$ elements
 - c. Because it needs $a + b + c$ operations
 - d. Because it uses three processors
2. Why are algorithms faster than $\mathcal{O}(n^3)$ for matrix multiplication rarely used in practice?
 - a. They give wrong results
 - b. They need more memory than any computer has

- c. They are much harder to parallelize efficiently on GPUs
 - d. They only work for small matrices
3. What does it mean for an implementation to be memory-bound?
- a. Its run time is dominated by moving data around rather than by computing
 - b. It cannot run without a GPU
 - c. Its big-O complexity is cubic
 - d. It uses less memory than a compute-bound one
4. Why does having k processors not change the big-O complexity?
- a. Because processors cannot share data
 - b. Because it gives at best a constant $1/k$ speedup, and big-O ignores constant factors
 - c. Because big-O only applies to GPUs
 - d. Because k processors are always slower

Answers

1. **a)** Applying the definition directly requires ac inner products, each over b elements, so the work grows like abc .
2. **c)** Asymptotically better algorithms exist, with exponent below 2.4, but they parallelize poorly on highly parallel hardware such as GPUs.
3. **a)** Asymptotic analysis ignores data movement, which can become the main bottleneck; then the implementation is memory-bound, and only a profiler can tell.
4. **b)** Parallelism provides at most a constant factor of $1/k$, and the asymptotic notation discards constants.

Worked examples

Counting the work of a matrix product How many inner products, and how many scalar multiplications, does a naive implementation need to multiply $X \sim (100, 200)$ by $Y \sim (200, 50)$?

The output Z has shape $(a, c) = (100, 50)$, so there are $100 \times 50 = 5,000$ elements to compute.

Each element is one inner product of dimension $b = 200$, which needs 200 multiplications.

Total multiplications: $5,000 \times 200 = 1,000,000$, which is exactly $a \cdot b \cdot c$, matching $\mathcal{O}(abc)$.

Answer: 5,000 inner products and 1,000,000 multiplications.

Practice

1. Two square matrices of size (n, n) are multiplied naively. If n doubles, by what factor does the work grow?

Hint: The naive algorithm is cubic in n .

Answer: By $2^3 = 8$, since the work grows like n^3 .

2. In PyTorch, a tensor of shape $(8, 3, 4)$ is matrix-multiplied by a tensor of shape $(8, 4, 6)$. What is the shape of the result and why?

Hint: The leading dimension is treated as a batch.

Answer: $(8, 3, 6)$: the first dimension is a batch of 8, and each pair of slices, $(3, 4)$ times $(4, 6)$, gives a $(3, 6)$ matrix.

3. Explain in your own words why $\text{GDT}(X, Y)$ for two tensors of shape (a, b, c) is just a dot product.

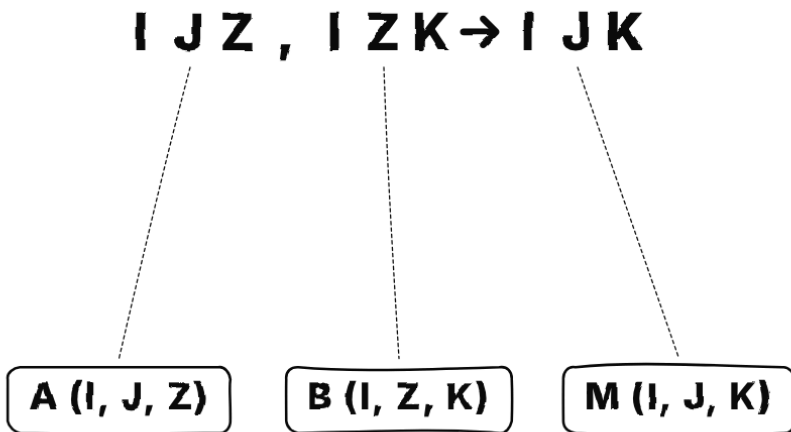
Hint: Imagine laying each tensor out as one long vector.

Answer: GDT multiplies the tensors element by element and sums everything; if both tensors are flattened into vectors of length abc in the

same order, that is exactly the sum of elementwise products, the ordinary dot product.

Einstein's notation, and derivatives

This is an optional section that covers einsum, a set of conventions that allows the user to specify practically every tensor operation, including reductions, sums and multiplications, with a simple syntax based on text strings. Box C.2.4 shows two examples of einsum in PyTorch: a batched matrix multiply, written with the string “ $i j z, i z k$, to $i j k$ ”, and a generalized dot product, written “ $i j k, i j k$, to nothing”.



PYTORCH = JAX

Reading an einsum String

To introduce the notation, let us consider again the two examples shown before in equations E.2.9 and E.2.10, writing down all the axes

explicitly: $M_{i j k} = \sum_z A_{i j z} B_{z k}$, and $M = \sum_i \sum_j \sum_k X_{i j k} Y_{i j k}$.

In line with Einstein’s notation, in a simplified version that ignores the distinction between upper and lower indices, we can simplify the two equations by removing the sums, under the convention that any index appearing on the right but not on the left is summed over. So $M_{i j k} = A_{i j z} B_{z k}$ stands for the sum over z , and $M = X_{i j k} Y_{i j k}$ stands for the sum over i, j and k .

Then we can condense the two definitions by isolating the indices in a single string, where the operands are now on the left: “ $i j z, i z k$, to $i j k$ ” for the batched matrix multiply, and “ $i j k, i j k$, to nothing” for the generalized dot product.

There is a direct one-to-one correspondence between the definitions in equations E.2.13 and E.2.14 and their simplified string definitions. This is implemented in most frameworks in the einsum operation, as Box C.2.4 shows.

The advantage of this notation is that we do not need to remember the API of a framework to implement a given operation, and translating from one framework to another is transparent because the einsum syntax is equivalent. For example, PyTorch has several matrix multiplication methods, including matmul and bmm, with different broadcasting rules and shape constraints, and einsum provides a uniform syntax for all of them. In addition, the einsum definition of our batched matrix multiplication is identical to, for example, the definition in JAX, as Box C.2.5 shows, with the very same string.

Working with transposed axes is also simple. For example, for $A \sim (n, a, b)$ and $B \sim (n, c, b)$, a batched multiplication of $[A]_i$ times $[B^T]_i$ is

obtained by switching the corresponding axes in the einsum definition, to “ ijz, ikz , to ijk ”.

$$\partial$$

$$ijz, ikz \rightarrow ijk$$

$$\updownarrow$$

$$ijz, ikz \rightarrow ijk$$

GRADIENTS AND JACOBIANS

Transposing by Swapping Letters

For these reasons, einsum and its generalizations, like the popular einops package, have recently gained wide popularity.

Section 2.2, Gradients and Jacobians. As the name differentiable implies, gradients play a pivotal role in the book, providing a way to optimize our models through semi-automatic mechanisms deriving from gradient descent. We recall here some basic definitions and concepts concerning differentiation of multi-valued functions. We focus on properties that will be essential later, partially at the expense of mathematical precision.

Derivatives of scalar functions. Starting from a simple function $y = f(x)$ with a scalar input and a scalar output, its derivative is defined as follows.

Definition, derivative: the derivative of $f(x)$ is defined as $f'(x) = \lim_{h \rightarrow 0} (f(x + h) - f(x)) / h$.

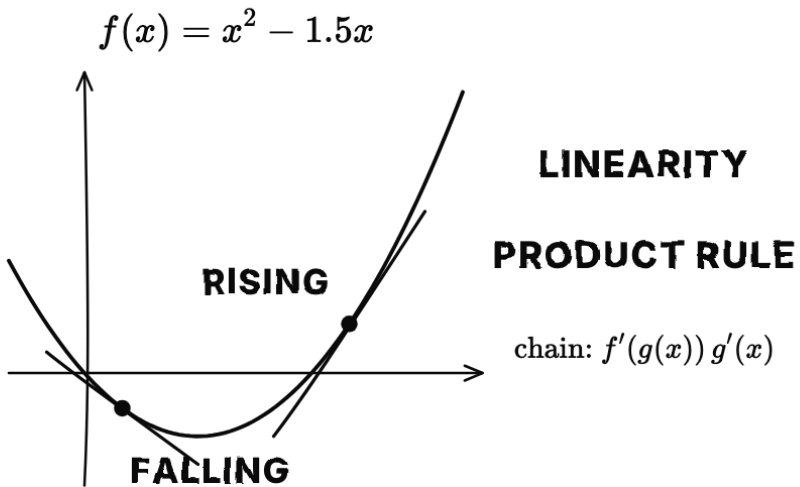
We use a variety of notations to denote derivatives. ∂ will denote generically derivatives and gradients of any dimension, vectors and matrices; ∂_x or $\partial/\partial x$ highlights the input argument we are differentiating with respect to, when needed; while $f'(x)$ is specific to scalar functions, and it is sometimes called Lagrange's notation.

We are not concerned here with the existence of the derivative of the function, which is not guaranteed everywhere even for a continuous function, and which we assume as given. We will only touch upon this point when discussing derivatives of non-smooth functions, such as the absolute value $f(x) = \text{abs}(x)$ at 0, later on in Chapter 6.

Derivatives of simple functions can be obtained by direct application of the definition; for example, the derivatives of a polynomial, a logarithm or a sine should be familiar: $\partial x^p = p x^{p-1}$, $\partial \log(x) = 1/x$, and $\partial \sin(x) = \cos(x)$.

Geometrically, the derivative can be understood as the slope of the tangent passing through a point, or equivalently as the best first-order approximation of the function itself at that point, as shown in Figure F.2.1. This is a fundamental point of view, because the slope of the line tells us how the function is evolving in a close neighborhood: for a positive slope, the function is increasing on the right and decreasing on the left, again for a sufficiently small interval, while for a negative slope the opposite is true. As we will see, this insight extends to

vector-valued functions. The figure plots the function $f(x) = x^2 - 1.5x$, along with its derivatives at two separate points.



Slopes of $x^2 - 1.5x$

We recall some important properties of derivatives that extend to the multi-dimensional case. Linearity: the derivative is linear, so the derivative of a sum is the sum of derivatives, $\partial[f(x) + g(x)] = f'(x) + g'(x)$. The product rule: $\partial[f(x)g(x)] = f'(x)g(x) + f(x)g'(x)$. The chain rule: the derivative of a function composition is given by multiplying the corresponding derivatives, $\partial[f(g(x))] = f'(g(x)) g'(x)$.

Questions

1. In Einstein's notation as used in the book, which indices are summed over?
 - a. Every index that appears twice on the left
 - b. Any index that appears on the right of the equals sign but not on the left
 - c. Only the index z
 - d. None; sums must always be written
2. What does the einsum string “ $ij\ k, ij\ k, \text{ to nothing}$ ” compute?
 - a. A batched matrix multiply
 - b. A transpose
 - c. A generalized dot product: every index is summed, leaving a single number
 - d. An element-wise product kept as a tensor
3. What does a positive derivative at a point tell you about the function nearby?
 - a. The function is increasing to the right and decreasing to the left, in a small enough interval
 - b. The function is positive there
 - c. The function has a maximum there
 - d. The function is constant nearby
4. What does the chain rule say about the derivative of $f(g(x))$?
 - a. It equals $f'(x) + g'(x)$
 - b. It equals $f'(g(x))$ times $g'(x)$
 - c. It equals $f'(x)$ times $g'(x)$
 - d. It equals $f(g'(x))$

Answers

1. **b)** The convention removes the sum signs: an index present on the right but missing on the left is summed over.
2. **c)** Nothing remains on the output side, so all three indices are summed: the generalized dot product.
3. **a)** The derivative is the slope of the tangent; a positive slope means the function rises to the right and falls to the left, locally.
4. **b)** The derivative of a composition is the product of the derivatives, the outer one evaluated at $g(x)$.

Worked examples

Slopes of the book's example function For $f(x) = x^2 - 1.5x$, the function plotted in Figure F.2.1, find the derivative and its value at $x = 0$ and at $x = 1$. Say whether f is increasing or decreasing near each point.

Differentiate term by term, using linearity and the rule for powers: the derivative of x^2 is $2x$ and the derivative of $1.5x$ is 1.5 .

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

At $x = 0$: $f'(0) = -1.5$, a negative slope, so f is decreasing near 0 .

At $x = 1$: $f'(1) = 2 - 1.5 = 0.5$, a positive slope, so f is increasing near 1 .

Answer: $f'(x) = 2x - 1.5$; $f'(0) = -1.5$ (decreasing), $f'(1) = 0.5$ (increasing).

Practice

1. Use the product rule to differentiate $f(x) = x \cdot \sin(x)$.

Hint: Take the first factor as x and the second as $\sin(x)$.

Answer: $f'(x) = 1 \cdot \sin(x) + x \cdot \cos(x) = \sin(x) + x \cos(x)$.

2. Use the chain rule to differentiate $h(x) = \log(x^2)$ for $x > 0$, and simplify.

Hint: The outer function is $\log$, the inner one x^2 .

Answer: $h'(x) = (1/x^2) \cdot 2x = 2/x$.

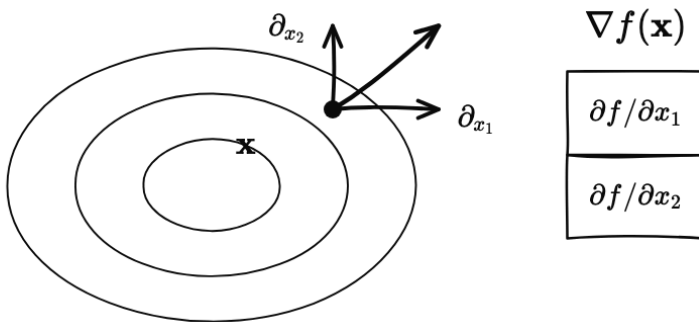
3. Write an einsum string for the matrix-vector product $z = Wx$, where W has indices i, j and x has index j .

Hint: Which index disappears from the output?

Answer: “ $i j, j, \text{ to } i$ ”: j appears on the input side but not in the output, so it is summed over, giving $z_i = \sum_j W_{ij} x_j$.

Gradients, directional derivatives and Jacobians

Consider now a function $y = f(\mathbf{x})$ taking a vector $\mathbf{x} \sim (d)$ as input. Talking about infinitesimal perturbations here does not make sense unless we specify the direction of the perturbation: while in the scalar case we only had “left” and “right”, in this case we have infinitely many possible directions in Euclidean space. In the simplest case, we can consider moving along the i -th axis, keeping all other values fixed: $\partial_{x_i} f(\mathbf{x}) = \partial y / \partial x_i = \lim_{h \rightarrow 0} (f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})) / h$. Here $\mathbf{e}_i \sim (d)$ is the i -th basis vector, the i -th row of the identity matrix: its j -th element is 1 if $i = j$, and 0 otherwise.



$$\mathbf{e}_i = [0, 1, 0]$$

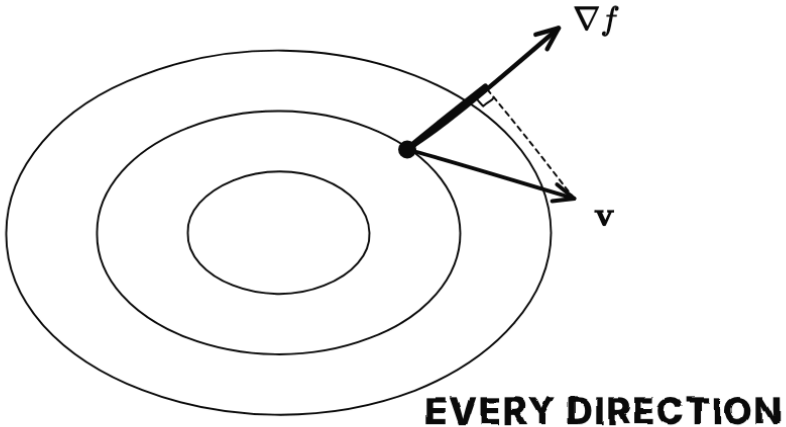
Partial Derivatives Make a Gradient

Equation E.2.17 is called a partial derivative. Stacking all the partial derivatives together gives us a d -dimensional vector called the gradient of the function.

Definition, gradient: the gradient of a function $y = f(\mathbf{x})$ is the column vector $\nabla f(\mathbf{x}) = \partial f(\mathbf{x})$, whose entries are the partial derivatives from $\partial_{x_1} f(\mathbf{x})$ down to $\partial_{x_d} f(\mathbf{x})$.

Because gradients are fundamental, we use the special notation $\nabla f(\mathbf{x})$ to distinguish them. What about displacements in a general direction $\mathbf{v}$? In this case we obtain the directional derivative: $D_{\mathbf{v}} f(\mathbf{x}) = \lim_{h \rightarrow 0} (f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})) / h$.

$$D_{\mathbf{v}}f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h} = \langle \nabla f(\mathbf{x}), \mathbf{v} \rangle$$



Any Direction from the Gradient

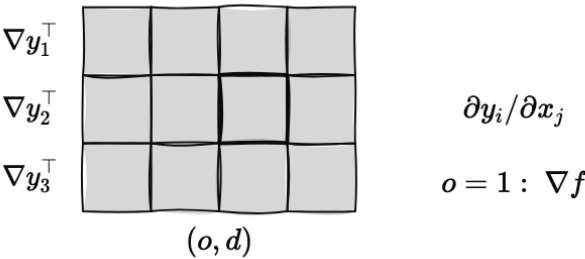
Movement in space can be decomposed into individual displacements along each axis; hence it is easy to prove that the directional derivative is given by the dot product of the gradient with the displacement vector $\mathbf{v}$: $D_{\mathbf{v}}f(\mathbf{x}) = \langle \nabla f(\mathbf{x}), \mathbf{v} \rangle = \sum_i \partial_{x_i} f(\mathbf{x}) v_i$, where each v_i is the displacement on the i -th axis.

Hence, knowing how to compute the gradient of a function is enough to compute all possible directional derivatives.

Jacobians. Let us now consider the generic case of a function $\mathbf{y} = f(\mathbf{x})$ with a vector input $\mathbf{x} \sim (d)$ as before, and this time a vector output $\mathbf{y} \sim (o)$. As we will see, this is the most general case we need to consider. Because we have more than one output, we compute a gradient for

each output, and their stack provides an (o, d) matrix called the Jacobian of f.

$$\mathbf{x} \sim (d) \rightarrow f \rightarrow \mathbf{y} \sim (o)$$



$$\partial[f \circ g] = \partial f \cdot \partial g$$



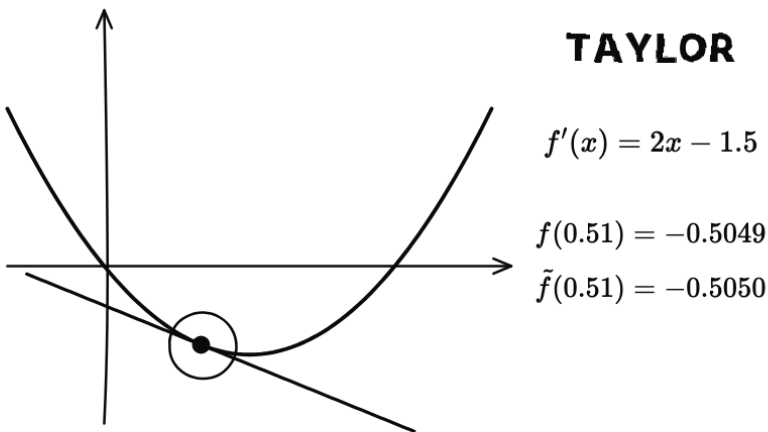
The Jacobian: Gradients Stacked

Definition, Jacobian: the Jacobian matrix of a function $\mathbf{y} = f(\mathbf{x})$, with $\mathbf{x} \sim (d)$ and $\mathbf{y} \sim (o)$, is the matrix $\partial f(\mathbf{x}) \sim (o, d)$ whose element in row i and column j is $\partial y_i / \partial x_j$, from $\partial y_1 / \partial x_1$ in the top-left corner to $\partial y_o / \partial x_d$ in the bottom-right.

We recover the gradient for $o = 1$, and the standard derivative for $d = o = 1$. Jacobians inherit all the properties of derivatives: importantly, the Jacobian of a composition of functions is now a matrix multiplication of the corresponding individual Jacobians, $\partial[f(g(\mathbf{x}))] = [\partial f(\bullet)] \partial g(\mathbf{x})$, where the first Jacobian is evaluated at $g(\mathbf{x}) \sim (h)$.

Like in the scalar case, gradients and Jacobians can be understood as linear functions tangent to a specific point. In particular, the gradient is the best “first-order approximation” in the following sense. For a point $\mathbf{x}_0$, the best linear approximation in an infinitesimal neighborhood of $f(\mathbf{x}_0)$ is given by $\tilde{f}(\mathbf{x}) = f(\mathbf{x}_0) + \langle \partial f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle$, where the gradient plays the part of the slope of the line, and $\mathbf{x} - \mathbf{x}_0$ is the displacement from $\mathbf{x}_0$.

$$\tilde{f}(\mathbf{x}) = f(\mathbf{x}_0) + \langle \partial f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle$$



The Tangent as an Approximation

This is called Taylor's theorem. Box C.2.6 and Figure F.2.2 visualize it in the scalar case $f(x) = x^2 - 1.5x$. The box writes the function, its derivative $2x - 1.5$ computed manually for now, and the linearization at $x = 0.5$. As a numerical check, at $x = 0.51$ the function gives -0.5049 and the linearization gives -0.5050 .

Questions

1. How do you get the directional derivative of f along any direction $\mathbf{v}$ once you know the gradient?
 - a. Take the dot product of the gradient with $\mathbf{v}$
 - b. Multiply the gradient by the length of $\mathbf{v}$ only
 - c. Take the largest entry of the gradient
 - d. You must compute a new limit for every direction
2. What is the shape of the Jacobian of a function from vectors of size d to vectors of size o ?
 - a. (d, o)
 - b. (o, d)
 - c. (d)
 - d. (o)
3. How does the chain rule look for Jacobians?
 - a. The Jacobian of a composition is the sum of the Jacobians
 - b. It is the Hadamard product of the Jacobians
 - c. It is the matrix product of the individual Jacobians, the outer one evaluated at $g(\mathbf{x})$
 - d. It only holds for scalar functions
4. In the book's check, why are $f(0.51) = -0.5049$ and its linearization -0.5050 so close?
 - a. Because f is linear
 - b. Because the linearization is the best first-order approximation near the point 0.5 , and 0.51 is very close to it

- c. Because the derivative is zero at 0.5
- d. Because rounding hides the difference

Answers

1. **a)** Movement decomposes along the axes, so the directional derivative is $\langle \nabla f(\mathbf{x}), \mathbf{v} \rangle$; the gradient is enough for every direction.
2. **b)** There is one gradient row for each of the o outputs, each with d entries, so the Jacobian is o by d .
3. **c)** Jacobians inherit the properties of derivatives; for a composition they multiply as matrices.
4. **b)** Taylor's theorem says the tangent line is the best linear approximation in a small neighborhood; a step of 0.01 stays in that neighborhood.

Worked examples

The book's linearization at 0.5 For $f(x) = x^2 - 1.5x$, build the first-order approximation at $x = 0.5$ and compare it with f at $x = 0.51$.

Evaluate the function at the point: $f(0.5) = 0.25 - 0.75 = -0.5$.

Evaluate the derivative $2x - 1.5$ at the point: $2 \times 0.5 - 1.5 = -0.5$, the slope of the tangent.

The linearization is $\tilde{f}(x) = -0.5 + (-0.5)(x - 0.5)$; at 0.51 the displacement is 0.01, so $\tilde{f}(0.51) = -0.5 - 0.005 = -0.505$.

The true value is $f(0.51) = 0.2601 - 0.765 = -0.5049$, which differs by only 0.0001.

Answer: $\tilde{f}(0.51) = -0.5050$ against $f(0.51) = -0.5049$.

Practice

1. For $f(x_1, x_2) = x_1^2 + 3x_1x_2$, find the gradient at $(1, 2)$ and the directional derivative along $\mathbf{v} = (1, -1)$.

Hint: Differentiate with respect to each variable while keeping the other fixed, then take a dot product with $\mathbf{v}$.

Answer: $\partial f / \partial x_1 = 2x_1 + 3x_2 = 2 + 6 = 8$ and $\partial f / \partial x_2 = 3x_1 = 3$, so the gradient is $(8, 3)$. Along $\mathbf{v}$: $8 \times 1 + 3 \times (-1) = 5$.

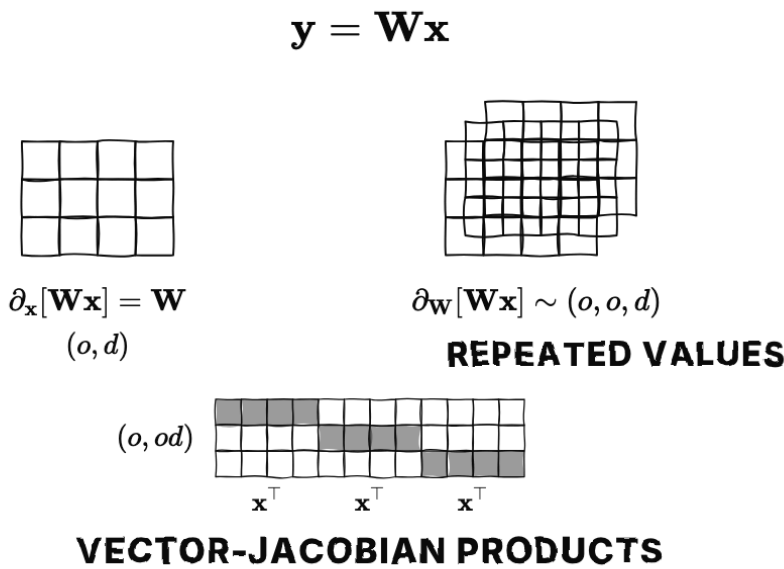
2. Find the Jacobian of $\mathbf{y} = (x_1x_2, x_1 + x_2)$ at the point $(2, 3)$, and give its shape.

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

Answer: Row 1: $(\partial(x_1x_2)/\partial x_1, \partial(x_1x_2)/\partial x_2) = (x_2, x_1) = (3, 2)$. Row 2: $(1, 1)$. The Jacobian is $[[3, 2], [1, 1]]$, of shape $(2, 2)$.

The dimensions of Jacobians, and gradient descent

We close with a pedantic note on dimensionality that will be useful in the following. Consider the function $\mathbf{y} = \mathbf{W}\mathbf{x}$. When viewed as a function of $\mathbf{x}$, the derivative is, as before, an (o, d) matrix, and it can be shown that $\partial_{\mathbf{x}}[\mathbf{W}\mathbf{x}] = \mathbf{W}$.



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

Figure F.2.2 shows the function $f(\mathbf{x}) = \mathbf{x}^2 - 1.5\mathbf{x}$ and its first-order approximation at 0.5.

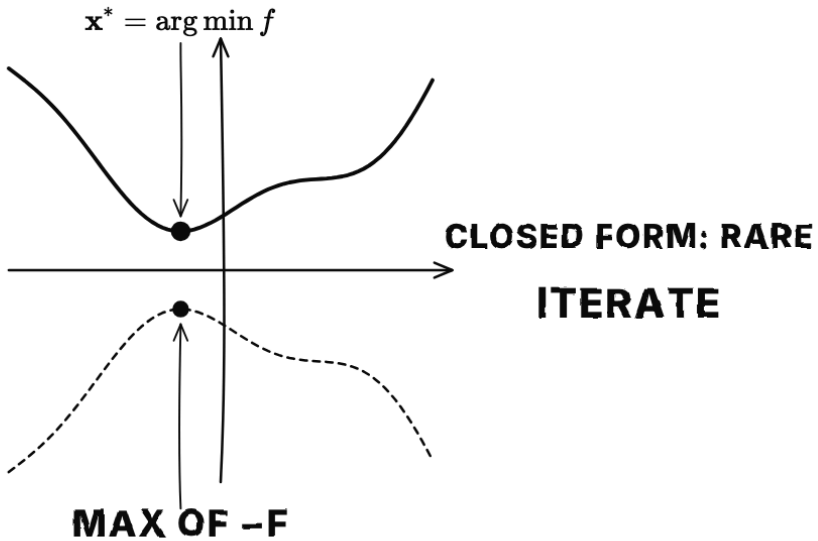
When viewed as a function of $\mathbf{W}$, instead, the input is itself an (o, d) matrix, and the “Jacobian” in this case has shape (o, o, d) , as the box that follows shows. However, we can always imagine an identical, isomorphic, function taking as input the vectorized version of $\mathbf{W}$, $\text{vect}(\mathbf{W}) \sim (od)$, in which case the Jacobian will be a matrix of shape (o, od) .

The box works out the Jacobian. To compute the Jacobian $\partial_{\mathbf{W}} \mathbf{W} \mathbf{x}$, we can rewrite the expression element-wise as $y_i = \sum_j W_{ij} x_j$, from which we immediately find that $\partial y_i / \partial W_{ij} = x_j$. Note that to materialize the Jacobian explicitly, that is, to store it in memory, we would need a lot of repeated values. As we will see in Chapter 6, this can be avoided because in practice we only care about the application of the Jacobian to another tensor.

This quick example clarifies what we mean by our statement that working with vector inputs and outputs is enough from a notational point of view. However, it will be important to keep this point in mind in Chapter 6, when we will use matrix Jacobians for simplicity of notation, in particular to avoid the proliferation of indices, but the sizes of these Jacobians may “hide” inside the actual shapes of the inputs and the outputs, most importantly the batch sizes. We will see in Chapter 6 that explicit computation of Jacobians can be avoided in practice by considering the so-called vector-Jacobian products. This can also be formalized by viewing Jacobians as abstract linear maps.

Section 2.3, Gradient descent. To understand the usefulness of having access to gradients, consider the problem of minimizing a generic function $f(\mathbf{x})$, with $\mathbf{x} \sim (d)$: $\mathbf{x}^* = \text{argmin}_{\mathbf{x}} f(\mathbf{x})$. Here, similarly to argmax , $\text{argmin} f(\mathbf{x})$ denotes the operation of finding the value of $\mathbf{x}$ corresponding to the lowest possible value of $f(\mathbf{x})$. We assume the

function has a single output, which is single-objective optimization, and that the domain over which we are optimizing $\mathbf{x}$ is unconstrained.

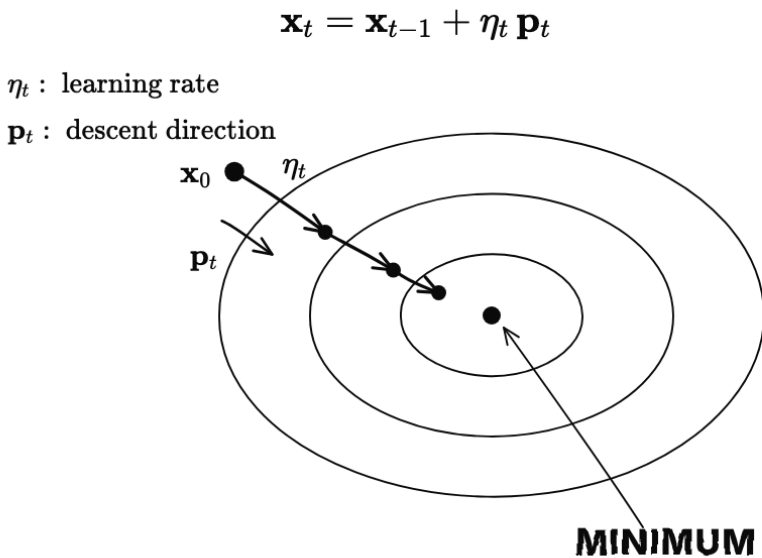


Looking for the Lowest Point

In the rest of the book $\mathbf{x}$ will encode the parameters of our model, and f will describe the performance of the model itself on our data, a setup called supervised learning that we introduce in the next chapter. We can consider minimizing instead of maximizing with no loss of generality, since minimizing $f(\mathbf{x})$ is equivalent to maximizing $-f(\mathbf{x})$ and vice versa; to visualize this, think of a function in one dimension and flip it across the x -axis, picturing what happens to its low points.

In very rare cases, we may be able to express the solution in closed form; we will see one example in the context of least-squares

optimization in Section 4.1.2. In general, however, we are forced to resort to iterative procedures. Suppose we start from a random guess $\mathbf{x}_0$ and that, at every iteration, we take a step, which we decompose in terms of its magnitude η_t , the length of the step, and the direction $\mathbf{p}_t$: $\mathbf{x}_t = \mathbf{x}_{t-1} + \eta_t \mathbf{p}_t$, the guess at iteration t being the previous guess plus the displacement at iteration t .

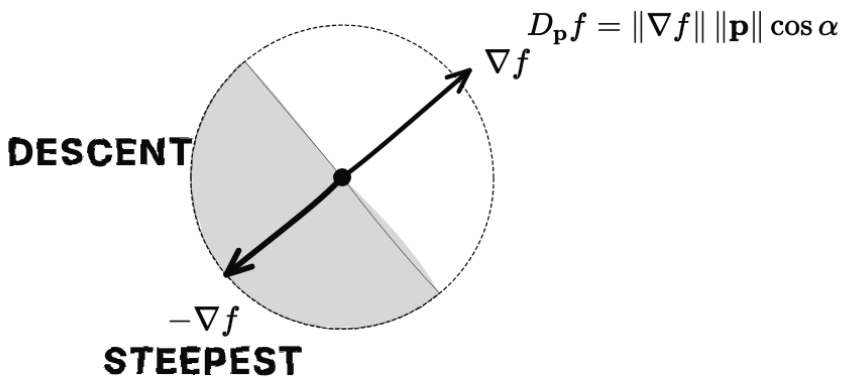


Step by Step Downhill

We call η_t the step size, or, in machine learning terminology, the learning rate, for reasons that will become clear in the next chapter. A direction $\mathbf{p}_t$ for which there exists an η_t such that $f(\mathbf{x}_t) \leq f(\mathbf{x}_{t-1})$ is called a descent direction. If we can select a descent direction at every iteration, and if we are careful in the choice of step size, the

iterative algorithm in equation E.2.26 will converge to a minimum, in a sense to be described shortly.

For differentiable functions, we can precisely quantify all descent directions by using the directional derivative from equation E.2.20, as they can be defined as the directions inducing a negative change with respect to our previous guess $\mathbf{x}_{t-1}$: if $\mathbf{p}_t$ is a descent direction, then $D_{\mathbf{p}_t} f(\mathbf{x}_{t-1}) \leq 0$.



Which Directions Go Down

Using what we learned in Section 2.2 and the definition of the dot product in terms of cosine similarity from equation E.2.2, we get $D_{\mathbf{p}_t} f(\mathbf{x}_{t-1}) = \langle \nabla f(\mathbf{x}_{t-1}), \mathbf{p}_t \rangle = \|\nabla f(\mathbf{x}_{t-1})\| \|\mathbf{p}_t\| \cos(\alpha)$, where α is the angle between $\mathbf{p}_t$ and $\nabla f(\mathbf{x}_{t-1})$.

Considering the expression on the right, the first term is a constant with respect to $\mathbf{p}_t$. Because we have assumed $\mathbf{p}_t$ only encodes the direction of movement, we can also safely restrict it to $\|\mathbf{p}_t\| = 1$, rendering the second term another constant. Hence, by the properties of the cosine, we deduce that any $\mathbf{p}_t$ whose angle with $\nabla f(\mathbf{x}_{t-1})$ is between $\pi/2$ and $3\pi/2$ is a descent direction. Among these, the direction $\mathbf{p}_t = -\nabla f(\mathbf{x}_{t-1})$, with an angle of π , has the lowest possible directional derivative, and we refer to it as the steepest descent direction.

Putting together this insight with the iterative procedure in equation E.2.26 gives us an algorithm to minimize any differentiable function, which we call steepest gradient descent.

Definition, steepest gradient descent: given a differentiable function $f(\mathbf{x})$, a starting point $\mathbf{x}_0$, and a step size sequence η_t , gradient descent proceeds as $\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla f(\mathbf{x}_{t-1})$.

We will not be concerned with the problem of finding an appropriate step size, which we will simply assume “small enough” so that the gradient descent iteration provides a reduction in f . In the next section we focus on what points are obtained by running gradient descent from a generic initialization. Note that gradient descent is as efficient as the procedure we use to compute the gradient: we introduce a general algorithm to this end in Chapter 6.

Questions

1. For $\mathbf{y} = \mathbf{W}\mathbf{x}$ viewed as a function of $\mathbf{W} \sim (\mathbf{o}, d)$, what shape does the Jacobian have?

- a. (o, d)
 - b. (o, o, d), or (o, od) if $\mathbf{W}$ is flattened
 - c. (d, o)
 - d. (o)
2. Which direction $\mathbf{p}_t$ has the lowest directional derivative, the steepest descent?
- a. The gradient itself
 - b. Any direction at a right angle to the gradient
 - c. Minus the gradient, at an angle of π to it
 - d. Any unit vector
3. Why can the book study minimization only, without loss of generality?
- a. Because maximization is impossible for differentiable functions
 - b. Because minimizing f is the same as maximizing $-f$
 - c. Because all functions have a single minimum
 - d. Because gradients only point downhill
4. Why does the book avoid materializing Jacobians such as $\partial_{\mathbf{W}} \mathbf{W}\mathbf{x}$ in memory?
- a. They would hold a lot of repeated values, and in practice only their application to another tensor is needed
 - b. They cannot be computed
 - c. They are always zero
 - d. They are only defined for scalars

Answers

1. **b)** The input is itself an o by d matrix, so the Jacobian has shape (o, o, d) ; flattening $\mathbf{W}$ into a vector of od entries makes it an (o, od) matrix.
2. **c)** With the norms fixed, the directional derivative is proportional to $\cos(\alpha)$, which is lowest, -1 , at $\alpha = \pi$: the negative gradient.
3. **b)** Flipping the function across the x -axis turns its high points into low points, so one problem can be turned into the other.
4. **a)** The Jacobian's entries are just the x_j repeated, and Chapter 6 shows that only products with it, vector-Jacobian products, are needed.

Worked examples

Two steps of gradient descent Minimize $f(x) = x^2$ by gradient descent from $x_0 = 3$ with a constant step size $\eta = 0.1$. Compute two steps and the value of f at each point.

The gradient of x^2 is $2x$, and the update is $x_t = x_{t-1} - \eta \cdot 2x_{t-1}$.

Step 1: the gradient at 3 is 6, so $x_1 = 3 - 0.1 \times 6 = 2.4$.

Step 2: the gradient at 2.4 is 4.8, so $x_2 = 2.4 - 0.1 \times 4.8 = 1.92$.

The function values fall from $f(3) = 9$ to $f(2.4) = 5.76$ to $f(1.92) = 3.6864$, so each step is a descent.

Answer: $x_1 = 2.4$, $x_2 = 1.92$; f goes $9 \rightarrow 5.76 \rightarrow 3.6864$.

Practice

1. At some point the gradient of f is $(2, -1)$. Is $p = (1, 1)$ a descent direction? Is $p = (-1, 0)$?

Hint: Compute the directional derivative $\langle \nabla f, p \rangle$ and check its sign.

Answer: For $(1, 1)$: $2 \times 1 + (-1) \times 1 = 1 > 0$, so it is not a descent direction. For $(-1, 0)$: $2 \times (-1) + 0 = -2 < 0$, so it is a descent direction.

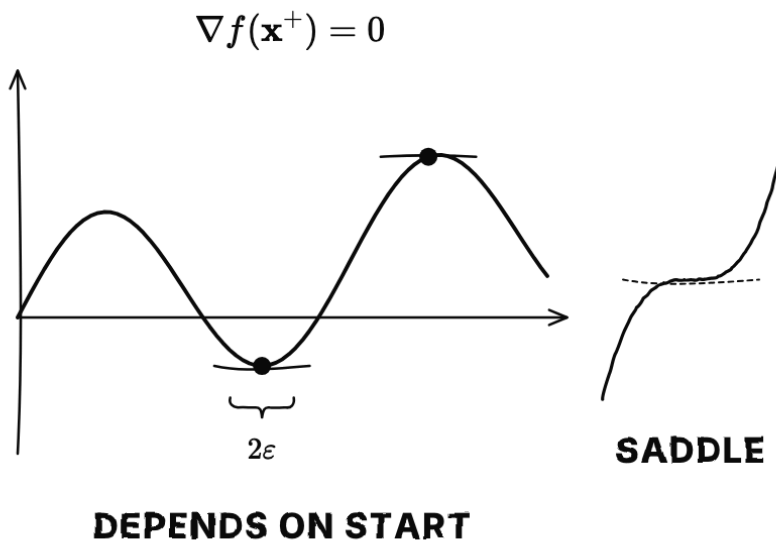
2. For $y = Wx$ with W of shape $(3, 4)$, give the shape of the Jacobian with respect to x and with respect to W , and with respect to the flattened W .

Hint: With respect to x it is W itself.

Answer: With respect to x : $(3, 4)$, equal to W . With respect to W : $(3, 3, 4)$. With respect to $\text{vect}(W)$, a vector of 12 entries: $(3, 12)$.

Convergence of gradient descent

When discussing the convergence of gradient descent, we need to clarify what we mean by “a minimizer” of a function. If you do not care about convergence and you trust gradient descent, proceed with no hesitation to the next section.



Minima, Maxima and Saddles

Definition, minimum: a local minimum of $f(\mathbf{x})$ is a point $\mathbf{x}^+$ such that, for some $\varepsilon > 0$, $f(\mathbf{x}^+) \leq f(\mathbf{x})$ for all $\mathbf{x}$ with $\|\mathbf{x} - \mathbf{x}^+\| < \varepsilon$, that is, for every point in a ball of size ε centered in $\mathbf{x}^+$.

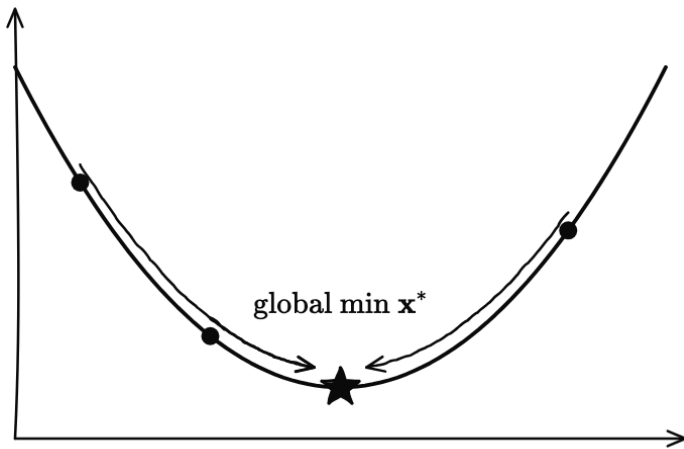
In words, the value $f(\mathbf{x}^+)$ is a minimum if we consider a sufficiently small neighborhood of $\mathbf{x}^+$. Intuitively, at such a point the slope of the tangent will be 0, and the gradient everywhere else in the neighborhood of $\mathbf{x}^+$ will point upwards. We can formalize the first idea by the concept of stationary points.

Definition, stationary points: a point $\mathbf{x}^+$ is called a stationary point of $f(\mathbf{x})$ if $\nabla f(\mathbf{x}^+) = \mathbf{0}$.

Stationary points are not limited to minima: they can be maxima, which are the minima of $-f(\mathbf{x})$, or saddle points, which are inflection points where the curvature of the function is changing, as in the example of Figure F.2.3. In general, without any constraint on f , gradient descent can only be proven to converge to a generic stationary point, depending on its initialization.

The figure is a simple example of a saddle point; try visualizing the tangent line at that point to see that it is indeed stationary.

Can we do better? Picture a parabola: in this case, the function does not have any saddle points, and it only has a single minimum. This minimum is also special, in the sense that at that point the function attains its lowest possible value across the entire domain: we say this is a global minimum.

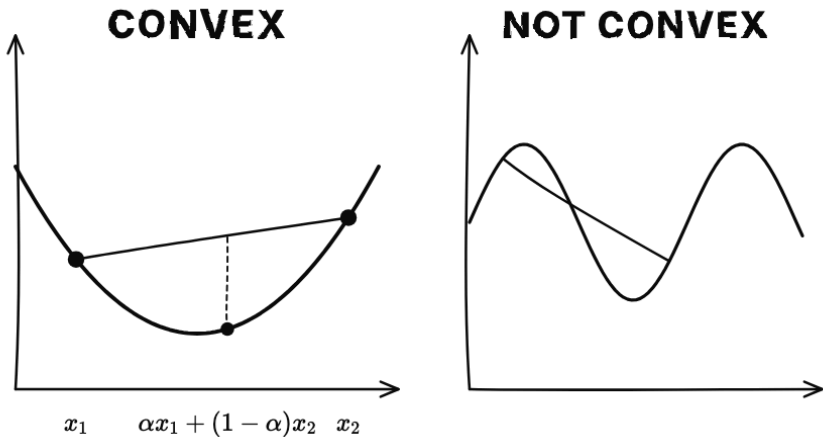


A Parabola Has One Global Minimum

Definition, global minimum: a global minimum of $f(\mathbf{x})$ is a point $\mathbf{x}^*$ such that $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for any possible input $\mathbf{x}$.

Intuitively, gradient descent will converge to this global minimum if run on a parabola, from any possible initialization, because all gradients will point towards it. We can generalize this idea with the concept of convexity of a function. There are many possible definitions of convexity; we choose the one below for simplicity of exposition.

$$f(\alpha x_1 + (1 - \alpha)x_2) \leq \alpha f(x_1) + (1 - \alpha)f(x_2)$$



ANY START → GLOBAL MIN

Convex: Below Every Chord

Definition, convex function: a function $f(\mathbf{x})$ is convex if, for any two points $\mathbf{x}_1$ and $\mathbf{x}_2$ and any $\alpha \in [0, 1]$, we have $f(\alpha \mathbf{x}_1 + (1 - \alpha)\mathbf{x}_2) \leq \alpha f(\mathbf{x}_1) + (1 - \alpha)f(\mathbf{x}_2)$.

The left-hand side of equation E.2.28 is the value of f at any point inside the interval ranging from $\mathbf{x}_1$ to $\mathbf{x}_2$, while the right-hand side is the corresponding value on the line segment connecting $f(\mathbf{x}_1)$ and $f(\mathbf{x}_2)$. If the function is always below the line joining any two points, it is convex; as an example, a parabola pointing upwards is convex.

Convexity qualifies the simplicity of optimizing the function, in the following sense. For a generic non-convex function, gradient descent converges to a stationary point, and nothing more can be said unless

we look at higher-order derivatives, the derivatives of the derivatives. For a convex function, gradient descent will converge to a global minimum, irrespective of initialization. If the inequality in equation E.2.28 is satisfied in a strict way, which is strict convexity, the global minimizer will also be unique.

This is a hard property: to find a global minimum in a non-convex problem with gradient descent, the only solution is to run the optimizer infinitely many times from every possible initialization, turning it into an NP-hard task.

This discussion has a strong historical significance. As we will see in Chapter 5, any non-trivial model is non-convex, meaning that its optimization problem may have several stationary points. This is in contrast to alternative algorithms for supervised learning, such as support vector machines, which maintain non-linearity while allowing for convex optimization. Interestingly, complex differentiable models seem to work well even in the face of this restriction, in the sense that their optimization, when started from a reasonable initialization, converges to points with good empirical performance.

Questions

1. Without any constraint on f , what can gradient descent be proven to reach?
 - a. Always a global minimum
 - b. A generic stationary point, which depends on the initialization
 - c. Always a local maximum

- d. Nothing at all
-
- 2. What does convexity guarantee for gradient descent?
 - a. Convergence to a global minimum, whatever the initialization
 - b. Convergence in exactly one step
 - c. Convergence to a saddle point
 - d. That the gradient is never zero
-
- 3. Why is the non-convexity of neural networks historically significant?
 - a. It made neural networks impossible to train
 - b. It contrasts with methods like support vector machines that keep non-linearity with convex optimization, yet networks still reach good points from reasonable starts
 - c. It proves networks always find global minima
 - d. It means networks need no gradients
-
- 4. What is a saddle point?
 - a. A global minimum
 - b. A stationary point where the curvature of the function changes, neither a minimum nor a maximum
 - c. A point where the gradient is very large
 - d. The starting point of gradient descent

Answers

1. **b)** Stationary points include minima, maxima and saddle points; for a generic f gradient descent only reaches one of them, depending on where it starts.

- 2. a)** For a convex function, gradient descent converges to a global minimum irrespective of initialization.
- 3. b)** Any non-trivial model is non-convex, unlike SVMs; still, differentiable models started sensibly converge to points that perform well.
- 4. b)** Saddle points are stationary, the gradient is zero, but they are inflection points where curvature changes.

Worked examples

Checking the convexity inequality Check the convexity inequality for $f(x) = x^2$ with $x_1 = 0$, $x_2 = 2$ and $\alpha = 0.5$.

The point inside the interval is $\alpha x_1 + (1 - \alpha) x_2 = 0.5 \times 0 + 0.5 \times 2 = 1$.

The left-hand side is the function there: $f(1) = 1$.

The right-hand side is the line between the endpoints: $0.5 \times f(0) + 0.5 \times f(2) = 0.5 \times 0 + 0.5 \times 4 = 2$.

Since $1 \leq 2$, the function lies below the segment at this point, as convexity requires.

Answer: $1 \leq 2$: the inequality holds.

Practice

- 1.** Show with $x_1 = -1$, $x_2 = 1$ and $\alpha = 0.5$ that $f(x) = -x^2$ is not convex.

Hint: Compute both sides of the convexity inequality.

Answer: The midpoint is 0, and $f(0) = 0$. The segment gives $0.5 \times f(-1) + 0.5 \times f(1) = 0.5 \times (-1) + 0.5 \times (-1) = -1$. Since 0 is not ≤ -1 , the inequality fails, so $-x^2$ is not convex.

- 2.** The function $f(x) = x^3$ has $f'(0) = 0$. Is $x = 0$ a minimum? What kind of stationary point is it?

Hint: Look at the values of f just left and just right of 0.

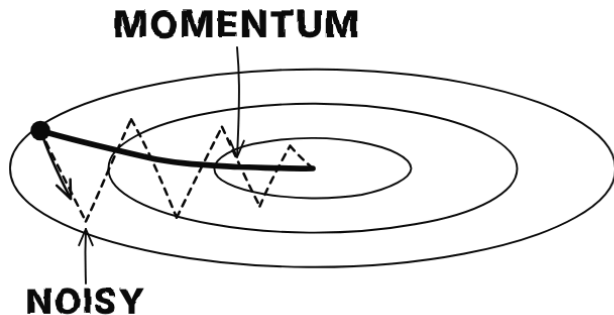
Answer: No. f is negative just left of 0 and positive just right, so 0 is neither a minimum nor a maximum: it is an inflection, saddle-type stationary point where the curvature changes.

Accelerating gradient descent

The negative gradient describes the direction of steepest descent, but only in an infinitesimally small neighborhood of the point. As we will see in Chapter 5, where we introduce stochastic optimization, these directions can be extremely noisy, especially when dealing with large models. A variety of techniques have been developed to accelerate the convergence of the optimization algorithm by selecting better descent directions. For computational reasons, we are especially interested in methods that do not require higher-order derivatives, such as the Hessian, or multiple calls to the function.

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

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

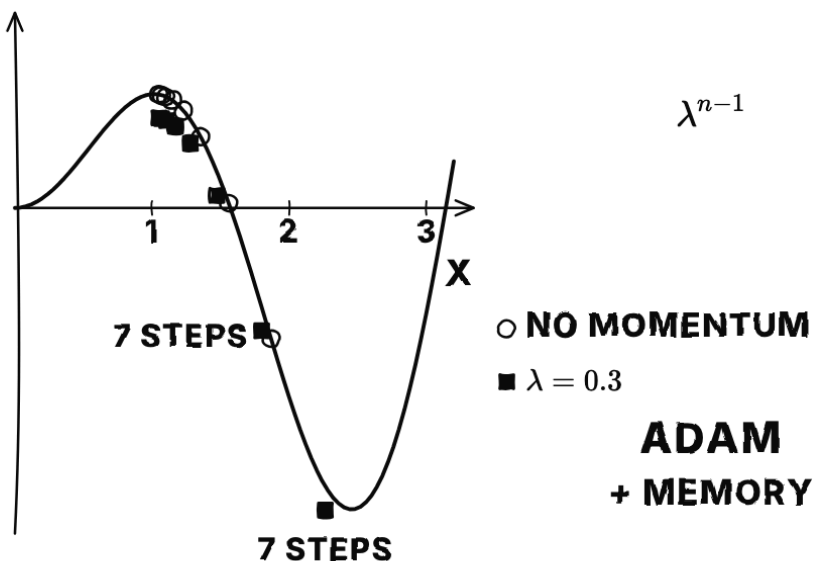


Momentum Smooths the Path

We describe here one such technique, momentum, and refer to other sources for a broader introduction, such as a 2016 blog post by Sebastian Ruder. If you picture gradient descent as a ball “rolling down a hill”, the movement is relatively erratic, because each gradient can point in a completely different direction; in fact, for a perfect choice of step size and a convex loss function, any two gradients in subsequent iterations will be orthogonal. We can smooth this behavior by introducing a “momentum” term that conserves some direction from the previous gradient iteration: $\mathbf{g}_t = -\eta_t \nabla f(\mathbf{x}_{t-1}) + \lambda \mathbf{g}_{t-1}$, the steepest descent plus the momentum term, and then $\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{g}_t$.

Here we initialize $\mathbf{g}_0 = \mathbf{0}$; Figure F.2.4 shows an example. The coefficient λ determines how much the previous term is dampened. In fact, unrolling two terms gives $\mathbf{g}_t = -\eta_t \nabla f(\mathbf{x}_{t-1}) - \lambda \eta_t \nabla f(\mathbf{x}_{t-2}) + \lambda^2 \mathbf{g}_{t-2}$.

The figure compares gradient descent with and without momentum when minimizing the function $x \sin(2x)$, starting from $x = 1 + \varepsilon$, with $\lambda = 0.3$.



$x \sin 2x$, With and Without Momentum

Generalizing, the iteration at time $t - n$ gets dampened by a factor λ^{n-1} . Momentum can be shown to accelerate training by smoothing the optimization path. Another common technique is adapting the step size for each parameter based on the magnitude of the gradients. A common optimization algorithm combining several of these ideas is Adam. One advantage of Adam is that it is found to be relatively robust to the choice of its hyper-parameters, a hyper-parameter being a parameter selected by the user, as opposed to being learnt by gradient descent, with the default choice in most frameworks being a good starting point in the majority of cases. Designing novel optimizers that can “unseat” Adam, or its variations such as AdamW, from its place as the de facto default optimizer in deep learning

remains an open research problem, with recent work on designing customized optimizers for neural networks from first principles.

One disadvantage of using accelerated optimization algorithms can be increased storage requirements: for example, momentum requires us to store the previous gradient iteration in memory, doubling the space needed by the optimization algorithm, although in most cases the memory required to compute the gradient is the most influential factor, as we will see in Section 6.3.

From theory to practice. A box about the exercises explains that this book does not have classical end-of-chapter exercises, which are covered in many existing textbooks. Instead, I propose a self-learning path to help you explore two frameworks, JAX and PyTorch, as you progress in the book. Solutions to the exercises will be published on the book's website. These sections are full of links to online material, which might have expired or moved by the time you search for them.

Questions

1. What does the momentum term add to plain gradient descent?
 - a. A second derivative of the function
 - b. A fraction λ of the previous step, which smooths the erratic path
 - c. A random perturbation at each step
 - d. A larger step size only
2. Why is plain gradient descent's path described as erratic?

- a. Because each gradient can point in a completely different direction from the previous one
 - b. Because the step size is random
 - c. Because it uses second derivatives
 - d. Because it always overshoots the minimum
3. What is a cost of using momentum?
- a. It cannot be used with convex functions
 - b. It needs the Hessian
 - c. It must store the previous step, doubling the optimizer's memory
 - d. It needs several calls to the function per step
4. Why is Adam a popular default optimizer?
- a. It is relatively robust to the choice of its hyper-parameters, and the defaults usually work
 - b. It finds the global minimum of any function
 - c. It needs no gradients
 - d. It uses no memory

Answers

- 1. b)** Momentum keeps some direction from the previous iteration, $\lambda \mathbf{g}_{t-1}$, so the ball rolling down the hill moves less erratically.
- 2. a)** Each gradient may point elsewhere; with a perfect step size on a convex loss, consecutive gradients are even orthogonal.
- 3. c)** The previous gradient iteration must be kept in memory, doubling the space needed by the optimization algorithm itself.

4. a) Adam combines several ideas and is found robust to its hyper-parameters, with framework defaults a good starting point.

Worked examples

Two steps with momentum Minimize $f(x) = x^2$ from $x_0 = 3$ with $\eta = 0.1$, $\lambda = 0.5$ and $g_0 = 0$, using momentum. Compare x_2 with plain gradient descent, which gives 1.92 after two steps.

The gradient of x^2 is $2x$; the updates are $g_t = -\eta \nabla f(x_{t-1}) + \lambda g_{t-1}$ and $x_t = x_{t-1} + g_t$.

Step 1: the gradient at 3 is 6, so $g_1 = -0.6 + 0.5 \times 0 = -0.6$ and $x_1 = 3 - 0.6 = 2.4$.

Step 2: the gradient at 2.4 is 4.8, so $g_2 = -0.48 + 0.5 \times (-0.6) = -0.78$ and $x_2 = 2.4 - 0.78 = 1.62$.

Momentum carried part of the first step forward, so after two steps it is closer to the minimum at 0 than plain descent.

Answer: $x_1 = 2.4$ and $x_2 = 1.62$ with momentum, against 1.92 without.

Practice

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

Hint: Momentum stores one previous step per parameter.

Answer: The parameters take 4 billion bytes, about 4 GB. Momentum stores one more value per parameter, so it adds about another 4 GB, doubling the optimizer's share.

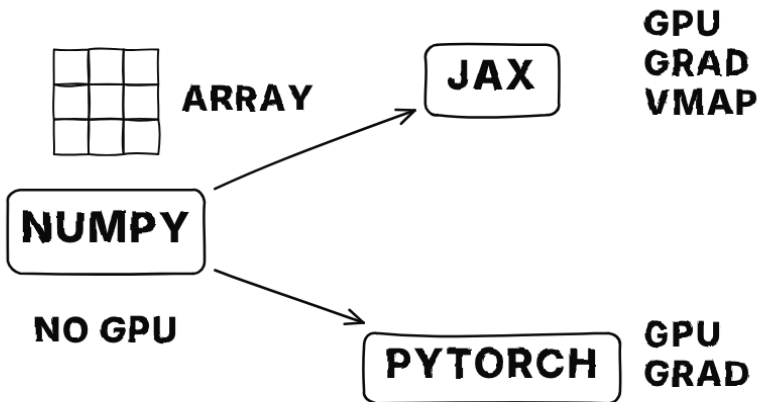
2. With $\lambda = 0.3$, what is λ^3 , and what does the book's dampening rule say it measures?

Hint: The iteration at time $t - n$ is dampened by λ^{n-1} .

Answer: $\lambda^3 = 0.027$. By the rule λ^{n-1} , it is the dampening factor of the iteration at time $t - 4$, which therefore still contributes, but only about 3 percent as strongly.

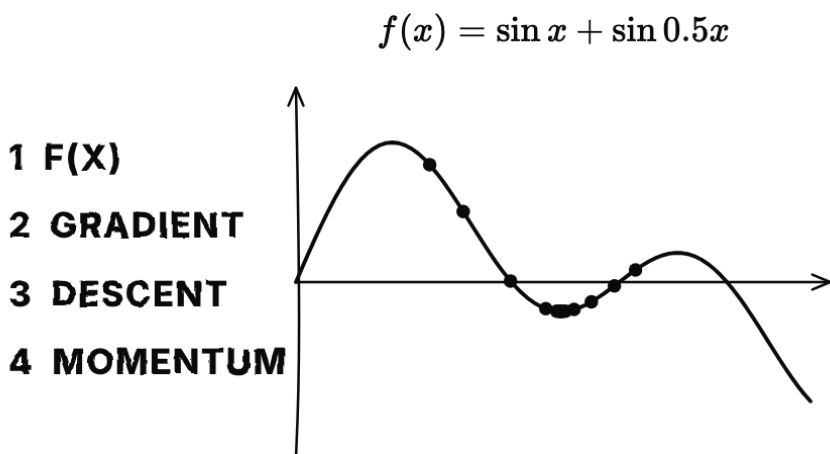
From theory to practice, and what a dataset is

Starting from the basics. The starting block for any designer of differentiable models is a careful study of NumPy. NumPy implements a generic set of functions to manipulate multidimensional arrays, what we call tensors in the book, as well as functions to index and transform their content. You can read more in the library's quick start. You should feel comfortable handling arrays in NumPy, most notably their indexing: the `numpy-100` repository by Nicolas Rougier provides a nice, slow-paced series of exercises to test your knowledge.



Moving to a realistic framework. Despite its influence, NumPy is limited in its support for parallel hardware such as GPUs, unless additional libraries are used, and by its lack of automatic differentiation, introduced in Chapter 6. JAX replicates the NumPy interface while adding extended hardware support, the automatic computation of gradients, and additional transformations such as the vectorized map, `vmap`. Frameworks such as PyTorch also implement a NumPy-like interface at their core, but they make minor adjustments in nomenclature and functionality, and they add high-level utilities for building differentiable models. Take your time to skim the documentation of the array type in JAX and the tensor type in PyTorch, to understand how much they have in common with NumPy. For now, you can ignore high-level modules such as PyTorch's neural network module. We will have more to say about how these frameworks are designed in Chapter 6, after we introduce their gradient computation mechanism.

Implementing a gradient descent algorithm. To become proficient with all three frameworks, NumPy, JAX and PyTorch, I suggest replicating the exercise below three times; each variant should only take a few minutes if you know the syntax. Consider a two-dimensional function $f(\mathbf{x})$, with $\mathbf{x} \sim (2)$, where we take the domain to be $[0, 10]$: $f(\mathbf{x}) = \sin(x_1)\cos(x_2) + \sin(0.5x_1)\cos(0.5x_2)$. I asked ChatGPT to generate a nice function with several minima and maxima; nothing else in the book is generated by a language model, which I feel is becoming an important disclaimer to make.

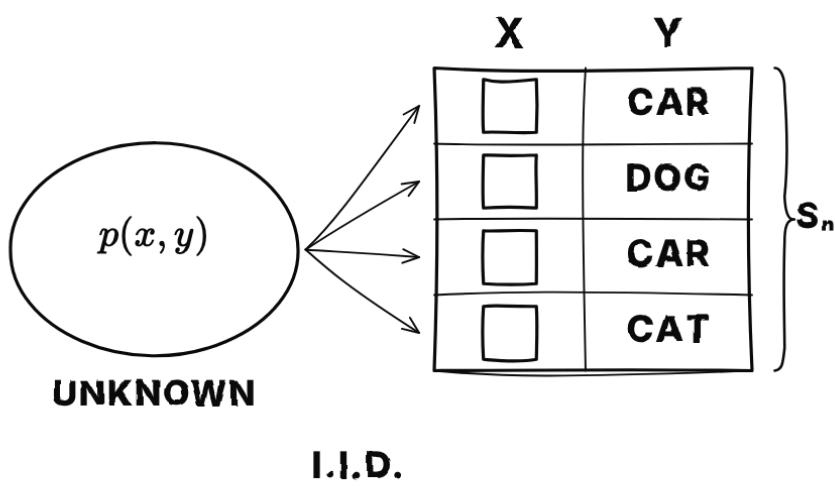


Gradient Descent on a Bumpy Function

Before proceeding in the book, repeat this for each framework. First, implement the function in a vectorized way: given a matrix $\mathbf{X} \sim (n, 2)$ of n inputs, it should return a vector $f(\mathbf{X}) \sim (n)$ where $[f(\mathbf{X})]_i = f(\mathbf{X}_i)$. Second, implement another function to compute its gradient, hard-coded, since we have not touched automatic differentiation yet. Third, write a basic gradient descent procedure and visualize the paths taken by the optimization process from multiple starting points. Fourth, try adding a momentum term and visualizing the norm of the gradients, which should converge to zero as the algorithm moves towards a stationary point.

If you are using JAX or PyTorch to solve the exercise, point three is a good place to experiment with `vmap` for vectorizing a function.

Chapter 3, Datasets and losses. In short, this chapter formalizes the supervised learning scenario. We introduce the concepts of datasets, losses, empirical risk minimization, and the basic assumptions made in supervised learning. We close by providing a probabilistic formulation of supervised learning built on the notion of maximum likelihood. This short chapter serves as the backbone for the rest of the book.



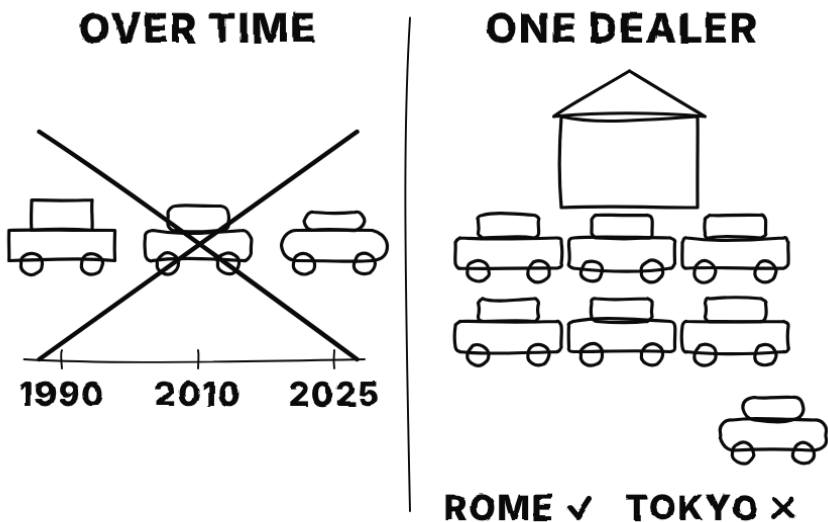
A Dataset of Pairs

Section 3.1, What is a dataset? We consider a scenario in which manually coding a certain function is unfeasible, for example recognizing objects in real-world images, but gathering examples of the desired behaviour is sufficiently easy. Examples of this abound,

ranging from speech recognition to robot navigation. We formalise this idea with the following definition.

Definition, dataset: a supervised dataset $\mathcal{S}_n$ of size n is a set of n pairs, $\mathcal{S}_n = (x_1, y_1), \dots, (x_n, y_n)$, where each (x_i, y_i) is an example of an input-output relationship we want to model. We further assume that each example is an independent and identically distributed, i.i.d., draw from some unknown, and unknowable, probability distribution $p(x, y)$.

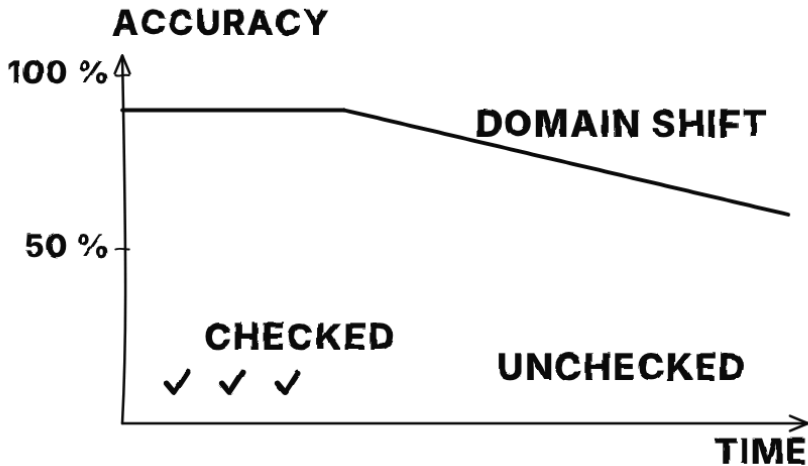
See the appendix if, upon reading the definition, you want to brush up on probability theory. The last assumption appears technical, but it is there to ensure that the relationship we are trying to model is meaningful. In particular, samples being identically distributed means that we are trying to approximate something which is sufficiently stable and unchanging through time. As a representative example, consider the task of gathering a dataset to recognise car models from photos. This assumption will be satisfied if we collect images over a short time span, but it will be invalid if we collect images from the last few decades, since car models will have changed over time. In the latter case, training and deploying a model on this dataset will fail, as it will be unable to recognise new models or will have sub-optimal performance when used.



When i.i.d. Breaks

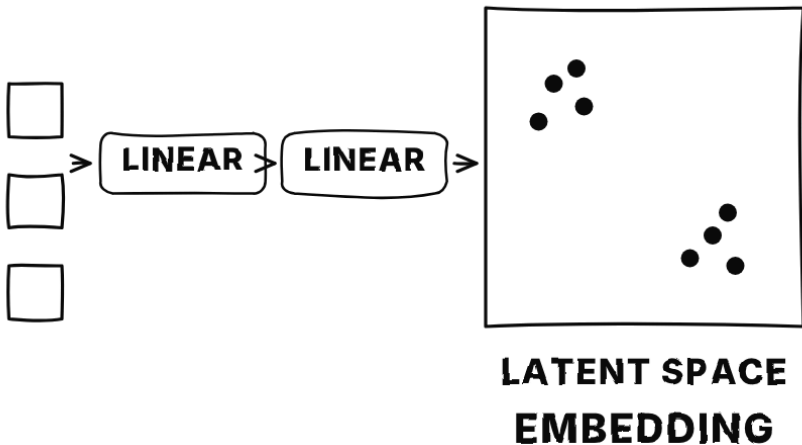
Similarly, samples being independently distributed means that our dataset has no bias in its collection, and that it is sufficiently representative of the entire distribution. Going back to the previous example, gathering images close to a Tesla dealership will be invalid, since we will collect an overabundance of images of a certain type while losing images of other makes and models. Note that the validity of these assumptions depends on the context: a car dataset collected in Italy may be valid when deploying our model in Rome or Milan, while it may be invalid when deploying our model in Tokyo or in Taiwan. The i.i.d. assumption should always be checked carefully, to ensure we are applying our supervised learning tools to a valid scenario. Interestingly, modern LLMs are trained on such large distributions of data that even understanding which tasks are truly in-

distribution and which are out-of-distribution, and how much the models are able to generalize, becomes blurred.



Domain Shift Over Time

A box adds more on the i.i.d. property. Importantly, ensuring the i.i.d. property is not a one-shot process, and it must be checked constantly during the lifetime of a model. In the case of car classification, if unchecked, subtle changes in the distribution of cars over time will degrade the performance of a machine learning model, an example of domain shift. As another example, a recommender system will change the way users interact with a certain app, as they will start reacting to the suggestions of the recommender system itself. This creates feedback loops that require constant re-evaluation of the performance of the system and of the app.



Loops and Embeddings

A figure shows that differentiable models process data by transforming it sequentially via linear algebra operations. In many cases, after we optimize these programs, the internal representations of the input data in the model, what we call a pre-trained model, have geometric properties: for example, semantically similar images are projected to points that are close in this “latent” space. Transforming data from a non-metric space, the original input images, to a metric space is called embedding the data.

Questions

1. What does JAX add on top of the NumPy interface?
 - a. Nothing; it is a renamed copy
 - b. Extended hardware support, automatic gradients, and transformations such as vmap
 - c. Only a neural network module
 - d. A different indexing convention that starts at one
2. Why would a car-photo dataset gathered over several decades break the i.i.d. assumption?
 - a. Because old photos are lower resolution
 - b. Because car models change over time, so the samples are not identically distributed
 - c. Because there would be too many photos
 - d. Because the photos would not be independent of each other's lighting
3. Why is collecting photos near a Tesla dealership a problem?
 - a. It biases the collection toward one type of car, so the samples are not representative: the independence part fails
 - b. Teslas are hard to recognise
 - c. It breaks the identically distributed part only
 - d. It makes the dataset too small
4. Why must the i.i.d. property be checked during a model's whole lifetime?
 - a. Because models forget their training
 - b. Because distributions drift, for example through domain shift or feedback loops like a recommender changing user behaviour

- c. Because frameworks change their defaults
- d. Because datasets are deleted after training

Answers

1. **b)** JAX replicates NumPy's interface while adding hardware support, automatic computation of gradients and transformations like the vectorized map.
2. **b)** Identically distributed means the thing being learnt is stable over time; car models change across decades.
3. **a)** Independently distributed means no bias in collection; a single dealership gives an overabundance of one make.
4. **b)** Subtle changes in the data over time degrade performance, and a recommender system even changes the behaviour it learns from.

Worked examples

One gradient step on the book's practice function For $f(x) = \sin(x_1)\cos(x_2) + \sin(0.5x_1)\cos(0.5x_2)$, compute the gradient at $(\pi, 0)$, take one gradient-descent step with $\eta = 0.1$, and check that f decreases.

The partial derivatives are $\partial f / \partial x_1 = \cos(x_1)\cos(x_2) + 0.5\cos(0.5x_1)\cos(0.5x_2)$ and $\partial f / \partial x_2 = -\sin(x_1)\sin(x_2) - 0.5\sin(0.5x_1)\sin(0.5x_2)$.

At $(\pi, 0)$: $\partial f / \partial x_1 = (-1)(1) + 0.5 \times \cos(\pi/2) \times 1 = -1$, and $\partial f / \partial x_2 = 0$, because $\sin(0) = 0$; so the gradient is $(-1, 0)$.

The step is $x - \eta \nabla f = (\pi, 0) - 0.1 \times (-1, 0) = (\pi + 0.1, 0)$, about $(3.2416, 0)$.

Check: $f(\pi, 0) = 0 + \sin(\pi/2) = 1$, while $f(\pi + 0.1, 0) = \sin(\pi + 0.1) + \sin(\pi/2 + 0.05) \approx -0.0998 + 0.9988 = 0.8990$, which is lower.

Answer: Gradient $(-1, 0)$; new point about $(3.2416, 0)$; f falls from 1 to about 0.899.

Practice

1. Your vectorized f receives a matrix X of shape $(5, 2)$. What shape must the output have, and what is its third element?

Hint: The book asks for $[f(X)]_i = f(X_i)$.

Answer: A vector of shape (5) ; its third element is f applied to the third row of X .

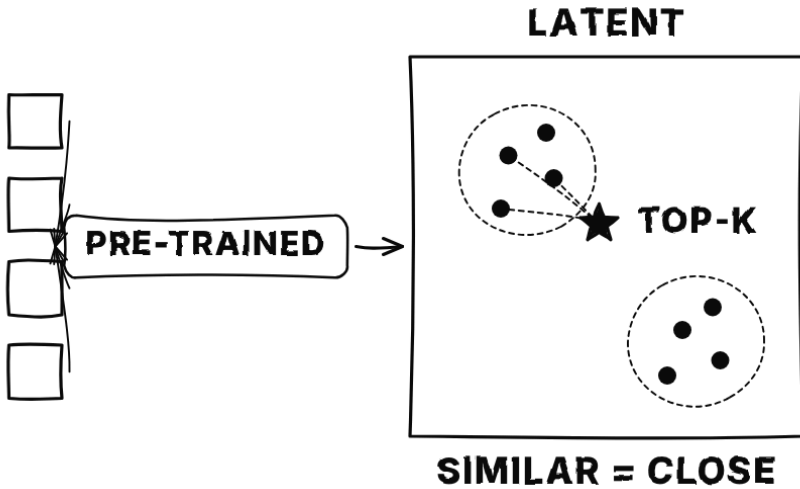
2. A spam filter trained on 2015 emails is deployed today. Name which part of the i.i.d. assumption is most at risk and why.

Hint: Ask whether what is being learnt has stayed the same over time.

Answer: The identically distributed part: spam styles and topics drift over the years, so today's emails come from a different distribution than the training data, a case of domain shift.

Variants of supervised learning

There exist many variations on the standard supervised learning scenario, although most successful applications make use of supervised learning in some form or another. For example, some datasets may not have targets y_i available, in which case we talk about unsupervised learning. Typical applications of unsupervised learning are clustering algorithms, in which we want to aggregate our input data into clusters such that points in a cluster are similar and points in different clusters are dissimilar. As another example, in a retrieval system we may want to search a large database for the top- k elements most similar to a query given by a user.



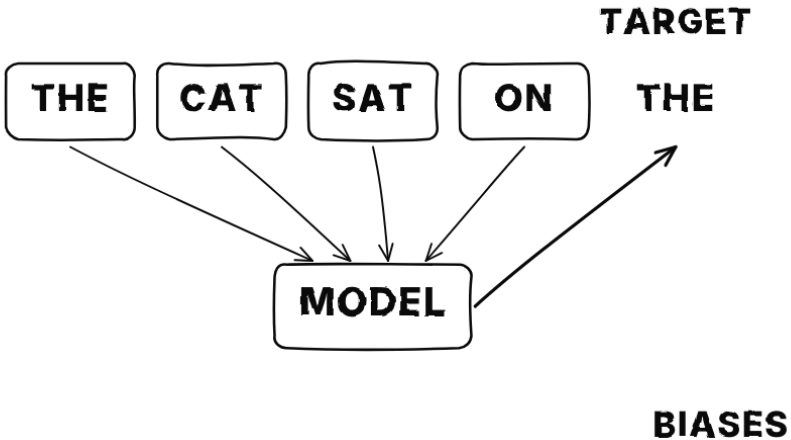
Clusters in a Latent Space

When dealing with complex data such as images, this is non-trivial, because distances between images are ill-defined if we operate on pixels: even small perturbations can modify millions of pixels. However, assume we have available some differentiable model that we have already optimized for some other task which we assume sufficiently generic, for example image classification. We call it a pre-trained model. As we will see, the internal states of this model can be interpreted as vectors in a high-dimensional space. In many cases, these vectors are shown to have useful geometrical properties, in the sense that objects that are semantically similar are sent, embedded, into points that are close in these representations. Hence, we can use these latent representations with standard clustering models, such as

Gaussian mixture models. Figure F.3.1 gives a high-level overview of this idea.

What if we do not have access to a pre-trained model? A common variation of unsupervised learning is called self-supervised learning, SSL. The aim of self-supervised learning is to automatically find some supervised objective in a generic unsupervised dataset, in order to optimize a model that can be used in a large set of downstream tasks. For example, if we have access to a large corpus of text, we can always optimize a program to predict how a small piece of text is likely to continue. The realization that neural networks can also perform an efficient embedding of text when pre-trained in a self-supervised way had a profound impact on the community. Large-scale web datasets are also full of biases, profanity, and vulgar content; recognizing that models trained on this data internalize these biases was another important realization, and it is one of the major criticisms of closed-source foundation models.

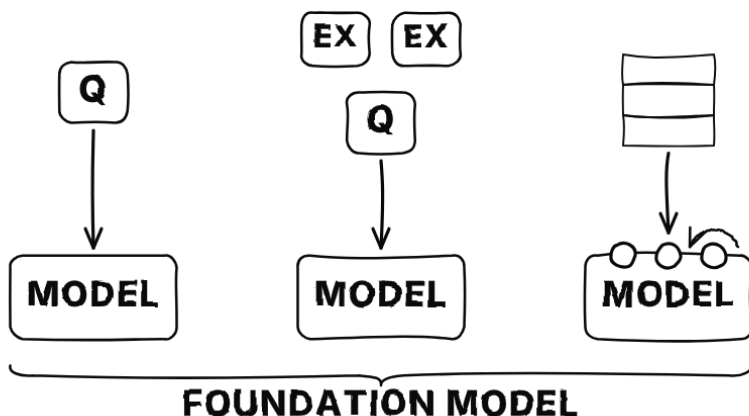
SELF-SUPERVISED



Learning from the Text Itself

Figure F.3.2 shows three ways of using trained models. Zero-shot: a question is directly given to the model, which can be achieved with generative language models, introduced in Chapter 8. Few-shot prompting is similar, but a few examples are provided as input. Both techniques can be employed only if the underlying model shows a large amount of generalization capability. Fine-tuning: the model is optimized via gradient descent on a small dataset of examples, which proceeds similarly to training the model from scratch.

ZERO-SHOT FEW-SHOT FINE-TUNE

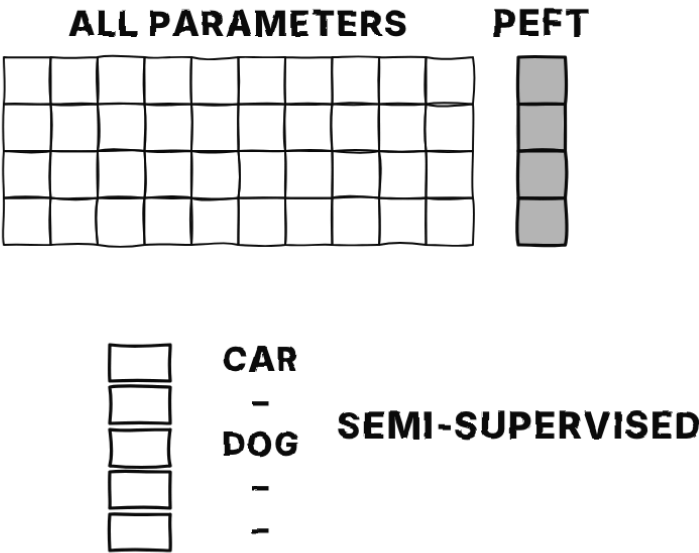


Zero-shot, Few-shot, Fine-tuning

As we will see in Chapters 8 and 10, LLMs can be seen as modern iterations on this basic idea, since optimizing models such as GPT or Llama always starts with basic self-supervised training in terms of next-token prediction. These models are sometimes called foundation models. In the simplest case, they can be used out of the box for a new task, such as answering a query: in this case, we say they are used in a zero-shot fashion. For LLMs, it is also possible to provide a small number of examples of a new task as the input prompt, in which case we talk about few-shot prompting. In the most general case, we can take a pre-trained foundation model and optimize its parameters by gradient descent on a new task: this is called fine-tuning the model. Figure F.3.2 compares the three approaches. In this book we focus on

building models from scratch, but fine-tuning can be done by similar means.

Fine-tuning is made especially easy by the presence of large open-source repositories online. Fine-tuning can be done on the full set of parameters of the starting model, or by considering only a smaller subset or a small number of additional parameters: this is called parameter-efficient fine-tuning, PEFT. Few-shot learning can also be done by fine-tuning the model; in cases in which fine-tuning is not needed, we say the model is performing in-context learning. We will consider parameter-efficient fine-tuning techniques in the next volume.



Tuning Few Parameters

Many other variations of supervised learning are possible, which we do not have space to list in detail here, except for some generic hints. If only parts of a dataset are labeled, we have a semi-supervised scenario; we will see some examples of semi-supervised learning in Chapter 12. Additionally, we can have scenarios with multiple datasets belonging to “similar” distributions, or the same distribution over different periods of time, giving rise to countless problems depending on the order in which the tasks or the data are provided, including domain adaptation, meta-learning, continual learning, metric learning, unlearning, and so on. Some of these will be treated in the next volume.

Questions

1. Why can't we simply cluster images by comparing their pixels?
 - a. Because pixels are not numbers
 - b. Because distances on pixels are ill-defined: small perturbations can change millions of pixels
 - c. Because images have too few pixels
 - d. Because clustering only works on text
2. What makes self-supervised learning possible on an unlabeled text corpus?
 - a. Humans label every sentence
 - b. A supervised objective is found automatically in the data, such as predicting how a piece of text continues
 - c. The model is given the answers in advance

- d. Clustering replaces training
- 3. What distinguishes fine-tuning from few-shot prompting?
 - a. Fine-tuning changes the model's parameters by gradient descent; few-shot prompting only puts examples in the input
 - b. Few-shot prompting needs more data
 - c. Fine-tuning needs no examples
 - d. They are the same thing
- 4. What is a semi-supervised scenario?
 - a. One with no data at all
 - b. One where only parts of the dataset are labeled
 - c. One with two models
 - d. One where labels change over time

Answers

- 1. **b)** Pixel distances do not reflect meaning; the book instead uses the internal vectors of a pre-trained model, where similar objects land close together.
- 2. **b)** SSL finds a supervised objective inside unsupervised data, for example next-token prediction, as LLMs do.
- 3. **a)** Few-shot prompting provides examples in the prompt, in-context; fine-tuning optimizes the parameters on a small dataset.
- 4. **b)** If only part of a dataset has labels, the scenario is semi-supervised.

Practice

- 1. You want to find, among a million holiday photos, the ten most similar to a new one. Using the passage, describe a workable

approach.

Hint: Do not compare pixels; think of a pre-trained model.

Answer: Pass every photo through a pre-trained model, such as an image classifier, and take its internal vector for each photo. Semantically similar photos are embedded close together, so you retrieve the ten photos whose vectors are closest to the new photo's vector, a top-k retrieval in the latent space.

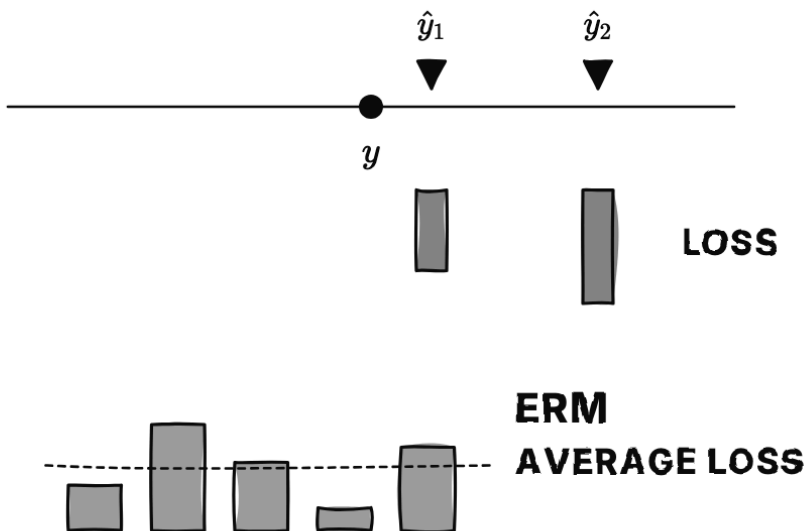
2. Name the three ways of using a trained model in Figure F.3.2 and say which ones change its parameters.

Hint: Two only change the input.

Answer: Zero-shot, where the question is given directly; few-shot prompting, where a few examples are added to the input; and fine-tuning, where the model is optimized by gradient descent on a small dataset. Only fine-tuning changes the parameters.

Loss functions

Once data has been gathered, we need to formalize our idea of “approximating” the desired behavior, which we do by introducing the concept of loss functions.



A Loss Ranks Predictions

Definition, loss function: given a desired target y and the predicted value $\hat{y} = f(x)$ from a model f , a loss function $l(y, \hat{y}) \in \mathbb{R}$ is a scalar, differentiable function whose value correlates with the performance of the model. That is, $l(y, \hat{y}_1) < l(y, \hat{y}_2)$ means that the prediction $\hat{y}_1$ is

better than the prediction $\hat{y}_2$ when considering the reference value, the target y .

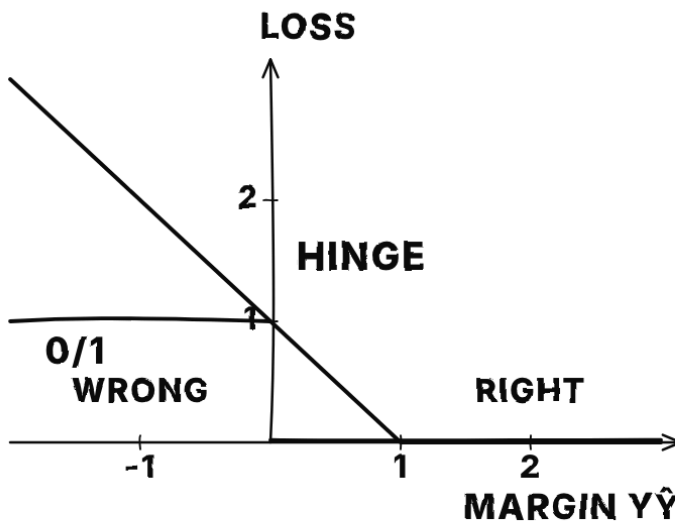
A loss function embeds our understanding of the task and our preferences in the space of solutions on a real-valued scale that can be exploited in an optimization algorithm. Being differentiable, it allows us to turn our learning problem into a mathematical optimization problem that can be solved via gradient descent by minimizing the average loss on our dataset.

To this end, given a dataset $\mathcal{S}_n$ of pairs (x_i, y_i) and a loss function l , a sensible optimization task to solve is the minimum average loss on the dataset achievable by any possible differentiable model f : $f^* = \operatorname{argmin}_f (1/n) \sum_{i=1}^n l(y_i, f(x_i))$, the average over the dataset of the loss of the prediction on the i -th sample.

For historical reasons, equation E.3.1 is referred to as empirical risk minimization, ERM, where risk is used as a generic synonym for loss. The box that follows says more on the origin of the term.

In equation E.3.1 we are implicitly assuming that we are minimizing across the space of all possible functions defined on our input x . We will see shortly that our models can always be parameterized by a set of tensors w , called the parameters of the model, and minimization is done by searching for the optimal value of these parameters via numerical optimization, which we denote by $f(x, w)$. Hence, given a dataset $\mathcal{S}_n$, a loss function l , and a model space f , we can train our model by optimizing the empirical risk of equation E.3.1 via gradient descent, equation E.2.27: $w^* = \operatorname{argmin}_w (1/n) \sum_{i=1}^n l(y_i, f(x_i, w))$, where the minimization is now done with respect to the parameter tensor w .

On the differentiability of the loss. Before proceeding, we make two observations on the ERM framework. First, note that the differentiability requirement on l is fundamental. Consider a simple binary classification task, which we will introduce properly in the next chapter, where y can only take two values, -1 or $+1$. Given a real-valued model $f(x) \in \mathbb{R}$, we can equate the two decisions with the sign of f , which we denote as $\text{sign}(f(x))$, and define a 0/1 loss as follows: $l(y, \hat{y})$ is 0 if $\text{sign}(\hat{y}) = y$, and 1 otherwise.



0/1 Loss versus Hinge Loss

While this aligns with our intuitive notion of “being right”, it is useless as a loss function, since its gradient will almost always be zero, except where the sign of f switches, and any gradient descent algorithm will remain stuck at initialization. A less intuitive quantity in this case is

the margin $y\hat{y}$, which is positive or negative depending on whether the sign of the model aligns, or does not align, with the desired one, but which varies continuously, unlike the 0/1 loss of equation E.3.3. A possible loss function in this case is the hinge loss $l(y, \hat{y}) = \max(0, 1 - y\hat{y})$, which is used to train support-vector models. Details apart, this shows the inherent tension between designing loss functions that encode our notion of performance while at the same time being useful for numerical optimization.

A box on risk and loss: empirical and expected risk minimization framed in this way are generally associated with the work of the Russian computer scientist Vladimir Vapnik, which gave rise to the field of statistical learning theory, SLT. Statistical learning theory is especially concerned with the behaviour of equation E.3.1 when seen as a finite-sample approximation of equation E.3.5, under some restricted class of functions f and some measure of underlying complexity. The counter-intuitive properties of modern neural networks, such as strong generalization long after overfitting should have been expected, have opened many new avenues of research in statistical learning theory; see also the introduction of Chapter 9.

Questions

1. Why is the 0/1 loss useless for gradient descent?
 - a. Because it is too expensive to compute
 - b. Because its gradient is almost always zero, so gradient descent stays stuck at initialization
 - c. Because it can be negative

- d. Because it only works for regression
2. What does empirical risk minimization minimize?
 - a. The largest loss on the dataset
 - b. The average loss of the model's predictions over the dataset
 - c. The number of parameters
 - d. The loss on future, unseen data directly
 3. What is the margin $y\hat{y}$ in binary classification?
 - a. A quantity that is positive when the model's sign agrees with the target and negative otherwise, varying continuously
 - b. The 0/1 loss multiplied by two
 - c. The distance between two classes in pixels
 - d. Always a positive number

Answers

1. **b)** The 0/1 loss is flat except where the sign of f switches, so its gradient gives no direction to move.
2. **b)** ERM searches for the model, in practice the parameters w , that minimizes the average of $l(y_i, f(x_i))$ over the n examples.
3. **a)** The margin is positive or negative depending on whether the signs agree, and unlike the 0/1 loss it changes continuously.

Worked examples

Hinge loss versus 0/1 loss Compute the hinge loss $\max(0, 1 - y\hat{y})$ and the 0/1 loss for three predictions: $y = +1$ with $\hat{y} = 0.3$; $y = +1$ with $\hat{y} = 2$; $y = -1$ with $\hat{y} = 0.5$. Then compute the average hinge loss.

First case: the margin is $1 \times 0.3 = 0.3$, so the hinge loss is $\max(0, 0.7) = 0.7$; the sign is right, so the 0/1 loss is 0.

Second case: the margin is 2, so the hinge loss is $\max(0, -1) = 0$; the 0/1 loss is 0.

Third case: the margin is $-1 \times 0.5 = -0.5$, so the hinge loss is $\max(0, 1.5) = 1.5$; the sign is wrong, so the 0/1 loss is 1.

The empirical risk under the hinge loss is the average: $(0.7 + 0 + 1.5) / 3 = 2.2 / 3 \approx 0.733$.

Answer: Hinge losses 0.7, 0 and 1.5 (average ≈ 0.733); 0/1 losses 0, 0 and 1.

Practice

1. In the first case above, the prediction moves from $\hat{y} = 0.3$ to $\hat{y} = 0.5$. How do the hinge loss and the 0/1 loss change, and what does that say about their gradients?

Hint: Recompute both losses.

Answer: The hinge loss falls from 0.7 to 0.5, while the 0/1 loss stays 0. The hinge loss reacts to the change, so it has a useful gradient; the 0/1 loss is flat there, with zero gradient.

2. Explain in your own words why risk is used as a synonym for loss in empirical risk minimization, and what is empirical about it.

Hint: Think of what the average is taken over.

Answer: Risk is a historical name for expected loss. It is empirical because the average is taken over the finite dataset we actually have, a sample of examples, rather than over the whole unknown distribution of data.

Expected risk, and choosing a loss

As a second observation, note that the empirical risk is always trivial to minimize, by defining $f(x)$ as follows: it returns y if the pair (x, y) is in the training set $\mathcal{S}_n$, and a default value $\bar{y}$, for example 0, otherwise.

MEMORISED

X	Y
x_1	y_1
x_2	y_2
x_3	y_3

LOSS 0

$x_{new} \rightarrow$ **0** **LOSS HIGH**

LEARNED?

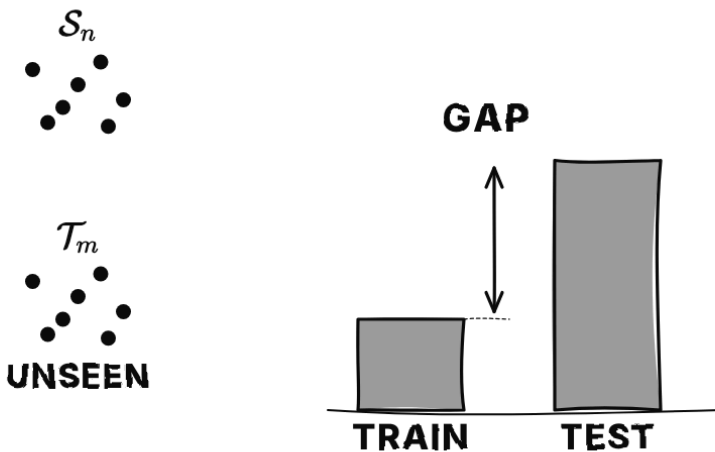
A Look-up Table Cheats

This is a look-up table that returns a prediction y if the pair (x, y) is contained in the dataset, while it defaults to some constant prediction $\bar{y}$, for example 0, otherwise. Assuming that the loss attains its lower bound whenever $y = \hat{y}$, this model will always achieve the lowest

possible value of empirical risk, while providing no actual practical value.

This shows the difference between memorization and learning, or optimization. Although we search for a model by optimizing some average loss quantity on our training data, as in equation E.3.1, our true objective is minimizing this quantity on some unknown, future input yet to be seen. The elements of our training set are only a proxy to this end. We can formalize this idea by defining the expected risk minimization problem.

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



The Generalization Gap

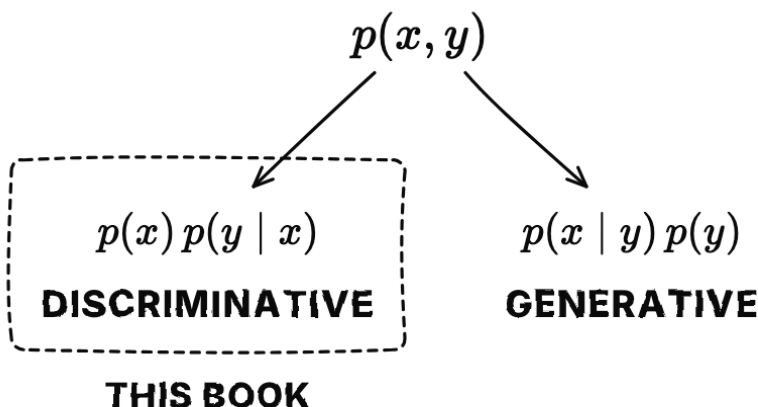
Definition, expected risk: given a probability distribution $p(x, y)$ and a loss function l , the expected risk, ER , is defined as $ER[f] = \mathbb{E}_{p(x,y)}$

$[\ell(y, f(x))]$.

Minimizing equation E.3.5 can be interpreted as minimizing the average, or expected, loss across all possible input-output pairs, for example all possible emails, that our model could see. Clearly, a model with low expected risk would be guaranteed to work correctly. However, the quantity in equation E.3.5 is unfeasible to compute in practice, as enumerating and labeling all data points is impossible. The empirical risk provides an estimate of the expected risk under the choice of a given dataset, and can be seen as a Monte Carlo approximation of the expected risk.

The difference in loss between the expected and the empirical risk is called the generalization gap: a pure memorization algorithm like that of equation E.3.4 will generalize poorly or, in other terms, it will overfit to the specific training data we provided. Generalization can be tested in practice by keeping a separate test dataset $\mathcal{T}_m$ with m data points never used during training, so that $\mathcal{S}_n \cap \mathcal{T}_m = \emptyset$. Then the difference in empirical loss between $\mathcal{S}_n$ and $\mathcal{T}_m$ can be used as an approximate measure of overfitting.

How to select a valid loss function? If you have not done so already, this is a good time to study, or skim, the material in the appendix, especially probability distributions, sufficient statistics, and maximum likelihood estimation.



Two Ways to Split $p(x, y)$

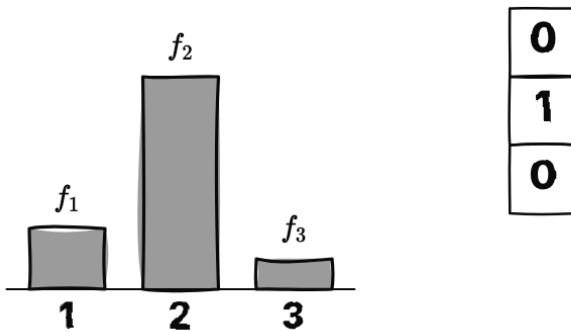
As we will see in the next chapters, the loss encodes our a priori knowledge of the task to be solved, and it has a large impact on performance. In some cases, simple considerations about the problem are enough to design valid losses, as was done for the hinge loss in the previous section.

However, it is possible to work in a more principled fashion by reformulating the entire training process in purely probabilistic terms, as we show now. This formulation provides an alternative viewpoint on learning, which may be more intuitive or more useful in certain scenarios. It is also the preferred viewpoint of many books. We provide the basic ideas in this section, and we consider specific applications later on in the book.

The key observation is the following. In Section 3.1, we started by assuming that our examples come from a distribution $p(x, y)$. By the product rule of probability, we can decompose $p(x, y)$ as $p(x, y) = p(x) p(y | x)$, such that $p(x)$ gives the probability of observing each input x , and the conditional term $p(y | x)$ describes the probability of observing a certain output y given an input x . We can also decompose it as $p(x, y) = p(x | y) p(y)$; methods that need to estimate $p(x)$ or $p(x | y)$ are called generative, while methods that estimate $p(y | x)$ are called discriminative. Apart from language modeling, in this book we focus on the latter case, and we consider generative modeling more broadly in the next volume. Approximating $p(y | x)$ with a function $f(x)$ makes sense if we assume that the probability mass is mostly concentrated around a single point y , that is, $p(y | x)$ is close to a so-called Dirac delta function, and it drastically simplifies the overall problem formulation.

SUM = 1

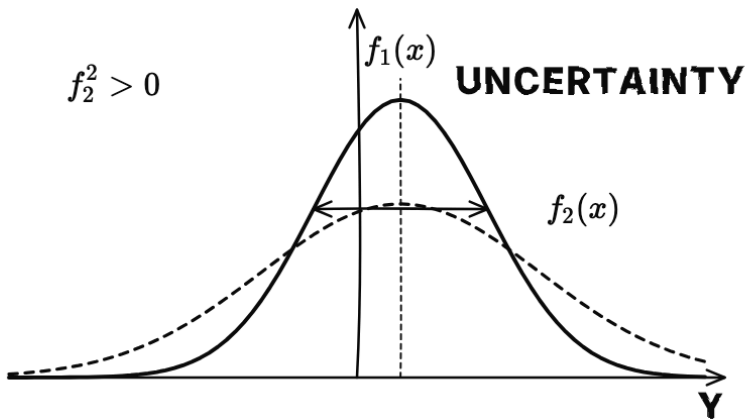
ONE-HOT Y



A Model Outputs a Distribution

However, we can relax this by assuming that our model $f(x)$ does not provide the prediction directly, but is used instead to parameterize the sufficient statistics of a conditional probability distribution $p(y \mid f(x))$ over possible outputs. For example, consider a classification problem where y can take three possible values, 1, 2 or 3. We can assume our model has three outputs that parameterize a categorical distribution over these classes, such that $p(y \mid f(x)) = \prod_{i=1}^3 f_i(x)^{y_i}$. Here $\mathbf{y} \sim \text{Binary}(3)$ is the one-hot encoding of the class y , a vector of all zeros except for the element of the class, which is 1, as introduced formally in Section 4.2, and $f(x) \sim \Delta(3)$ are the predicted probabilities for each class. As another example, assume we want to predict a single scalar value $y \in \mathbb{R}$, which is regression. We can model this with a two-valued

function $f(x) \sim (2)$ such that the prediction is a Gaussian with an appropriate mean and variance: $p(y | f(x)) = \mathcal{N}(y | f_1(x), f_2^2(x))$.



Regression as a Bell Curve

Here the second output of $f(x)$ is squared to ensure that the predicted variance remains positive. As can be seen, this is a very general setup that subsumes our previous discussion, and it provides more flexibility to the designer, as choosing a specific parameterization for $p(y | x)$ can be easier than choosing a specific loss function $l(y, \hat{y})$. In addition, this framework provides a more immediate way to model uncertainty, such as the variance in equation E.3.6.

Questions

1. Why does a look-up table that memorizes the training set have no practical value, even though it minimizes the empirical risk?
 - a. Because it is too slow
 - b. Because our true objective is low loss on unseen future inputs, where it just returns a default value
 - c. Because look-up tables cannot be differentiated
 - d. Because it needs a test set
2. How can the generalization gap be estimated in practice?
 - a. By computing the expected risk exactly
 - b. By comparing the empirical loss on the training set with the loss on a separate test set never used in training
 - c. By counting the parameters
 - d. By training twice on the same data
3. In the probabilistic view of classification with three classes, what does the model output?
 - a. A single class number
 - b. Three probabilities, one per class, that parameterize a categorical distribution
 - c. A loss value
 - d. The one-hot encoding of the input
4. Why is the second output of the regression model squared?
 - a. To make the mean larger

- b. So that the predicted variance of the Gaussian is always positive
- c. To speed up training
- d. To turn regression into classification

Answers

1. **b)** Memorization reaches the lowest empirical risk but says nothing about new inputs; the training set is only a proxy for future data.
2. **b)** The expected risk cannot be computed, so a held-out test set approximates it, and the difference in losses measures overfitting.
3. **b)** $f(x) \sim \Delta(3)$ gives the predicted probabilities of the three classes, the sufficient statistics of a categorical distribution.
4. **b)** A variance must be positive; squaring $f_2(x)$ guarantees it.

Worked examples

Probability of the true class A three-class model predicts $f(x) = (0.2, 0.7, 0.1)$. The true class is 2. Write its one-hot encoding and compute $p(y | f(x))$ with the book's product formula.

The one-hot encoding of class 2 has a 1 in position 2 and zeros elsewhere: $y = (0, 1, 0)$.

The formula multiplies $f_i(x)$ raised to y_i for $i = 1, 2, 3$.

Any number raised to 0 is 1, so only the second factor counts: $0.2^0 \times 0.7^1 \times 0.1^0 = 1 \times 0.7 \times 1$.

The result is simply the probability the model gives to the true class.

Answer: $p(y | f(x)) = 0.7$

Practice

1. A regression model outputs $f(x) = (3.0, -0.5)$. What Gaussian does it predict for y ?

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

Answer: A normal distribution with mean 3.0 and variance $(-0.5)^2 = 0.25$, so a standard deviation of 0.5.

2. A model has an average loss of 0.1 on its training set and 0.6 on a held-out test set. What is the approximate generalization gap and what does it suggest?

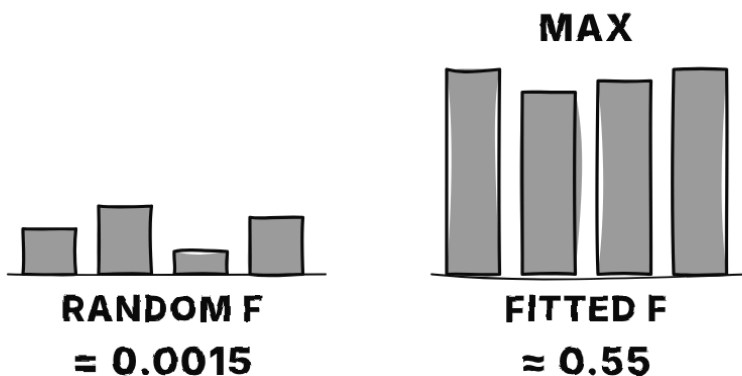
Hint: Subtract the two empirical losses.

Answer: About 0.5. The model does much better on data it has seen than on new data, which suggests it is overfitting, leaning towards memorization.

Maximum likelihood, and Bayesian learning

How can we train a probabilistic model? Remember that we assumed the samples in our dataset $\mathcal{S}_n$ to be i.i.d. samples from a probability distribution $p(x, y)$. Hence, given a model $f(x)$, the probability assigned to the dataset itself by a specific choice of function f is given by the product over the samples in the dataset: $p(\mathcal{S}_n | f) = \prod_{i=1}^n p(y_i | f(x_i))$.

$$p(\mathcal{S}_n | f) = \prod_i p(y_i | f(x_i))$$



Likelihood: Multiply the Probabilities

The quantity $p(\mathcal{S}_n | f)$ is called the likelihood of the dataset. For a random choice of $f(x)$, the model will assign probabilities more or less at random across all possible inputs and outputs, and the likelihood of our specific dataset will be small. A sensible strategy, then, is to select the model such that the likelihood of the dataset is instead maximized. This is a direct application of the maximum likelihood approach, described in the appendix.

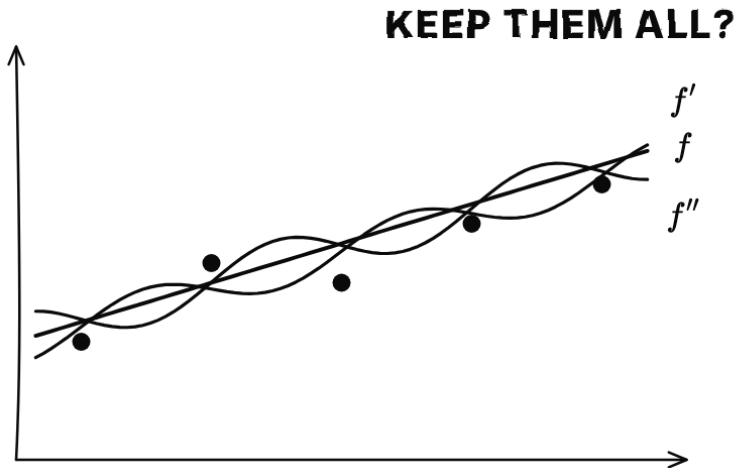
Definition, maximum likelihood: given a dataset $\mathcal{S}_n$ of pairs (x_i, y_i) and a family of probability distributions $p(y | f(x))$ parameterized by $f(x)$, the maximum likelihood solution is given by $f^* = \operatorname{argmax}_f \prod_{i=1}^n p(y_i | f(x_i))$.

While we are again left with an optimization problem, it now follows directly from the laws of probability once all probability distributions are chosen, in contrast to before, where the specific loss was part of the design space. The two viewpoints, however, are closely connected. Working in log-space and switching to a minimization problem, we obtain that the $\operatorname{arg max}$ of $\log \prod_i p(y_i | f(x_i))$ equals the $\operatorname{arg min}$ of $\sum_i -\log p(y_i | f(x_i))$.

Hence, the two formulations are identical if we identify $-\log p(y | f(x))$ as a “pseudo-loss” to be optimized. As we will see, all loss functions used in practice can be obtained under the maximum likelihood principle for specific choices of this term. Both viewpoints are interesting, and we urge readers to keep them in mind as we progress in the book.

Section 3.3, Bayesian learning. We discuss here a further generalization of the probabilistic formulation, called Bayesian neural networks, BNNs, which is of interest in the literature. We only provide

the general idea, and we refer the reader to one of many in-depth tutorials for more details.

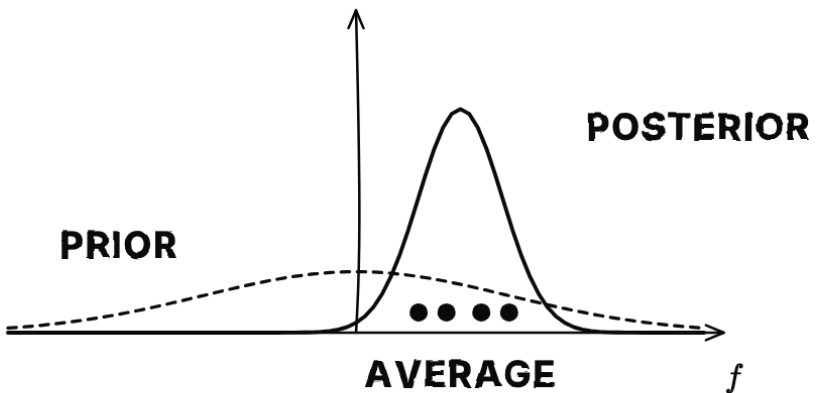


Many Functions Fit the Data

By designing a probability function $p(y \mid f(x))$ instead of $f(x)$ directly, we can handle situations where more than one prediction is of interest, that is, where the probability function has more than a single mode. However, our procedure still returns a single function $f(x)$ out of the space of all possible functions, while it may happen that more than a single parameterization across the entire model space is valid. In this case, it could be useful to have access to all of them for a more faithful prediction.

Once again, we can achieve this objective by designing another probability distribution and then letting the rules of probability guide us. Since we are now planning to obtain a distribution across all possible functions, we start by defining a prior probability distribution $p(f)$ over all possible functions; recall that in practice f is described by a finite set of parameters, in which case the prior $p(f)$ becomes a prior over these weights. For example, we will see that in many situations functions with a smaller norm are preferred, as they are more stable, in which case we could define a prior $p(f) \propto 1/\|f\|$ for some norm $\|f\|$ of f .

$$p(f | \mathcal{S}_n) = \frac{p(\mathcal{S}_n | f) p(f)}{p(\mathcal{S}_n)} \quad = \text{EVIDENCE}$$



From Prior to Posterior

Once a dataset is observed, the probability over f shifts depending on the prior and the likelihood, and the update is given by Bayes'

theorem: $p(f \mid \mathcal{S}_n) = p(\mathcal{S}_n \mid f) p(f) / p(\mathcal{S}_n)$, where $p(f)$ is the prior, before observing the dataset, and $p(f \mid \mathcal{S}_n)$ the posterior, after observing it.

The term $p(f \mid \mathcal{S}_n)$ is called the posterior distribution function, while the term $p(\mathcal{S}_n)$ in the denominator is called the evidence, and it is needed to ensure that the posterior is properly normalized. Assume for now that we have access to the posterior. Unlike before, the distribution can encode a preference for more than a single function f , which may provide better predictive power. Given an input x , we can make a prediction by averaging all possible models based on their posterior weight: $p(y \mid x) = \int p(y \mid f(x)) p(f \mid \mathcal{S}_n) df$, which is approximately $(1/k) \sum_{i=1}^k p(y \mid f_i(x)) p(f_i \mid \mathcal{S}_n)$.

In equation E.3.10 we have approximated the integral with a Monte Carlo average over k random samples from the posterior distribution, $f_k \sim p(f \mid \mathcal{S}_n)$. The overall beauty of this setup is marred by the fact that the posterior is in general impossible to compute in closed form, except for very specific choices of prior and likelihood. Lacking this, one is forced to approximate solutions, either by Markov chain Monte Carlo or by variational inference. We will see in Section 9.3.1 one example of a Bayesian treatment of the model's parameters, called Monte Carlo dropout.

MAP

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

LIKELIHOOD REGULARIZER

ML: DATA RANDOM BAYES: F RANDOM

MAP: Likelihood Plus Regularizer

We remark on two interesting facts about the posterior before closing this section. First, suppose we are only interested in the function having the highest posterior density. In this case, the evidence term can be ignored and the solution decomposes into two separate terms: $f^* = \operatorname{argmax}_f p(\mathcal{S}_n | f) p(f) = \operatorname{argmax}_f (\log p(\mathcal{S}_n | f) + \log p(f))$, a likelihood term plus a regularization term.

This is called the maximum a posteriori, MAP, solution. If all functions have the same weight a priori, that is, if $p(f)$ is uniform over the function space, then the second term is a constant and the problem reduces to the maximum likelihood solution. In general, however, the maximum a posteriori solution will impose a penalty on functions deviating too much from our prior distribution. We will see

this is a useful idea to combat overfitting and to impose specific constraints on the function f . The term $\log p(f)$ is generally called a regularizer over the function space, as it pushes the solution towards the basin of attraction defined by the prior distribution. The difference between maximum likelihood and maximum a posteriori solutions is loosely connected to the difference between the frequentist and Bayesian interpretations of probability, that is, probabilities as frequencies of events or probabilities as a measure of uncertainty. From a very high-level point of view, maximum likelihood sees the parameters as an unknown fixed term and the data as a random sample, while a Bayesian treatment sees the data as fixed and the parameters as random variables.

Questions

1. Why can the likelihood of a dataset be written as a product over its samples?
 - a. Because the samples are assumed i.i.d., so their probabilities multiply
 - b. Because all samples have the same label
 - c. Because the model is linear
 - d. Because the dataset is small
2. How does maximum likelihood connect to minimizing a loss?
 - a. It does not; they are unrelated
 - b. Taking logs and flipping the sign turns it into minimizing the sum of $-\log p(y_i | f(x_i))$, a pseudo-loss
 - c. It requires the hinge loss

- d. It replaces gradient descent
3. When does the maximum a posteriori solution reduce to maximum likelihood?
- a. When the prior is uniform, so $\log p(f)$ is a constant
 - b. When the dataset is empty
 - c. When the evidence is zero
 - d. Never
4. Why are Bayesian neural networks hard to use exactly?
- a. Because priors cannot be defined
 - b. Because the posterior is in general impossible to compute in closed form, so approximations such as MCMC or variational inference are needed
 - c. Because they have no likelihood
 - d. Because they need labeled test sets

Answers

- 1. **a)** Independent and identically distributed samples have a joint probability equal to the product of the individual probabilities.
- 2. **b)** Maximizing the log of the product equals minimizing the sum of negative log-probabilities, which plays the role of a loss.
- 3. **a)** With equal prior weight for all functions, the regularization term is constant and only the likelihood remains.
- 4. **b)** Except for very specific priors and likelihoods, the posterior has no closed form.

Worked examples

Likelihood and negative log-likelihood On three examples, a model gives the true labels probabilities 0.9, 0.8 and 0.5. Compute the likelihood of the dataset and the negative log-likelihood, using natural logarithms.

With i.i.d. samples, the likelihood is the product: $0.9 \times 0.8 \times 0.5 = 0.36$.

The pseudo-loss is the sum of $-\log p$ for each example: $-\ln 0.9 \approx 0.105$, $-\ln 0.8 \approx 0.223$, $-\ln 0.5 \approx 0.693$.

Their sum is about 1.022.

Check: $-\ln 0.36 \approx 1.022$, the same, since the log of a product is the sum of the logs.

Answer: Likelihood 0.36; negative log-likelihood ≈ 1.022 .

Practice

1. A second model gives the same three true labels probabilities 0.6, 0.6 and 0.6. Which model does maximum likelihood prefer?

Hint: Compute its likelihood and compare with 0.36.

Answer: Its likelihood is $0.6^3 = 0.216$, lower than 0.36, so maximum likelihood prefers the first model.

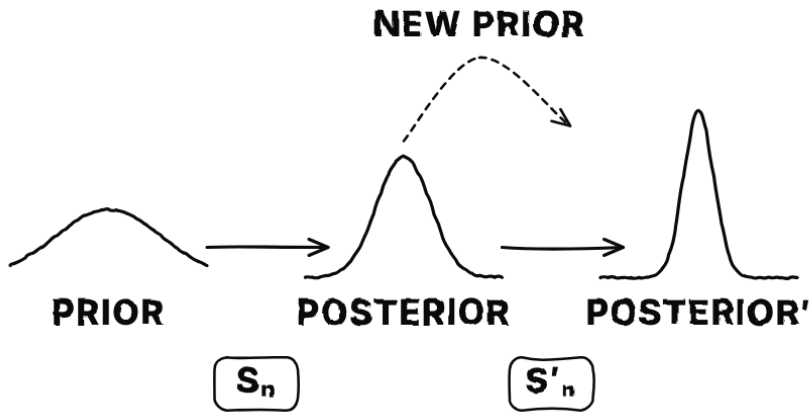
2. With equal priors of 0.5 for the two models above, use Bayes' theorem to find the posterior probability of the first model.

Hint: Multiply each likelihood by its prior, then divide by the sum of both, the evidence.

Answer: Prior times likelihood: $0.5 \times 0.36 = 0.18$ and $0.5 \times 0.216 = 0.108$. The evidence is 0.288, so the first model's posterior is $0.18 / 0.288 = 0.625$.

Linear models, and the squared loss

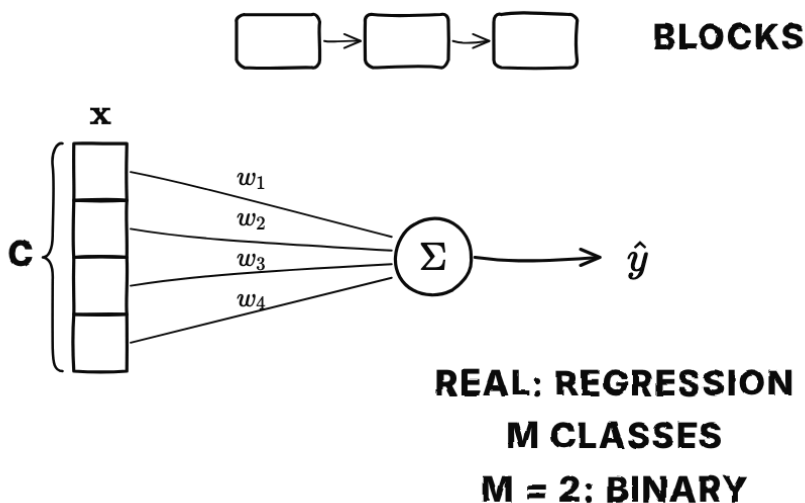
Second, the full Bayesian treatment provides a simple way to incorporate new data, for example a new dataset $\mathcal{S}'_n$ from the same distribution. To do that, we replace the prior function in equation E.3.8 with the posterior distribution that we computed on the first portion of the dataset, which now represents the starting assumption on the possible values of f , and which gets updated by looking at new data. Think of the original prior function as the distribution on f after having observed an initial empty set of values. This can mitigate issues when training models online, most notably the so-called catastrophic forgetting of old information.



NO FORGETTING

Yesterday's Posterior, Today's Prior

Chapter 4, Linear models. In short: programming is done by choosing the appropriate sequence of primitive operations to solve a task. By analogy, building a model is done by choosing the correct sequence of differentiable blocks. In this chapter we introduce the simplest block, linear models, which assume that inputs act additively on the output via a weighted average. In a sense, all differentiable models are smart variations and compositions of linear blocks.



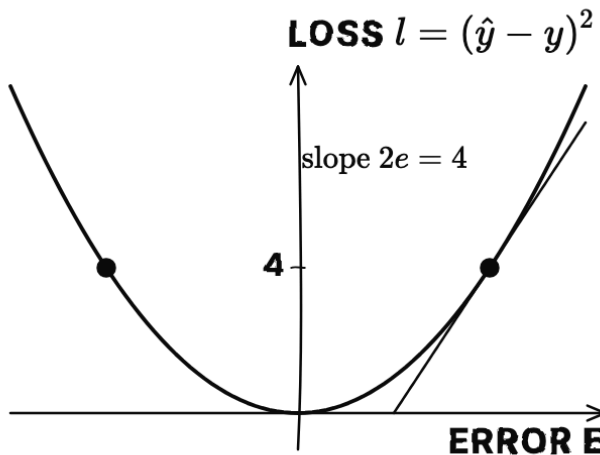
A Linear Model: Weighted Sum

Section 4.1, Least-squares regression. Summarizing the previous chapter, a supervised learning problem can be defined by choosing the input type x , the output type y , the model f , and the loss function l . In this chapter we consider the simplest possible choices for all of them.

The input is a vector $x \sim (c)$, corresponding to a set of features, for example c personal features of a client of a bank. We use the scalar c , short for channels, to denote the number of features, to be consistent with the following chapters. The output is a single real value $y \in \mathbb{R}$. In the unconstrained case, we say this is a regression task. If y can only take one out of m possible values, from 1 to m , we say this is a classification task. In the special case of $m = 2$, we say this is a binary

classification task. We take f to be a linear model, providing us with simple closed-form solutions in some cases, most notably least-squares regression, in Section 4.1.2.

The basic shapes to remember are summarized in Table T.4.1: n is the size of the dataset, c the number of features, and m the number of classes; for uniformity, we will use the same letters as much as possible throughout the book. We begin by discussing the choice of loss in the regression case. We start from the regression case since, as we show later, classification can be solved by small modifications to the regression case.



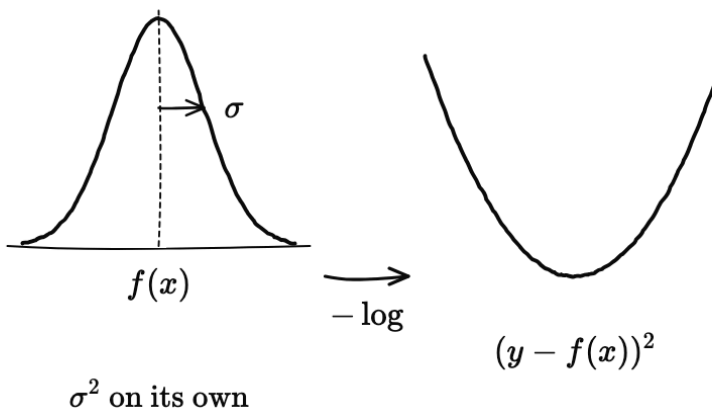
The Squared Loss

The squared loss and variants. Finding a loss for regression is relatively simple, since the prediction error $e = \hat{y} - y$ between the predicted output of the model, $\hat{y} = f(\mathbf{x})$, and the true desired output y is a well-defined target, being a continuous function of the model's output that decreases monotonically. Since in general we do not care about the sign of the prediction error, a common choice is the squared loss: $l(\hat{y}, y) = (\hat{y} - y)^2$.

Here and in the following we use the symbol $\hat{y}$ to denote the prediction of a model. As we will see, working with equation E.4.1 grants several benefits to our solution. Among others, the gradient of the squared loss is a linear function of the model's output, allowing us to solve for the optimal solution in closed form.

Recalling the maximum likelihood principle, the squared loss can be obtained by assuming that the outputs of the model follow a Gaussian distribution centered in $f(\mathbf{x})$ and with a constant variance σ^2 : $p(y | f(\mathbf{x})) = \mathcal{N}(y | f(\mathbf{x}), \sigma^2)$.

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



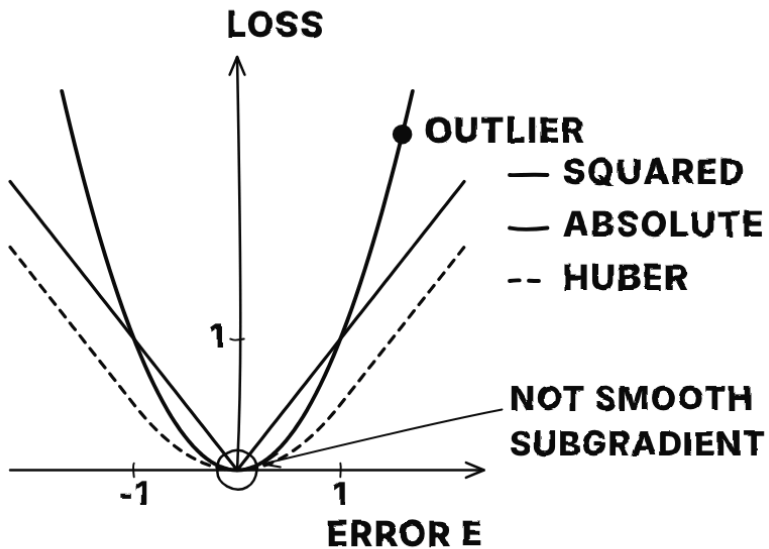
Gaussian Noise Gives Squared Loss

In this case the log-likelihood for a single point can be written, recalling that the log of a product is the sum of the logs and the log of a power is the exponent times the log, as $\log p(y \mid f(x), \sigma^2) = -\log \sigma - \frac{1}{2} \log(2\pi) - (y - f(x))^2 / (2\sigma^2)$.

Minimizing equation E.4.2 for f , we see that the first two terms on the right-hand side are constant, and the third reverts to the squared loss. Minimizing for σ^2 can be done independently of the optimization of f , with a simple closed-form solution, equation E.4.9 below.

Coming up with variants of the squared loss is also easy. For example, one drawback of the squared loss is that higher errors are penalized with a strength that grows quadratically in the error, which may give

undue influence to outliers, that is, points that are badly mislabeled. Other choices that diminish the influence of outliers can be the absolute value loss, $l(\hat{y}, y) = \text{abs}(\hat{y} - y)$, or the Huber loss, a combination of the squared loss and the absolute loss. The Huber loss is $\frac{1}{2}(y - \hat{y})^2$ when the absolute error is at most 1, and $\text{abs}(y - \hat{y}) - \frac{1}{2}$ otherwise.



Squared, Absolute and Huber

Figure F.4.1 visualizes the squared loss, the absolute loss, and the Huber loss with respect to the prediction error $e = \hat{y} - y$.

The Huber loss is quadratic in the proximity of 0 error, and linear otherwise, with the $-\frac{1}{2}$ term added to ensure continuity.

The absolute loss seems an invalid choice in our context, since it has a point of non-differentiability at 0, due to the absolute value. We will see later that functions with one, or a small number, of points of this form are not truly problematic. Mathematically, they can be handled by the notion of subgradient, a slight generalization of the derivative. Practically, you can imagine that if we start from a random initialization, gradient descent will never reach these points with perfect precision, and the derivative of the absolute value of ϵ for any $\epsilon > 0$ is always defined.

Questions

1. Under which assumption does maximum likelihood give exactly the squared loss?
 - a. Outputs follow a Gaussian centered on $f(\mathbf{x})$ with constant variance
 - b. Outputs follow a uniform distribution
 - c. Outputs are binary
 - d. The model has no parameters
2. Why might one prefer the Huber or absolute loss to the squared loss?
 - a. They are always differentiable
 - b. The squared loss grows quadratically with the error, giving outliers undue influence; the others grow only linearly for large errors
 - c. They give closed-form solutions

- d. They ignore the sign of the error while the squared loss does not
3. Why is the absolute loss's corner at 0 not a real problem in practice?
- a. Because gradient descent from a random start essentially never lands exactly on 0, and elsewhere the derivative exists; subgradients handle the point mathematically
 - b. Because the absolute loss is never used
 - c. Because the corner is smoothed by the GPU
 - d. Because the loss is constant there
4. In the book's view, what are linear models to all differentiable models?
- a. Unrelated
 - b. The simplest block; all differentiable models are smart variations and compositions of linear blocks
 - c. Always better
 - d. A special case of classification only

Answers

- 1. a)** With a Gaussian of constant variance, the log-likelihood has two constant terms and a third proportional to $(y - f(x))^2$, the squared loss.
- 2. b)** Large errors are penalized quadratically by the squared loss, so badly mislabeled points dominate; linear growth limits their pull.
- 3. a)** One point of non-differentiability is handled by subgradients, and in practice it is never hit with perfect precision.
- 4. b)** The chapter opens by calling linear models the simplest differentiable block, of which all others are variations and compositions.

Worked examples

Three losses on two errors Compute the squared, absolute and Huber losses for a small error $e = 0.5$ and a large error $e = 4$.

Squared loss: $0.5^2 = 0.25$ and $4^2 = 16$.

Absolute loss: 0.5 and 4 .

Huber loss: for 0.5 the absolute error is at most 1 , so it is $\frac{1}{2} \times 0.5^2 = 0.125$; for 4 it is $4 - \frac{1}{2} = 3.5$.

The large error weighs 64 times the small one under the squared loss but only 28 times under Huber, which is why Huber limits outliers.

Answer: Squared 0.25 and 16 ; absolute 0.5 and 4 ; Huber 0.125 and 3.5 .

Practice

1. Check that the two pieces of the Huber loss meet at an absolute error of 1 .

Hint: Evaluate both formulas at error 1 .

Answer: The quadratic piece gives $\frac{1}{2} \times 1^2 = 0.5$ and the linear piece gives $1 - \frac{1}{2} = 0.5$: they agree, which is what the $-\frac{1}{2}$ term is there for.

2. A bank model uses $c = 12$ client features and a dataset of $n = 5,000$ clients. What is the shape of one input, and is predicting a credit limit in euros a regression or a classification task?

Hint: Look at what kind of value the output takes.

Answer: One input is a vector $\mathbf{x} \sim (12)$. A credit limit is an unconstrained real value, so it is a regression task.