Alice's Adventures in a differentiable wonderland

A primer on designing neural networks

Vol. I - A tour of the land

Simone Scardapane

"For, you see, so many out-of-the-way things had happened lately, that Alice had begun to think that very few things indeed were really impossible."

— Chapter 1, Down the Rabbit-Hole

Foreword

This book is an introduction to the topic of (deep) **neural networks** (NNs), the core technique at the hearth of large language models, generative artificial intelligence - and many other applications. Because the term *neural* comes with a lot of historical baggage, and because NNs are simply compositions of differentiable primitives, I refer to them – when feasible – with the simpler term **differentiable models**.

In 2009, I stumbled almost by chance upon a paper by Yoshua Bengio on the power of 'deep' NNs [Ben09], at the same time when automatic differentiation libraries like Theano [ARAA+16] were becoming popular. Like Alice, I had stumbled upon a strange programming realm - a *differentiable* wonderland where simple things, such as selecting an element, were incredibly hard, and other things, such as recognizing cats, were amazingly simple.

I have spent more than ten years reading about, implementing, and teaching about these models. This book is a rough attempt at condensing something of what I have learned in the process, with a focus on their design and most common components. Because the field is evolving quickly, I have tried to strike a good balance between theory and code, historical considerations and recent trends. I assume the reader has some exposure to machine learning and linear algebra, but I try to cover the preliminaries when necessary.

Gather round, friends: it's time for our beloved Alice's adventures in a differentiable wonderland!

Contents

1 | Introduction

Neural networks have become an integral component of our everyday's world, either openly (e.g., in the guise of **large language models**, LLMs), or hidden from view, by powering or empowering countless technologies and scientific discoveries [WFD+23] 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.

The power of scaling is embodied in the relatively recent concept of **neural scaling laws**, which has driven massive investments in artificial intelligence (AI) [KMH+20, HBE+24]: 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 [HBE+24]. The tremendous power of combining simple, general-purpose tools with exponentially increased computational power in AI was called the *bitter lesson* by R. Sutton.[1]

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 [ZBH+21] 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 [ASA+23]. 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

[1] R. Sutton, *The Bitter Lesson*, http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

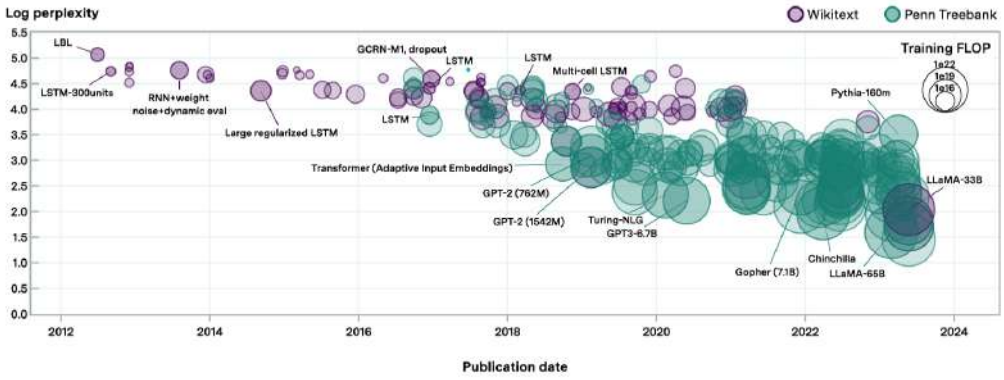


Figure F.1.1: *Performance in **language modeling** – predicting the continuation of a sentence –, evaluated here in terms of **perplexity**, has steadily improved, while the size of the models has constantly increased. The increase in performance is also matched by equivalent data scaling, with variations in modelling becoming asymptotically less significant. Reproduced from [HBE+24].*

strong generalization capabilities and a fascinating low dependency on pure memorization, i.e., **overfitting** [PBE+22].

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,[2] has also shifted how we interact with these models. Many tasks can now be solved by simply *prompting* (i.e., interacting with text or visual instructions) a pre-trained model found on the web [ASA+23], 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 programs your libraries in, e.g., 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 seems: nowadays, few teams worldwide have the compute and the technical expertise to design and release truly large-scale models such as the Llama LLMs [TLI+23], just like few companies have the resources to build enterprise CRM software.

And in the same way, just like open-source software provides endless possibilities for customizing or designing from scratch your programs, customer-grade hardware and a bit of ingenuity gives you a vast array of options to experiment with differentiable models, from **fine-tuning** them for your tasks [LTM+22] to merging different models [AHS23], quantizing them for low-power hardware, testing their robustness, or even

[2] https://huggingface.co/

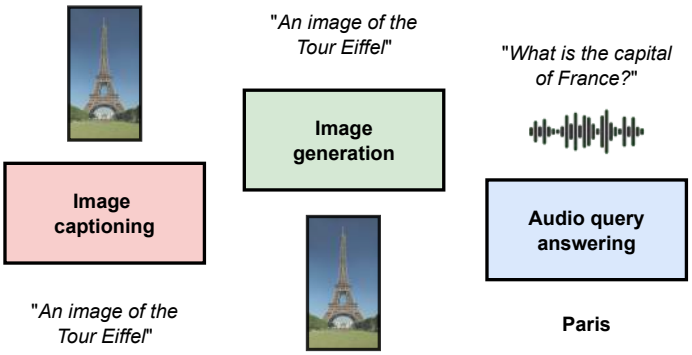


Figure F.1.2: *Most tasks can be categorized based on the desired input - output we need: image generation wants an image (an ordered grid of pixels) from a text (a sequence of characters), while the inverse (image captioning) is the problem of generating a caption from an image. As another example, audio query answering requires a text from an audio (another ordered sequence, this time numerical). Fascinatingly, the design of the models follow similar specifications in all cases.*

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 idiosincrasies 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.

About this book

We assume our readers are familiar with the basics of **machine learning** (ML), and more specifically **supervised learning** (SL). SL can be used to solve complex tasks by gathering data on a desired behavior, and 'training' (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' decision (Figure F.1.2).

In general, learning is a **search** problem. We start by defining a program with a large number of *degree-of-freedoms* (that we call parameters), and we manipulate the parameters until the model performance is satisfying. To make this idea practical, we

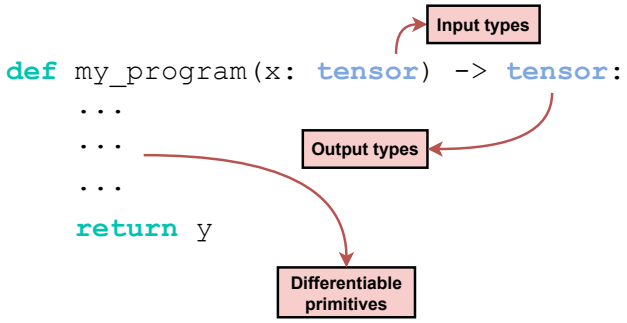


Figure F.1.3: *Neural networks are sequences of differentiable **primitives** which operate on structured arrays (**tensors**): each primitive can be categorized based on its input/output signature, which in turn defines the rules for composing them.*

need efficient ways of searching for the optimal configuration even in the presence of millions (or billions, or trillions) of such parameters. As the name implies, **differentiable models** do this by restricting the selection of the model to differentiable components, i.e., 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 their 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 (Part I, **Compass and Needle**), going from Chapter 2 to Chapter 6.

By viewing neural networks as simply compositions of differentiable primitives we can ask two basic questions (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 (what we will call **tensors**) of numbers, 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 differentiable component, the **convolutional** operator (Part II, 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* (a.k.a., composed of many steps in sequence) models, as long 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.

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

The book is complemented by a website[3] where I collect additional chapters and material on topics of interest that do not focus on a specific type of data, including **generative modelling**, **conditional computation**, **transfer learning**, and **explainability**. These chapters are more research-oriented in nature and can be read in any order. Hopefully they will be part of a second volume if time allows.

What is "differentiable programming"?

Neural networks have a long and rich history. The name itself is a throwback to early attempts at modelling (biological) neurons in the 20th century, and similar terminology has remained pervasive: to be consistent with existing frameworks, in the upcoming chapters we may refer to *neurons*, *layers*, or, e.g., *activations*. After multiple waves of interest, the period between 2012 and 2017 saw an unprecedented rise in complexity in 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** (Chapter 10) in 2017: just like 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 done for videos, graphs (Chapter 12), and audio, culminating in the current excitement around LLMs, multimodal networks, and generative models.[4]

This period paralleled a quick evolution in terminology, from the **connectionism** of the 80s [RHM86] to the use of **deep learning** for referring to modern networks in opposition to the smaller, *shallower* models of the past [Ben09, LBH15]. Despite this, all these terms remain inexorably vague, because modern (artificial) networks retain almost no resemblance to biological neural networks and neurology [ZER+23]. Looking at modern neural networks, their essential characteristic is being composed by 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

[3] https://sscardapane.it/alice-book

[4] This is not the place for a complete historical overview of modern neural networks; for the interested reader, I refer to [Met22] as a great starting point.

The New York Times

NEW NAVY DEVICE LEARNS BY DOING; Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

July 8, 1958

AI hype - except it is 1958, and the US psychologist Frank Rosenblatt has gathered up significant media attention with his studies on "perceptrons", one of the first working prototypes of neural networks.

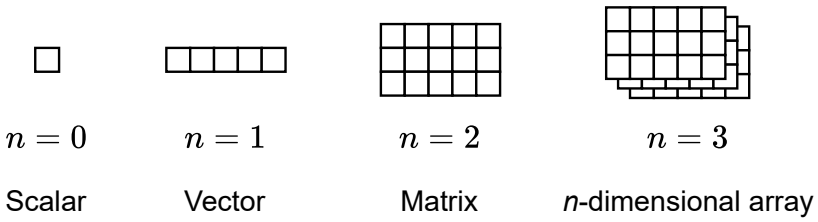
of **differentiable programming**, an emerging discipline that blends computer science and optimization to study differentiable computer programs more broadly [BR24].[5]

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 XIX century [Sti81]; so-called "fully-connected networks" in the form we use later on can be dated back to the 1980s [RHM86]; convolutional models were known and used already at the end of the 90s [LBBH98].[6] However, it took many decades to have sufficient data and power to realize how well they can perform given enough data and enough parameters.

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.

[5] Like many, I was inspired by a 'manifesto' published by Y. LeCun on Facebook in 2018: https://www.facebook.com/yann.lecun/posts/10155003011462143. For the connection between neural networks and open-source programming (and development) I am also thankful to a second manifesto, published by C. Raffel in 2021: https://colinraffel.com/blog/a-call-to-build-models-like-we-build-open-source-software.html.

[6] For a history of NNs up to this period through interviews to some of the main characters, see [AR00]; for a large opinionated history there is also an *annotated history of neural networks* by J. Schmidhuber: https://people.idsia.ch/~juergen/deep-learning-history.html.



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

Notation and symbols

The fundamental data type when dealing with differentiable models is a **tensor**,[7] which we define as an n-dimensional array of objects, typically real-valued numbers. With the necessary apology to any mathematician reading us,[8] we call n the **rank** of the tensor. The notation in the book varies depending on n:

- A single-item tensor ($n = 0$) is just a single value (a **scalar**). For scalars, we use lowercase letters, such as x or y.[9]
- Columns of values ($n = 1$) are called **vectors**. For vectors we use a lowercase bold font, such as $\mathbf{x}$. The corresponding row vector is denoted by $\mathbf{x}^\top$ when we need to distinguish them. We can also ignore the transpose for readability, if the shape is clear from context.
- Rectangular array of values ($n = 2$) are called a **matrix**. 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 $\mathcal{X}$, that we reserve for sets or probability distributions.

[7] In the scientific literature, tensors have a more precise definition as multilinear operators [Lim21], while the objects we use in the book are simpler multidimensional arrays. Although technically a misnomer, the use of *tensor* is so widespread that we keep this convention here.

[8] Assuming anyone is actually reading us.

[9] 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.

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:

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

This is a rank-4 tensor with shape $(b, h, w, 3)$. Some dimensions can be pre-specified (e.g., 3), while other dimensions can be denoted by variables. We use the same symbol to denote drawing from a probability distribution, e.g., $\varepsilon \sim \mathcal{N}(0, 1)$, 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 for $\mathbf{X} \sim (n, d)$ instead of $\mathbf{X} \in \mathbb{R}^{n \times d}$. Finally, we may want to constrain the elements of a tensor, for which we use a special notation:

1. $\mathbf{x} \sim \text{Binary}(c)$ denotes a tensor with only binary values, i.e., elements from the set $\{0, 1\}$.

2. $\mathbf{x} \sim \Delta(a)$ denotes a vector belonging to the so-called **simplex**, i.e., $x_i \geq 0$ and $\sum_i x_i = 1$. For tensors with higher rank, e.g., $\mathbf{X} \sim \Delta(n, c)$, we assume the normalization is applied with respect to the last dimension (e.g., in this case each row of $\mathbf{X}_i$ belongs to the simplex).

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

- A **bottle** to emphasize some definitions. We have many definitions, especially in the early chapters, and we use this symbol to visually discriminate the most important ones.

- A **clock** for sections we believe crucial to understand the rest of the book – please do not skip these!

- On the contrary, a **teacup** for more relaxed sections – these are generally discursive and mostly optional in relation to the rest of the book.

Final thoughts before departing

The book stems from my desire to give a coherent form to the lectures I prepared for a course called **Neural Networks for Data Science Applications**, which I have been teaching in the Master Degree in Data Science at Sapienza University of Rome

for a few years. 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 I 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.

There are already a number of excellent (and recent) books on the topic of modern, deep neural networks, including [Pri23, ZLLS23, BB23, Fle23, HR22]. This book covers a similar content to all of these in the beginning, while the exposition and some additional parts (or a few sections in the advanced chapters) intersect less, and they 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 [ARAA+16] 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 (a) common today, and (b) 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 (e.g., from batch normalization to layer normalization). However, keep in mind that hundreds more exist: I invite you for this to an exploration of 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.[10]

[10] See for example this blog post by A. Karpathy: http://karpathy.github.io/2019/04/25/recipe/, or his recent **Zero to Hero** video series: https://karpathy.ai/zero-to-hero.html.

Acknowledgments

Equations' coloring is thanks to the beautiful `st--/annotate-equations` package.[11] 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 from the original Arthur Rackham's 1907 illustrations, thanks to Wikimedia.[12] I thank Roberto Alma for extensive feedback on a previous draft of the book and for encouraging me to publish the book. I also thank Corrado Zoccolo and Emanuele Rodolà for providing corrections and suggestions to the current version, and everyone who provided me feedback via email.

[11] https://github.com/st--/annotate-equations/tree/main

[12] https://commons.wikimedia.org/wiki/Category:Alice%27s_adventures_in_Wonderland_(Rackham,_1907)

Part I
Compass and needle

“Would you tell me, please, which way I ought to go from here?”
“That depends a good deal on where you want to get to,” said the Cat.
“I don’t much care where” said Alice.
“Then it doesn’t matter which way you go,” said the Cat.

— Chapter 6, Pig and Pepper

2 | Mathematical preliminaries

About this chapter

We compress here the mathematical concepts required to follow the book. We assume prior knowledge on 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 (e.g., tensors) and their 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 Appendix A, with a focus on the **maximum likelihood** principle. This chapter is full of content and definitions: bear with me for a while!

2.1 Linear algebra

We recall here some basic concepts from linear algebra that will be useful in the following (and to agree on a shared notation). Most of the book revolves around the idea of a **tensor**.

Definition D.2.1 (Tensors) *A **tensor** X is an n-dimensional array of elements of the same type. We use $X \sim (s_1, s_2, \dots, s_n)$ 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, 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 (e.g., TPUs, 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 (the specific format of which we will consider later on), but they can also be defined for integers (e.g., in classification) or for strings (e.g., for text). Tensors can be **indexed** to get **slices** (subsets) of their values, and most conventions from NumPy indexing[1] apply. For simple equations we use pedices: for example, for a 3-dimensional tensor $X \sim (a, b, c)$ we can write X_i to denote a slice of size (b, c) or X_{ijk} 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:

$$[X]_{ijk}$$

to visually split the indexing part from the rest, where the argument of $[\,\bullet\,]$ can also be an expression.

2.1.1 Common vector operations

We are mostly concerned with models that can be written as composition 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 1-dimensional tensors. Linear algebra books are concerned with distinguishing between column vectors $\mathbf{x}$ and row vectors $\mathbf{x}^\top$, 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 2-dimensional tensors of shape $(1, d)$ or $(d, 1)$, which are different from 1-dimensional tensors of shape (d). This is important to keep in mind because most frameworks implement broadcasting rules[2] inspired by NumPy, giving rise to non-intuitive behaviors. See Box C.2.1 for an example of a very common error arising in implicit broadcasting of tensors' shapes.

[1] See https://numpy.org/doc/stable/user/basics.indexing.html for a review. For readability in the book we index from 1, not from 0.

[2] See: https://numpy.org/doc/stable/user/basics.broadcasting.html. In a nutshell, broadcasting aligns the tensors' shape from the right, and repeats a tensor whenever possible to match the two shapes.

```python
import torch
x = torch.randn((4, 1))   # "Column" vector
y = torch.randn((4,))     # 1-dimensional tensor
print((x + y).shape)
# [Out]: (4,4) (because of broadcasting!)
```

Box C.2.1: *An example of (probably incorrect) broadcasting, resulting in a matrix output from an elementwise operation on two vectors due to their shapes. The same result can be obtained in practically any framework (NumPy, TensorFlow, 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:

$$z_i = ax_i + by_i$$

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 (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 D.2.2 (Inner product) *The inner product between two vectors $\mathbf{x}, \mathbf{y} \sim (d)$ is given by:*

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\top \mathbf{y} = \sum_i x_i y_i \tag{E.2.1}$$

The notation $\langle \bullet, \bullet \rangle$ 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 $\mathbf{y} = [-4.0, 0.05, 0.1]$:

$$\langle \mathbf{x}, \mathbf{y} \rangle = -0.4 + 0 - 0.03 = -0.43$$

A simple geometric interpretation of the dot product is given by its relation with the

angle α between the two vectors:

$$\mathbf{x}^\top \mathbf{y} = \|\mathbf{x}\|\|\mathbf{y}\| \cos(\alpha) \tag{E.2.2}$$

Hence, for two normalized vectors such that $\|\cdot\| = 1$, 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 (two vectors pointing in the same direction) and -1 (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 unitary norm), if we fix $\mathbf{x}$, then:

$$\mathbf{y}^* = \arg\max \ \langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x} \tag{E.2.3}$$

where $\arg\max$ denotes the operation of finding the value of $\mathbf{x}$ corresponding to the highest possible value of its argument. From (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 $\mathbf{w}$ (**template matching**).

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:

$$\langle \mathbf{x}, \mathbf{1} \rangle = \sum_{i=1}^{d} x_i$$

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.

2.1.2 Common matrix operations

In the 2-dimensional case we have matrices:

$$\mathbf{X} = \begin{bmatrix} X_{11} & \cdots & X_{1d} \\ \vdots & \ddots & \vdots \\ X_{n1} & \cdots & X_{nd} \end{bmatrix} \sim (n, d)$$

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 $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, where the stack is organized in a row-wise fashion:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix}$$

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 D.2.3 (Matrix multiplication) *Given two matrices* $\mathbf{X} \sim (a,b)$ *and* $\mathbf{Y} \sim (b,c)$*, matrix multiplication* $\mathbf{Z} = \mathbf{XY}$*, with* $\mathbf{Z} \sim (a,c)$ *is defined element-wise as:*

$$Z_{ij} = \langle \mathbf{X}_i, \mathbf{Y}_j^\top \rangle \tag{E.2.4}$$

i.e., 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:

$$\mathbf{z} = \mathbf{Wx} \tag{E.2.5}$$

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

Definition D.2.4 (Hadamard multiplication) *The* ***Hadamard multiplication*** *of two matrices of the same shape is done element-wise:*

$$[\mathbf{X} \odot \mathbf{Y}]_{ij} = X_{ij} Y_{ij}$$

While Hadamard multiplication does not have all the interesting algebraic properties

```
X = torch.randn((5, 5))
X = torch.exp(X)                # Element-wise exponential
X = torch.linalg.matrix_exp(X) # Matrix exponential
```

Box C.2.2: *Difference between the element-wise exponential of a matrix and the matrix exponential as defined in linear algebra textbooks. Specialized linear algebra operations are generally encapsulated in their own sub-package.*

of standard matrix multiplication, it is commonly used in differentiable models for performing *masking* operations (e.g., 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.

On the definition of matrix multiplication

Why is matrix multiplication defined as (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 matrix multiplication $\mathbf{AB}$ corresponds to function composition $f \circ g$, where $(f \circ g)(\mathbf{x}) = f(g(\mathbf{x}))$ and $g(\mathbf{x}) = \mathbf{Bx}$.

Sometimes we write expressions such as $\exp(\mathbf{X})$, which are to be interpreted as *element-wise* applications of the operation:

$$[\exp(\mathbf{X})]_{ij} = \exp(X_{ij}) \tag{E.2.6}$$

By comparison, "true" matrix exponentiation is defined for a squared matrix as:

$$\text{mat-exp}(\mathbf{X}) = \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{X}^k \tag{E.2.7}$$

Importantly, (E.2.6) can be defined for tensors of any shape, while (E.2.7) is only valid for (squared) matrices. This is why all frameworks, like PyTorch, have specialized modules that collect all matrix-specific operations, such as inverses and determinants. See Box C.2.2 for an example.

Finally, we can write *reduction* operations (sum, mean, ...) across axes without specify-

ing 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:

```
r = X.sum(axis=1)
```

Computational complexity and matrix multiplication

I will use matrix multiplication to introduce the topic of *complexity* of an operation. Looking at (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 (what we call the *time* complexity), while the memory requirement for a sequential implementation is simply the size of the output matrix (what we call instead the *space* complexity).

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:

$$f(x) \leq cg(x) \quad \text{for any } x \geq x_0 \tag{E.2.8}$$

meaning 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 to practical performance gains, which depends on many details such as what hardware is used, what parallelism is supported, and so on.[3] As an example, it is known that the best

[3] When you call a specific primitive in a linear algebra framework, such as matrix multiplication `A @ B` in PyTorch, the specific 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, etc. This is made even more complex by compilers such as XLA

asymptotic algorithm for multiplying two square matrices of size (n, n) scales as $\mathcal{O}(n^c)$ for a constant $c < 2.4$ [CW82], 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.

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 $\frac{1}{k}$ 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 the other, which can become the major bottleneck in many situations.[4] In these cases, we say the implementation is *memory-bound* as opposed to *compute-bound*. Practically, 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.

2.1.3 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 (e.g., matrix exponentials, matrix multiplication, determinants, ...). When moving to higher dimensions, most of the operations we are interested into are either batched variants of matrix operations, or specific combinations of matrix operations and reduction operations.

As an example of the former, consider two tensors $X \sim (n, a, b)$ and $Y \sim (n, b, c)$. **Batched matrix multiplication** (BMM) is defined as:

$$[\text{BMM}(X, Y)]_i = \mathbf{X}_i \mathbf{Y}_i \sim (n, a, c) \tag{E.2.9}$$

Operations in most frameworks operate transparently on batched versions of their arguments, which are assumed like in this case to be *leading dimensions* (the first dimensions). For example, batched matrix multiplication in PyTorch is the same as standard matrix multiplication, see Box C.2.3.

As an example of a reduction operation, consider two tensors $X, Y \sim (a, b, c)$. A gen-

(https://openxla.org/xla), which can optimize code by fusing and optimizing operations with a specific target hardware in mind.

[4] https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html

```
X = torch.randn((4, 5, 2))
Y = torch.randn((4, 2, 3))
(torch.matmul(X, Y)).shape # Or X @ Y
# [Out]: (4, 5, 3)
```

Box C.2.3: *BMM in PyTorch is equivalent to standard matrix multiplication. Practically every operation is implemented to run on generically batched inputs.*

eralized version of the dot product (GDT) can be written as:

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

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.

2.1.4 Einstein's notation

This is an optional section that covers `einsum`,[5] a set of conventions allowing the user to specify most tensor operations with a unified syntax. Let us consider again the two examples shown before, writing down explicitly all the axes:

$$\text{Batched matrix multiply:} \qquad M_{ijk} = \sum_z A_{ijz} B_{izk} \tag{E.2.10}$$

$$\text{Generalized dot product:} \qquad M = \sum_i \sum_j \sum_k X_{ijk} Y_{ijk} \tag{E.2.11}$$

In line with **Einstein's notation**,[6] 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:

$$M_{ijk} = A_{ijz} B_{izk} \triangleq \sum_z A_{ijz} B_{izk} \tag{E.2.12}$$

$$M = X_{ijk} Y_{ijk} \triangleq \sum_i \sum_j \sum_k X_{ijk} Y_{ijk} \tag{E.2.13}$$

[5] https://numpy.org/doc/stable/reference/generated/numpy.einsum.html

[6] See https://en.wikipedia.org/wiki/Einstein_notation. The notation we use is a simplified version which ignores the distinction between upper and lower indices.

```
# Batched matrix multiply
M = torch.einsum('ijz,izk->ijk', A, B)
# Generalized dot product
M = torch.einsum('ijk,ijk->', A, B)
```

Box C.2.4: *Examples of using einsum in PyTorch.*

```
M = jax.numpy.einsum('ijz,izk->ijk', A, B)
```

Box C.2.5: *Example of using einsum in JAX - compare with Box C.2.4.*

Then, we can condense the two definitions by isolating the indices in a unique string (where the operands are now on the left):

- '*ijz,izk→ijk*' (batched matrix multiply);
- '*ijk,ijk→*' (generalized dot product).

There is a direct one-to-one correspondence between the definitions in (E.2.12)-(E.2.13) and their simplified string definition. This is implemented in most frameworks in the `einsum` operation, see Box C.2.4.

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 the other 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, e.g., the definition in JAX, see Box C.2.5.

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 $[\mathbf{A}]_i$ times $[\mathbf{B}^\top]_i$ is obtained by switching the corresponding axes in the einsum definition:

```
M = torch.einsum('ijz,ikz->ijk', A, B)
```

Because of these reasons, einsum and its generalizations (like the popular `einops`[7] package) have gained a wide popularity recently.

[7] http://einops.rocks

2.2 Gradients and Jacobians

As the name *differentiable* implies, gradients play a pivotal role in the book, by allowing us to optimize our models through semi-automatic mechanisms deriving from gradient descent. To this end, we recall here some basic definitions and concepts concerning differentiation of multi-valued functions. We focus on properties that will be essential for later, partially at the expense of mathematical precision.

2.2.1 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 D.2.5 (Derivative) *The **derivative** of $f(x)$ is defined as:*

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{E.2.14}$$

We use a variety of notation to denote derivatives: ∂ will denote generically derivatives and gradients of any dimension (vectors, matrices); ∂_x or $\frac{\partial}{\partial x}$ to highlight 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 about the existence of the derivative of the function (which is not guaranteed everywhere even for a continuous function), which we assume as given. We will only touch upon this point when discussing derivatives of non-smooth functions, such as $f(x) = |x|$ in 0 later on in Chapter 6.

Derivatives of simple functions can be obtained by direct application of the definition, e.g., the derivative of a polynomial should be familiar:

$$\begin{aligned} \partial x^p &= \lim_{h \to 0} \frac{\overbrace{(x+h)^p}^{\text{Expandable via the binomial theorem}} - x^p}{h} = \lim_{h \to 0} \frac{1}{h} \left(p x^{p-1} h + \frac{p(p-1)}{2} x^{p-2} h^2 + \ldots + h^p \right) \\ &= \underbrace{p x^{p-1}}_{\text{Independent from } h} + \lim_{h \to 0} \left(\frac{p(p-1)}{2} x^{p-2} h^2 + \ldots + h^{p-1} \right) = p x^{p-1} \end{aligned}$$

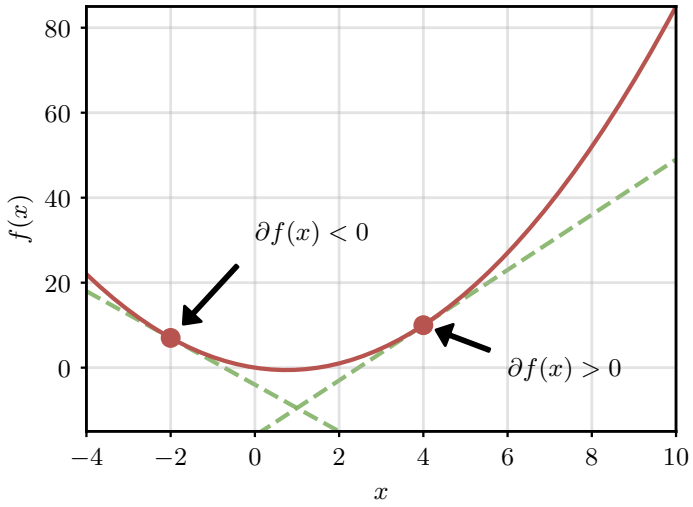


Figure F.2.1: *Plot of the function $f(x) = x^2 - 1.5x$, shown along with the derivatives on two separate points.*

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 in 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.

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:
$$\partial\Big[f(x) + g(x)\Big] = f'(x) + g'(x)\,.$$
- **Product rule**:
$$\partial\Big[f(x)g(x)\Big] = f'(x)g(x) + f(x)g'(x)\,,$$
- **Chain rule**: the derivative of function composition is given by multiplying the corresponding derivatives:
$$\partial\Big[f(g(x))\Big] = f'(g(x))g'(x) \tag{E.2.15}$$

2.2.2 Gradients and directional derivatives

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 this perturbation (while in the scalar case we only had "left" and "right", in this case we have infinite possible directions in the Euclidean space). In the simplest case, we can consider moving along the i-th axis, keeping all other values fixed:

$$\partial_{x_i} f(\mathbf{x}) = \frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}, \tag{E.2.16}$$

where $\mathbf{e}_i \sim (d)$ is the i-th basis vector (the i-th row of the identity matrix):

$$[\mathbf{e}_i]_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{E.2.17}$$

(E.2.16) is called a **partial derivative**. Stacking all partial derivatives together gives us a d-dimensional vector called the **gradient** of the function.

Definition D.2.6 (Gradient) *The **gradient** of a function $y = f(\mathbf{x})$ is given by:*

$$\nabla f(\mathbf{x}) = \partial f(\mathbf{x}) = \begin{bmatrix} \partial_{x_1} f(\mathbf{x}) \\ \vdots \\ \partial_{x_d} f(\mathbf{x}) \end{bmatrix} \tag{E.2.18}$$

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**:

$$\mathrm{D}_{\mathbf{v}} f(\mathbf{x}) = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h}, \tag{E.2.19}$$

Movement in space can be decomposed by considering 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 $\mathbf{v}$:

$$\mathrm{D}_{\mathbf{v}} f(\mathbf{x}) = \langle \nabla f(\mathbf{x}), \mathbf{v} \rangle = \sum_i \partial_{x_i} f(\mathbf{x}) v_i \tag{E.2.20}$$

Displacement on the i-th axis

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

2.2.3 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 can compute a gradient for each output, and their stack provides an (o, d) matrix we call the **Jacobian** of f.

Definition D.2.7 (Jacobian) *The **Jacobian** matrix of a function $\mathbf{y} = f(\mathbf{x})$, $\mathbf{x} \sim (d)$, $\mathbf{y} \sim (o)$ is given by:*

$$\partial f(\mathbf{x}) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_o}{\partial x_1} & \cdots & \frac{\partial y_o}{\partial x_d} \end{pmatrix} \sim (o, d) \tag{E.2.21}$$

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:

$$\partial\,[f(g(\mathbf{x}))] = [\partial f(\bullet)]\,\partial g(\mathbf{x}) \tag{E.2.22}$$

where the first derivative is evaluated in $g(\mathbf{x}) \sim (h)$. See [PP08, Chapter 2] for numerical examples of worked out gradients and Jacobians. 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:

$$\widetilde{f}(\mathbf{x}) = f(\mathbf{x}_0) + \langle\, \underbrace{\partial f(\mathbf{x}_0)}_{\text{Slope of the line}},\, \underbrace{\mathbf{x} - \mathbf{x}_0}_{\text{Displacement from } \mathbf{x}_0} \,\rangle$$

This is called **Taylor's theorem**. See Box C.2.6 and Figure F.2.2 for a visualization in the scalar case $f(x) = x^2 - 1.5x$.

```python
# Generic function
f = lambda x: x**2-1.5*x

# Derivative (computed manually for now)
df = lambda x: 2*x-1.5

# Linearization at 0.5
x=0.5
f_linearized = lambda h: f(x) + df(x)*(h-x)

# Comparing the approximation to the real derivative
print(f(x + 0.01))             # [Out]: -0.5049
print(f_linearized(x + 0.01)) # [Out]: -0.5050
```

Box C.2.6: *Example of computing a first-order approximation (scalar case). The result is plotted in Figure F.2.2.*

On the dimensionality of the Jacobians

We close with a pedantic note on dimensionality that will be useful in the following. Consider the following 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:

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

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) (see box in the following page). 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).

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. Importantly, 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 - see [BR24] for a formal overview of this topic.

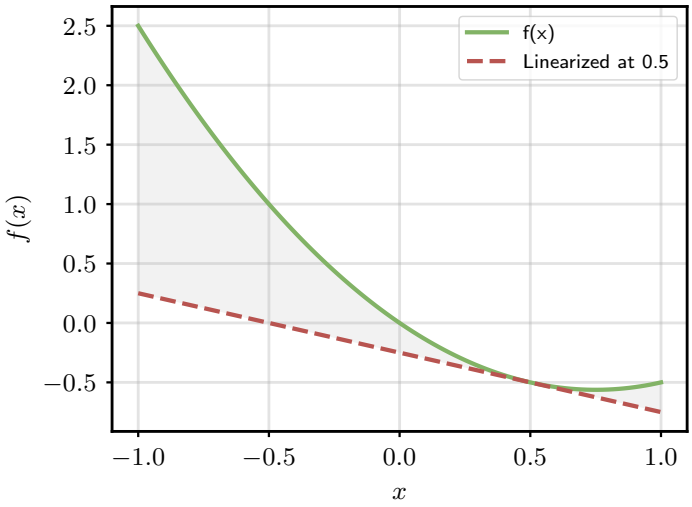


Figure F.2.2: *The function $f(x) = x^2 - 1.5x$ and its first-order approximation shown in* 0.5.

Working 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:

$$\frac{\partial y_i}{\partial W_{ij}} = x_j \tag{E.2.23}$$

Note that to materialize the Jacobian explicitly (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 on another tensor.

2.3 Numerical optimization and gradient descent

To understand the usefulness of having access to gradients, consider the problem of minimizing a generic function $f(\mathbf{x})$, with $\mathbf{x} \sim (d)$:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x}) \tag{E.2.24}$$

where, similarly to arg max, arg min $f(\mathbf{x})$ 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 (**single-objective optimization**), and that the domain over which we are optimizing $\mathbf{x}$ is unconstrained.

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 1D and rotate 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 4.1.3). In general, however, we are forced to resort to iterative procedures. Suppose we start from a random guess $\mathbf{x}_0$ and that, for every iteration, we take a step, that we decompose in terms of its magnitude η_t (the length of the step) and the direction $\mathbf{p}_t$:

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \eta_t \mathbf{p}_t \tag{E.2.25}$$

Guess at iteration t

Displacement at iteration t

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 for every iteration, and if we are careful in the choice of step size, the iterative algorithm in (E.2.25) 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 (E.2.19), as they can be defined as the directions inducing a negative change with respect to our previous guess $\mathbf{x}_{t-1}$:

$$\mathbf{p}_t \text{ is a descent direction } \Rightarrow \; \mathrm{D}_{\mathbf{p}_t} f(\mathbf{x}_{t-1}) \leq 0$$

Using what we learned in Section 2.2 and the definition of the dot product in terms of

cosine similarity from (E.2.2) we get:

$$\mathrm{D}_{\mathbf{p}_t} f(\mathbf{x}_{t-1}) = \langle \nabla f(\mathbf{x}_{t-1}), \mathbf{p}_t \rangle = \|\nabla f(\mathbf{x}_{t-1})\| \|\mathbf{p}_t\| \cos(\alpha)$$

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 is between $\pi/2$ and $3\pi/2$ with $\nabla f(\mathbf{x}_{t-1})$ 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 (E.2.25) gives us an algorithm to minimize any differentiable function, that we call **(steepest) gradient descent**.

Definition D.2.8 ((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:*

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla f(\mathbf{x}_{t-1}) \tag{E.2.26}$$

We will not be concerned with the problem of finding an appropriate step size, which we will just 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 efficient algorithm to this end in Chapter 6.

2.3.1 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 to go well, proceed with no hesitation to the next section.

Definition D.2.9 (Minimum) *A **local minimum** of $f(\mathbf{x})$ is a point $\mathbf{x}^+$ such that the*

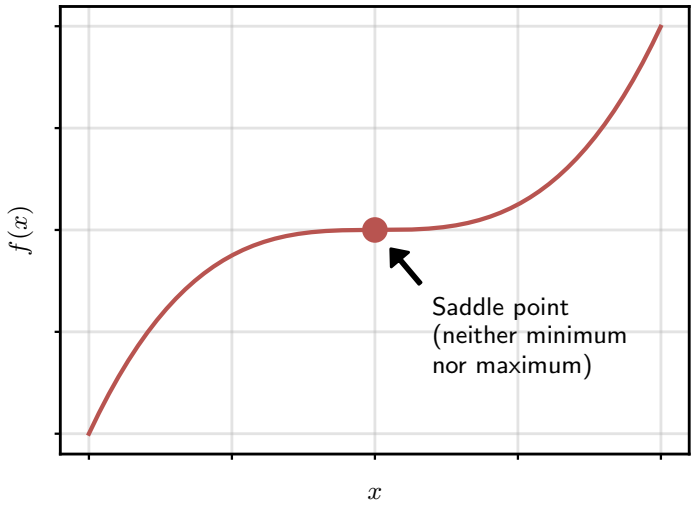


Figure F.2.3: *Simple example of a saddle point (try visualizing the tangent line in that point to see it is indeed stationary).*

following is true for some $\varepsilon > 0$*:*

$$f(\mathbf{x}^+) \leq f(\mathbf{x}) \quad \forall \mathbf{x} : \|\mathbf{x} - \mathbf{x}^+\| < \varepsilon$$

Ball of size ε *centered in* $\mathbf{x}^+$

In words, the value of $f(\mathbf{x}^+)$ is a minimum if we consider a sufficiently small neighborhood of $\mathbf{x}^+$. Intuitively, in 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 D.2.10 (Stationary points) *A* ***stationary point*** *of* $f(\mathbf{x})$ *is a point* $\mathbf{x}^+$ *such that* $\nabla f(\mathbf{x}^+) = 0$.

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

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 the function in that point attains its lowest possible value across the entire domain: we say this is a **global minimum**.

Definition D.2.11 (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.

Definition D.2.12 (Convex function) *A function $f(\mathbf{x})$ is convex if for any two points $\mathbf{x}_1$ and $\mathbf{x}_2$ and $\alpha \in [0,1]$ we have:*

$$f(\underbrace{\alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2}_{\text{Interval from } \mathbf{x}_1 \text{ to } \mathbf{x}_2}) \leq \overbrace{\alpha f(\mathbf{x}_1) + (1-\alpha) f(\mathbf{x}_2)}^{\text{Line segment from } f(\mathbf{x}_1) \text{ to } f(\mathbf{x}_2)} \tag{E.2.27}$$

The left-hand side in (E.2.27) is the value of f on any point inside the interval ranging from $\mathbf{x}_1$ to $\mathbf{x}_2$, while the right-hand side is the corresponding value on a line 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 [JK+17]:

1. For a generic *non-convex* function, gradient descent converges to a *stationary point*. Nothing more can be said unless we look at higher-order derivatives (derivatives of the derivatives).

2. For a *convex* function, gradient descent will converge to a *global minimum*, irrespective of initialization.

3. If the inequality in (E.2.27) is satisfied in a strict way (**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 infinite times from any possible initialization, turning it into an NP-hard task [JK+17].

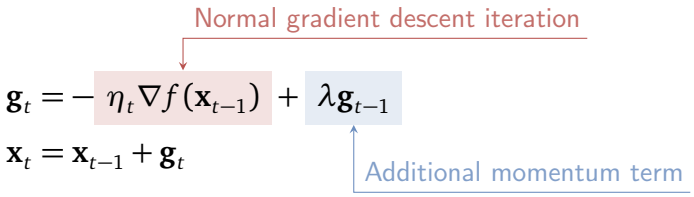
This discussion has a strong historical significance. As we will see in Chapter 5, any

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 such restriction, in the sense that their optimization, when started from a reasonable initialization, converge to points with good empirical performance.

2.3.2 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 9 (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 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 (e.g., the Hessian), or multiple calls to the function.

We describe here one such technique, **momentum**, and we refer to [ZLLS23, Chapter 12], for a broader introduction.[8] 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:



where we initialize $\mathbf{g}_0 = \mathbf{0}$. See Figure F.2.4 for an example.

The coefficient λ determines how much the previous term is dampened. In fact, un-

[8] See also this 2016 blog post by S. Ruder: https://www.ruder.io/optimizing-gradient-descent/.

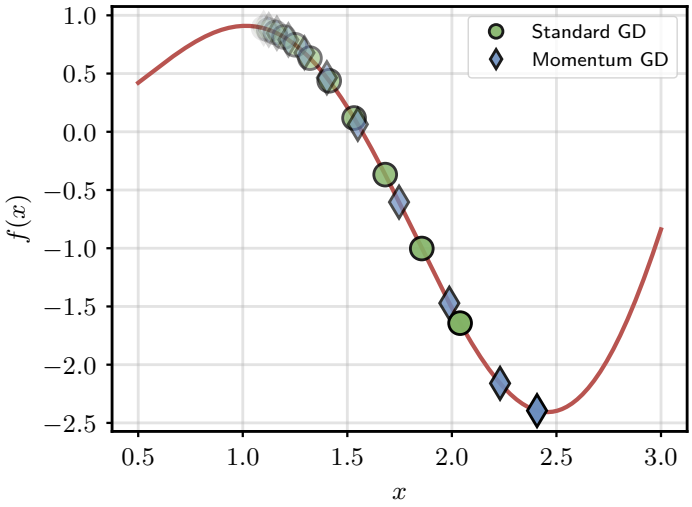


Figure F.2.4: *First iterations of standard GD and GD with momentum when minimizing* $f(x) = x\sin(2x)$ *starting from* $x = 1 + \varepsilon$*, with* $\lambda = 0.3$.

rolling two terms:

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

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 [SMDH13]. Another common technique is adapting the step size for each parameter based on the gradients' magnitude [ZLLS23]. A common optimization algorithm combining several of these ideas is Adam [KB15]. One advantage of Adam is that it is found to be relatively robust to the choice of its **hyper-parameters**,[9] with the default choice in most frameworks being a good starting point in the majority of cases.

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 in terms of memory, as we will see in Section 6.3).

[9] A hyper-parameter is a parameter which is selected by the user, as opposed to being learnt by gradient descent.

From theory to practice

About the exercises

This book does not have classical end-of-chapter exercises, which are covered in many existing textbooks. By contrast, I describe a self-learning path to help you explore two frameworks (JAX and PyTorch) as you progress in the book. Solutions to the more practical exercises will be published on the book's website.[a] These sections are full of URLs linking to online material – they might be expired or moved by the time you search for them.

[a] https://www.sscardapane.it/alice-book

Starting from the basics: NumPy

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 long as functions to index and transform their content. You can read more on the library's quick start.[10] You should feel comfortable in handling arrays in NumPy, most notably for their indexing: the `rougier/numpy-100`[11] repository provides a nice, slow-paced series of exercises to test your knowledge.

Moving to a realistic framework

Despite its influence, NumPy is limited, in particular in his support for parallel hardware such as GPUs (unless additional libraries are used), and for his lack of automatic differentiation (introduced in Chapter 6). JAX replicates the NumPy's interface while adding extended hardware support, the automatic computation of gradients, and additional transformations such as the **vectorized map** (`jax.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 several high-level utilities for building differentiable models. For the purposes of this chapter, you can skim the documentation of `jax.numpy.array` and `torch.tensor` to understand how

[10] https://numpy.org/doc/stable/user/quickstart.html
[11] https://github.com/rougier/numpy-100

much they have in common with NumPy. For now, you can ignore high-level modules such as `torch.nn`. 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, PyTorch), I suggest to replicate the exercise below thrice – each variant should only take a few minutes if you know the syntax. Consider a 2D function $f(\mathbf{x})$, $\mathbf{x} \sim (2)$, where we take the domain to be $[0, 10]$:[12]

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

Before proceeding in the book, repeat this for each framework:

1. Implement the function in a **vectorized** way, i.e., given a matrix $\mathbf{X} \sim (n, 2)$ of n inputs, it should return a vector $f(\mathbf{X}) \sim (n)$ where $[f(\mathbf{X})]_i = f(\mathbf{X}_i)$.
2. Implement another function to compute its gradient (hard-coded – we have not touched automatic differentiation yet).
3. Write a basic gradient descent procedure and visualize the paths taken by the optimization process from multiple starting points.
4. 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 (3) is a good place to experiment with `vmap` for vectorizing a function.

[12] I asked ChatGPT to generate a nice function with several minima and maxima. Nothing else in the book is LLM-generated, which I feel is becoming an important disclaimer to make.

3 | Datasets and losses

About this chapter

This chapter formalizes the supervised learning scenario. We introduce the concepts of datasets, loss functions, and empirical risk minimization, stressing 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.

3.1 What is a dataset?

We consider a scenario in which manually coding a certain function is unfeasible (e.g., recognizing objects from 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 D.3.1 (Dataset) *A **supervised dataset** $\mathscr{S}_n$ of size n is a set of n pairs $\mathscr{S}_n = \{(x_i, y_i)\}_{i=1}^n$, 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 **identically** and **independently** distributed (i.i.d.) draw from some unknown (and unknowable) probability distribution $p(x, y)$.*

See Appendix A 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 collecting 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.

Similarly, samples being **independently distributed** means that our dataset has no bias in its collection, and 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 loosing on images of other makers 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 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 what tasks are truly *in-distribution* against what is *out-of-distribution* (and how much the models are able to generalize) becomes blurred [YCC+24].

3.1.1 Variants of supervised learning

There exists 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 available **targets** y_i, 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 between clusters are dissimilar [HTF09]. As another example, in a **retrieval** system we may want to search a large database for the top-k most similar elements to a user-given query.

When dealing with complex data such as images, this is non-trivial because distances on images are ill-defined if we operate on pixels (i.e., 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, e.g., 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

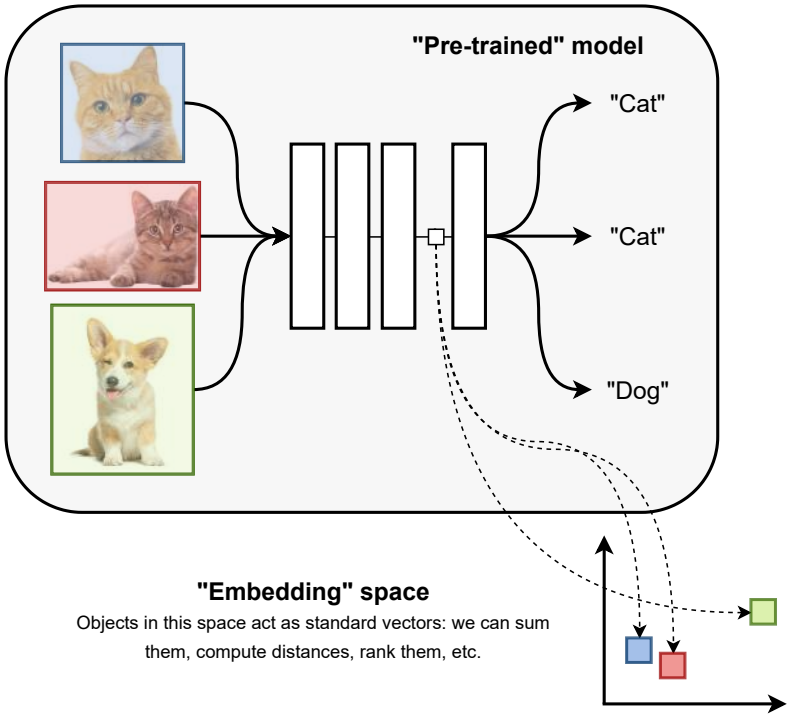


Figure F.3.1: *Differentiable models process data by transforming it sequentially via linear algebra operations. In many cases, after we optimize these programs, the internal representations of the input data of the model (what we call a **pre-trained** model) have geometric properties: for example, semantically similar images are projected to points that are close in this "latent" space. Transforming data from a non-metric space (original input images) to a metric space (bottom right) is called **embedding** the data.*

standard clustering models, such as Gaussian mixture models [HHWW14]. See Figure F.3.1 for 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, [ZJM+21]). The aim of SSL is to automatically find some supervised objective from 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 [RWC+19]. 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 [MSC+13].[1]

[1]Large-scale web datasets are also full of biases, profanity, and vulgar content. Recognizing that

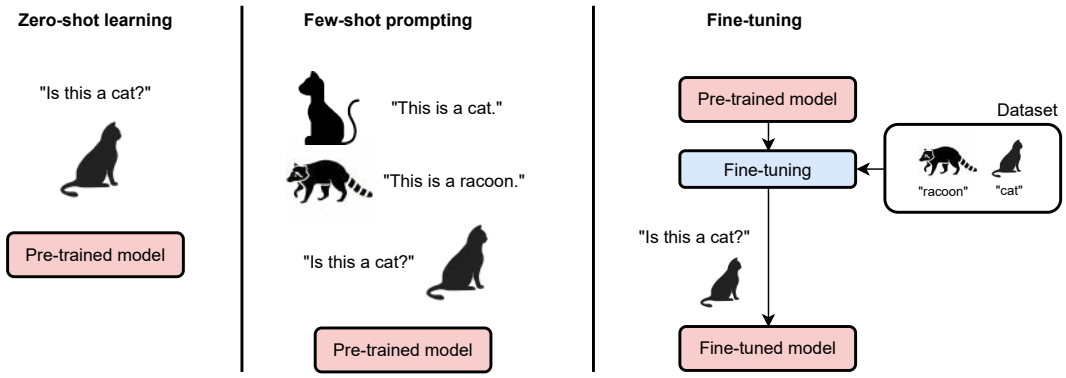


Figure F.3.2: *Three ways of using trained models.* ***Zero-shot****: a question is directly given to the model. This can be achieved with generative language models (introduced in Chapter 8).* ***Few-shot prompting*** *is similar, but a few examples are provided as input. Both techniques can be employed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

As we will see in Chapter 8 and Chapter 10, LLMs can be seen as modern iterations on this basic idea, since optimizing models such as GPT or Llama [TLI+23] always start by a 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 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. See Figure F.3.2 for a comparison of 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.[2] 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) [LDR23].[3] We will consider PEFT

models trained on this data internalize these biases was another important realization [BCZ+16] and it is one of the major criticisms of closed-source foundation models [BGMMS21].

[2] https://huggingface.co/models

[3] 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** [ASA+23].

techniques in the next volume.

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 [BNS06]. 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 period of times, giving rise to countless tasks depending on the order in which the tasks or the data are provided, including **domain adaptation**, **meta-learning** [FAL17], **continual learning** [PKP+19, BBCJ20], **metric learning**, **unlearning**, etc. Some of these will be treated in the next volume.

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 suggestions of the recommender system itself. This creates **feedback loops** [CMMB22] that require constant re-evaluation of the performance of the system and of the app.

3.2 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**.

Definition D.3.2 (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, i.e., $l(y, \hat{y}_1) < l(y, \hat{y}_2)$ means that the prediction $\hat{y}_1$ is better than the prediction $\hat{y}_2$ when considering the reference value (target) y.*

A loss function embeds our understanding of the task and our preferences in the solutions' space 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 op-

timization problem that can be solved via gradient descent by minimizing the average loss on our dataset.

To this end, given a dataset $\mathscr{S}_n = \{(x_i, y_i)\}$ and a loss function $l(\cdot,\cdot)$, a sensible optimization task to solve is the minimum average loss on the dataset achievable by any possible *differentiable* model f:

Average over the dataset

$$f^* = \arg\min_f \frac{1}{n}\sum_{i=1}^{n} l(y_i,\ f(x_i)\ ) \tag{E.3.1}$$

Prediction of f on the i-th sample from the dataset

For historical reasons, (E.3.1) is referred to as **empirical risk minimization** (ERM), where *risk* is used as a generic synonym for *loss*. See also the box in the next page for more on the origin of the term.

In (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 **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 $\mathscr{S}_n$, a loss function l, and a model space f, we can **train** our model by optimizing the empirical risk (E.3.1) via gradient descent (E.2.26):

$$w^* = \arg\min_w \frac{1}{n}\sum_{i=1}^{n} l(y_i, f(x_i, w)) \tag{E.3.2}$$

where the minimization is now done with respect to the parameter's 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 (that we will introduce properly in the next chapter), where $y \in \{-1, +1\}$ 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:

$$l(y, \hat{y}) = \begin{cases} 0 & \text{if } \operatorname{sign}(\hat{y}) = y \\ 1 & \text{otherwise} \end{cases} \tag{E.3.3}$$

While this aligns with our intuitive notion of "being right", it is useless as loss function since its gradient will almost always be zero (except when 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 [negative] depending on whether the sign of the model aligns [or does not align] with the desired one, but it varies continuously differently from 0/1 loss in (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.

3.2.1 Expected risk and overfitting

As a second observation, note that the empirical risk is always trivial to minimize, by defining:

$$f(x) = \begin{cases} y & \text{if } (x, y) \in \mathcal{S}_n \\ \bar{y} & \text{otherwise} \end{cases}. \tag{E.3.4}$$

x is in the training set

Default value, e.g., 0

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 $\bar{y}$ (e.g., 0) otherwise. Assuming that the loss is lower-bounded 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** (optimization). Although we search for a model by optimizing some average loss quantity on our training data, as in (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.

Definition D.3.3 (Expected risk) *Given a probability distribution $p(x, y)$ and a loss function l, the **expected risk** (ER) is defined as:*

$$\text{ER}[f] = \mathbb{E}_{p(x,y)}[l(y, f(x))] \tag{E.3.5}$$

Minimizing (E.3.5) can be interpreted as minimizing the average (expected) loss across *all possible input-output pairs* (e.g., 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 (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 ER term.

The difference in loss between the expected and the empirical risk is called the **generalization gap**: a pure memorization algorithm like (E.3.4) will have poor generalization 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, $\mathcal{S}_n \cap \mathcal{T}_m = \emptyset$. Then, the difference in empirical loss between $\mathcal{S}_n$ and $\mathcal{T}_m$ can be used as an approximate measure of overfitting.

Risk and loss

Empirical and expected risk minimization framed in this way are generally associated with the work of the Russian computer scientist V. Vapnik [Vap13], which gave rise to the field of *statistical learning theory* (SLT). SLT is especially concerned with the behaviour of (E.3.1) when seen as a finite-sample approximation of (E.3.5) under some restricted class of functions f and measure of underlying complexity [PS+03, SSBD14, MRT18]. 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 SLT [PBL20]. See also the introduction of Chapter 9.

3.2.2 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 Appendix A, especially probability distributions, sufficient statistics, and maximum likelihood estimation.

As we will see in the next chapters, the loss encodes our a priori knowledge on the task to be solved, and it has a large impact on performance. In some cases, simple considerations on the problem are enough to design valid losses (e.g., as done for the hinge loss in Section 3.2).

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 [BB23]. 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 $p(x, y) = p(x)p(y \mid x)$, such that $p(x)$ depends on the probability of observing each input x, and the conditional term $p(y \mid x)$ describes the probability of observing a certain output y given an input x.[4] Approximating $p(y \mid x)$ with a function $f(x)$ makes sense if we assume that the probability mass is mostly concentrated around a single point y, i.e., $p(y \mid x)$ is close to a so-called Dirac delta function, and it drastically simplifies the overall problem formulation.

However, we can relax this by assuming that our model $f(x)$ does not provide directly the prediction, but it 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 $y \in \{1, 2, 3\}$ can take three possible values. We can assume our model has three outputs that parameterize a categorical distribution over these classes, such that:

$$p(\mathbf{y} \mid f(x)) = \prod_{i=1}^{3} f_i(x)^{y_i}$$

where $\mathbf{y} \sim \text{Binary}(3)$ is the one-hot encoding of the class y[5] 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 $y \in \mathbb{R}$ (**regression**). We can model this with a two-valued function $f(x) \sim (2)$ such that the prediction is a Gaussian with appropriate mean and variance:

[4] We can also decompose it as $p(x, y) = p(x \mid y)p(y)$. Methods that require to estimate $p(x)$ or $p(x \mid y)$ are called **generative**, while methods that estimate $p(y \mid x)$ are called **discriminative**. Apart from language modeling, in this book we focus on the latter case. We consider generative modelling more broadly in the next volume.

[5] Given an integer i, its one-hot representation is a vector of all zeros except the i-th element, which is 1. This is introduced formally in Section 4.2.

$$p(y \mid f(x)) = \mathcal{N}(y \mid f_1(x), \underbrace{f_2^2(x)}_{\text{Squared to ensure positivity}}) \tag{E.3.6}$$

where 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 $p(y \mid x)$ 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 (E.3.6).

3.2.3 Maximum likelihood

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 f of function is given by the product of each sample in the dataset:

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

The quantity $p(\mathcal{S}_n \mid f)$ 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 (see Section A.6 in Appendix A).

Definition D.3.4 (Supervised learning as maximum likelihood) *Given a dataset $\mathcal{S}_n = \{(x_i, y_i)\}$ and a family of probability distributions $p(y \mid f(x))$ parameterized by $f(x)$, the **maximum likelihood** (ML) solution is given by:*

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

While we are again left with an optimization problem, it now follows directly from the

laws of probability once all probability distributions are chosen, which is 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:

$$\arg\max_f \left\{ \log \prod_{i=1}^{n} p(y_i \mid f(x_i)) \right\} = \arg\min_f \left\{ \sum_{i=1}^{n} -\log(p(y_i \mid f(x_i)) \right\}$$

Hence, the two formulations are identical if we identify $-\log(p(y \mid f(x))$ as a "pseudo-loss" to be optimized. As we will see, all loss functions used in practice can be obtained under the ML 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.

3.3 Even more probability: 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, e.g., [JLB+22], for more details.

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 (i.e., 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 than more than a single parameterization across the entire model's 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 (once again, remember than in the rest of the book f will be described by a finite set of parameters, in which case the prior $p(f)$ would be a prior over these weights). For example, we will see that in many situations functions with smaller norm are preferred (as they are more stable), in which case we could define a prior $p(f) \propto \frac{1}{\|f\|}$ for some norm $\|f\|$ of f.

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**:

$$p(f \mid \mathscr{S}_n) = \frac{p(\mathscr{S}_n \mid f)\, p(f)}{p(\mathscr{S}_n)} \tag{E.3.7}$$

Prior (*before* observing the dataset)

Posterior (*after* observing the dataset)

The term $p(f \mid \mathscr{S}_n)$ is called the **posterior distribution function**, while the term $p(\mathscr{S}_n)$ 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. Differently from before, the distribution can encode 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's weight:

$$p(y \mid x) = \int_f p(y \mid f(x))\; p(f \mid \mathscr{S}_n) \approx \frac{1}{k} \sum_{i=1}^{k} p(y \mid f_i(x)) p(f_i \mid \mathscr{S}_n) \tag{E.3.8}$$

Prediction of $f(x)$

Weight assigned to f

Monte Carlo approximation

where on the right term of (E.3.8) we have approximated the integral with a Monte Carlo average over k random samples from the posterior distribution $f_k \sim p(f \mid \mathscr{S}_n)$. 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 [Bis06]. Lacking this, one is forced to approximated solutions, either by **Markov chain Monte Carlo** or by **variational inference** [JLB+22]. We will see in Section 9.3.1 one example of Bayesian treatment of the model's parameters called **Monte Carlo dropout**.

We remark on two interesting facts about the posterior before closing this section. First, suppose we are only interested about the function having highest posterior density. In this case, the evidence term can be ignored and the solution decomposed into two separate terms:

$$f^* = \arg\max_f p(\mathscr{S}_n \mid f)p(f) = \tag{E.3.9}$$

$$\arg\max_f \left\{ \log p(\mathscr{S}_n \mid f) + \log p(f) \right\} \tag{E.3.10}$$

Likelihood term

Regularization term

This is called the **maximum a posteriori** (MAP) solution. If all functions have the same weight a priori (i.e., $p(f)$ is uniform over the function's space), then the second term is a constant and the problem reduces to the maximum likelihood solution. In general, however, the MAP solution will impose a penalty to functions deviating too much from our prior distribution. We will see this is a useful idea to combat overfitting and impose specific constraints on the function f. The term $\log p(f)$ is generally called a **regularizer** over the function's space as it pushes the solution towards the basin of attraction defined by the prior distribution.[6]

Second, the full Bayesian treatment provides a simple way to incorporate new data, e.g., a new dataset $\mathscr{S}'_n$ from the same distribution. To do that, we replace the prior function in (E.3.7) 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 which gets updated by looking at new data.[7] This can mitigate issues when training models online, most notably the so-called *catastrophic forgetting* of old information [KPR+17].

[6] The difference between maximum likelihood and maximum a posteriori solutions is loosely connected to the difference between the **frequentist** and **Bayesian** interpretation of probability [Hac19], i.e., probabilities as frequency of events or probabilities as a measure of uncertainty. From a very high-level point of view, ML 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.

[7] Think of the original prior function as the distribution on f after having observed an initial *empty set* of values.

4 | Linear models

About this chapter

Programming software 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 possible block, so-called 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.

4.1 Least-squares regression

4.1.1 Problem setup

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, namely:

- The input is a vector $\mathbf{x} \sim (c)$, corresponding to a set of features (e.g., 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, i.e., $y \in \{1, \dots, m\}$, 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

Table T.4.1: *Basic shapes to remember for this chapter. For uniformity, we will use the same letters as much as possible throughout the book.*

n	**size of the dataset**
c	**features**
m	**classes**

in some cases, most notably **least-squares regression** (Section 4.1.3).

The basic shapes to remember are summarized in Table T.4.1. 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.

4.1.2 Regression losses: 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 $\hat{y} = f(x)$ 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**:

$$l(\hat{y}, y) = (\hat{y} - y)^2 \tag{E.4.1}$$

Here and in the following we use the symbol $\hat{y}$ to denote the prediction of a generic model. As we will see, working with (E.4.1) provides a number of interesting 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 it in closed-form for the optimal solution.

Recalling the maximum likelihood principle (Section A.6), 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:

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

In this case the log-likelihood (for a single point) can be written as:[1]

[1] Recalling that $\log(ab) = \log(a) + \log(b)$ and $\log(a^b) = b \log(a)$.

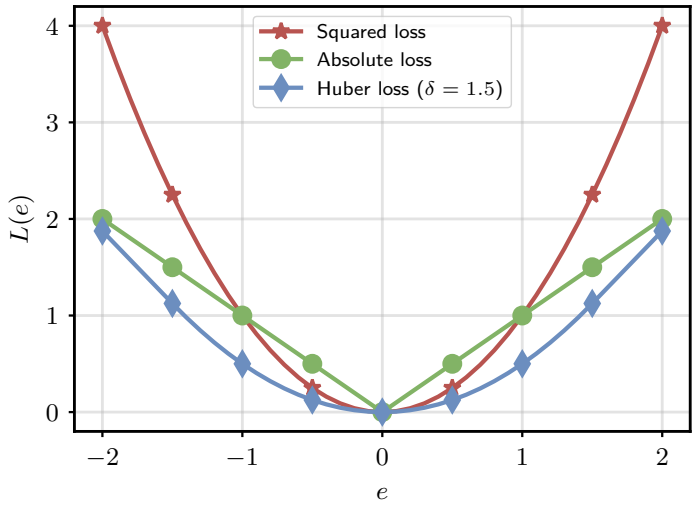


Figure F.4.1: *Visualization of the squared loss, the absolute loss, and the Huber loss with respect to the prediction error $e = (\hat{y} - y)$.*

$$\log(p(y \mid f(\mathbf{x}), \sigma^2)) = -\log(\sigma) - \frac{1}{2}\log(2\pi) - \frac{1}{2\sigma^2}(y - f(\mathbf{x}))^2 \tag{E.4.2}$$

Minimizing (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 from the optimization of f, with a simple closed-form solution (see below, equation (E.4.9)).

Coming up with variants to the squared loss is also easy. For example, one drawback of the squared loss is that higher errors will be penalized with a strength that grows quadratically in the error, which may provide undue influence to **outliers**, i.e., points that are badly mislabeled. Other choices that diminish the influence of outliers can be the **absolute value loss** $l(\hat{y}, y) = |\hat{y} - y|$ or the Huber loss (a combination of the squared loss and the absolute loss):

$$\text{Huber loss: } L(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq 1 \\ \left(|y - \hat{y}| - \frac{1}{2}\right) & \text{otherwise} \end{cases} \tag{E.4.3}$$

which is quadratic in the promixity of 0 error, and linear otherwise (with the $-\frac{1}{2}$ term added to ensure continuity). See Figure F.4.1 for a visualization of these losses with respect to the prediction error.

The absolute loss seems an invalid choice in our context, since it has a point of non-

differentiability in 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 derivatives of $|\varepsilon|$ for any $\varepsilon > 0$ is always defined.

4.1.3 The least-squares model

With a loss function in hand, we consider the following model (a linear model) to complete the specification of our first supervised learning problem.

Definition D.4.1 (Linear models) *A **linear model** on an input* $\mathbf{x}$ *is defined as:*

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$

where $\mathbf{w} \sim (c)$ *and* $b \in \mathbb{R}$ *(the bias) are trainable parameters.*

The intuition is that the model assigns a fixed weight w_i to each input feature x_i, and provides a prediction by linearly summing all the effects for a given input $\mathbf{x}$, reverting to a default prediction equal to b whenever $\mathbf{x} = \mathbf{0}$. Geometrically, the model defines a line for $d = 1$, a plane for $d = 2$, and a generic hyperplane for $d > 1$. From a notational perspective, we can sometimes avoid writing a bias term by assuming a constant term of 1 as the last feature of $\mathbf{x}$:

$$f\left(\begin{bmatrix}\mathbf{x}\\1\end{bmatrix}\right) = \mathbf{w}^\top \begin{bmatrix}\mathbf{x}\\1\end{bmatrix} = \mathbf{w}_{1:c}^\top \mathbf{x} + w_{c+1}$$

Combining the linear model, the squared loss, and an empirical risk minimization problem we obtain the **least-squares optimization problem**.

Definition D.4.2 (Least-squares) *The **least-squares** optimization problem is given by:*

$$\mathbf{w}^*, b^* = \arg\min_{\mathbf{w},b} \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \mathbf{w}^\top \mathbf{x}_i - b\right)^2 \tag{E.4.4}$$

Before proceeding to the analysis of this problem, we rewrite the least-squares in a **vec-**

```python
def linear_model(w: Float[Tensor, "c"],
                 b: Float,
                 X: Float[Tensor, "n c"])
                 -> Float[Tensor, "n"]:
  return X @ w + b
```

Box C.4.1: *Computing a batched linear model as in* (E.4.5). *For clarity, we are showing the array dimensions as type hints using* `jaxtyping` (*https://docs.kidger.site/jaxtyping/*).

torized form that only involves matrix operations (matrix products and norms). This is useful because, as already stated, modern code for training differentiable models is built around n-dimensional arrays, with optimized hardware to perform matrix operations on them. To this end, we first stack all the inputs and outputs of our training set into an **input matrix**:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix} \sim (n, c)$$

and a similar **output vector** $y = [y_1, \ldots, y_n]^\top$. We can write a batched model output (the model output for a mini-batch of values) as:

$$f(\mathbf{X}) = \mathbf{X}\mathbf{w} + \mathbf{1}b \tag{E.4.5}$$

Same bias b for all n predictions

Equations like (E.4.5) can be replicated almost line-by-line in code - see Box C.4.1 for an example in PyTorch.

Of only marginal interest for now but of more importance for later, we note that the row ordering of the input matrix and of the output vector are fundamentally arbitrary, in the sense that permuting their rows will only result in a corresponding permutation of the rows of $f(\mathbf{X})$. This is a simple example of a phenomenon called **permutation equivariance** that will play a much more important role later on.

```
def least_squares_solve(w: Float[Tensor, "c"],
                        X: Float[Tensor, "n c"],
                        y: Float[Tensor, "n"],
                        numerically_stable = True) \
                        -> Float[Tensor, "c"]:
  # Explicit solution
  if not numerically_stable:
    return torch.linalg.inv(X.T @ X) @ X.T @ y
  else:
    return torch.linalg.solve(X.T @ X, X.T @ y)
```

Box C.4.2: *Solving the least-squares problem with the closed-form solution. The numerically stable variant calls a solver specialized for systems of linear equations.*

The vectorized least-squares optimization problem becomes:

$$\mathrm{LS}(\mathbf{w}, b) = \frac{1}{n} \|\mathbf{y} - \mathbf{X}\mathbf{w} - \mathbf{1}b\|^2 \tag{E.4.6}$$

where we recall that the norm of a vector is defined as $\|\mathbf{e}\|^2 = \sum_i e_i^2$.

4.1.4 Solving the least-squares problem

To solve the least-squares problem through gradient descent, we need the equations for its gradient. Although we will soon develop a general algorithmic framework to compute these gradients automatically (Chapter 6), it is instructive to look at the gradient itself in this simple scenario. Ignoring the bias (for the reasons stated above, we can incorporate it in the weight vector), and other constant terms we have:

$$\nabla \mathrm{LS}(\mathbf{w}) = \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y})$$

The LS problem is convex in the weights of the model, as can be understood informally by noting that the equations describe a paraboloid in the space of the weights (a quadratic function). The global minima are then described by the equations:

$$\mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \;\Rightarrow\; \mathbf{X}^\top \mathbf{X}\mathbf{w} = \mathbf{X}^\top \mathbf{y}$$

These are called the **normal equations**. Importantly, the normal equations describe

a linear system of equations in $\mathbf{w}$,[2] meaning that under the appropriate conditions (invertibility of $\mathbf{X}^\top\mathbf{X}$) we can solve for the optimal solution as:

$$\mathbf{w}_* = \left(\mathbf{X}^\top\mathbf{X}\right)^{-1}\mathbf{X}^\top\mathbf{y} \tag{E.4.7}$$

Tidbits of information

The matrix $\mathbf{X}^\dagger = \left(\mathbf{X}^\top\mathbf{X}\right)^{-1}\mathbf{X}^\top$ is called the **pseudoinverse** (or **Moore-Penrose inverse**) of the non-square matrix $\mathbf{X}$, since $\mathbf{X}^\dagger\mathbf{X} = \mathbf{I}$. Performing the inversion in (E.4.7) is not always possible: for example, if one feature is a scalar multiple of the other, the matrix $\mathbf{X}$ does not have full rank (this is called **collinearity**). Finally, note that the predictions of the least-squares model can be written as $\hat{\mathbf{y}} = \mathbf{M}\mathbf{y}$, with $\mathbf{M} = \mathbf{X}\mathbf{X}^\dagger$. Hence, least-squares can also be interpreted as performing a weighted average of the training labels, where the weights are given by a projection on the column space induced by $\mathbf{X}$. This is called the **dual** formulation of least-squares. Dual formulations provide an intrinsic level of debugging of the model, as they allow to check which inputs were the most relevant for a prediction by checking the corresponding dual weights [ICS22].

This is the only case in which we will be able to express the optimal solution in a closed-form way, and it is instructive to compare this solution with the gradient descent one. To this end, we show in Box C.4.2 an example of solving the least-squares in closed form using (E.4.7), and in Box C.4.3 the equivalent gradient descent formulation. A prototypical evolution of the loss in the latter case is plotted in Figure F.4.2. Since we selected a very small learning rate, each step in the gradient descent procedure provides a stable decrease in the loss, until convergence. Practically, convergence could be checked by numerical means, e.g., by evaluating the difference in norm between two iterations for some numerical threshold $\varepsilon > 0$:

$$\|\mathbf{w}_{t+1} - \mathbf{w}_t\|^2 < \varepsilon \tag{E.4.8}$$

As we will see, understanding when more complex models have converged will be a more subtle task.

Considering again the Gaussian log-likelihood in (E.4.2), we can also optimize the term

[2] That is, we can write them as $\mathbf{A}\mathbf{w} = \mathbf{b}$, with $\mathbf{A} = \mathbf{X}^\top\mathbf{X}$ and $\mathbf{b} = \mathbf{X}^\top\mathbf{y}$.

```python
def least_squares_gd(X: Float[Tensor, "n c"],
                     y: Float[Tensor, "n"],
                     learning_rate=1e-3) \
                     -> Float[Tensor, "c"]:
    # Initializing the parameters
    w = torch.randn((X.shape[1], 1))

    # Fixed number of iterations
    for i in range(15000):
      # Note the sign: the derivative has a minus!
      w = w + learning_rate * X.T @ (y - X @ w)

    return w
```

Box C.4.3: *Same task as Box C.4.2, solved with a naive implementation of gradient descent with a fixed learning rate that defaults to* $\eta = 0.001$.

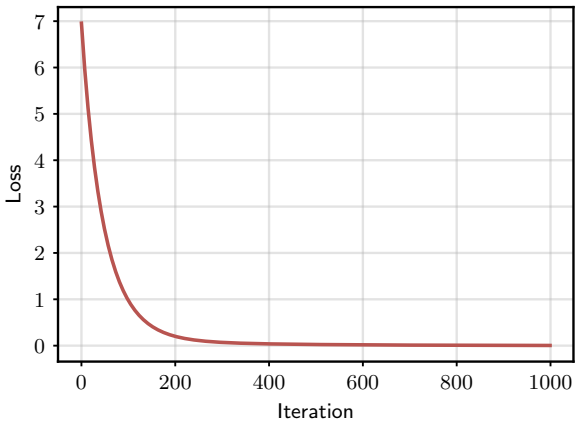


Figure F.4.2: *An example of running code from Box C.4.2, where the data is composed of* $n = 10$ *points drawn from a linear model* $\mathbf{w}^\top \mathbf{x} + \varepsilon$, *with* $w_i \sim \mathcal{N}(0,1)$ *and* $\varepsilon \sim \mathcal{N}(0, 0.01)$. *Details apart, note the very smooth descent: each step provides a decrease in loss.*

with respect to σ^2 once the weights have been trained, obtaining:

$$\sigma_*^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}_*^\top \mathbf{x}_i)^2 \,. \tag{E.4.9}$$

which has the intuitive meaning that the variance of the model is constant (by definition) and given by the average squared prediction error on our training data. More sophisticated probabilistic models can be obtained by assuming the variance itself is predicted by the model (**heteroscedastic** models), see [Bis06].

4.1.5 Some computational considerations

Even if the inverse can be computed, the quality of the solution will depend on the condition number of $\mathbf{X}^\top\mathbf{X}$, and large numerical errors can occur for poorly conditioned matrices.[3] In addition, the computational cost of solving (E.4.7) may be prohibitive. The matrix inversion will scale, roughly, as $\mathcal{O}(c^3)$. As for the matrix multiplications, the algorithm requires a multiplication of a $c \times n$ matrix with another $n \times c$ one, and a multiplication between a $c \times c$ matrix and a $c \times n$ one. Both these operations will scale as $\mathcal{O}(c^2 n)$.

In general, we will always prefer algorithms that scale linearly both in the feature dimension c and in the batch size n, since super-linear algorithms will become quickly impractical (e.g., a batch of 32 RGB images of size 1024×1024 has $c \approx 1e^7$). We can avoid a quadratic complexity in the equation of the gradient by computing the multiplications in the correct order, i.e., computing the matrix-vector product $\mathbf{Xw}$ first. Hence, pure gradient descent is linear in both c and n, but only if proper care is taken in the implementation: generalizing this idea is the fundamental insight for the development of **reverse-mode automatic differentiation**, a.k.a. **back-propagation** (Section 6.3).

4.1.6 Regularizing the least-squares solution

Looking again at the potential instability of the inversion operation, suppose we have a dataset for which the matrix is almost singular, but we still wish to proceed with the closed-form solution. In that case, it is possible to slightly modify the problem to achieve a solution which is "as close as possible" to the original one, while being feasible to compute. For example, a known trick is to add a small multiple, $\lambda > 0$, of the identity matrix to the matrix being inverted:

$$\mathbf{w}^* = \left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^\top\mathbf{y}$$

This pushes the matrix to be "more diagonal" and improves its condition number. Backtracking to the original problem, we note this is the closed form solution of a modified optimization problem:

$$\text{LS-Reg}(\mathbf{w}) = \frac{2}{n}\|\mathbf{y} - \mathbf{Xw}\|^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

[3] The condition number of a matrix $\mathbf{A}$ is defined as $\kappa(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|$ for some choice of matrix norm $\|\bullet\|$. Large conditions number can make the inversion difficult, especially if the floating-point precision is not high.

This problem is called **regularized least-squares** (or **ridge regression**), and the red part in the loss is an instance of ℓ_2-regularization (or, more generally, regularization). Note that regularization does not depend on the dataset, as it simply encodes a preference for a certain type of solution (in this case, low-norm weights), where the strength of the preference itself is defined by the hyper-parameter λ. From a Bayesian perspective (Section 3.3), the regularized least-squares corresponds to a MAP solution when defining a Gaussian prior over the weights centered in zero with constant variance.

4.2 Linear models for classification

We now move to classification, in which $y_i \in \{1, \dots, m\}$, where m defines the number of **classes**. As we will see later, this is a widely influential problem, encompassing a range of tasks in both computer vision (e.g., image classification) and natural language processing (e.g., next-token prediction). We can tackle this problem by slight variations with respect to the regression case.

While we can solve the task by regressing directly on the integer value y_i, it is instructive to consider why this might not be a good idea. First, it is difficult for a model to directly predict an integer value, since this requires some thresholding that would render its gradient zero almost everywhere. Instead, we could regress on a real value $\tilde{y}_i \in [1, m]$ inside the interval from 1 to m (as we will show, bounding the output of the model inside an interval can be done easily). During inference, given the output $\hat{y}_i = f(\mathbf{x}_i)$, we map back to the original domain by rounding:

$$\text{Predicted class} = \text{round}(\hat{y}_i)$$

For example, $\hat{y}_i = 1.3$ would be mapped to class 1, while $\hat{y}_i = 3.7$ would be mapped to class 4. Note that this is a post-hoc processing of the values that is only feasible at inference time. The reason this is not a good modelling choice is that we are introducing a spurious ordering of the classes which might be exploited by the model itself, where class 2 is "closer" to class 3 than it is to class 4. We can avoid this by moving to a classical **one-hot encoded** version of y, which we denote by $\mathbf{y}^{\text{oh}} \sim \text{Binary}(m)$:

$$[\mathbf{y}^{\text{oh}}]_j = \begin{cases} 1 & \text{if } y = j \\ 0 & \text{otherwise} \end{cases}$$

For example, in the case of three classes, we would have $\mathbf{y}^{\text{oh}} = [1 \;\; 0 \;\; 0]^\top$ for class 1, $\mathbf{y}^{\text{oh}} = [0 \;\; 1 \;\; 0]^\top$ for class 2, and $\mathbf{y}^{\text{oh}} = [0 \;\; 0 \;\; 1]^\top$ for class 3 (this representation should

be familiar to readers with some background in machine learning, as it is a standard representation for categorical variables).

One-hot vectors are unordered, in the sense that given two generic outputs $\mathbf{y}_1^{\text{oh}}$ and $\mathbf{y}_2^{\text{oh}}$, their Euclidean distance is either 0 (same class) or $\sqrt{2}$ (different classes). While we can perform a multi-valued regression directly on the one-hot encoded outputs, with the mean-squared error known as the **Brier score** in this case, we show below that a better and more elegant solution exists, in the form of **logistic regression**.

4.2.1 The probability simplex and the softmax function

We cannot train a model to directly predict a one-hot encoded vector (for the same reasons described above), but we can achieve something similar by a slight relaxation. To this end, we re-introduce the **probability simplex**.

> **Definition D.4.3 (Probability simplex)** *The **probability simplex** Δ_n is the set of vectors $\mathbf{x} \sim \Delta(n)$ such that:*
>
> $$x_i \geq 0, \sum_i x_i = 1$$

Geometrically, you can picture the set of one-hot vectors as the vertices of an n-dimensional polytope, and the simplex as its convex hull: values inside the simplex, such as $[0.2, 0.05, 0.75]$, do not precisely correspond to a vertex, but they allow for gradient descent because we can smoothly move inside the polytope. Given a value $\mathbf{x} \in \Delta_n$, we can project to its closest vertex (the predicted class) as:

$$\arg\max_i \{\mathbf{x}_i\}$$

As the name implies, we can interpret values inside the simplex as probability distributions, and projection on the closest vertex as finding the mode (the most probable class) in the distribution. In this interpretation, a one-hot encoded vector is a "special case" where all the probability mass is concentrated on a single class (which we know to be the correct one).

In order to predict a value in this simplex, we need two modifications to the linear model from (E.4.4): first, we need to predict an entire vector simultaneously; and second, we need to constrain the outputs to lie in the simplex. First, we modify the

linear model to predict an m-dimensional vector:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{E.4.10}$$

where $\mathbf{W} \sim (m, c)$ can be interpreted as m linear regression models running in parallel, and $\mathbf{b} \sim (m)$. This output is unconstrained and it is not guaranteed to be in the simplex. The idea of logistic regression is to combine the linear model in (E.4.10) with a simple, parameter-free transformation that projects inside the simplex, called the **softmax** function.

Definition D.4.4 (Softmax function) *The **softmax** function is defined for a generic vector* $\mathbf{x} \sim (m)$ *as:*

$$[\text{softmax}(\mathbf{x})]_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{E.4.11}$$

Let us decompose the terms in (E.4.11) into the basic computations that are executed by introducing two intermediate terms. First, the numerator of the softmax converts each number to a positive value h_i by exponentiation:

$$h_i = \exp(x_i) \tag{E.4.12}$$

Second, we compute a normalization factor Z as the sum of these new (non-negative) values:

$$Z = \sum_j h_j \tag{E.4.13}$$

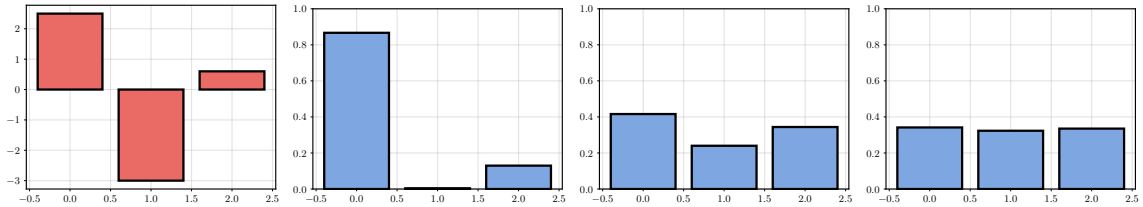
The output of the softmax is then given by dividing h_i by Z, thus ensuring that the new values sum to 1:

$$y_i = \frac{h_i}{Z} \tag{E.4.14}$$

Another perspective comes from considering a more general version of the softmax, where we add an additional hyper-parameter $\tau > 0$ called the **temperature**:

$$\text{softmax}(\mathbf{x}; \tau) = \text{softmax}(\mathbf{x}/\tau)$$

The softmax keeps the relative ordering among the values of x_i for all values of τ,



(a) *Inputs* **(b)** $\tau = 1$ **(c)** $\tau = 10$ **(d)** $\tau = 100$

Figure F.4.3: *Example of softmax applied to a three-dimensional vector (a), with temperature set to 1 (b), 10 (c), and 100 (d). As the temperature increases, the output converges to a uniform distribution. Note that inputs can be both positive or negative, but the outputs of the softmax are always constrained in* $[0, 1]$.

but their absolute distance is increased or decreased based on the temperature. In particular, we have the following two limiting cases:

$$\lim_{\tau \to \infty} \text{softmax}(\mathbf{x}; \tau) = 1/c \tag{E.4.15}$$

$$\lim_{\tau \to 0} \text{softmax}(\mathbf{x}; \tau) = \arg\max_i \ \mathbf{x} \tag{E.4.16}$$

For infinite temperature, relative distances will disappear and the output reverts to a uniform distribution. At the contrary, at 0 temperature the softmax reverts to the (poorly differentiable) argmax operation. Hence, softmax can be seen as a simple differentiable approximation to the argmax, and a better name should be **softargmax**. However, we will retain the most standard name here. See Figure F.4.3 for a visualization of a softmax applied on a generic three-dimensional vector with different temperature values.

4.2.2 The logistic regression model

We can summarize our previous discussion by combining the softmax in (E.4.11) with the linear model in (E.4.10) to obtain a linear model for classification:

$$\hat{\mathbf{y}} = \text{softmax}\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right)$$

The pre-normalized outputs $\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{b}$ are called the **logits** of the model, a name that

will be discussed in more detail in the next section.

The only thing left to complete the specification of the logistic regression model is a loss function. We can achieve this easily by considering the probabilistic viewpoint from Section 3.2.2. Because our outputs are restricted to the probability simplex, we can interpret them as the parameters of a categorical distribution:

Exponent is always either 0 or 1

$$p(\mathbf{y}^{\text{oh}} \mid \hat{\mathbf{y}}) = \prod_i \hat{y}_i^{\,y_i^{\text{oh}}}$$

One-hot encoded class

Computing the maximum likelihood solution in this case (try it) gets us the **cross-entropy loss**.

Definition D.4.5 (Cross-entropy loss) *The **cross-entropy** loss function between* $\mathbf{y}^{oh}$ *and* $\hat{\mathbf{y}}$ *is given by:*

$$\text{CE}(\mathbf{y}^{\text{oh}}, \hat{\mathbf{y}}) = -\sum_i y_i^{\text{oh}} \log(\hat{y}_i) \tag{E.4.17}$$

The loss can also be derived as the KL divergence between the two probability distributions. While unintuitive at first, it has a very simple interpretation by noting that only one value of $\mathbf{y}^{\text{oh}}$ will be non-zero, corresponding to the true class $y = \arg\max_i \{y_i^{\text{oh}}\}$. We can then simplify the loss as:

Probability assigned to the true class

$$\text{CE}(y, \hat{\mathbf{y}}) = -\log(\hat{y}_y) \tag{E.4.18}$$

From (E.4.18), we see that the effect of minimizing the CE loss is to maximize the output probability corresponding to the true class. This works since, due to the denominator in the softmax, any increase in one output term will automatically lead to a decrease of the other terms. Putting everything together, we obtain the logistic regression optimization problem:

$$\text{LR}(\mathbf{W}, \mathbf{b}) = \frac{1}{n} \sum_{i=1}^{n} \text{CE}\left(\mathbf{y}_i^{\text{oh}}, \text{softmax}(\mathbf{W}\mathbf{x}_i + \mathbf{b})\right).$$

Differently from least-squares, we cannot compute a closed-form solution anymore, but we can still proceed with gradient descent. We will show in the next section an example of gradient in this case, and in Section 6.3 a generic technique to compute gradients in cases such as this one.

4.3 Additional topics on classification

4.3.1 Binary classification

Consider now the specific case of $m = 2$. In this case we have $y \in \{0, 1\}$, and the problem reduces to **binary classification**, sometimes called **concept learning** (as we need to learn whether a certain binary "concept" is present or absent in the input). With a standard logistic regression, this would be modelled by a function having two outputs. However, because of the softmax denominator, the last output of a logistic regression is always redundant, as it can be inferred knowing that the outputs must sum to 1:

$$f_m(\mathbf{x}) = \sum_{i=1}^{m-1} f_i(\mathbf{x})$$

Based on this, we can slightly simplify the formulation by considering a scalar model with a single output $f(\mathbf{x}) \in [0, 1]$, such that:

$$\text{Predicted class} = \text{round}(f(\mathbf{x})) = \begin{cases} 0 & \text{if } f(\mathbf{x}) \leq 0.5 \\ 1 & \text{otherwise} \end{cases}$$

To achieve the desired normalization in $[0, 1]$, the first output of a two-valued softmax can be rewritten as $\frac{\exp(x_1)}{1+\exp(x_1)}$, and we can further simplify it by dividing both sides by $\exp(x_1)$. The result is the **sigmoid** function.

Definition D.4.6 (Sigmoid function) *The **sigmoid** function $\sigma(s) : \mathbb{R} \to [0, 1]$ is given by:*

$$\sigma(s) = \frac{1}{1 + \exp(-s)}$$

The sigmoid provides a generic transformation projecting any real value to the $[0, 1]$ interval (with the two extremes being reached only asymptotically). Its graph is shown in Figure F.4.4.

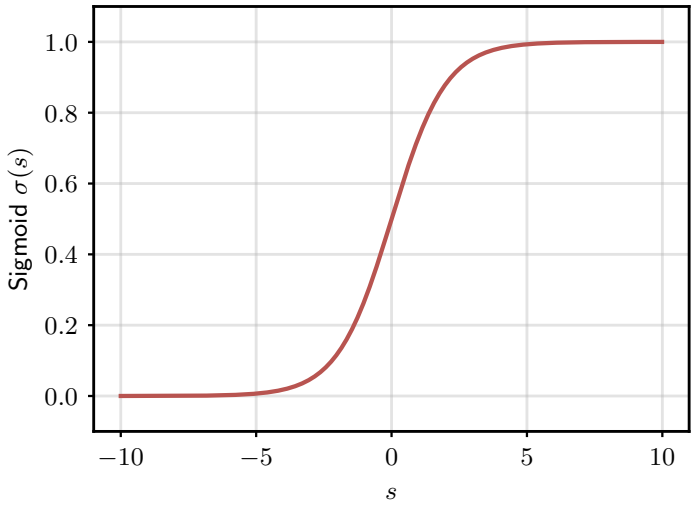


Figure E4.4: *Plot of the sigmoid function. Note that* $\sigma(0) = 0.5$.

The **binary logistic regression** model is obtained by combining a one-dimensional linear model with a sigmoid rescaling of the output:

$$f(\mathbf{x}) = \sigma\left(\mathbf{w}^\top \mathbf{x} + b\right)$$

The cross-entropy similarly simplifies to:

$$\text{CE}(\hat{y}, y) = \underbrace{-y \log(\hat{y})}_{\text{Loss for class 1}} \underbrace{-(1-y)\log(1-\hat{y})}_{\text{Loss for class 2}} \tag{E.4.19}$$

Hence, in the binary classification case we can solve the problem with two equivalent approaches: (a) a two-valued model with the standard softmax, or (b) a simplified one-valued output with a sigmoid output transformation.

As an interesting side-note, consider the gradient of the binary logistic regression model with respect to $\mathbf{w}$ (a similar gradient can also be written for the standard multi-class case):

$$\nabla \text{CE}(f(\mathbf{x}), y) = (f(\mathbf{x}) - y)\mathbf{x}$$

Note the similarity with the gradient of a standard linear model for regression. This similarity can be further understood by rewriting our model as:

```
# Binary cross-entropy
torch.nn.functional.binary_cross_entropy
# Binary cross-entropy accepting logits
torch.nn.functional.binary_cross_entropy_with_logits
# Standard cross-entropy, only works with logits
torch.nn.functional.cross_entropy
# Cross-entropy accepting log f(x) as inputs
torch.nn.functional.nll_loss
```

Box C.4.4: *Cross entropy losses in PyTorch. Some losses are only defined starting from the logits of the model, instead of the post-softmax output. These are the functional variants of the losses - equivalent object-oriented variants are also present in most frameworks.*

$$\underbrace{\mathbf{w}^\top \mathbf{x} + b}_{\text{Logits}} = \underbrace{\log\left(\frac{y}{1-y}\right)}_{\text{Sigmoid inverse: } \sigma^{-1}(y)} \tag{E.4.20}$$

This clarifies why we were referring to the model as a "linear model" for classification: we can always rewrite it as a purely linear model in terms of a non-linear transformation of the output (in this case, the inverse of the sigmoid, also known as the **log-odds**). In fact, the logistic regression model is part of a broader family of models extending this idea, called **generalized linear models**. For the curious reader, the name of the logit can be understood in this context in reference to the probit function.[4]

4.3.2 The logsumexp trick

This is a more technical subsection that clarifies an implementation aspect of what we described up to now. Looking at frameworks like TensorFlow or PyTorch, we can find multiple existing implementations of the cross-entropy loss, based on whether the output is described as an integer or as a one-hot encoded vector. This can be understood easily, as we have already seen that we can formulate the cross-entropy loss in both cases. However, we can also find variants that accept logits instead of the softmax-normalized outputs, as shown in Box C.4.4.

To understand why we would need this, consider the i-th term of the cross-entropy in

[4] https://en.wikipedia.org/wiki/Probit

terms of the logits $\mathbf{p}$:

$$-\log\left(\frac{\exp p_i}{\sum_j \exp p_j}\right).$$

This term can give rise to several numerical issues, notably due to the interplay between the (potentially unbounded) logits and the exponentiation. To solve this, we first rewrite it as:

$$-\log\left(\frac{\exp p_i}{\sum_j \exp p_j}\right) = -p_i + \underbrace{\log\left(\sum_j \exp p_j\right)}_{\triangleq \text{logsumexp}(\mathbf{p})}$$

The first term does not suffer from instabilities, while the second term (the **logsumexp** of the logits) is a function of the entire logits' vector, and it can be shown to be invariant for a given scalar $c \geq 0$ in the following sense:[5]

$$\text{logsumexp}(\mathbf{p}) = \text{logsumexp}(\mathbf{p} - c) + c$$

Note that $\nabla\text{softmax}(\bullet) = \text{logsumexp}(\bullet)$. By taking $c = \max(\mathbf{p})$ we can prevent numerical problems by bounding the maximum logit value at 0. However, this is only possible if we have access to the original logits, which is why numerically stable variants of the cross-entropy require them as inputs. This creates a little amount of ambiguity, in that the softmax can now be included as either part of the model, or as part of the loss function.

4.3.3 Calibration and classification

We close the chapter by briefly discussing the important topic of **calibration** of the classifier. To understand it, consider the following fact: although our model provides an entire distribution over the possible classes, our training criterion only targets the maximization of the true class. Hence, the following sentence is justified:

> The predicted class of $f(\mathbf{x})$ is $\arg\max_i [f(\mathbf{x})]_i$.

Instead, this more general sentence might not be correct:

> The probability of $\mathbf{x}$ being of class i is $[f(\mathbf{x})]_i$.

[5] https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/

When the confidence scores of the network match the probability of a given prediction being correct, we say the network's outputs are **calibrated**.

> **Definition D.4.7 (Calibration)** *A classification model $f(\mathbf{x})$ giving in output the class probabilities is said to be calibrated if the following holds for any possible prediction:*
>
> $$[f(\mathbf{x})]_i = p(y = i \mid \mathbf{x})$$

Although the cross entropy should recover the conditional probability distribution over an unrestricted class of models and in the limit of infinite data [HTF09], in practice the mismatch between the two may be high [BGHN24], especially for the more complex models we will introduce later on.

To understand the difference between accuracy and calibration, consider these two scenarios. First, consider a binary classification model that has perfect accuracy, but always predicts the true class with 0.8 confidence. In this case, the model is clearly *underconfident* in its predictions, since by looking at the confidence we may assume that 20% of them would be incorrect. Second, consider a 4 class problem with perfectly balanced classes, with a model that always predict $[0.25, 025, 0.25, 0.25]$. In this case, the model is perfectly calibrated, but useless from the point of view of accuracy.

Having access to a calibrated model is very important in situations in which different predictions may have different costs. This can be formalized by defining a so-called *cost matrix* assigning a cost C_{ij} for any input of class i predicted as class j. A standard example is a binary classification problem having the matrix of costs shown in Table T.4.2.

Table T.4.2: *Example of cost matrix for a classification problem having asymmetric costs of misclassification.*

	True class 0	True class 1
Predicted class 0	0	10
Predicted class 1	1	0

We can interpret Table T.4.2 as follows: making a correct prediction incurs no cost, while making a false negative mistake (0 instead of 1) is 10 times more costly than making a false positive mistake. As an example, an incorrect false negative mistake in a medical diagnosis is much worse than a false positive error, in which a further test may correct the mistake. A calibrated model can help us in better estimating the

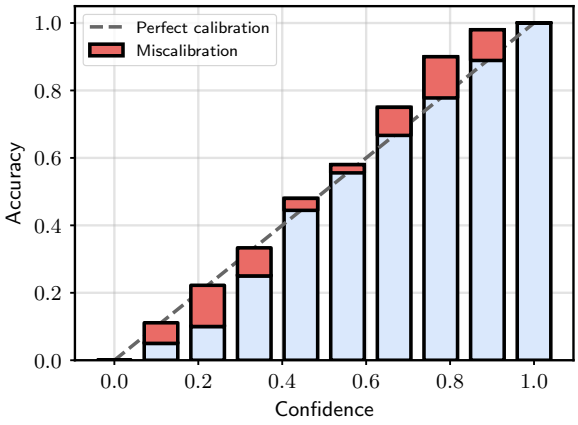


Figure F.4.5: *An example of reliability plot with $b = 10$ bins. The blue bars show the average accuracy of the model on that bin, while the red bars show the miscalibration for the bin, which can be either under-confident (below the diagonal) or over-confident (above the diagonal). The weighted sum of the red blocks is the ECE in* (E.4.21).

average risk of its deployment, and to fine-tune our balance of false positive and false negative mistakes.

To see this, denote by $\mathbf{C} \sim (m, m)$ the generic matrix of costs for a multiclass problem (like the 2×2 matrix in Table T.4.2). The rational choice is to select a class which minimizes the expected cost based on the scores assigned by our model:

$$\arg\min_i \sum_{j=1}^{m} C_{ij}[f(\mathbf{x})]_j$$

If $C_{ij} = 1$ whenever $i \neq j$ and 0 otherwise, this reduces to selecting the argmax of f, but for a general matrix of costs the choice of predicted class will be influenced by the relative costs of making specific mistakes. This is a simple example of **decision theory** [Bis06].

4.3.4 Estimating the calibration error

To estimate whether a model is calibrated we can bin its predictions, and compare its calibration to the accuracy in each bin. To this end, suppose we split the interval $[0, 1]$ into b equispaced bins, each of size $1/b$. Take a validation set of size n, and denote by $\mathscr{B}_i$ the elements whose confidence falls into bin i. For each bin, we can further compute the average confidence p_i of the model (which will be, approximately, in the middle of the bin), and the average accuracy a_i. Plotting the set of pairs (a_i, p_i) on an histogram is called a **reliability diagram**, as shown in Figure F.4.5. To have a single, scalar metric of calibration we can use, for example, the **expected calibration error** (ECE):

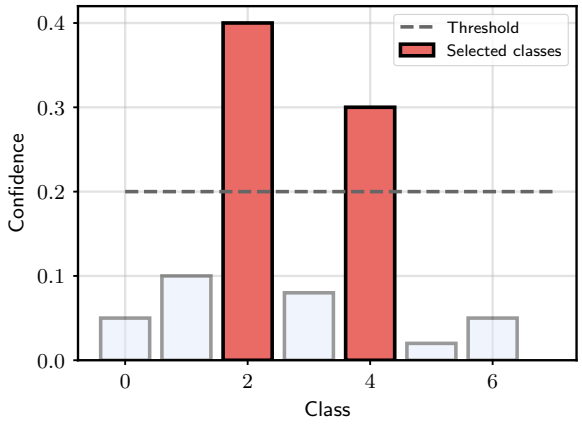


Figure F.4.6: *Calibration by turning the model's output into a set: we return all classes whose predicted probability exceeds a given threshold. By properly selecting the threshold we can bound the probability of the true class being found in the output set.*

$$\text{ECE} = \sum_i \frac{|\mathcal{B}_i|}{n} |a_i - p_i| \tag{E.4.21}$$

Calibration for bin i

Fraction of validation set falling into bin i

Other metrics, such as the maximum over the bins, are also possible. If the model is found to be uncalibrated, modifications need to be made. Examples include rescaling the predictions via temperature scaling [GPSW17] or optimizing with a different loss function such as the focal loss [MKS+20].

We close by mentioning an alternative to direct calibration of the model, called **conformal prediction**, which has become popular recently [AB21]. Suppose we fix a threshold γ, and we take the set of classes predicted by the model whose corresponding probability is higher than γ:

$$\mathscr{C}(\mathbf{x}) = \{i \mid [f(\mathbf{x})]_i > \gamma\} \tag{E.4.22}$$

i.e., the answer of the model is now a *set* $\mathscr{C}(\mathbf{x})$ of potential classes. An example is shown in Figure F.4.6. The idea of conformal prediction is to select the minimum γ such that the probability of finding the correct class y in the set is higher than a user-defined error α:[6]

$$p(y \in \mathscr{C}(\mathbf{x})) \geq 1 - \alpha \tag{E.4.23}$$

Intuitively, there is an inversely proportional relation between γ and α. Conformal pre-

[6]Note that it is always possible to satisfy this property by selecting $\gamma = 0$, i.e., including all classes in the output set.

diction provides automatic algorithms to guarantee (E.4.23) at the cost of not having a single class in output anymore.

From theory to practice

From Chapter 2 you should have a good grasp of NumPy, JAX, and PyTorch's `torch.tensor`. They are all okay for this chapter, and nothing else is required.

I suggest a short exercise to let you train your first differentiable model:

1. Load a toy dataset: for example, one of those contained in scikit-learn datasets module.[7]

2. Build a linear model (for regression or classification depending on the dataset). Think about how to make the code as modular as possible: as we will see, you will need at least two functions, one for initializing the parameters of the model and one for computing the model's predictions.

3. Train the model via gradient descent. For now you can compute the gradients manually: try to imagine how you can make also this part modular, i.e., how do you change the gradient's computation if you want to dynamically add or remove the bias from a model?

4. Plot the loss function and the accuracy on an independent test set. If you know some standard machine learning, you can compare the results to other supervised learning models, such as a decision tree or a k-NN, always using scikit-learn.

[7] https://scikit-elearn.org/stable/datasets/toy_dataset.html

5 | Fully-connected models

About this chapter

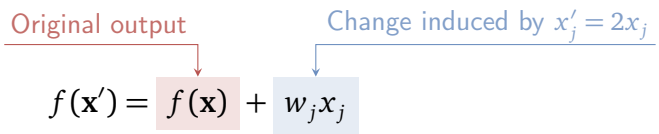
Standard programming is done by concatenating together the proper primitive operations. In this chapter we show that we can do something similar for differentiable models, by composing a sequence of so-called *fully-connected layers*. For historical reasons, these models are also known as multilayer perceptrons (MLPs). MLPs are built by interleaving linear blocks (similar to Chapter 4) with non-linear functions, sometimes called *activation functions*.

5.1 The limitations of linear models

Linear models are fundamentally limited, in the sense that by definition they cannot model non-linear relationships across features. As an example, consider two input vectors $\mathbf{x}$ and $\mathbf{x}'$, which are identical except for a single feature indexed by j:

$$x'_i = \begin{cases} x_i & \text{if } i \neq j \\ 2x_i & \text{otherwise} \end{cases}$$

For example, this can represent two clients of a bank, which are identical in all aspects except for their income, with $\mathbf{x}'$ having double the income of $\mathbf{x}$. If f is a linear model (with no bias) we have:



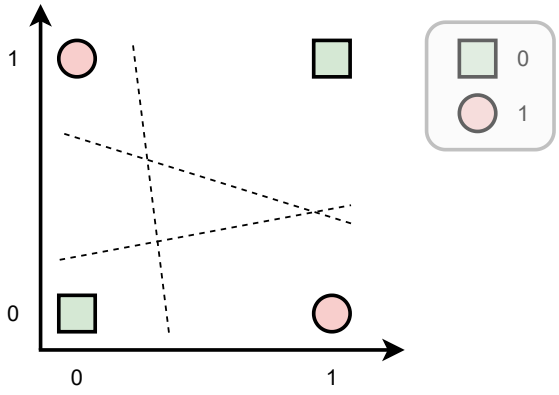


Figure F.5.1: *Illustration of the XOR dataset: green squares are values of one class, red circles are values of another class. No linear model can separate them perfectly (putting all squares on one side and all circles on the other side of the decision boundary). We say that the dataset is not* ***linearly separable****.*

Hence, the only consequence of the change in input is a small linear change of output dictated by w_j. Assume we are scoring the users, we may wish to model relationships such as "*an income of 1500 is low, except if the age* < 30".[1] Clearly, this cannot be done with a linear model due to the analysis above.

The prototypical example of this is the XOR dataset, a two-valued dataset where each feature can only take values in $\{0, 1\}$. Hence, the entire dataset is given by only 4 possibilities:

$$f([0,0]) = 0\ ,\ f([0,1]) = 1\ ,\ f([1,0]) = 1\ ,\ f([1,1]) = 0$$

where the output is positive whenever *only one* of the two inputs is positive. Despite its simplicity, this is also **non-linearly separable**, and cannot be solved with 100% accuracy by a linear model - see Figure F.5.1 for a visualization.

5.2 Composition and hidden layers

A powerful idea in programming is decomposition, i.e., breaking down a problem into its constituent parts recursively, until each part can be expressed in simple, manageable operations. Something similar can be achieved in our case by imagining that our model f is, in fact, the composition of two trainable operations:

$$f(\mathbf{x}) = (f_2 \circ f_1)(\mathbf{x})$$

[1] You probably shouldn't do credit scoring with machine learning anyways.

where $f_2 \circ f_1$ is the composition of the two functions: $(f_2 \circ f_1)(\mathbf{x}) = f_2(f_1(\mathbf{x}))$, and we assume that each function instantiates its own set of trainable parameters. We can keep subdividing the computations:

$$f(\mathbf{x}) = (f_l \circ f_{l-1} \circ \cdots \circ f_2 \circ f_1)(\mathbf{x})$$

where we now have a total of l functions that are being composed. Note that as long as each f_i does not change the "type" of its input data, we can chain together as many of these transformations as we want, and each one will add its own set of trainable parameters.

For example, in our case the input $\mathbf{x}$ is a vector, hence any vector-to-vector operation (e.g., a matrix multiplication $f_i(\mathbf{x}) = \mathbf{W}\mathbf{x}$) can be combined together an endless number of times. However, some care must be taken. Suppose we chain together two different linear projections:

$$\mathbf{h} = f_1(\mathbf{x}) = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 \tag{E.5.1}$$
$$y = f_2(\mathbf{h}) = \mathbf{w}_2^\top \mathbf{h} + b_2 \tag{E.5.2}$$

It is easy to show that the two projections "collapse" into a single one:

$$y = \underbrace{(\mathbf{w}_2^\top \mathbf{W}_1)}_{\triangleq \mathbf{A}}\mathbf{x} + \underbrace{(\mathbf{w}_2^\top \mathbf{b}_1 + b_2)}_{\triangleq \mathbf{c}} = \mathbf{A}\mathbf{x} + \mathbf{c}$$

The idea of **fully-connected** (FC) models, also known as **multi-layer perceptrons** (MLPs) for historical reasons, is to insert a simple elementwise non-linearity $\phi : \mathbb{R} \to \mathbb{R}$ in-between projections to avoid the collapse:

Element-wise non-linearity

$$\mathbf{h} = f_1(\mathbf{x}) = \phi\,(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \tag{E.5.3}$$

$$y = f_2(\mathbf{h}) = \mathbf{w}_2^\top \mathbf{h} + b_2 \tag{E.5.4}$$

The second block can be linear, as in (E.5.4), or it can be wrapped into another non-linearity depending on the task (e.g., a softmax function for classification). The function ϕ can be any non-linearity, e.g., a polynomial, a square-root, or the sigmoid func-

tion σ. As we will see in the next chapter, choosing it has a strong effect on the gradients of the model and, consequently, on optimization, and the challenge is to select a ϕ which is "non-linear enough" to prevent the collapse while staying as close as possible to the identity in its derivative. A good default choice is the so-called **rectified linear unit** (ReLU).

Definition D.5.1 (Rectified linear unit) *The **rectified linear unit** (ReLU) is defined elementwise as:*

$$\text{ReLU}(s) = \max(0, s) \tag{E.5.5}$$

We will have a lot more to say on the ReLU in the next chapter. With the addition of ϕ, we can now chain as many transformations as we want:

$$y = \mathbf{w}_l^\top \phi\left(\mathbf{W}_{l-1}\left(\phi\left(\mathbf{W}_{l-2}\phi\left(\cdots\right) + \mathbf{b}_{l-2}\right)\right) + \mathbf{b}_{l-1}\right) + b_l \tag{E.5.6}$$

In the rest of the chapter we focus on analyzing training and approximation properties of this class of models. First, however, a brief digression on naming conventions.

On the terminology used in differentiable models

As we already mentioned, neural networks have a long history and a long baggage of terminology, which we briefly summarize here. Each f_i is called a **layer** of the model, with f_l being the **output layer**, $f_i, i = 1, \ldots, l-1$ the **hidden layers** and, with a bit of notational overloading, $\mathbf{x}$ being the **input layer**. With this terminology, we can restate the definition of the **fully-connected layer** in batched form below.

Definition D.5.2 (Fully-connected layer) *For a batch of n vectors, each of size c, represented as a matrix $\mathbf{X} \sim (n, c)$, a **fully-connected** (FC) layer is defined as:*

$$\text{FC}(\mathbf{X}) = \phi\left(\mathbf{X}\mathbf{W} + \mathbf{b}\right) \tag{E.5.7}$$

The parameters of the layer are the matrix $\mathbf{W} \sim (c', c)$ and the bias vector $\mathbf{b} \sim (c')$, for a total of $(c'+1)c$ parameters (assuming ϕ does not have parameters). Its hyperparameters are the width c' and the non-linearity ϕ.

The outputs $f_i(\mathbf{x})$ are called the **activations** of the layer, where we can sometimes distinguish between the **pre-activation** and the **post-activation** (before and after the

```python
class FullyConnectedLayer(nn.Module):
  def __init__(self, c: int, cprime: int):
    super().__init__()
    # Initialize the parameters
    self.W = nn.Parameter(torch.randn(c, cprime))
    self.b = nn.Parameter(torch.randn(1, cprime))

  def forward(self, x):
    return relu(x @ self.W + self.b)
```

Box C.5.1: *The FC layer in* (E.5.7) *implemented as an object in PyTorch. We require a special syntax to differentiate trainable parameters, such as* $\mathbf{W}$*, from other non-trainable tensors: in PyTorch, this is obtained by wrapping the tensors in a* `Parameter` *object. PyTorch also has its collection of layers in* `torch.nn`*, including the FC layer (implemented as* `torch.nn.Linear`*).*

non-linearity). The non-linearity ϕ itself can be called the **activation function**. Each output of f_i is called a **neuron**. Although much of this terminology is outdated, it is still pervasive and we will use it when needed.

The size of the each layer (the shape of the output) is an hyperparameter that can be selected by the user, as it only influences the input shape of the next layer, which is known as the **width** of the layer. For a large number of layers, the number of hyper-parameters grows linearly and their selection becomes a combinatorial task. We will return on this point in Chapter 9, when we discuss the design of models with dozens (or hundreds) of layers.

The layer concept is also widespread in common frameworks. A layer such as (E.5.7) can be defined as an object having two functions: an initialization function that randomly initializes all parameters of the model based on the selected hyper-parameters, and a call function that provides the output of the layer itself. See Box C.5.1 for an example. Then, a model can be defined by chaining together instances of such layers. For example, in PyTorch this can be achieved by the `Sequential` object:

```python
model = nn.Sequential(
    FullyConnectedLayer(3, 5),
    FullyConnectedLayer(5, 4)
)
```

Note that from the point of view of their input-output signature, there is no great difference between a layer as defined in Box C.5.1 and a model as defined above, and we could equivalently use `model` as a layer of a larger one. This compositionality is

a defining characteristic of differentiable models.

5.2.1 Approximation properties of MLPs

Training MLPs proceeds similarly to what we discussed for linear models. For example, for a regression task, we can minimize the mean-squared error:

$$\min_{\{\mathbf{W}_k,\mathbf{b}_k\}_{k=1}^{l}} \frac{1}{n}\sum_i (y_i - f(\mathbf{x}_i))^2$$

where the minimization is now done on all parameters of the model simultaneously. We will see in the next lecture a general procedure to compute gradients in this case.

For now, we note that the main difference with respect to having a linear model is that adding an hidden layer makes the overall optimization problem non-convex, with multiple local optima depending on the initialization of the model. This is an important aspect historically, as alternative approaches to supervised learning (e.g., support vector machines [HSS08]) provide non-linear models while remaining convex. However, the results of the last decade show that highly non-convex models can achieve significantly good performance in many tasks.[2]

From a theoretical perspective, we can ask what is the significance of having added hidden layers, i.e., if linear models can only solve tasks which are linearly separable, what is instead the class of functions that can be approximated by adding hidden layers? As it turns out, having a single hidden layer is enough to have **universal approximation** capabilities. A seminal result in this sense was proved by G. Cybenko in 1989 [Cyb89].

> **Theorem 5.1 (Universal approximation of MLPs [Cyb89])** *Given a continuous function $g : \mathbb{R}^d \to \mathbb{R}$, we can always find a model $f(\mathbf{x})$ of the form (E.5.3)-(E.5.4) (an MLP with a single hidden layer) and sigmoid activation functions, such that for any $\varepsilon > 0$:*
>
> $$|f(\mathbf{x}) - g(\mathbf{x})| \leq \varepsilon \ , \ \forall \mathbf{x}$$
>
> *where the result holds over a compact domain. Stated differently, one-hidden-layer MLPs are "dense" in the space of continuous functions.*

[2] The reason differentiable models generalize so well is an interesting, open research question, to which we return in Chapter 9. Existing explanations range from an implicit bias of (stochastic) gradient descent [PPVF21] to intrinsic properties of the architectures themselves [AJB+17, TNHA24].

The beauty of this theorem should not distract from the fact that this is purely a theoretical construct, that makes use of the fact that the width of the hidden layer of the model can grow without bounds. Hence, for any $\mathbf{x}$ for which the previous inequality does not hold, we can always add a new unit to reduce the approximation error (see Appendix B). In fact, it is possible to devise classes of functions on which the required number of hidden neurons grows exponentially in the number of input features [Ben09].[3]

Many other authors, such as [Hor91], have progressively refined this result to include models with fundamentally any possible activation function, including ReLUs. In addition, universal approximation can also be proved for models having finite *width* but possibly infinite *depth* [LPW+17]. A separate line of research has investigated the approximation capabilities of *overparameterized* models, in which the number of parameters exceeds the training data. In this case, training to a global optimum can be proved in many interesting scenarios [DZPS19, AZLL19] (informally, for sufficiently many parameters, the model can achieve the minimum of the loss on each training sample and, hence, the global minimum of the optimization problem). See Appendix B for a one-dimensional visualization of Cybenko's theorem.

Approximation and learning capabilities of differentiable models are immense fields of study, with countless books devoted to them, and we have only mentioned some significant results here. In the rest of the book, we will be mostly concerned with the effective design of the models themselves, whose behavior can be more complex and difficult to control (and design) than these theorems suggest.

5.3 Stochastic optimization

To optimize the models we can perform gradient descent on the corresponding empirical risk minimization problem. However, this can be hard to achieve when n (the size of the dataset) grows very large. We will see in the next chapter that computing the gradient of the loss requires a time linear in the number of examples, which becomes unfeasible or slow for n in the order of 10^4 or more, especially for large models (memory issues aside).

Fortunately, the form of the problem lends itself to a nice approximation, where we use subsets of the data to compute a descent direction. To this end, suppose that for iteration t of gradient descent we sample a subset $\mathcal{B}_t \subset \mathcal{S}_n$ of r points (with $r \ll n$)

[3] One of these problems, the *parity* problem, is closely connected to the XOR task: https://blog.wtf.sg/posts/2023-02-03-the-new-xor-problem/.

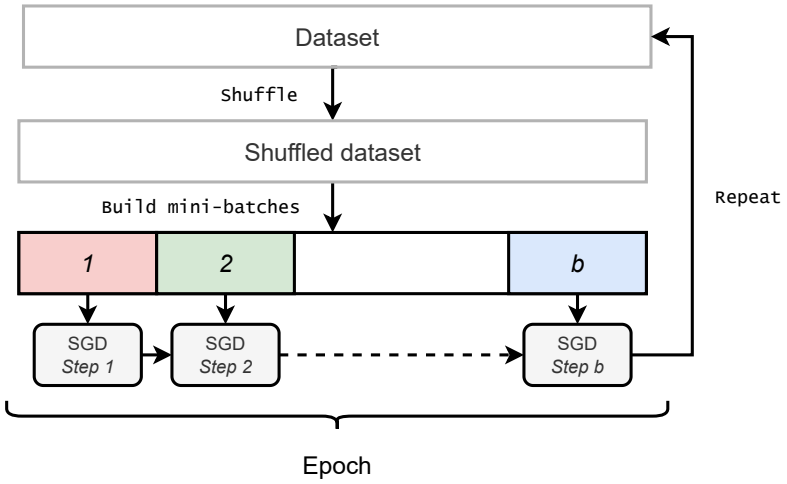


Figure F.5.2: *Building the mini-batch sequence: after shuffling, stochastic optimization starts at mini-batch* 1*, which is composed of the first* r *elements of the dataset. It proceeds in this way to mini-batch* b *(where* $b = \frac{n}{r}$*, assuming the dataset size is perfectly divisible by* r*). After one such epoch, training proceed with mini-batch* $b + 1$*, which is composed of the first* r *elements of the shuffled dataset. The second epoch ends at mini-batch* $2b$*, and so on.*

from the dataset, which we call a **mini-batch**. We can compute an approximated loss by only considering the mini-batch as:

$$\widetilde{L}_t = \frac{1}{r} \sum_{(x_i, y_i) \in \mathcal{B}_t} l(y_i, f(x_i)) \approx \frac{1}{n} \sum_{(x_i, y_i) \in \mathcal{S}_n} l(y_i, f(x_i)) \tag{E.5.8}$$

Mini-batch — Full dataset

If we assume the elements in the mini-batch are sampled i.i.d. from the dataset, $\widetilde{L}_t$ is a Monte Carlo approximation of the full loss, and the same holds for its gradient. However, its computational complexity grows only with r, which can be controlled by the user. Roughly speaking, lower dimensions r of the mini-batch result in faster iterations with higher gradient variance, while higher r results in slower, more precise iterations. For large models, memory is in general the biggest bottleneck, and the mini-batch size r can be selected to fill up the available hardware for each iteration.

Gradient descent applied on mini-batches of data is an example of **stochastic gradient descent** (SGD). Due to the properties discussed above, SGD can be proven to converge to a minimum in expectation, and it is the preferred optimization strategy when train-

```
# A dataset composed by two tensors
dataset = torch.utils.data.TensorDataset(
    torch.randn(1000, 3), torch.randn(1000, 1))

# The data loader provides shuffling and mini-batching
dataloader = torch.utils.data.DataLoader(dataset,
                shuffle=True, batch_size=32)

for xb, yb in dataloader:
  # Iterating over the mini-batch sequence (one epoch)
  # xb has shape (32, 3), yb has shape (32, 1)
```

Box C.5.2: *Building the mini-batch sequence with PyTorch's data loader: all frameworks provide similar tools.*

ing differentiable models.

The last remaining issue is how to select the mini-batches. For large datasets, sampling elements at random can be expensive, especially if we need to move them back and forth from the GPU memory. An intermediate solution that lends itself to easier optimization is the following:

1. Begin by shuffling the dataset.

2. Then, subdivide the original dataset into mini-batches of r *consecutive* elements and process each of them sequentially. Assuming a dataset of size $n = rb$, this results in b mini-batches and hence b steps of SGD. If we are executing the code on a GPU, this step includes sending the mini-batch to the GPU memory.

3. After completing all mini-batches constructed in this way, return to point 1 and iterate.

One complete loop of this process is called an **epoch** of training, and it is a very common hyper-parameter to specify (e.g., for a dataset of 1000 elements and mini-batches of 20 elements, "*training for 5 epochs*" means training for 250 iterations). The expensive shuffling operation is only done once per epoch, while in-between an epoch mini-batches can be quickly pre-fetched and optimized by the framework. This is shown schematically in Figure F.5.2. Most frameworks provide a way to organize the dataset into elements that can be individually indexed, and a separate interface to build the mini-batch sequence. In PyTorch, for example, this is done by the `Dataset` and `DataLoader` interfaces, respectively - see Box C.5.2.

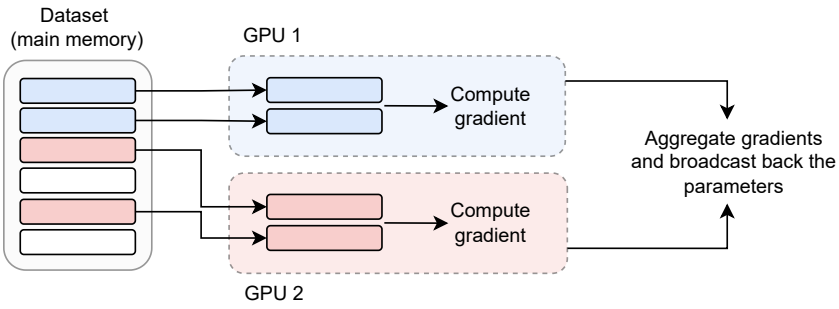


Figure F.5.3: *A simple form of distributed stochastic optimization: we process one mini-batch per available machine or GPU (by replicating the weights on each of them) and sum or average the corresponding gradients before broadcasting back the result (which is valid due to the linearity of the gradient operation). This requires a synchronization mechanism across the machines or the GPUs.*

This setup also leads itself to a simple form of parallelism across GPUs or across machines. If we assume each machine is large enough to hold an entire copy of the model's parameters, we can process different mini-batches in parallel over the machines and then sum their local contributions for the final update, which is then broadcasted back to each machine. This is called a **data parallel** setup in PyTorch,[4] and it is shown visually in Figure F.5.3. More complex forms of parallelism, such as **tensor parallelism**, are also possible, but we do not cover them in this book.

5.4 Activation functions

We close the chapter by providing a brief overview on the selection of activation functions. As we stated in the previous section, almost any element-wise non-linearity is theoretically valid. However, not all choices have good performance. As an example, consider a simple polynomial function, for some user-defined positive integer p:

$$\phi(s) = s^p$$

For large p, this will grow rapidly on both sides, compounding across layers and resulting in models which are hard to train and with numerical instabilities.

Historically, neural networks were introduced as approximate models of biological neu-

[4] https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

rons (hence, the name *artificial NNs*). In this sense, the weights $\mathbf{w}^\top$ in the dot product $\mathbf{w}^\top\mathbf{x}$ were simple models of synapses, the bias b was a threshold, and the neuron was "activated" when the cumulative sum of the inputs surpassed the threshold:

$$s = \mathbf{w}^\top\mathbf{x} - b \ , \ \phi(s) = \mathbb{I}_{s\geq 0}$$

where $\mathbb{I}_b$ is an indicator function which is 1 when b is true, 0 otherwise. Because this activation function is non-differentiable, the sigmoid $\sigma(s)$ can be used as a soft-approximation. In fact, we can define a generalized sigmoid function with a tunable slope a as $\sigma_a(s) = \sigma(as)$, and we have:

$$\lim_{a\to\infty} \sigma_a(s) = \mathbb{I}_{s\geq 0}$$

Another common variant was the hyperbolic tangent, which is a scaled version of the sigmoid in $[-1,+1]$:

$$\tanh(s) = 2\sigma(s) - 1$$

Modern neural networks, popularized by AlexNet in 2012 [KSH12], have instead used the ReLU function in (E.5.5). The relative benefits of ReLU with respect to sigmoid-like functions will be discussed in the next chapter. We note here that ReLUs have several counter-intuitive properties. For example, they have a point of non-differentiability in 0, and they have a large output sparsity since all negative inputs are set to 0. This second property can result in what is known as "dead neurons", wherein certain units have a constant 0 output for all inputs. This can be solved by a simple variant of ReLU, known as **Leaky ReLU**:

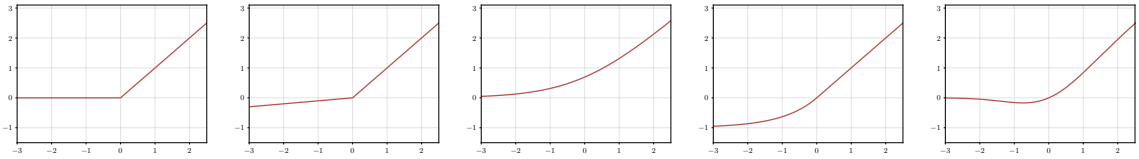
$$\text{LeakyReLU}(s) = \begin{cases} s & \text{if } s \geq 0 \\ \alpha s & \text{otherwise} \end{cases} \tag{E.5.9}$$

for a very small α, e.g., $\alpha = 0.01$. We can also train a different α for each unit (as the function is differentiable with respect to α). In this case, we call the AF a **parametric ReLU** (PReLU) [HZRS15]. Trainable activation functions are, in general, an easy way to add a small amount of flexibility with a minor amount of parameters – in the case of PReLU, one per neuron.

Fully-differentiable variants of ReLU are also available, such as the **softplus**:

$$\text{softplus}(s) = \log(1 + \exp(s)) \tag{E.5.10}$$

The softplus does not pass through the origin and it is always greater than 0. Another



(a) *ReLU* **(b)** *LeakyReLU* **(c)** *Softplus* **(d)** *ELU* **(e)** *GELU*

Figure F.5.4: *Visual comparison of ReLU and four variants: LeakyReLU* (E.5.9)*, Softplus* (E.5.10)*, ELU* (E.5.11)*, and GELU. LeakyReLU is shown with* $\alpha = 0.1$ *for better visualization, but in practice* α *can be closer to* 0 *(e.g.,* 0.01*)*..

variant, the **exponential linear unit** (ELU), preserves the passage at the origin while switching the lower bound to -1:

$$\mathrm{ELU}(s) = \begin{cases} s & \text{if } s \geq 0 \\ \exp(s) - 1 & \text{otherwise} \end{cases} \tag{E.5.11}$$

Yet another class of variants can be defined by noting the similarity of ReLU with the indicator function. We can rewrite the ReLU as:

$$\mathrm{ReLU}(s) = s\mathbb{I}_{s\geq 0}$$

Hence, ReLU is identical to the indicator function on the negative quadrant, while replacing 1 with s on the positive quadrant. We can generalize this by replacing the indicator function with a weighting factor $\beta(s)$:

$$\mathrm{GeneralizedReLU}(s) = \beta(s)s$$

Choosing $\beta(s)$ as the cumulative Gaussian distribution function, we obtain the **Gaussian ELU** (GELU) [HG16], while for $\beta(s) = \sigma(s)$ we obtain the **sigmoid linear unit** (SiLU) [HG16], also known as the **Swish** [RZL17]. We plot some of these AFs in Figure F.5.4. Apart from some minor details (e.g., monotonicity in the negative quadrant), they are all relatively similar, and it is in general very difficult to obtain a significant boost in performance by simply replacing the activation function.

Multiple trainable variants of each function can be obtained by adding trainable parameters to the functions. For example, a common trainable variant of the Swish with four parameters $\{a, b, c, d\}$ is obtained as:

$$\text{Trainable-Swish}(s) = \sigma(as + b)(cs + d) \tag{E.5.12}$$

We can also design **non-parametric** activation functions, in the sense of activation functions that do not have a fixed number of trainable parameters. For example, consider a generic set of (non-trainable) scalar functions ϕ_i indexed by an integer i. We can build a fully flexible activation function as a linear combination of n such bases:

$$\phi(s) = \sum_{i=1}^{n} \alpha_i \phi_i(s) \tag{E.5.13}$$

where n is an hyper-parameter, while the coefficients α_i are trained by gradient descent. They can be the same for all functions, or different for each layer and/or neuron. Based on the choice of ϕ_i we obtain different classes of functions: if each ϕ_i is a ReLU we obtain the **adaptive piecewise linear** (APL) function [AHSB14], while for more general kernels we obtain the **kernel activation function** (KAF) [MZBG18, SVVTU19]. Even more general models can be obtained by considering functions with multiple inputs and multiple outputs [LCX+23]. See [ADIP21] for a survey.

In general, there is no answer to the question of "what is the best AF", as it depends on the specific problem, dataset, and architecture. Apart from its performance, ReLU is a common choice also because highly optimized code kernels are available and it adds a minor cost overhead. It is important to consider the fundamental computational trade-off that, for a given budget, more complex AFs can result in having smaller width or smaller depth, potentially hindering the performance of the entire architecture. For this reason, AFs with a lot of trainable parameters are less common.

Design variants

Not every layer fits into the framework of *linear projections* and *element-wise* non-linearities. For example, the **gated linear unit** (GLU) [DFAG17] combines the structure of (E.5.12) with multiplicative (Hadamard) interactions:

$$f(\mathbf{x}) = \sigma(\mathbf{W}_1\mathbf{x}) \odot (\mathbf{W}_2\mathbf{x}) \tag{E.5.14}$$

where $\mathbf{W}_1$ and $\mathbf{W}_2$ are trained. Another common variant, the SwiGLU, replaces the sigmoid in (E.5.14) with a Swish function [Sha20]. In a **maxout** network [GWFM+13] each unit produces the maximum of k (hyper-parameter) different projections. Replacing the linear projection $\mathbf{W}$ with a matrix of trainable non-linearities $W_{ij} \to \phi_{ij}(x_j)$ of the form (E.5.13) has also been proposed recently under the name of **Kolmogorov-Arnold networks** (KAN, [LWV+24]).

From theory to practice

This chapter has introduced two key requirements for any general-purpose framework for training differentiable models:

1. A way to handle large datasets that need to be shuffled, separated into mini-batches, and moved back and forth from the GPU. In PyTorch, most of this is implemented via the `Dataset` and `DataLoader` interfaces, as in Box C.5.2.[5]

2. A mechanism to build models from the combination of basic blocks, known as *layers*. In PyTorch, layers are implemented in the `torch.nn` module, and they can be composed via the `Sequential` interface or by subclassing the `Module` class, as in Box C.5.1.

I suggest you now try to replicate one of the many quick guides available on the documentation of PyTorch.[6] Everything should be reasonably clear, apart from the gradient computation mechanism, introduced in the next chapter. This is also a good time to investigate HuggingFace Datasets, which combines a vast repository of datasets with an interface to process and cache them which is framework-agnostic and backed by Apache Arrow.[7]

JAX does not provide high-level utilities. For data loading you can use any existing tool, including PyTorch's data loaders and HuggingFace Datasets. For building models, the easiest way is to rely on an external library. Because JAX is fully functional, object-oriented abstractions like Box C.5.1 are not possible. My personal suggestion is Equinox [KG21], which provides a class-like experience by combining the basic data structure of JAX (the `pytree`) with callable nodes.

[5] https://pytorch.org/tutorials/beginner/basics/data_tutorial.html
[6] https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html
[7] https://huggingface.co/docs/datasets/en/quickstart

6 | Automatic differentiation

About this chapter

The previous chapter highlighted the need for an efficient, automatic procedure to compute gradients of any possible sequence of operations. In this chapter we describe such a method, called **back-propagation** in the neural network's literature or **reverse-mode automatic differentiation** in the computer science one. Its analysis has several insights, ranging from the model's choice to the memory requirements for optimizing it.

6.1 Problem setup

We consider the problem of efficiently computing gradients of generic *computational graphs*, such as those induced by optimizing a scalar loss function on a fully-connected model, a task called **automatic differentiation** (AD) [BPRS18]. You can think of a computational graph as the set of atomic operations (which we call **primitives**) obtained by running the program itself. We will consider sequential graphs for brevity, but everything can be easily extended to acyclic and even more generic computational graphs [GW08, BR24] .

The problem may seem trivial, since the chain rule of Jacobians (Section 2.2, (E.2.22)) tells us that the gradient of function composition is simply the matrix product of the corresponding Jacobian matrices. However, efficiently implementing this is the key challenge, and the resulting algorithm (**reverse-mode AD** or **backpropagation**) is a cornerstone of neural networks and differentiable programming in general [GW08, BR24]. Understanding it is also key to understanding the design (and the differences) of most frameworks for implementing and training such programs (such as TensorFlow

or PyTorch or JAX). A brief history of the algorithm can be found in [Gri12].

To setup the problem, we assume we have at our disposal a set of **primitives**:

$$\mathbf{y} = f_i(\mathbf{x}, \mathbf{w}_i)$$

Each primitive represents an operation on an input vector $\mathbf{x} \sim (c_i)$, parameterized by the vector $\mathbf{w}_i \sim (p_i)$ (e.g., the weights of a linear projection), and giving as output another vector $\mathbf{y} \sim (c_i')$.

There is a lot of flexibility in our definition of primitives, which can represent basic linear algebra operations (e.g., matrix multiplication), layers in the sense of Chapter 5 (e.g., a fully-connected layer with an activation function), or even larger blocks or models. This recursive composability is a key property of programming and extends to our case.

We only assume that for each primitive we know how to compute the partial derivatives with respect to the two input arguments, which we call the **input Jacobian** and the **weight Jacobian** of the operation:

$$\begin{aligned} \textbf{Input Jacobian:} \quad & \partial_{\mathbf{x}}[f(\mathbf{x}, \mathbf{w})] \sim (c', c) \\ \textbf{Weight Jacobian:} \quad & \partial_{\mathbf{w}}[f(\mathbf{x}, \mathbf{w})] \sim (c', p) \end{aligned}$$

These are reasonable assumptions since we restrict our analysis to differentiable models. Continuous primitives with one or more points of non-differentiability, such as the ReLU, can be made to fit into this framework with the use of **subgradients** (Section 6.4.4). Non differentiable operations such as sampling or thresholding can also be included by finding a relaxation of their gradient or an equivalent estimator [NCN+23]. We cover the latter case in the next volume.

On our notation and higher-order Jacobians

We only consider vector-valued quantities for readability, as all resulting gradients are matrices. In practice, existing primitives may have inputs, weights, or outputs of higher rank. For example, consider a basic fully-connected layer on a mini-batched input:

$$f(\mathbf{X}, \mathbf{W}) = \mathbf{X}\mathbf{W} + \mathbf{b}$$

In this case, the input $\mathbf{X}$ has shape (n, c), the weights have shape (c, c') and (c') (with c' a hyper-parameter), and the output has shape (n, c'). Hence, the input Jacobian has shape (n, c', n, c), and the weight Jacobian has shape (n, c', c, c'), both having rank 4.

In our notation, we can consider the equivalent flattened vectors $\mathbf{x} = \text{vect}(\mathbf{X})$ and $\mathbf{w} = [\text{vect}(\mathbf{W}); \mathbf{b}]$, and our resulting "flattened" Jacobians have shape (nc', nc) and (nc', cc') respectively. This is crucial in the following, since every time we refer to "*the input size c*" we are referring to "*the product of all input shapes*", including eventual mini-batching dimensions. This also shows that, while we may know how to compute the Jacobians, we may not wish to fully *materialize* them in memory due to their large dimensionality.

As a final note, our notation aligns with the way these primitives are implemented in a functional library, such as JAX. In an object-oriented framework (e.g., TensorFlow, PyTorch), we saw that layers are implemented as objects (see Box C.5.1 in the previous chapter), with the parameters being a property of the object, and the function call being replaced by an object's method. This style simplifies certain practices, such as deferred initialization of all parameters until the input shapes are known (**lazy initialization**), but it adds a small layer of abstraction to consider to translate our notation into workable code. As we will see, these differences are reflected in turn in the way AD is implemented in the two frameworks.

6.1.1 Automatic differentiation: problem statement

With all these details out of the way, we are ready to state the AD task. Consider a sequence of l primitive calls, followed by a final summation:

$$\begin{aligned} \mathbf{h}_1 &= f_1(\mathbf{x}, \mathbf{w}_1) \\ \mathbf{h}_2 &= f_2(\mathbf{h}_1, \mathbf{w}_2) \\ &\vdots \\ \mathbf{h}_l &= f_l(\mathbf{h}_{l-1}, \mathbf{w}_l) \\ y &= \sum \mathbf{h}_l \end{aligned}$$

This is called an **evaluation trace** of the program. Roughly, the first $l-1$ operations can represent several layers of a differentiable model, operation l can be a per-input loss (e.g., cross-entropy), and the final operation sums the losses of the mini-batch. Hence, the output of our program is always a scalar, since we require it for numerical optimization. We abbreviate the previous program as $F(\mathbf{x})$.

Definition D.6.1 (Automatic differentiation) *Given a program $F(\mathbf{x})$ composed of a sequence of differentiable primitives,* ***automatic differentiation*** *(AD) refers to the task of simultaneously and efficiently computing all weight Jacobians of the program given knowledge of the computational graph and all individuals input and weight Jacobians:*

$$AD(F(\mathbf{x})) = \left\{\partial_{\mathbf{w}_i} y\right\}_{i=1}^{l}$$

As we will see, there are two major classes of AD algorithms, called **forward-mode** and **backward-mode**, corresponding to a different ordering in the composition of the individual operations. We will also see that the backward-mode (called **back-propagation** in the neural networks' literature) is significantly more efficient in our context. While we focus on a simplified scenario, it is relatively easy to extend our derivation to acyclic graphs of primitives (as already mentioned), and also to situations where parameters are shared across layers (**weight sharing**). We will see an example of weight sharing in Chapter 13.

6.1.2 Numerical and symbolic differentiation

Before moving on to forward-mode AD, we comment on the difference between AD and other classes of algorithms for differentiating functions. First, we could directly apply the definition of gradients (Section 2.2) to obtain a suitable numerical approximation of the gradient. This process is called **numerical differentiation**. However, each scalar value to be differentiated requires 2 function calls in a naive implementation, making this approach unfeasible except for numerical checks over the implementation.

Second, consider this simple function:

$$f(x) = a\sin(x) + bx\sin(x)$$

We can ask a symbolic engine to pre-compute the full, symbolic equation of the derivative. This is called **symbolic differentiation** and shown in Python in Box C.6.1.

In a realistic implementation, the intermediate value $h = \sin(x)$ would be computed only once and stored in an intermediate variable, which can also be reused for the corresponding computation in the gradient trace (and a similar reasoning goes for the $\cos(x)$ term in the derivative). This is less trivial than it appears: finding an optimal implementation for the Jacobian which avoids any unnecessary computation is an NP-complete task (**optimal Jacobian accumulation**). However, we will see that we can exploit the structure of our program to devise a suitably efficient implementation of AD

```python
import sympy as sp
x, a, b = sp.symbols('x a b')
y = a*sp.sin(x) + b*x*sp.sin(x)
sp.diff(y, x) # [Out]: acos(x)+bxcos(x)+bsin(x)
```

Box C.6.1: *Symbolic differentiation in Python using SymPy.*

that is significantly better than a symbolic approach like the above (and it is, in fact, equivalent to a symbolic approach allowing for the presence of subsequences [Lau19]).

6.2 Forward-mode automatic differentiation

We begin by recalling the chain rule of Jacobians. Consider a combination of two primitive functions:

$$\mathbf{h} = f_1(\mathbf{x})\,,\ \mathbf{y} = f_2(\mathbf{h})$$

In terms of their gradients, we have:

$$\partial_{\mathbf{x}} \mathbf{y} = \partial_{\mathbf{h}} \mathbf{y} \cdot \partial_{\mathbf{x}} \mathbf{h}$$

If $\mathbf{x}$, $\mathbf{h}$, and $\mathbf{y}$ have dimensions a, b, and c respectively, the previous Jacobian requires the multiplication of a $c \times b$ matrix (in green) with a $b \times a$ one (in red). We can interpret the rule as follows: if we have already computed f_1 and its Jacobian (red term), once we apply f_2 we can "update" the gradient by multiplying with the corresponding Jacobian (green term).

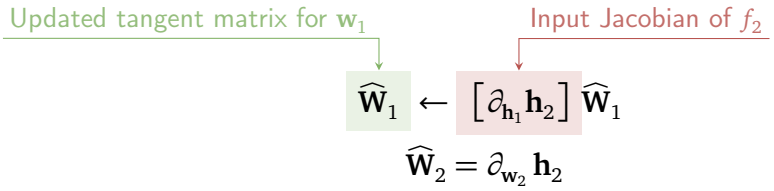
We can immediately apply this insight to obtain a working algorithm called **forward-mode automatic differentiation** (F-AD). The idea is that every time we apply a primitive function, we initialize its corresponding weight Jacobian (called **tangent** in this context), while simultaneously updating all previous tangent matrices. Let us see a simple worked-out example to illustrate the main algorithm.

Consider the first instruction, $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$, in our program. Because nothing has been stored up to now, we initialize the tangent matrix for $\mathbf{w}_1$ as its weight Jacobian:

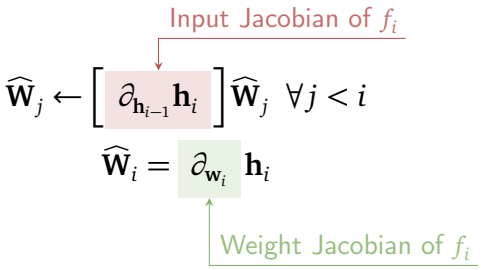
$$\widehat{\mathbf{W}}_1 = \partial_{\mathbf{w}_1} \mathbf{h}_1$$

We now proceed to the second instruction, $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$. We update the previous

tangent matrix while simultaneously initializing the second one:



The update requires the input Jacobian of the primitive, while the second term requires the weight Jacobian of the primitive. Abstracting away, consider the generic i-th primitive given by $\mathbf{h}_i = f_i(\mathbf{h}_{i-1}, \mathbf{w}_i)$. We initialize the tangent matrix for $\mathbf{w}_i$ while simultaneously updating *all* previous matrices:



There are $i-1$ updates in the first row (one for each tangent matrix we have already stored in memory), with the red term – the input Jacobian of the i-th operation – being shared for all previous tangents. The last operation in the program is a sum, and the corresponding gradient gives us the output of the algorithm:[1]

$$\nabla_{\mathbf{w}_i} y = \mathbf{1}^\top \widehat{\mathbf{W}}_i \;\; \forall i \tag{E.6.1}$$

Done! Let us analyze the algorithm in more detail. First, all the operations we listed can be easily *interleaved* with the original program, meaning that the space complexity will be roughly proportional to the space complexity of the program we are differentiating.

On the negative side, the core operation of the algorithm (the update of $\widehat{\mathbf{W}}_i$) requires a multiplication of two matrices, generically shaped (c_i', c_i) and (c_i, p_j), where c_i, c_i' are input/output shapes, and p_j is the shape of $\mathbf{w}_j$. This is an extremely expensive operation: for example, assume that inputs and outputs are both shaped (n, d), where

[1] To be fully consistent with notation, the output of (E.6.1) is a row vector, while we defined the gradient as a column vector. We will ignore this subtle point for simplicity until it is time to define vector-Jacobian products later on the in the chapter.

n is the mini-batch dimension and d represents the input/output features. Then, the matrix multiplication will have complexity $\mathcal{O}(n^2 d^2 p_j)$, which is quadratic in both mini-batch size and feature dimensionality. This can easily become unfeasible, especially for high-dimensional inputs such as images.

We can obtain a better trade-off by noting that the last operation of the algorithm is a simpler matrix-vector product, which is a consequence of having a scalar output. This is explored in more detail in the next section.

6.3 Reverse-mode automatic differentiation

To proceed, we unroll the computation of a single gradient term corresponding to the i-th weight matrix:

$$\nabla_{\mathbf{w}_i} y = \mathbf{1}^\top \left[\partial_{\mathbf{h}_{l-1}} \mathbf{h}_l\right] \cdots \left[\partial_{\mathbf{h}_i} \mathbf{h}_{i+1}\right] \left[\partial_{\mathbf{w}_i} \mathbf{h}_i\right] \tag{E.6.2}$$

Remember that, notation apart, (E.6.2) is just a potentially long series of matrix multiplications, involving a constant term (a vector $\mathbf{1}$ of ones), a series of input Jacobians (the red term) and a weight Jacobian of the corresponding weight matrix (the green term). Let us define a shorthand for the red term:

$$\widetilde{\mathbf{h}}_i = \mathbf{1}^\top \prod_{j=i+1}^{l} \partial_{\mathbf{h}_{j-1}} \mathbf{h}_j \tag{E.6.3}$$

Because matrix multiplication is associative, we can perform the computations in (E.6.2) in any order. In F-AD, we proceeded from the right to the left, since it corresponds to the ordering in which the primitive functions were executed. However, we can do better by noting two interesting aspects:

1. The leftmost term in (E.6.2) is a product between a vector and a matrix (which is a consequence of having a scalar term in output), which is computationally better than a product between two matrices. Its output is also another vector.

2. The term in (E.6.3) (the product of all input Jacobians from layer i to layer l) can be computed recursively starting from the *last* term and iteratively multiplying by the input Jacobians in the reverse order.

We can put together these observations to develop a second approach to automatic differentiation, that we call **reverse-mode automatic differentiation** (R-AD), which

is outlined next.

1. Differently from F-AD, we start by executing the *entire* program to be differentiated, storing all intermediate outputs.

2. We inizialize a vector $\widetilde{\mathbf{h}} = \mathbf{1}^\top$, which corresponds to the leftmost term in (E.6.2).

3. Moving in reverse order, i.e., for an index i ranging in $l, l-1, l-2, \dots, 1$, we first compute the gradient with respect to the i-th weight matrix as:

$$\partial_{\mathbf{w}_i} y = \widetilde{\mathbf{h}} \left[\partial_{\mathbf{w}_i} \mathbf{h}_i \right]$$

   which is the i-th gradient we need. Next, we update our "back-propagated" input Jacobian as:

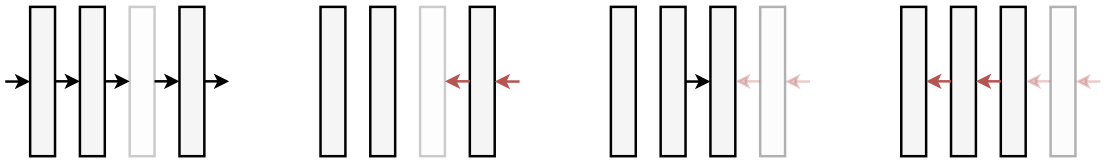
$$\widetilde{\mathbf{h}} \leftarrow \widetilde{\mathbf{h}} \left[\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i \right]$$

Steps (1)-(3) describe a program which is roughly symmetrical to the original program, that we call the **dual** or **reverse** program. The terms $\widetilde{\mathbf{h}}$ are called the **adjoints** and they store (sequentially) all the gradients of the output with respect to the variables $\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_l$ in our program.[2]

In the terminology of neural networks, we sometimes say that the original (**primal**) program is a **forward pass** (not to be confused with forward-mode), while the reverse program is a **backward pass**. Differently from F-AD, in R-AD the full primal program must be executed before the reverse program can be run, and we need specialized mechanisms to store all intermediate outputs to "unroll" the computational graph. Different frameworks implement this differently, as outlined next.

Computationally, R-AD is significantly more efficient than F-AD. In particular, both operations in step (3) of R-AD are vector-matrix products scaling only linearly in all shape quantities. The tradeoff is that executing R-AD requires a large amount of memory, since all intermediate values of the primal program must be stored on disk with a suitable strategy. Specific techniques, such as **gradient checkpointing**, can be used to improve on this tradeoff by increasing computations and partially reducing the memory requirements. This is done by only storing a few intermediate outputs (called **checkpoints**) while recomputing the remaining values during the backward pass. See Figure F.6.1 for a visualization.

[2] Compare this with F-AD, where the tangents represented instead the gradients of the $\mathbf{h}_i$ variables with respect to the weights.



(a) (b) (c) (d)

Figure F.6.1: *An example of **gradient checkpointing**. (a) We execute a forward pass, but we only store the outputs of the first, second, and fourth blocks (**checkpoints**). (b) The backward pass (red arrows) stops at the third block, whose activations are not available. (c) We run a second forward pass starting from the closest checkpoint to materialize again the activations. (d) We complete the forward pass. Compared to a standard backward pass, this requires 1.25x more computations. In general, the less checkpoints are stored, the higher the computational cost of the backward pass.*

6.4 Practical considerations

6.4.1 Vector-Jacobian products

Looking at step (3) in the R-AD algorithm, we can make an interesting observation: the only operation we need is a product between a row vector $\mathbf{v}$ and a Jacobian of f (either the input or the weight Jacobian). We call these two operations the **vector-Jacobian products** (VJPs) of f.[3] In the next definition we restore dimensional consistency by adding a transpose to the vector.

Definition D.6.2 (Vector-Jacobian product (VJP)) *Given a function $\mathbf{y} = f(\mathbf{x})$, with $\mathbf{x} \sim (c)$ and $\mathbf{y} \sim (c')$, its VJP is another function defined as:*

$$\text{vjp}_f(\mathbf{v}) = \mathbf{v}^\top \partial f(\mathbf{x}) \tag{E.6.4}$$

where $\mathbf{v} \sim (c')$. If f has multiple parameters $f(\mathbf{x}_1, \dots, \mathbf{x}_n)$, we can define n individual VJPs denoted as $\text{vjp}_{f,\mathbf{x}_1}(\mathbf{v})$, ..., $\text{vjp}_{f,\mathbf{x}_n}(\mathbf{v})$.

In particular, in our case we can define two types of VJPs, corresponding to the input

[3] By contrast, F-AD can be formulated entirely in terms of the transpose of the VJP, called a **Jacobian-vector product** (JVP). For a one-dimensional output, the JVP is the directional derivative (E.2.19) from Section 2.2. Always by analogy, the VJP represents the application of a linear map connected to infinitesimal variations of the *output* of the function, see [BR24].

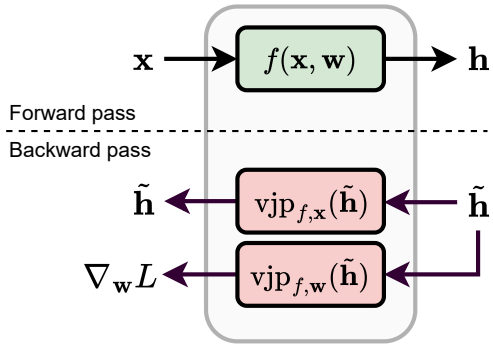


Figure F.6.2: *For performing R-AD, primitives must be augmented with two VJP operations to be able to perform a backward pass, corresponding to the input VJP* (E.6.5) *and the weight VJP* (E.6.6)*. One call for each is sufficient to perform the backward pass through the primitive, corresponding to* (E.6.7)-(E.6.8)*.*

and the weight argument respectively:

$$\mathrm{vjp}_{f,\mathbf{x}}(\mathbf{v}) = \mathbf{v}^\top \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w}) \tag{E.6.5}$$

$$\mathrm{vjp}_{f,\mathbf{w}}(\mathbf{v}) = \mathbf{v}^\top \partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w}) \tag{E.6.6}$$

We can now rewrite the two operations in step (3) of the R-AD algorithm as two VJP calls of the primitive function with the adjoint values (ignoring the i indices for readability), corresponding to the adjoint times the weight VJP, and the adjoint times the input VJP:

$$\partial_{\mathbf{w}} y = \mathrm{vjp}_{f,\mathbf{w}}\left(\tilde{\mathbf{h}}\right) \tag{E.6.7}$$

$$\tilde{\mathbf{h}} \leftarrow \mathrm{vjp}_{f,\mathbf{h}}\left(\tilde{\mathbf{h}}\right) \tag{E.6.8}$$

Hence, we can implement an entire automatic differentiation system by first choosing a set of primitives operations, and then augmenting them with the corresponding VJPs, without having to materialize the Jacobians in memory at any point. This is shown schematically in Figure F.6.2.

In fact, we can recover the Jacobians' computation by repeatedly calling the VJPs with the basis vectors $\mathbf{e}_1, \ldots, \mathbf{e}_n$, to generate them one row at a time, e.g., for the input Jacobian we have:

$$\partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w}) = \begin{bmatrix} \mathrm{vjp}_{f,\mathbf{x}}(\mathbf{e}_1) \\ \mathrm{vjp}_{f,\mathbf{x}}(\mathbf{e}_2) \\ \vdots \\ \mathrm{vjp}_{f,\mathbf{x}}(\mathbf{e}_n) \end{bmatrix}$$

To understand why this reformulation can be convenient, let us look at the VJPs of a fully-connected layer, which is composed of linear projections and (elementwise)

non-linearities. First, consider a simple linear projection with no bias:

$$f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$$

The input Jacobian here is simply $\mathbf{W}$, but the weight Jacobian is a rank-3 tensor (Section 2.2). By comparison, the input VJP has no special structure:

$$\text{vjp}_{f,\mathbf{x}}(\mathbf{v}) = \mathbf{v}^\top \mathbf{W}^\top = [\mathbf{W}\mathbf{v}]^\top \tag{E.6.9}$$

The weight VJP, instead, turns out to be a simple outer product, which avoids rank-3 tensors completely:

$$\text{vjp}_{f,\mathbf{w}}(\mathbf{v}) = \mathbf{v}\mathbf{x}^\top \tag{E.6.10}$$

Working out the VJP

To compute (E.6.10), we can write $y = \mathbf{v}^\top \mathbf{W}\mathbf{x} = \sum_i \sum_j W_{ij} v_i x_j$, from which we immediately get $\frac{\partial y}{\partial W_{ij}} = v_i x_j$, which is the elementwise definition of the outer product.

Hence, every time we apply a linear projection in the forward pass, we modify the back-propagated gradients by the transpose of its weights, and we perform an outer product to compute the gradient of $\mathbf{W}$.

Consider now an element-wise activation function with no trainable parameters, e.g., the ReLU:

$$f(\mathbf{x}, \{\}) = \phi(\mathbf{x})$$

Because we have no trainable parameters, we need only consider the input VJP. The gradient is a diagonal matrix having as elements the derivatives of ϕ:

$$[\partial_{\mathbf{x}} \phi(\mathbf{x})]_{ii} = \phi'(x_i)$$

The input VJP is a multiplication of a diagonal matrix by a vector, which is equivalent to an Hadamard product (i.e., a scaling operation):

$$\text{vjp}_{\mathbf{x}}(f, \mathbf{v}) = \mathbf{v} \odot \phi'(\mathbf{x}) \tag{E.6.11}$$

Interestingly, also in this case we can compute the VJP without having to materialize the full diagonal matrix.

```python
# Original function (sum-of-squares)
def f(x: Float[Array, "c"]):
  return (x**2).sum()

grad_f = func.grad(f)
print(grad_f(torch.randn(10)).shape)
# [Out]: torch.Size([10])
```

Box C.6.2: *Gradient computation as a higher-order function. The* `torch.func` *interface replicates the JAX API. In practice, the function can be traced (e.g., with* `torch.compile`*) to generate an optimized computational graph.*

6.4.2 Implementing a R-AD system

There are many ways to implement the R-AD system, ranging form Wengert lists (as done in TensorFlow) to source-to-source code transformations [GW08]. Here, we discuss briefly some common implementations in existing frameworks.

First, describing primitives as functions with two arguments $f(\mathbf{x}, \mathbf{w})$ aligns with functional frameworks such as JAX, where everything is a function. Consider a function $f(\mathbf{x})$ with a c-dimensional input and a c'-dimensional output. From this point of view, a VJP can be implemented as a higher-order function with signature:

$$(\mathbb{R}^c \to \mathbb{R}^{c'}) \to \mathbb{R}^c \to (\mathbb{R}^{c'} \to \mathbb{R}^c) \tag{E.6.12}$$

i.e., given a function f and an input $\mathbf{x}'$, a VJP returns another function that can be applied to a c'-dimensional vector $\mathbf{v}$ to return $\mathbf{v}^\top \partial f(\mathbf{x}')$. Similarly, the gradient for a one-dimensional function can be implemented as another higher-order function with signature:

$$(\mathbb{R}^c \to \mathbb{R}) \to (\mathbb{R}^c \to \mathbb{R}^c) \tag{E.6.13}$$

taking as input the function $f(\mathbf{x})$ and returning another function that computes $\nabla f(\mathbf{x})$. In JAX, these ideas are implemented in the functions `jax.grad` and `jax.jvp` respectively, which is also replicated in PyTorch in the `torch.func` module - see Box C.6.2 for an example.[4]

As we mentioned, in practice our models are implemented as compositions of objects

[4] Many operations, such as computing an Hessian, can be achieved by smartly composing JVPs and VJPs based on their signatures: https://jax.readthedocs.io/en/latest/notebooks/autodiff_cookbook.html.

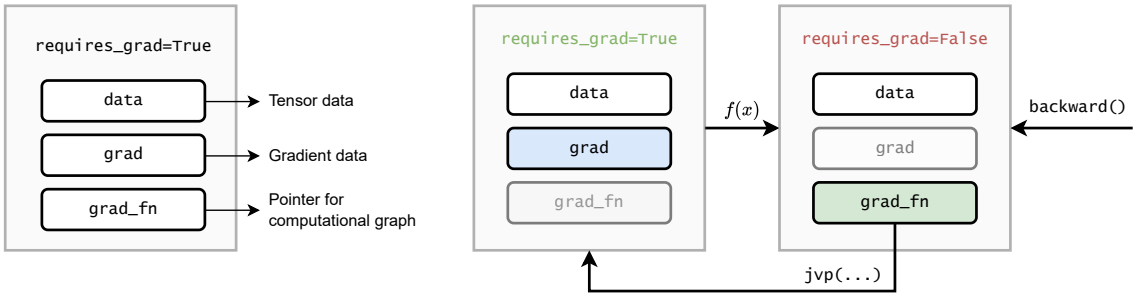


Figure F.6.3: *Left: in PyTorch, a tensor is augmented with information about its gradient (empty at initialization), and about the operation that created it. Right: during a backward pass, the* `grad_fn` *property is used to traverse the computational graph in reverse, and gradients are stored inside the tensor's* `grad` *property whenever* `requires_grad` *is explicitly set to* ***`True`*** *(to avoid consumming unnecessary memory).*

whose parameters are encapsulated as properties (Box C.5.1). One possibility is to "purify" the object to turn it into a pure function, e.g.:[5]

```
# Extract the parameters
params = dict(model.named_parameters())
# Functional call over the model's forward function
y = torch.func.functional_call(model, params, x)
```

More in general, frameworks like PyTorch are augmented with techniques to handle this scenario directly, without introducing intermediate operations. In PyTorch, for example, tensors' objects are augmented with information about the operation that generated them (Figure F.6.3, left). Whenever a `backward()` call is requested on a scalar value, these properties are used to traverse the computational graph in reverse, storing the corresponding gradients *inside* the tensors that requires them (Figure F.6.3, right).

This is just a high-level overview of how these systems are implemented in practice, and we are leaving behind many details, for which we refer to the official documentations.[6]

[5] https://sjmielke.com/jax-purify.htm

[6] I definitely suggest trying to implement an R-AD system from scratch: many didactical implementations can be found online, such as https://github.com/karpathy/micrograd.

6.4.3 Choosing an activation function

Coincidentally, we can now motivate why ReLU is a good choice as activation function. A close look at (E.6.11) tells us that every time we add an activation function in our model, the adjoints in the backward pass are scaled by a factor of $\phi'(\mathbf{x})$. For models with many layers, this can give rise to two pathological behaviors:

1. If $\phi'(\cdot) < 1$ everywhere, there is the risk of the gradient being shrank to 0 exponentially fast in the number of layers. This is called the **vanishing gradient** problem.

2. Conversely, if $\phi'(\cdot) > 1$ everywhere, the opposite problem appears, with the gradients exponentially converging to infinity in the number of layers. This is called the **exploding gradient** problem.

These are serious problems in practice, because libraries represent floating point numbers with limited precision (typically 32 bits or lower), meaning that underflows or overflows can manifest quickly when increasing the number of layers.

Linear non-linear models

Surprisingly, a stack of linear layers implemented in floating point precision is not fully linear because of small discontinuities at machine precision! This is generally not an issue, but it can be exploited to train fully-linear deep neural networks.[a]

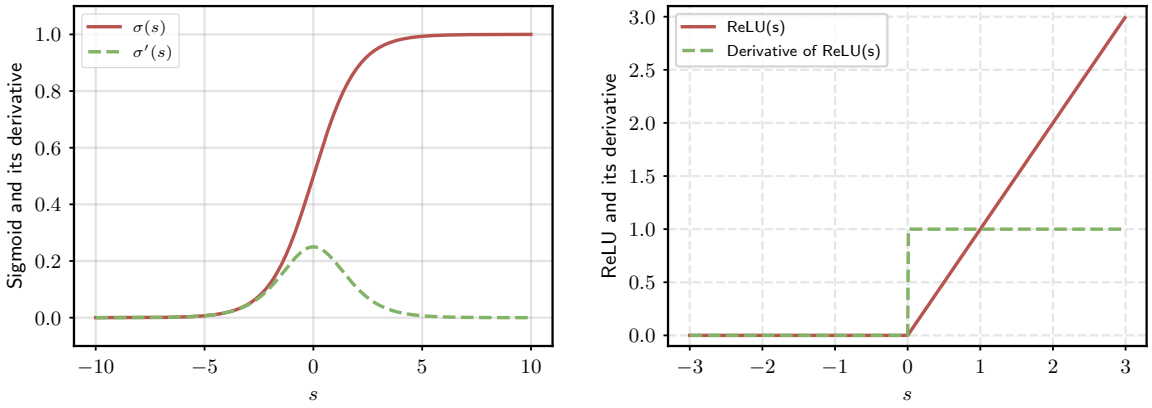
[a] https://openai.com/research/nonlinear-computation-in-deep-linear-networks

As an example of how vanishing gradients can appear, consider the sigmoid function $\sigma(s)$. We already mentioned that this was a common AF in the past, due to it being a soft approximation to the step function. We also know that $\sigma'(s) = \sigma(s)(1-\sigma(s))$. Combined with the fact that $\sigma(s) \in [0,1]$, we obtain that:

$$\sigma'(s) \in [0, 0.25]$$

Hence, the sigmoid is a prime candidate for vanishing gradient issues: see Figure F.6.4a.

Designing an AF that never exhibits vanishing or exploding gradients is non trivial, since the only function having $\phi'(s) = 1$ everywhere is a constant function. We then need a function which is "linear enough" to avoid gradient issues, but "non-linear"



(a) *Sigmoid* **(b)** *ReLU*

Figure F.6.4: *(a) Plot of the sigmoid function (red) and its derivative (green). (b) Plot of ReLU (red) and its derivative (green).*

enough to separate the linear layers. The ReLU ends up being a good candidate since:

$$\partial_s \text{ReLU}(s) = \begin{cases} 0 & s < 0 \\ 1 & s > 0 \end{cases}$$

The gradient is either zeroed-out, inducing sparsity in the computation, or multiplied by 1, avoiding scaling issues - this is shown in Figure F.6.4b.

As a side note, the ReLU's gradient is identical irrespective of whether we replace the input to the ReLU layer with its output (since we are only masking the negative values while keeping the positive values untouched). Hence, another benefit of using ReLU as activation function is that we can save a small bit of memory when performing R-AD, by overwriting the layer's input in the forward pass without impacting the correctness of the AD procedure: this is done in PyTorch, for example, by setting the `in_place` parameter.[7]

6.4.4 Subdifferentiability and correctness of AD

There is a small detail we avoided discussing until now: the ReLU is non-differentiable in 0, making the overall network non-smooth. What happens in this case? The "pragmatic" answer is that, by minimizing with stochastic gradient descent from a random

[7] https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

(non-zero) initialization, the probability of ending up exactly in $s = 0$ is practically null, while the gradient is defined in $\text{ReLU}(\varepsilon)$ for any $|\varepsilon| > 0$.

For a more technical answer, we can introduce the concept of **subgradient** of a function.

> **Definition D.6.3 (Subgradient)** *Given a convex function $f(x)$, a subgradient in x is a point z such that, for all y:*
>
> $$f(y) \geq f(x) + z(y - x)$$

Note the similarity with the definition of convexity: a subgradient is the slope of a line "tangent" to $f(x)$, such that the entire f is lower bounded by it. If f is differentiable in x, then only one such line exists, which is the derivative of f in x. In a non-smooth point, multiple subgradients exists, and they form a set called the **subdifferential** of f in x:

$$\partial_x f(x) = \{z \,|\, z \text{ is a subgradient of } f(x)\}$$

With this definition in hand, we can complete our analysis of the gradient of ReLU by replacing the gradient with its subdifferential in 0:

$$\partial_s \text{ReLU}(s) = \begin{cases} \{0\} & s < 0 \\ \{1\} & s > 0 \\ [0, 1] & s = 0 \end{cases}$$

Hence, any value in $[0, 1]$ is a valid subgradient in 0, with most implementations in practice favoring $\text{ReLU}'(0) = 0$. Selecting subgradients at every step of an iterative descent procedure is called **subgradient descent**.

In fact, the situation is even more tricky, because the subgradient need not be defined for non-convex functions. In that case, one can resort to generalizations that relax the previous definition to a local neighborhood of x, such as the Clarke subdifferential.[8] Subdifferentiability can also create problems in AD, where different implementations of the same functions can provide different (possibly invalid) subgradients, and more refined concepts of chain rules must be considered for a formal proof [KL18, BP20].[9]

[8] https://en.wikipedia.org/wiki/Clarke_generalized_derivative

[9] Consider this example reproduced from [BP20]: define two functions, $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ and $\text{ReLU}_3(s) = 0.5(\text{ReLU}(s) + \text{ReLU}_2(s))$. They are both equivalent to ReLU, but in PyTorch a backward pass

From theory to practice

If you followed the exercises in Chapter 5, you already saw an application of R-AD in both PyTorch and JAX, and this chapter (especially Section 6.4.2) should have clarified their implementation.

It is a good idea to try and re-implement a simple R-AD system, similar to the one of PyTorch. For example, focusing on scalar-valued quantities, the `micrograd` repository[10] is a very good didactical implementation. The only detail we do not cover is that, once you move to a general acyclic graph, an ordering of the variables in the computational graph before the backward pass is essential to avoid creating wrong backpropagation paths. In micrograd, this is achieved via a non-expensive topological sorting of the variables.

It is also interesting to try and implement a new primitive (in the sense used in this chapter) in PyTorch, which requires specifying its forward pass along with its JVPs.[11] One example can be one of the trainable activation functions from Section 5.4. This is a didactical exercise, in the sense that this can be implemented equivalently by subclassing `nn.Module` and letting PyTorch's AD engine work out the backward pass.

All these steps can also be replicated in JAX:

- Implement a didactic version of JAX with `autodidax`: https://jax.readthedocs.io/en/latest/autodidax.html
- Write out a new primitive by implementing the corresponding VJP: https://jax.readthedocs.io/en/latest/notebooks/Custom_derivative_rules_for_Python_code.html
- Read the **JAX Autodiff** Cookbook[12] to discover advanced use-cases for the automatic differentiation engine, such as higher-order derivatives, Hessians, and more.

in 0 returns 0.0 for ReLU, 1.0 for ReLU_2, and 0.5 for ReLU_3.

[10] https://github.com/karpathy/micrograd

[11] https://pytorch.org/docs/master/notes/extending.html

[12] https://jax.readthedocs.io/en/latest/notebooks/autodiff_cookbook.html

Part II
A strange land

“Curiouser and curiouser!” cried Alice (she was so much surprised, that for the moment she quite forgot how to speak good English).

— Chapter 2, The Pool of Tears

7 | Convolutional layers

About this chapter

In this chapter we introduce our second core layer, the **convolutional layer**, which is designed to work with images (or, more in general, sequential data of any kind) by exploiting two key ideas that we call *locality* and *parameter sharing*.

Fully-connected layers are important historically, but less so from a practical point of view: on unstructured data (what we also call **tabular** data, as it can be easily represented as a table) MLPs are generally outperformed by other alternatives, such as random forests or well tuned support vector machines [GOV22]. This is not true, however, as soon as we consider other types of data, having some structure that can be exploited in the design of the layers and of the model.

In this chapter we consider the image domain, while in the next chapters we also consider applications to time series, audio, graphs, and videos. In all these cases, the input has a sequential structure (either temporal, spatial, or of other type) that can be leveraged to design layers that are both performant, easily composable, and highly efficient in terms of parameters. Interestingly, we will see that possible solutions can be designed by taking as starting point a fully-connected layer, and then suitably restricting or generalizing it based on the properties of the input.

7.1 Towards convolutional layers

7.1.1 Why fully-connected layers are not enough

An image can be described by a tensor $X \sim (h, w, c)$, where h is the height of the image, w the width of the image, and c is the number of channels (which can be 1 for black and

white images, 3 for color images, or higher for, e.g., hyper-spectral images). Hence, a mini-batch of images will generally be of rank 4 with an additional leading batch dimension (b, h, w, c). The three dimensions are not identical, since h and w represent a spatial arrangement of *pixels*, while the channels c do not have a specific ordering, in the sense that storing images in an RGB or a GBR format is only a matter of convention.

On notation, channels, and features

We use the same symbol we used for features in the tabular case (c) because it will play a similar role in the design of the models, i.e., we can think of each pixel as described by a generic set of c *features* which are updated in parallel by the layers of the model. Hence, the convolutional layer will return a generic tensor (h, w, c') with an embedding of size c' for each of the hw pixels.

In order to use a fully-connected layer, we would need to "flatten" (vectorize) the image:

$$\mathbf{h} = \phi(\mathbf{W} \cdot \underbrace{\text{vect}(X)}_{\text{Flattened image}}) \tag{E.7.1}$$

where vect(x) is equivalent to `x.reshape(-1)` in PyTorch, and it returns for a generic rank-n tensor $x \sim (i_1, i_2, \ldots, i_n)$ an equivalent tensor $\mathbf{x} \sim \left(\prod_{j=1}^{n} i_j\right)$.

Although it should be clear this is an inelegant approach, it is worth emphasizing some of its disadvantages. First, we have lost a very important property from the previous section, namely, **composability**: our input is an image, while our output is a vector, meaning we cannot concatenate two of these layers. We can recover this by reshaping the output vector to an image:

$$H = \text{unvect}(\phi(\mathbf{W} \cdot \text{vect}(X))) \tag{E.7.2}$$

where we assume that the layer does not modify the number of pixels, and unvect reshapes the output to a (h, w, c') tensor, with c' an hyper-parameter.

This leads directly to the second issue, which is that the layer has a *huge* number of parameters. Considering, for example, a (1024, 1024) image in RGB, keeping the same dimensionality in output results in $(1024 * 1024 * 3)^2$ parameters (or $(hw)^2 cc'$) in general), which is in the order of 10^{13}! We can interpret the previous layer as follows: for each pixel, every channel in the output is a weighted combination of *all* channels

of *all* pixels in the input image. As we will see, we can obtain a more efficient solution by restricting this computation.

More on reshaping

In order to flatten (or more in general, reshape) a tensor, we need to decide an ordering in which to process the values. In practice, this is determined by the way the tensors are stored in memory: in most frameworks, the tensor's data is stored sequentially in a contiguous block of memory, in what is called a **strided layout**. Consider the following example:

```
torch.randn(32, 32, 3).stride() # [Out]: (96, 3, 1)
```

The stride is the number of steps that must be taken in memory to move of 1 position along that axis, i.e., the last dimension of the tensor is contiguous, while to move of one position in the first dimension we need 96 ($32 * 3$) steps. This is called a **row-major** ordering or, in image analysis, a **raster** order.[a] Every reshaping operation works by moving along this strided representation.

[a] https://en.wikipedia.org/wiki/Raster_scan

As a running example to visualize what follows, consider a 1D sequence (we will consider 1D sequences more in-depth later on; for now, you can think of this as "*4 pixels with a single channel*"):

$$\mathbf{x} = \begin{bmatrix} x_1, x_2, x_3, x_4 \end{bmatrix}$$

In this case, we do not need any reshaping operations, and the previous layer (with $c' = 1$) can be written as:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & W_{14} \\ W_{21} & W_{22} & W_{23} & W_{24} \\ W_{31} & W_{32} & W_{33} & W_{34} \\ W_{41} & W_{42} & W_{43} & W_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

7.1.2 Local layers

The spatial arrangement of pixels introduces a metric (a distance) between the pixels. While there are many valid notions of "distance", we will find it convenient to work

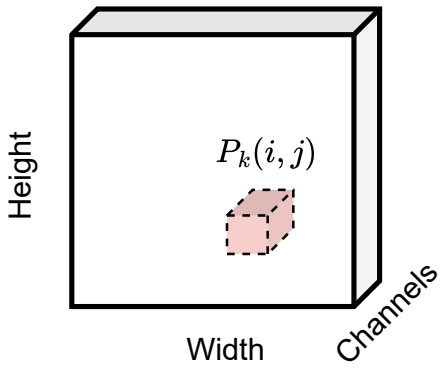


Figure F.7.1: *Given a tensor (h,w,c) and a maximum distance k, the **patch** $P_k(i,j)$ (shown in red) is a $(2k+1, 2k+1, c)$ tensor collecting all pixels at distance at most k from the pixel in position (i,j).*

with the following definition, which defines the distance between pixel (i,j) and (i',j') as the maximum distance across the two axes:

$$d((i,j),(i',j')) = \max(|i-i'|, |j-j'|) \tag{E.7.3}$$

How can we exploit this idea in the definition of a layer? Ideally, we can imagine that the influence of a pixel on another one decreases with a factor inversely proportional to their distance. Pushing this idea to its extreme, we can assume that the influence is effectively zero for a distance larger than some threshold. To formalize this insight, we introduce the concept of a **patch**.

Definition D.7.1 (Image patch) *Given an image X, we define the **patch** $P_k(i,j)$ as the sub-image centered at (i,j) and containing all pixels at distance equal or lower than k:*

$$P_k(i,j) = [X]_{i-k:i+k, j-k:j+k, :}$$

where distance is defined as in (E.7.3). This is shown visually in Figure F.7.1.

The definition is only valid for pixels which are at least k steps away from the borders of the image: we will ignore this point for now and return to it later. Each patch is of shape (s,s,c), where $s = 2k+1$, since we consider k pixels in each direction along with the central pixel. For reasons that will be clarified later on, we call s the **filter size** or **kernel size**.

Consider a generic layer $H = f(X)$ taking as input a tensor of shape (h,w,c) and returning a tensor of shape (h,w,c'). If the output for a given pixel only depends on a patch of predetermined size, we say that the layer is **local**.

Definition D.7.2 (Local layer) *Given an input image $X \sim (h, w, c)$, a layer $f(X) \sim (h, w, c')$ is **local** if there exists a k such that:*

$$[f(X)]_{ij} = f(P_k(i, j))$$

This has to hold for all pixels of the image.

We can transform the layer (E.7.1) into a local layer by setting to 0 all weights belonging to pixels outside the influence region (**receptive field**) of each pixel:

$$H_{ij} = \phi\left(\mathbf{W}_{ij} \cdot \text{vect}(P_k(i,j))\right)$$

Flattened patch (of shape $s^2c'c$)

Position-dependent weight matrix

We call this class of layers **locally-connected**. Note that we have a different weight matrix $\mathbf{W}_{ij} \sim (c', ssc)$ for each output pixel, resulting in $hw(s^2cc')$ parameters. By comparison, we had $(hw)^2cc'$ parameters in the initial layer, for a reduction factor of $\frac{s^2}{hw}$ in the number of parameters.

Considering our toy example, assuming for example $k = 1$ (hence $s = 3$) we can write the resulting operation as:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{12} & W_{13} & 0 & 0 \\ W_{21} & W_{22} & W_{23} & 0 \\ 0 & W_{31} & W_{32} & W_{33} \\ 0 & 0 & W_{41} & W_{42} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

Our operation is not defined for x_1 and x_4, in which case we have considered a "shortened" filter by removing the weights corresponding to undefined operations. Equivalently, you can think of adding 0 on the border whenever necessary:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & 0 & 0 & 0 \\ 0 & W_{21} & W_{22} & W_{23} & 0 & 0 \\ 0 & 0 & W_{31} & W_{32} & W_{33} & 0 \\ 0 & 0 & 0 & W_{41} & W_{42} & W_{43} \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}$$

This technique is called **zero-padding**. In an image, for a kernel size $2k+1$ we need exactly k rows and columns of 0 on each side to ensure that the operation is valid for each pixel. Otherwise, the output cannot be computed close to the borders, and the output tensor will have shape $(h-2k, w-2k, c')$. Both are valid options in most frameworks.

On our definition of patches

The definition of convolutions using the idea of patches is a bit unconventional, but I find it to greatly simplify the notation. I provide a more conventional, signal processing oriented definition later on. The two definitions are equivalent and can be used interchangeably. The patch-oriented definition requires an **odd** kernel size and does not allow for **even** kernel sizes, but these are uncommon in practice.

7.1.3 Translation equivariance and the convolutive layer

In a locally-connected layer, two identical patches can result in different outputs based on their location: some content on pixel $(5,2)$, for example, will be processed differently than the same content on pixel $(39,81)$ because the two matrices $\mathbf{W}_{5,2}$ and $\mathbf{W}_{39,81}$ are different. For the most part, however, we can assume that this information is irrelevant: informally, "a horse is a horse", irrespective of its positioning on the input image. We can formalize this with a property called **translation equivariance**.

Definition D.7.3 (Translation equivariance) *We say that a layer $H = f(X)$ is **translation equivariant** if:*

$$P_k(i,j) = P_k(i',j') \text{ implies } f(P_k(i,j)) = f(P_k(i',j'))$$

Identical patches — *Identical outputs*

To understand the nomenclature, note that we can interpret the previous definition as follows: whenever an object moves (translates) on the image from position (i,j) to position (i',j'), the output $f(P_k(i,j))$ that we we had in (i,j) will now be found in $f(P_k(i',j'))$. Hence, the activations of the layer are moving with the same (*èqui* in Latin) translational movement as the input. We will define more formally equivariance and invariance later on.

A simple way to achieve translation equivariance is given by **weight sharing**, i.e., let-

```
x = torch.randn(16, 3, 32, 32)
w = torch.randn(64, 3, 5, 5)
torch.nn.functional.conv2d(x, w, padding='same').shape
# [Out]: torch.Size([16, 64, 32, 32])
```

Box C.7.1: *Convolution in PyTorch. Note that the channel dimension is – by default – the first one after the batch dimension. The kernel matrix is organized as a (c', c, k, k) tensor. Padding can be specified as an integer or a string ('same' meaning that the output must have the same shape as the input, 'valid' meaning no padding).*

ting every position share the same set of weights:

$$H_{ij} = \phi(\mathbf{W} \cdot \text{vect}(P_k(i,j)))$$

Weight matrix does not depend on (i,j)

This is called a **convolutional layer**, and it is extremely efficient in terms of parameters: we only have a single weight matrix $\mathbf{W}$ of shape (c', ssc), which is independent from the resolution of the original image (once again, contrast this with a layer which is only locally-connected with $hw(s^2c'c)$ parameters: we have reduced them by another factor $\frac{1}{hw}$). We can write a variant with biases by adding c' additional parameters in the form of a bias vector $\mathbf{b} \sim (c')$. Because of its importance, we restate the full definition of the layer below.

Definition D.7.4 (Convolutional layer) *Given an image $X \sim (h, w, c)$ and a kernel size $s = 2k+1$, a **convolutional layer** $H = \text{Conv2D}(X)$ is defined element-wise by:*

$$H_{ij} = \mathbf{W} \cdot \text{vect}(P_k(i,j)) + \mathbf{b} \tag{E.7.4}$$

The trainable parameters are $\mathbf{W} \sim (c', ssc)$ and $\mathbf{b} \sim (c')$. The hyper-parameters are k, c', and (eventually) whether to apply zero-padding or not. In the former case the output has shape (h, w, c'), in the latter case it has shape $(h-2k, w-2k, c')$.

See Box C.7.1 for a code example. The equivalent object-oriented implementation can be found in `torch.nn.Conv2D`. By comparison, our toy example can be refined as follows:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_2 & W_3 & 0 & 0 \\ W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \\ 0 & 0 & W_1 & W_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{E.7.5}$$

where we now have only three weights $\mathbf{W} = [W_1, W_2, W_3]^\top$ (the zero-padded version is equivalent to before and we omit it for brevity). This weight matrix has a special structure, where each element across any diagonal is a constant (e.g., on the main diagonal we only find W_2). We call these matrices **Toeplitz matrices**,[1] and they are fundamental to properly implement a convolutional layer on modern hardware. Toeplitz matrices are an example of **structured** dense matrices [QPF+24]. Equation (E.7.5) should also clarify that a convolution remains a *linear* operation, albeit with a highly restricted weight matrix compared to a fully-connected one.

Convolutions and terminology

Our terminology comes (mostly) from signal processing. We can understand this by rewriting the output of the convolutional layer in a more standard form. To this end, we first rearrange the weight matrix into an equivalent weight tensor W of shape (s, s, c, c'), similar to the PyTorch implementation in Box C.7.1. For convenience, we also define a function that converts an integer i' from the interval $[1, \dots, 2k+1]$ to the interval $[i-k, \dots, i+k]$:

$$t(i) = i - k - 1 \tag{E.7.6}$$

where k is left implicit in the arguments of $t(\bullet)$. We now rewrite the output of the layer with explicit summations across the axes:

$$H_{ijz} = \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1} \sum_{d=1}^{c} [W]_{i',j',z,d} [X]_{i'+t(i),j'+t(j),d} \tag{E.7.7}$$

Check carefully the indexing: for a given pixel (i, j) and output channel z (a free index running from 1 to c'), on the spatial dimensions W must be indexed along $1, 2, \dots, 2k+1$, while X must be indexed along $i-k, i-k+1, \dots, i+k-1, i+k$. The index d runs instead over the input channels.

From the point of view of signal processing, equation (E.7.7) corresponds to a filtering

[1] https://en.wikipedia.org/wiki/Toeplitz_matrix

operation on the input signal X through a set of **finite impulse response** (FIR) filters [Unc15], implemented via a discrete convolution (apart from a sign change). Each filter here corresponds to a slice $W_{:,:,:,i}$ of the weight matrix. In standard signal processing, these filters can be manually designed to perform specific operations on the image. As an example, a 3×3 filter to detect ridges can be written as:[2]

$$W = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

In convolutional layers, instead, these filters can be randomly initialized and trained via gradient descent. We consider the design of **convolutional models** built on convolutional layers in the next section. Before continuing, we mention that an interesting aspect of convolutional layers is that the output maintains a kind of "spatial consistency" and it can be plotted: we call a slice $H_{:,:,i}$ of the output an **activation map** of the layer, representing how much the specific filter was "activated" on each input region. We will consider in more detail the exploration of these maps in the next volume.

7.2 Convolutional models

7.2.1 Designing convolutional "blocks"

With the definition of a convolutional layer in hand, we now turn to the task of building **convolutional models**, also called **convolutional neural networks** (CNNs). We consider the problem of image classification, although a lot of what we say can be extended to other cases. To begin with, we formalize the concept of **receptive field**.

> **Definition D.7.5 (Receptive field)** *Denote by X an image, and by $H = g(X)$ a generic intermediate output of a convolutional model, e.g., the result of applying 1 or more convolutional layers. The **receptive field** $R(i,j)$ of pixel (i,j) is the subset of X which contributed to its computation:*
>
> $$[g(X)]_{ij} = g(R(i,j)), \quad R(i,j) \subseteq X$$

For a single convolutional layer, the receptive field of a pixel is equal to a patch: $R(i,j) = P_k(i,j)$. However, it is easy to prove that for two convolutional layers in

[2] https://en.wikipedia.org/wiki/Kernel_(image_processing)

sequence with identical kernel size, the resulting receptive field is $R(i,j) = P_{2k}(i,j)$, then $P_{3k}(i,j)$ for three layers, and so on. Hence, the receptive field increases *linearly* in the number of convolutional layers. This motivates our notion of locality: even if a single layer is limited in its receptive field by the kernel size, a sufficiently large stack of them results in a *global* receptive field.

Consider now a sequence of two convolutional layers:

$$H = \text{Conv}(\text{Conv}(X))$$

Because convolution is a linear operation (see previous section), this is equivalent to a single convolution with a larger kernel size (as per the above). We can avoid this "collapse" in a similar way to fully-connected layers, by interleaving them with activation functions:

$$H = (\phi \circ \text{Conv} \circ \ldots \circ \phi \circ \text{Conv})(X) \tag{E.7.8}$$

To continue with our design, we note that in (E.7.8) the channel dimension will be modified by each convolutional layer, while the spatial dimensions will remain of the same shape (or will be slightly reduced if we avoid zero-padding). However, it can be advantageous in practice to eventually reduce this dimensionality if our aim is something like image classification.

Consider again the example of a horse appearing in two different regions across two different images. The translation equivariance property of convolutional layers guarantees that every feature found in region 1 in the first image will be found, correspondingly, in region 2 of the second image. However, if our aim is "horse classification", we eventually need one or more neurons activating for an horse *irrespective of where it is found* in the image itself: if we only consider shifts, this property is called **translation invariance**.

Many operations that reduce over the spatial dimensions are trivially invariant to translations, for example:

$$H' = \sum_{i,j} H_{ij} \quad \text{or} \quad H' = \max_{i,j}(H_{ij})$$

In the context of CNNs, this is called a **global pooling**. However, this destroys all spatial information present in the image. We can obtain a slightly more efficient solution with a partial reduction, called **max-pooling**.

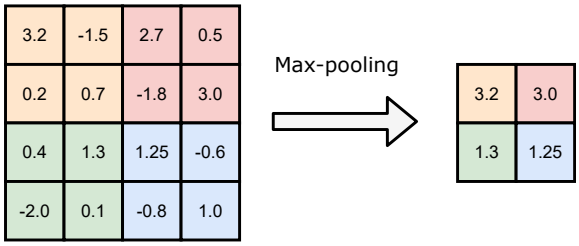


Figure F.7.2: *Visualization of 2x2 max-pooling on a (4,4,1) image. For multiple channels, the operation is applied independently on each channel.*

Definition D.7.6 (Max-pooling layer) *Given a tensor $X \sim (h, w, c)$, a max-pooling layer, denoted as MaxPool(X) $\sim (\frac{h}{2}, \frac{w}{2}, c)$, is defined element-wise as:*

$$[\text{MaxPool}(X)]_{ijc} = \max\Big([X]_{2i-1:2i,2j-1:2j,c} \Big)$$

2×2 *image patch*

Hence, we take 2×2 windows of the input, and we compute the maximum value independently for each channel (this is generalized trivially to larger windows). Max-pooling effectively halves the spatial resolution while leaving the number of channels untouched. An example is shown in Figure F.7.2.

We can build a convolutional "block" by stacking several convolutional layers with a max-pooling operation (see Figure F.7.3):

$$\text{ConvBlock}(X) = (\text{MaxPool} \circ \phi \circ \text{Conv} \circ \ldots \circ \phi \circ \text{Conv})(X)$$

And a more complex network by stacking together multiple such blocks:

$$H = (\text{ConvBlock} \circ \text{ConvBlock} \circ \ldots \circ \text{ConvBlock})(X) \tag{E.7.9}$$

This design has a large number of hyper-parameters: the output channels of each layer, the kernel size of each layer, etc. It is common to drastically reduce the search space for the design by making some simplifying assumptions. For example, the VGG design [SLJ+15] popularized the idea of maintaining the filter size constant in each layer (e.g., $k = 3$), while keeping the number of channels constant in each block and doubling them in-between every block.

An alternative way for reducing the dimensionality is to downsample the output of a convolutional layer: this is called the **stride** of the convolution. For example, a convolution with stride 1 is a normal convolution, while a convolution with stride 2 will

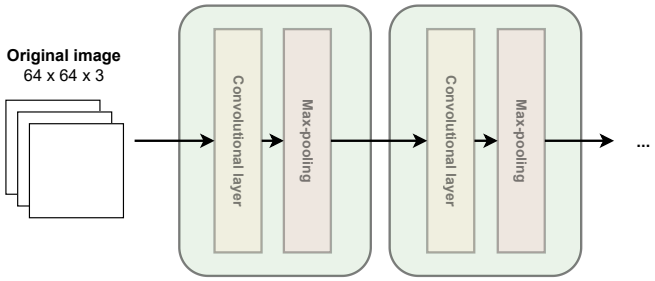


Figure E.7.3: *Abstracting away from "layers" to "blocks" to simplify the design of differentiable models.*

compute only one output pixel every 2, a convolution with stride 3 will compute one output every 3 pixels, and so on. Large strides and max-pooling can also be combined together depending on how the entire model is designed.

Invariance and equivariance

Informally, if T is a transformation on x from some set (e.g., all possible shifts), we say a function f is equivariant if $f(Tx) = Tf(x)$, and invariant if $f(Tx) = f(x)$. The space of all transformations form a group [BBL+17], and the matrix corresponding to a specific transformation is called a **representation** for that group. Convolutional layers are equivariant to translations by design, but other strategies can be found for more general forms of symmetries, such as averaging over the elements of the group (**frame averaging**, [PABH+21]). We will see other types of layers' equivariances in Chapter 12 and Chapter 10.

7.2.2 Designing the complete model

We can now complete the design of our model. By stacking together multiple convolutional blocks as in (E.7.9), the output H will be of shape (h', w', c'), where w' and h' depend on the number of max-pooling operations (or on the stride of the convolutional layers), while c' will depend only on the hyper-parameters of the last convolutional layer in the sequence. Note that each element H_{ij} will correspond to a "macro-region" in the original image, e.g., if $h', w' = 2$, H_{11} will correspond to the "top-left" quadrant in the original image. We can remove this spatial dependency by performing a final global pooling operation before classification.

The complete model, then, can be decomposed as three major components: a series of

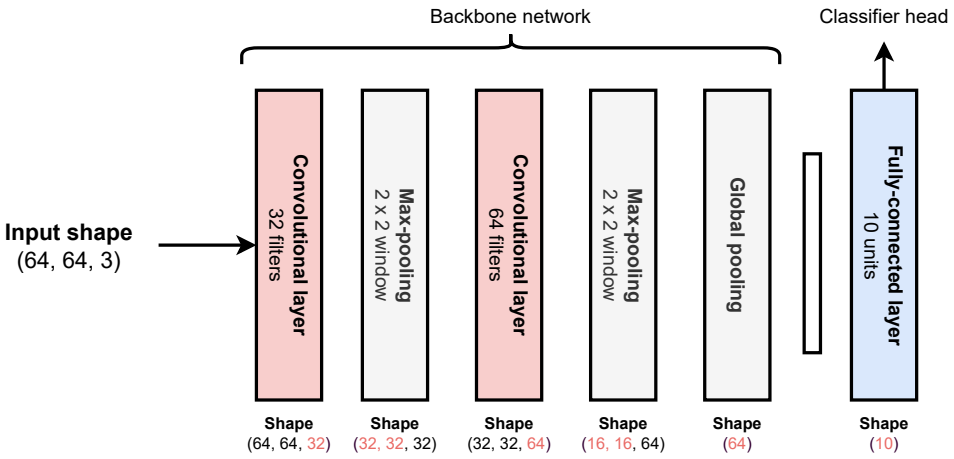


Figure F.7.4: *Worked-out design of a very simple CNN for image classification (assuming 10 output classes). We show the output shape for each layer on the bottom. The global pooling operation can be replaced with a flattening operation. The last (**latent**) representation before the classification head is very useful when fine-tuning large-scale pre-trained models – it is an **embedding** of the image in the sense of Section 3.1.1.*

convolutional blocks, a global average pooling, and a final block for classification.

$$H = (\text{ConvBlock} \circ \ldots \circ \text{ConvBlock})(X) \tag{E.7.10}$$

$$\mathbf{h} = \frac{1}{h'w'} \sum_{i,j} H_{ij} \tag{E.7.11}$$

$$y = \text{MLP}(\mathbf{h}) \tag{E.7.12}$$

where $\text{MLP}(\mathbf{h})$ is a generic sequence of fully-connected layers (a flattening operation can also be used in place of the global pooling). This is a prototypical example of a CNN. See Figure F.7.4 for a worked-out example.

This design has a few interesting properties we list here:

1. It can be trained like the models described in Chapter 4 and Chapter 5. For example, for classification, we can wrap the output in a softmax and train by minimizing the cross-entropy. The same rules of back-propagation described in Chapter 6 apply here.

2. Because of the global pooling operation, it does not depend on a specific input

resolution. However, it is customary to fix this during training and inference to simplify mini-batching (more on variable length inputs in the next chapter).

3. (E.7.11) can be thought of as a "feature extraction" block, while (E.7.12) as the "classification block". This interpretation will be very useful when we consider transfer learning in the next volume. We call the feature extraction block the **backbone** of the model, and the classification block the **head** of the model.

Notable types of convolution

We close the chapter by mentioning two instances of convolutional layers that are common in practice.

First, consider a convolutional layer with $k = 0$, i.e., a so-called 1×1 convolution. This corresponds to updating each pixel's embedding by a weighted sum of its channels, disregarding all other pixels:

$$H_{ijz} = \sum_{t=1}^{c} W_{zt} X_{ijt}$$

It is a useful operation for, e.g., modifying the channel dimension (we will see an example when dealing with residual connections in Chapter 9). In this case, the parameters can be compactly represented by a matrix $\mathbf{W} \sim (c', c)$. This is equivalent to a fully-connected layer applied on each pixel independently.

Second, consider an "orthogonal" variant to 1×1 convolutions, in which we combine pixels in a small neighborhood, but disregarding all channels except one:

$$H_{ijc} = \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1} W_{i',j',c} X_{i'+t(i),j'+t(j),c}$$

where $t(\bullet)$ is the offset defined in (E.7.6). In this case we have a rank-3 weight matrix W of shape (s, s, c), and each output channel $H_{:,:,c}$ is updated by considering only the corresponding input channel $X_{:,:,c}$. This is called a **depthwise convolution**, and it can be generalized by considering groups of channels, in which case it is called a **groupwise convolution** (with the depthwise convolution being the extreme case of a group size equal to 1).

We can also combine the two ideas and have a convolution block made of alternating 1×1 convolutions (to mix the channels) and depthwise convolutions (to mix the

pixels). This is called a **depthwise separable** convolution and it is common in CNNs targeted for low-power devices [HZC+17]. Note that in this case, the number of parameters for a single block (compared to a standard convolution) is reduced from $sscc'$ to $ssc + cc'$. We will see later how these decompositions, where the input is processed alternatively across separate axes, are fundamental for other types of architectures, such as transformers, in Chapter 10.

From theory to practice

All the layers introduced in this chapter (convolution, max-pooling) are implemented in the `torch.nn` module. The torchvision library provides datasets and functions to load images, as long as an interface to apply transformations to the images that will be very useful in the next chapter.[3]

Before proceeding, I suggest you follow and re-implement one of the many online tutorials on image classification in torchvision, which should now be relatively easy to follow.[4] Toy image datasets abound, including MNIST (digit classification) and CIFAR-10 (general image classification). Combining the torchvision loader with the layers in Equinox allows you to replicate the same tutorial in JAX, e.g., https://docs.kidger.site/equinox/examples/mnist/.

Implementing a convolution from scratch is also an interesting exercise, whose complexity depends on the level of abstraction. One possibility is to use the `fold`/`unfold` operations from PyTorch to extract the patches.[5] Premade kernels for convolutions will always be significantly faster, making this a purely didactic exercise.

If you have some signal processing background, you may know that convolution can also be implemented as multiplication by moving to the frequency domain. This is impractical for the small kernels used we tend to use, but it can be useful for very large (also known as *long*) convolutions, e.g., https://github.com/fkodom/fft-conv-pytorch. PyTorch also provides a differentiable Fast Fourier transform that you can use as a starting point.

[3] https://pytorch.org/vision/stable/transforms.html

[4] As an example from the official documentation: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

[5] See for example: https://github.com/loeweX/Custom-ConvLayers-Pytorch

8 | Convolutions beyond images

About this chapter

Convolutional models are a powerful baseline model in many applications, going far beyond image classification. In this chapter we provide an overview of several such extensions, including the use of convolutional layers for 1D and 3D data, text modeling, and autoregressive generation. Several of the concepts we introduce (e.g., masking, tokenization) are fundamental in the rest of the book.

8.1 Convolutions for 1D and 3D data

8.1.1 Beyond images: time series, audio, video, text

In the previous chapter we focused exclusively on images. However, many other types of data share similar characteristics, i.e., one or more "ordered" dimensions representing time or space, and one dimension representing features (the channels in the image case). Let us consider some examples:

1. **Time series** are collections of measurements of one or more processes (e.g., stocks prices, sensor values, energy flows). We can represent a time series as a matrix $\mathbf{X} \sim (t, c)$, where t is the length of the time series, and $\mathbf{X}_i \sim (c)$ are the c measurements at time t (e.g., c sensors from an EEG scan, or c stock prices). Each time instant is equivalent to a pixel, and each measurement is equivalent to a channel.

2. **Audio** files (speech, music) can also be described by a matrix $\mathbf{X} \sim (t, c)$, where t is now the length of the audio signal, while c are the channels of the recording (1 for a mono audio, 2 for a stereo signal, etc.).

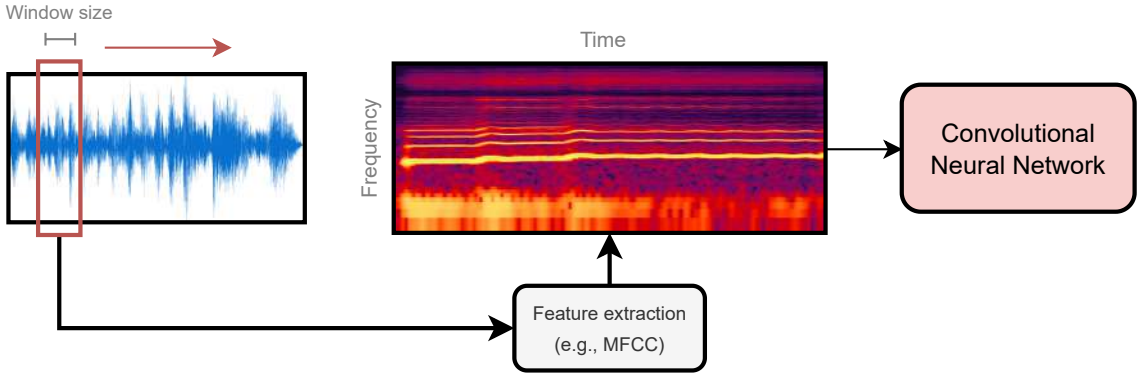


Figure F.8.1: *Audio can be represented as either a 1D sequence (left), or a 2D image in a time-frequency domain (middle). In the second case, we can apply the same techniques described in the previous chapter.*

> **Frequency-analysis**
>
> Audios can also be converted to an image-like format via frequency analysis (e.g., extracting the MFCC coefficients over small windows), in which case the resulting *time-frequency* images represent the evolution of the frequency content over the signal - see Figure F.8.1 for an example. With this preprocessing we can use standard convolutional models to process them.

3. **Videos** can be described by a rank-4 tensor $X \sim (t, h, w, c)$, where t is the number of *frames* of the video, and each frame is an image of shape (h, w, c). Another example is a volumetric scan in medicine, in which case t is the volume depth.

Time series, audio signals, and videos can be described by their **sampling rate**, which denotes how many samples are acquired per unit of time, sometimes expressed in samples per second, or hertz (Hz). For example, classical EEG units acquire signals at 240 Hz, meaning 240 samples each second. A stock can be checked every minute, corresponding to 1/60 Hz. By contrast, audio is acquired with very high frequency to ensure fidelity: for example, music can be acquired at $44.1e^3$ Hz (or 44.1 kHz). Typical acquisition **frame rates** for video are instead around 24 frames per second (fps) to ensure smoothness to the human eye.

Image resolution, audio sampling rate, and video frame rates all play similar roles in determining the precision with which a signal is acquired. For an image, we can assume a fixed resolution a priori (e.g., 1024×1024 pixels). This is reasonable, since images

can always be reshaped to a given resolution while maintaining enough consistency, except for very small resolutions. By contrast, audio and video durations can vary from input to input (e.g., a song of 30 seconds vs. a song of 5 minutes), and they cannot be reshaped to a common dimension, meaning that our datasets will be composed of **variable-length** data. In addition, audio resolution can easily grow very large: with a 44.1 kHz sampling rate, a 3-minute audio will have $\approx 8M$ samples.

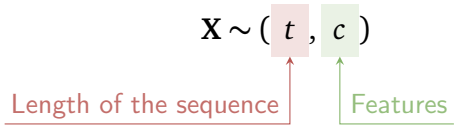
We also note that the dimensions in these examples can be roughly categorized as either "spatial dimensions" (e.g., images) or "temporal dimensions" (e.g., audio resolution). While images can be considered symmetric along their spatial axes (in many cases, an image flipped along the width is another valid image), time is *asymmetric*: an audio sample inverted on its temporal axis is in general invalid, and an inverted time series represents a series evolving from the future towards its past. Apart from exploiting this aspect in the design of our models (**causality**), we can also be interested in *predicting* future values of the signal: this is called **forecasting**.

Finally, consider a text sentence, such as "*the cat is on the table*". There are many ways to split this sentence into pieces. For example, we can consider its individual syllables: ["*the*", "*cat*", "*i*", "*s*", "*on*", "*the*", "*ta*", *ble*"]. This is another example of a sequence, except that each element of the sequence is now a categorical value (the syllable) instead of a numerical encoding. Hence, we need some way of encoding these values into features that can be processed by the model: splitting a text sequence into components is called **tokenization**, while turning each token into a vector is called **embedding** the tokens.

In the next sections we consider all these aspects (variable-length inputs, causality, forecasting, tokenization, and embedding) in turn, to see how we can build convolutional models to address them. Some of the techniques we introduce, such as masking, are very general and are useful also for other types of models, such as transformers. Other techniques, such as dilated convolutions, are instead specific to convolutional models.

8.1.2 1D and 3D convolutional layers

Let us consider how to define convolutions for 1D signals (e.g., time series, audio) and their extension to 3D signals (e.g., videos). Note that the dimensionality refers only to the number of dimensions along which we convolve (spatial or time), and does not include the channel dimension. Recall that, in the 1D case, we can represent the input as a single matrix:



We now replicate the derivation from Chapter 7. Given a patch size $s = 2k+1$, we define $P_k(i) \sim (s, c)$ as the subset of rows in $\mathbf{X}$ at distance at most k from i (ignoring border elements for which zero-padding can be used). A 1D convolutional layer $\mathbf{H} = \text{Conv1D}(\mathbf{X})$ outputs a matrix $\mathbf{H} \sim (t, c')$, with c' an hyper-parameter that defines the output dimensionality, defined row-wise as:

$$[\text{Conv1D}(X)]_i = \phi(\mathbf{W} \cdot \text{vect}(P_k(i)) + \mathbf{b}) \tag{E.8.1}$$

with trainable parameters $\mathbf{W} \sim (c', sc)$ and $\mathbf{b} \sim (c')$. Like in the 2D case, this layer is local (for a properly modified definition of locality) and equivariant to translations of the sequence.

In the 2D case, we also discussed an alternative notation with all indices explicitly summed over:

$$H_{ijz} = \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1} \sum_{d=1}^{c} [W]_{i',j',z,d} [X]_{i'+t(i),j'+t(j),d} \tag{E.8.2}$$

where $t(i) = i+k-1$ as in (E.7.6). Recall that we use t to index i' and j' differently for the two tensors: from 1 to $2k+1$ for W, and from $i-k$ to $i+k$ for X. The equivalent variant for (E.8.1) is obtained trivially by removing one summation index:

$$H_{iz} = \sum_{i'=1}^{2k+1} \sum_{d=1}^{c} [W]_{i',z,d} [X]_{i'+t(i),d} \tag{E.8.3}$$

where the parameters $W \sim (s, c', c)$ are now organized in a rank-3 tensor. By contrast, the 3D variant is obtained by adding a new summation over the third dimension with index p:

$$H_{pijz} = \sum_{p'=1}^{2k+1} \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1}, \sum_{d=1}^{c} [W]_{p',i',j',z,d} [X]_{p'+t(p),i'+t(i),j'+t(j),d}$$

We assume that the kernel size is identical across all dimensions for simplicity. With

similar reasonings we can derive a vectorized 3D variant of convolution, and also 1D and 3D variants of max pooling.

8.2 Convolutional models for 1D and 3D data

We now consider the design of convolutional models in the 1D case, with a focus on how to handle variable-length inputs and how to deal with text sequences. Several of the ideas we introduce are fairly generic for all differentiable models.

8.2.1 Dealing with variable-length inputs

Consider two audio files (or two time series, or two texts), described by their corresponding input matrices $\mathbf{X}_1 \sim (t_1, c)$ and $\mathbf{X}_2 \sim (t_2, c)$. The two inputs share the same number of channels c (e.g., the number of sensors), but they have different lengths, t_1 and t_2. Remember from our discussion in Section 7.1 that convolutions can handle (in principle) such **variable-length** inputs. In fact, denote by g a generic composition of 1D convolutions and max-pooling operations, corresponding to the feature extraction part of the model. The output of the block are two matrices:

$$\mathbf{H}_1 = g(\mathbf{X}_1), \mathbf{H}_2 = g(\mathbf{X}_2)$$

having the same number of columns but a different number of rows (depending on how many max-pooling operations or strided convolutions are applied on the inputs). After global average pooling, the dependence on the length disappears:

$$\mathbf{h}_1 = \sum_i \mathbf{H}_{1i}, \mathbf{h}_2 = \sum_i \mathbf{H}_{2i}$$

and we can proceed with a final classification on the vectors $\mathbf{h}_1$ and $\mathbf{h}_2$. However, while this is not a problem at the level of the model, it is a problem in practice, since mini-batches cannot be built from matrices of different dimensions, and thus operations cannot be easily vectorized. This can be handled by zero-padding the resulting mini-batch to the maximum dimension across the sequence length. Assuming for example, without lack of generality, $t_1 > t_2$, we can build a "padded" mini-batch as:

$$X = \text{stack}\left(\mathbf{X}_1, \begin{bmatrix} \mathbf{X}_2 \\ \mathbf{0} \end{bmatrix}\right)$$

```
# Sequences with variable length (3, 5, 2, respectively)
X1, X2, X3 = torch.randn(3, 8),
             torch.randn(5, 8),
             torch.randn(2, 8)

# Pad into a single mini-batch
X = torch.nn.utils.rnn.pad_sequence([X1, X2, X3],
                    batch_first=True)
print(X.shape) # [Out]: torch.Size([3, 5, 8])
```

Box C.8.1: *A padded mini-batch from three sequences of variable length (with $c = 8$). When using a* `DataLoader`*, padding can be achieved by over-writing the default* `collate_fn`*, which describes how the loader concatenates the individual samples.*

where stack operates on a new leading dimension, and the resulting tensor X has shape $(2, t_1, c)$. We can generalize this to any mini-batch by considering the largest length with respect to all elements of the mini-batch. For a convolution, this is not very different from zero-padding, and operating on the padded input will not influence significantly the operation (e.g., in audio, zero-padding is equivalent to adding silence at the end). See Box C.8.1 for an example of building a padded mini-batch.

Alternatively, we can build a masking matrix describing valid and invalid indexes in the mini-batched tensor:

$$\mathbf{M} = \begin{bmatrix} \mathbf{1}_{t_1} & \\ \mathbf{1}_{t_2} & \mathbf{0}_{t_1 - t_2} \end{bmatrix}$$

where the index denotes the size of the vectors. These masking matrices can be helpful to avoid invalid operations on the input tensor.

8.2.2 CNNs for text data

Let us consider now the problem of dealing with text data. As we mentioned previously, the first step in dealing with text is **tokenization**, in which we divide the text (a string) into a sequence of known symbols (also called **tokens** in this context). There are multiple types of tokenizers:

1. **Character tokenizer**: each character becomes a symbol.
2. **Word tokenizer**: each (allowed) word becomes a symbol.
3. **Subword tokenizer**: intermediate between a character tokenizer and a word

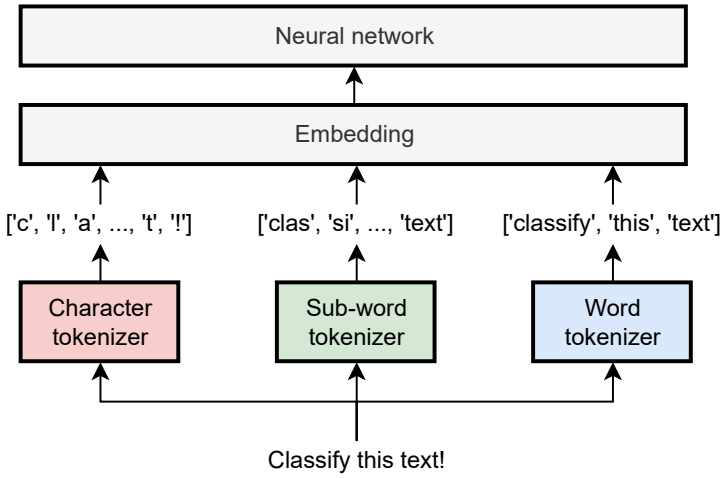


Figure F.8.2: *Starting from a text, multiple types of tokenizers are possible. In all cases, symbols are then embedded as vectors and processed by a generic 1D model.*

tokenizer, each symbol is possibly larger than a character but also smaller than a word.

This is shown schematically in Figure F.8.2. In all three cases, the user has to define a **dictionary** (**vocabulary**) of allowed tokens, such as all ASCII characters for a character tokenizer. In practice, one can select a desired size of the dictionary, and then look at the most frequent tokens in the text to fill it up, with every other symbol going into a special "out-of-vocabulary" (OOV) token. Subword tokenizers have many specialized algorithms to this end, such as byte-pair encoding (BPE) [SKF+99].[1]

Because large collections of text can have a wide variability, pre-trained subword tokenizers are a standard choice nowadays. As a concrete example, OpenAI has released an open-source version of its own tokenizer,[2] which is a subword model consisting of approximately 100k subwords (at the time of writing). Consider for example the encoding of "*This is perplexing!*" with this tokenizer, shown in Figure F.8.3. Some tokens correspond to entire words (e.g., "*This*"), some to pieces of a word (e.g, "*perplex*"), while others to punctuation marks. The sequence can be equivalently represented by a sequence of integers:

$$[2028, 374, 74252, 287, 0] \tag{E.8.4}$$

Each integer spans between 0 and the size of the vocabulary (in this case, roughly

[1] This is a short exposition focused on differentiable models, and we are ignoring many preprocessing operations that can be applied to text, such as removing stop words, punctuation, "stemming", and so on. As the size of the models has grown, these operations have become less common.

[2] https://github.com/openai/tiktoken

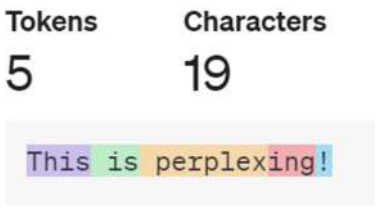


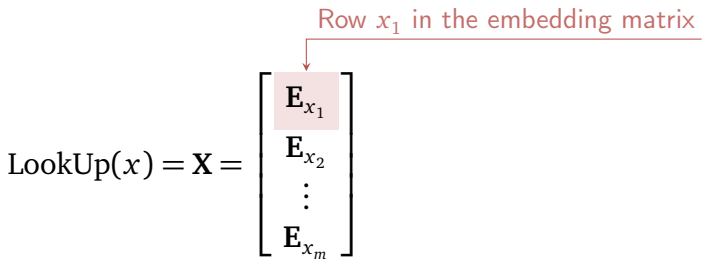
Figure F.8.3: *Example of applying the tiktoken tokenizer to a sentence.*

100k), and it uniquely identifies the token with respect to that vocabulary. In practice, nothing prevents us from adding "special" tokens to the sequence, such as tokens representing the beginning of the sentence (sometimes denoted as [BOS]), OOV tokens, or anything else. The [BOS] token will be of special significance in the next section.

Subword tokenization with very large dictionaries can be counter-intuitive at times: for example, common digits such as 52 have their unique token, while digits like 2512 can be split into a "251" token and a "2" token. For applications where processing numbers is important, specialized numerical tokenizers can be applied [GPE+23]. In general, visualizing the tokenization process is always important to debug the models' behaviour.

After the tokenization step, the tokens must be **embedded** into vectors to be used as inputs for a CNN. A simple one-hot encoding strategy here works poorly, since vocabularies are large and the resulting vectors would be significantly sparse. Instead, we have two alternative strategies: the first is to use *pretrained* networks that perform the embedding for us; we will consider this option later on, when we introduce transformers. In order to build some intuition for it, we consider here the second alternative, *training* the embeddings together with the rest of the network.

Suppose we fix an embedding dimension e as a hyper-parameter. Since the size n of the dictionary is also fixed, we can initialize a matrix of embeddings $\mathbf{E} \sim (n, e)$. We now define a look-up operation that replaces each integer with the corresponding row in $\mathbf{E}$. Denoting by x the sequence of IDs we have:



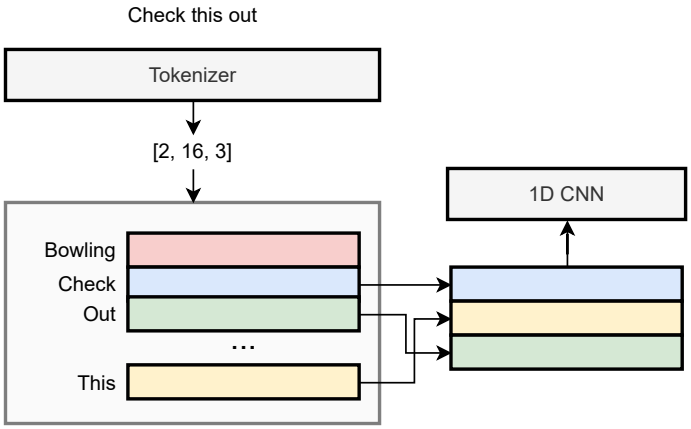


Figure F.8.4: *A lookup table to convert a sequence of tokens' IDs to their corresponding embeddings: the input is a list, the output is a matrix. The embeddings (shown inside the box) can be trained together with all the other parameters via gradient descent. We assume the size of the vocabulary is* $n = 16$.

The resulting input matrix $\mathbf{X}$ will have shape (m, e), where m is the length of the sequence. We can now apply a generic 1D convolutional model for, e.g., classifying the text sequence:

$$\hat{y} = \text{CNN}(\mathbf{X})$$

This model can be trained in a standard way depending on the task, except that gradient descent will be performed jointly on the parameters of the model and the embedding matrix $\mathbf{E}$. This is shown visually in Figure F.8.4, and an example of model's definition is given in Box C.8.2.

This idea is extremely powerful, especially because in many cases we find that the resulting embeddings can be manipulated algebraically as vectors, e.g., by looking at the closest embeddings in an Euclidean sense to find "semantically similar" words or sentences. This idea is at the core of the use of differentiable models in many sectors that necessitate retrieval or search of documents.

Differentiable models and embeddings

Once again, the idea of embedding is very general: any procedure that converts an object into a vector with algebraic characteristics is an embedding. For example, the output of the backbone of a trained CNN after global pooling can be understood as a high-level embedding of the input image, and it can be used to retrieve "similar" images by comparing it to all other embeddings.

```
class TextCNN(nn.Module):
  def __init__(self, n, e):
    super().__init__()
    self.emb = nn.Embedding(n, e)
    self.conv1 = nn.Conv1d(e, 32, 5, padding='same')
    self.conv2 = nn.Conv1d(32, 64, 5, padding='same')
    self.head = nn.Linear(64, 10)

  def forward(self, x):       # (*, m)
    x = self.emb(x)           # (*, m, e)
    x = x.transpose(1, 2)     # (*, e, m)
    x = relu(self.conv1(x))   # (*, 32, m)
    x = max_pool1d(x, 2)      # (*, 32, m/2)
    x = relu(self.conv2(x))   # (*, 64, m/2)
    x = x.mean(2)             # (*, 64)
    return self.head(x)       # (*, 10)
```

Box C.8.2: *A 1D CNN with trainable embeddings. n is the size of the dictionary, e is the size of each embedding. We use two convolutional layers with* 32 *and* 64 *output channels. The shape of the output for each operation in the forward pass is shown as a comment.*

8.2.3 Dealing with long sequences

Many of the sequences described before can be very long. In this case, the locality of convolutional layers can be a drawback, because we need a linearly increasing number of layers to process larger and larger receptive fields. We will see in the next chapters that other classes of models (e.g., transformers) can be designed to solve this problem. For now we remain in the realm of convolutions and we show one interesting solution, called **dilated** (or **atrous**, from the French *à trous*) convolutions, popularized in the WaveNet model for speech generation [ODZ+16].

We introduce an additional hyper-parameter called the **dilation rate**. A convolution with dilation rate of 1 is a standard convolution. For a dilation rate of 2, we modify the convolution operation to select elements for our patch by skipping one out of two elements in the sequence. Similarly, for a dilation rate of 4, we skip three elements over four, etc. We stack convolutional layers with exponentially increasing dilation rates, as shown in Figure F.8.5.

The number of parameters does not change, since the number of neighbors remain constant irrespective of the dilation rate. However, it is easy to show that the resulting receptive field in this case grows *exponentially fast* in the number of layers.

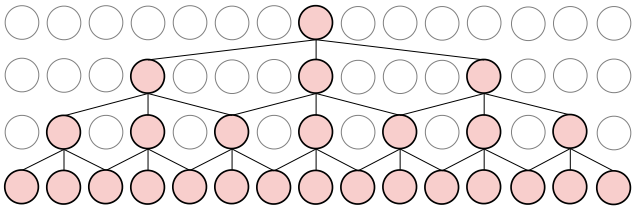


Figure F.8.5: *Convolutional layers with increasing dilation rates. Elements selected for the convolution are in red, the others are greyed out. We show the receptive field for a single output element.*

8.3 Forecasting and causal models

8.3.1 Forecasting sequences

One important aspect of working with sequences is that we can build a model to predict future elements, e.g., energy prices, turbulence flows, call center occupations, etc. Predicting tokens is also the fundamental building block for large language models and other recent breakthroughs. In a very broad sense, much of the current excitement around neural networks revolves around the question of how much a model can be expected to infer from next-token prediction on large corpora of text, and how much this setup can be replicated across different modalities (e.g., videos) and dynamics [WFD+23]. Formally, predicting the next element of a sequence is called **forecasting** in statistics and time series analysis. From now on, to be consistent with modern literature, we will use the generic term **token** to refer to each element of the sequence, irrespective of whether we are dealing with an embedded text token or a generic vector-valued input.

Stationarity and forecasting

Just like text processing, forecasting real-world time series has a number of associated problems (e.g., the possible non-stationarity of the time series, trends and seasonalities) that we do not consider here.[a] In practice, audio, text, and many other sequences of interest can be considered stationary and do not need special preprocessing. Like for text, for very large forecasting datasets and correspondingly large models, the impact of preprocessing tend to diminish [AST+24].

[a] https://filippomb.github.io/python-time-series-handbook/

The reason forecasting is an important problem is that we can train a forecasting model by just having access to a set of sequences, with no need for additional target labels: in modern terms, this is also called a **self-supervised learning** task, since the targets can be automatically extracted from the inputs.

To this end, suppose we fix a user-defined length t, and we extract all possible subsequences of length t from the dataset (e.g., with $t = 12$, all consecutive windows of 12 elements, or all sentences composed of 12 tokens, etc.). In the context of LLMs, the size of the input sequence is called the **context** of the model. We associate to each subsequence a target value which is the next element in the sequence itself. Thus, we build a set of pairs $(\mathbf{X}, \mathbf{y}), \mathbf{X} \sim (t, c)$, $y \sim (c)$ and our forecasting model is trained in a supervised way over this dataset:

$$f(\mathbf{X}) \approx \mathbf{y}$$

Note that a standard 1D convolutional model can be used as forecasting model, trained with either mean-squared error (for continuous time series) or cross-entropy (for categorical sequences, such as text). While the model is trained to predict a single step-ahead, we can easily use it to generate as many steps as we want by what is called an **autoregressive** approach, meaning that the model is predicting (*regressing*) on its own outputs. Suppose we predict a single step, $\widehat{\mathbf{y}} = f(\mathbf{X})$, and we create a "shifted" input by adding our predicted value to the input (removing the first element to avoid exceeding t elements):

$$\mathbf{X}' = \begin{bmatrix} \mathbf{X}_{2:t} \\ \widehat{\mathbf{y}} \end{bmatrix} \tag{E.8.5}$$

Window of $t-1$ input elements

Predicted value at time $t+1$

Forecasting discrete sequences

For a continuous time series this is trivial. For a time series with discrete values, f will return a probability vector over the possible values (i.e., possible tokens), and we can obtain $\widehat{\mathbf{y}}$ by taking its arg max, i.e., the token associated to the highest probability. Alternatively, we can sample a token proportionally to the predicted probabilities: see Section 8.4.1.

We can now run $f(\mathbf{X}')$ to generate the next input value in the sequence, and so on iteratively, by always updating our buffered input in a FIFO fashion. This approach is extremely powerful, but it requires us to fix a priori the input sequence length, which limits its applicability. To overcome this limitation, we need only a minor modification to our models.

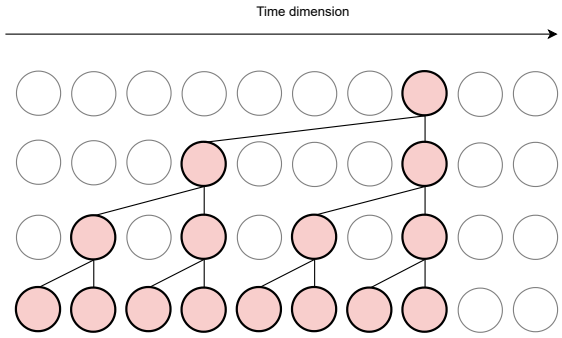


Figure F.8.6: *Overview of a 1D causal convolutional layer with (original) kernel size of 3 and exponentially increasing dilation rates. Zeroed out connections are removed, and we show the receptive field for a single output element.*

8.3.2 Causal models

Suppose we only have available a short sequence of 4 elements collected into a matrix $\mathbf{X} \sim (4, c)$, but we have trained a forecasting model on longer sequences with $t = 6$. In order to run the model on the shorter sequence, we can zero-pad the sequence with two zero vectors $\mathbf{0}$ at the beginning, but these will be interpreted by the model as actual values of the time series unless we mask its operations. Luckily, there is a simpler and more elegant approach in the form of **causal** models.

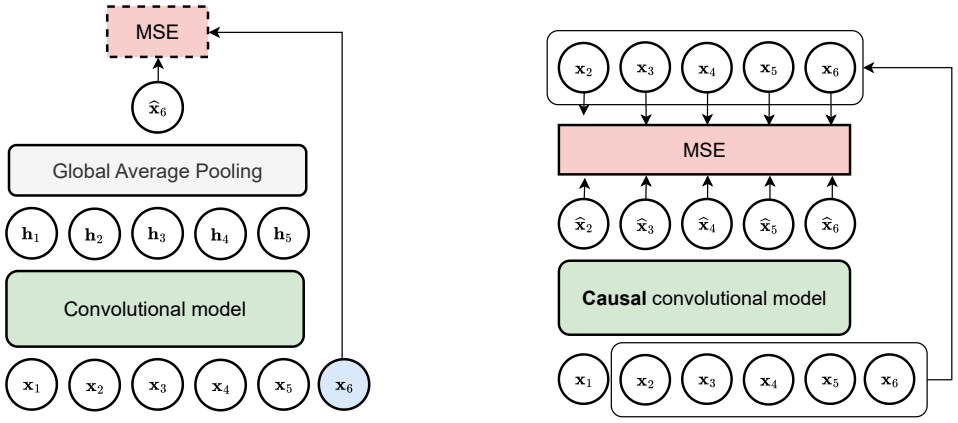
> **Definition D.8.1 (Causal layer)** *A layer $\mathbf{H} = f(\mathbf{X})$ is **causal** if $\mathbf{H}_i = f(\mathbf{X}_{:i})$, i.e., the value corresponding to the i-th element of the sequence depends only on elements "from its past".*

A model composed only of causal layers will, of course, be causal itself. For example, a convolutional layer with kernel size 1 is causal, since each element is processed considering only itself. However, a convolutional layer with kernel size 3 is not causal, since it is processed considering in addition one element to the left and one element to the right. We can convert any convolution into a causal variant by partially zero masking the weights corresponding to non-causal connections:

$$\mathbf{h}_i = \phi\left(\left[\,\mathbf{W} \odot \mathbf{M}\,\right]\text{vect}(P_k(i)) + \mathbf{b}\right)$$

Masked weight matrix

where $M_{ij} = 0$ if the weight corresponds to an element in the input such that $j > i$, 1 otherwise. Causal 1D convolutions can be combined with dilated kernels to obtain autoregressive models for audio, such as in the WaveNet model [ODZ+16] - see Figure F.8.6 for an example.



(a) *Non-causal model* **(b)** *Causal model*

Figure F.8.7: *Comparison between (a) a non-causal model for forecasting (predicting only a single element for the entire input sequence) and (b) a causal model trained to predict one output element for each input element in the sequence.*

Masking is easier to understand in the case of a single channel, in which case $\mathbf{M}$ is simply a lower-triangular binary matrix. The masking operation effectively reduces the number of parameters from $(2k+1)cc'$ to $(k+1)cc'$.

By stacking several causal convolutional layers, we can obtain a causal 1D model variant. Suppose we apply it on our input sequence, with a model that has no max-pooling operations. In this case, the output sequence has the same length as the input sequence:

$$\widehat{\mathbf{Y}} = f_{\text{causal}}(\mathbf{X})$$

In addition, any element in the output only depends on input elements in the same position or preceding it. Hence, we can define a more sophisticated forecasting model by predicting a value *for each element of the input sequence*. Practically, consider now a matrix output defined as:

$$\mathbf{Y} = \begin{bmatrix} \mathbf{X}_{2:t} \\ \mathbf{y} \end{bmatrix}$$

This is similar to the shifted input from (E.8.5), except that we are adding the true value as last element of the sequence. We can train this model by minimizing a loss on

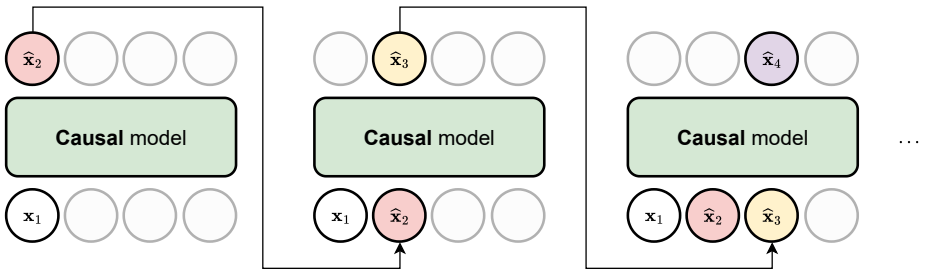


Figure F.8.8: *Inference with a causal CNN, generating a sequence step-by-step in an autoregressive way. Unused input tokens are greyed out. Generated tokens are colored with different colors to distinguish them.*

all elements, e.g., a mean-squared error:

$$l(\widehat{\mathbf{Y}}, \mathbf{Y}) = \|\widehat{\mathbf{Y}} - \mathbf{Y}\|^2 = \sum_{i=1}^{t} \|\widehat{\mathbf{Y}}_i - \mathbf{Y}_i\|^2 \tag{E.8.6}$$

Loss when predicting $\mathbf{X}_{i+1}$

We simultaneously predict the second element based on the first one, the third one based on the first two, etc. For a single input window, we have t separate loss terms, greatly enhancing the gradient propagation. A comparison between the two approaches is shown in Figure F.8.7: in Figure F.8.7a we show a non-causal convolutional model trained to predict the next element in the sequence, while in Figure F.8.7b we show a causal model trained according to (E.8.6).

More importantly, we can now use the model in an autoregressive way with any sequence length up to the maximum length of t. This can be seen easily with an example. Suppose we have $t = 4$, and we have observed two values $\mathbf{x}_1$ and $\mathbf{x}_2$. We call the model a first time by zero-padding the sequence to generate the third token:

$$\begin{bmatrix} - \\ \widehat{\mathbf{x}}_3 \\ - \\ - \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}\right)$$

We are ignoring all output values except the second one (in fact, the third and fourth outputs are invalid due to the zero-padding). We add $\widehat{\mathbf{x}}_3$ to the sequence and continue calling the model autoregressively (we show in color the predicted values):

$$\begin{bmatrix} - \\ - \\ \widehat{\mathbf{x}}_4 \\ - \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \widehat{\mathbf{x}}_3 \\ \mathbf{0} \end{bmatrix}\right), \begin{bmatrix} - \\ - \\ - \\ \widehat{\mathbf{x}}_5 \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \widehat{\mathbf{x}}_3 \\ \widehat{\mathbf{x}}_4 \end{bmatrix}\right), \begin{bmatrix} - \\ - \\ - \\ \widehat{\mathbf{x}}_6 \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_2 \\ \widehat{\mathbf{x}}_3 \\ \widehat{\mathbf{x}}_4 \\ \widehat{\mathbf{x}}_5 \end{bmatrix}\right) \dots$$

In the last step we removed one of the original inputs to keep the constraint on the size of the input. This is also shown in Figure F.8.8. Note that the model is trained only on real values, not on its own predictions: this is called **teacher forcing**. A variant of teacher forcing is to progressively replace some of the values in the mini-batches with values predicted by the model, as training proceeds and the model becomes more accurate.

Causal autoregressive models are especially interesting in the case of text sequences (where we only have a single channel, the index of the tokens), since we can start from a single [BOS] token representing the beginning of the sequence and generate text sentences from scratch, or *condition* the generation on a specific prompt by the user which is appended to the [BOS] token. A similar reasoning can be applied to audio models to generate speech or music [ODZ+16].

8.4 Autoregressive and generative models

8.4.1 A probabilistic formulation of generative models

An autoregressive model is a simple example of a **generative model**.[3] We will talk at length about other types of generative models in the next volume. For now, we provide some insights specific to autoregressive algorithms. We consider sequences with a single channel and discrete values, such as text. Autoregressive models over text tokens are the foundation of LLMs, and they can be used as the basis for multimodal architectures (Chapter 11).

Generative models are more naturally framed in the context of probabilities, so we begin by reframing our previous discussion with a probabilistic formalism. Denote by

[3] Remember from Chapter 3 that we assume our supervised pairs (x, y) come from some unknown probability distribution $p(x, y)$. By the product rule of probability we can decompose it equivalently as $p(y \mid x)p(x)$, or $p(x \mid y)p(y)$. Any model which approximates $p(x)$ or $p(x \mid y)$ is called **generative**, because you can use it to sample new input points. By contrast, a model that only approximates $p(y \mid x)$, like we did in the previous chapters, is called **discriminative**.

$\mathcal{X}$ the space of all possible sequences (e.g., all possible combinations of text tokens). In general, many of these sequences will be invalid, such as the sequence ["*tt*", "*tt*"] in English. However, even very uncommon sequences may appear at least once or twice in very large corpora of text (imagine a character yelling "*Scotttt!*").

We can generalize this by considering a probability distribution $p(x)$ over all possible sequences $x \in \mathcal{X}$. In the context of text, this is also called a **language model**. Generative modeling is the task of learning to sample efficiently from this distribution:[4]

$$x \sim p(x)$$

To see how this connects to our previous discussion, note that by the product rule of probability we can always rewrite $p(x)$ as:

$$p(x) = \prod_i p(x_i \mid x_{:i}) \tag{E.8.7}$$

where we condition each value x_i to all preceding values. If we assume that our model input length is large enough to accommodate all possible sequences, we can use a causal forecasting model to parameterize the probability distribution in (E.8.7):

$$p(x_i \mid x_{:i}) = \text{Categorical}(x_i \mid f(x_{:i}))$$

where we use a single, shared model for all time-steps. Maximum likelihood over this model is then equivalent to minimizing a cross-entropy loss over the predicted probabilities, as in Section 4.2.2.

8.4.2 Sampling in an autoregressive model

In general, sampling from a probability distribution is non-trivial. However, for autoregressive models we can exploit the product decomposition in (E.8.7) to devise a simple iterative strategy:

1. Sample $x_1 \sim p(x_1)$. This is equivalent to conditioning on the empty set $p(x_1 \mid \{\})$. In practice, we always condition on an initial fixed token, such as the [BOS] token, so that our input is never empty.

2. Sample $x_2 \sim p(x_2 \mid x_1)$ by running again the network with the value we sampled

[4] In this section $\sim$ is used to denote sampling from a probability distribution instead of the shape of a tensor.

at step (1), as in Figure F.8.8.

3. Sample $x_3 \sim p(x_3 \mid x_1, x_2)$.

4. Continue until we reach a desired sequence length or until we get to an end-of-sentence token.

We did this implicitly before by always sampling the element of highest probability:

$$x_i = \arg\max_i f(x_{:i})$$

However, we can also generalize this by sampling a value according to the probabilities predicted by f. Remember (Section 4.2.1) that the softmax can be generalized by considering an additional temperature parameter. By varying this parameter during inference, we can vary smoothly between always taking the argmax value (very low temperature) to having an almost uniform distribution over tokens (very high temperature).

In the context of probabilistic modelling, sampling in this way from this class of models is called **ancestral sampling**, while in the context of language modelling we sometimes use the term **greedy decoding**. The use of the term "greedy" and this brief discussion is enough to highlight one potential drawback of these models: while the product decomposition of $p(x)$ is exact, greedy decoding is not guaranteed to provide a sample corresponding to high values of $p(x)$.

To see this, note that f provides an estimate of the probability for a single token, but the probability of a sequence is given by a product of many such terms. Hence, sampling a token with high (local) probability at the beginning of a sequence may not correspond to a sequence having large (global) probability as a sentence. This is easy to visualize if you imagine the choice of the first token letting the decoding stage being "stuck" in a low-probability path.

A common mitigation to this problem is **beam search** (or **beam decoding**). In beam search, in the first step we sample k different elements (called the beams, with k being a user-defined parameter). In the second step, for each of our k beams we sample k possible continuations. Out of these k^2 pairs, we keep only the top-k values in terms of their product probability $p(x_1)p(x_2 \mid x_1)$ (or, equivalently, their log probability). We continue iteratively in this way until the end of the sequence.

Viewed under this lens, sampling the most probable sequence from our autoregressive

model is a combinatorial search problem (think of a tree, where for each token we expand across all possible next tokens, and so on). From the point of view of computer programming, beam search is then an example of **breadth-first** search over this tree.

8.4.3 Conditional modelling

As we mentioned earlier, in general we may not be interested so much in generating sequences from scratch, but in generating continuations of known sequences, such as a user's question or interaction. This can be formalized by considering *conditional* probability distributions in the form $p(x \mid c)$, where c is the conditioning argument, such as a user's prompt. Our previous discussion extends almost straightforwardly to this case. For example, the product decomposition is now written as:

$$p(x \mid c) = \prod_i p(x_i \mid x_{:i}, c)$$

where we condition on the previous inputs *and* the user's context. Sampling, decoding, etc., are extended in a similar way.

To perform conditional generation we parameterize $p(x_i \mid x_{:i}, c)$ with a neural network $f(x, c)$ such that:

$$p(x_i \mid x_{:i}, c) \approx \text{Categorical}(x_i \mid f(x_{:i}, c))$$

Hence, the major difference with the unconditional case is that we need a function $f(x, c)$ having two input arguments and which satisfies causality in the first argument. When working with autoregressive models, if both x and c are texts we can do this easily be considering c as part of the input sequence and working with a single concatenated input $x' = [c \| x]$. For example, with the user's prompt "*The capital of France*", taking for simplicity a word tokenizer we might have:[5]

$$f_{\text{causal}}([\text{The}, \text{capital}, \text{of}, \text{France}]) = \text{is}$$
$$f_{\text{causal}}([\text{The}, \text{capital}, \text{of}, \text{France}, \text{is}]) = \text{Paris}$$

Hence, we can handle unconditional and conditional modelling simultaneously with a single model.[6] In the next volume we will see other examples of conditional generative

[5] We ignore the presence of an end-of-sequence token (EOS) to stop the autoregressive generation.

[6] We will see in Chapter 11 that almost any type of data can be converted into a sequence of tokens. Suppose we are generating a text sequence conditioned on an image prompt (e.g., **image captioning**).

models in which more sophisticated strategies are needed. We will also extend upon this topic when we discuss decoder-only transformer models in Chapter 11.

From theory to practice

Working with text data is more complex than image classification, due to many subtleties involved with tokenization, data formatting, weird characters, and variable-length sequences. PyTorch has its own text library, `torchtext`, which at the time of writing is less documented than the main library and relies on another beta library (`torchdata`) to handle the data pipelines. Thus, we ignore it here, but we invite you to check it out on your own.

Hugging Face Datasets is probably the most versatile tool in this case, as it provides a vast array of datasets and pre-trained tokenizers, which can be exported immediately to PyTorch.[7] Familiarize yourself a bit with the library before proceeding with the exercise.

1. Choose a text classification dataset, such as the classic IMDB dataset.[8]
2. Tokenize it to obtain a dataset of the form (x, y), where x is a list of integers as in (E.8.4) and y is the text label.
3. Build and train a 1D CNN model similar to Box C.8.2. Experiment a bit with the model's design to see its impact on the final accuracy.

PyTorch does not have a quick way to make a 1D convolution causal, so we will postpone our autoregressive experiments for when we introduce transformers.[9] Training your own tokenizer is a very good didactic exercise, although it's far beyond the scope of the book. For an introduction, you can check this minimalistic BPE implementation: https://github.com/karpathy/minbpe.

If both text and images are converted to tokens having the same embedding size, we can apply an autoregressive model by concatenating the tokens from the two input types (also called **modalities** in this context), where we view the image tokens as the conditioning set c.

[7] See this tutorial for a guide: https://huggingface.co/docs/datasets/use_dataset.

[8] https://huggingface.co/datasets/stanfordnlp/imdb

[9] If you want to try, you can emulate a causal convolution with proper padding; see Lecture 10.2 here: https://fleuret.org/dlc/. The entire course is really good if you are looking for streamed lectures.

9 | Scaling up the models

About this chapter

We now turn to the task of designing differentiable models having dozens (or hundreds) of layers. As we saw, the receptive field of convolutional models grows linearly with the number of layers, motivating architectures with such depth. This can be done by properly stabilizing training using a plethora of methods, ranging from data augmentation to normalization of the hidden states.

9.1 The ImageNet challenge

Let us consider again the task of image classification, which holds a strong interest for neural networks, both practically and historically. In fact, interest in these models in the period 2012-2018 can be associated in large part to the **ImageNet Large Scale Visual Recognition Challenge**[1] (later **ImageNet** for simplicity). ImageNet was a yearly challenge that run from 2010 to 2017 to evaluate state-of-the-art models for image classification. The challenge was run on a subset of the entire ImageNet dataset, consisting of approximately 1M images tagged across 1k classes.

It is instructive to take a look at the early editions of the challenges. In 2010[2] and in 2011,[3] the winners were linear kernels methods built with a combination of specialized image descriptors and kernels, with a top-5% error of 28% (2010) and 26% (2011). Despite a number of promising results,[4] convolutional models trained by gra-

[1] https://image-net.org/challenges/LSVRC/
[2] https://image-net.org/challenges/LSVRC/2010/
[3] https://image-net.org/challenges/LSVRC/2011/
[4] https://people.idsia.ch/~juergen/computer-vision-contests-won-by-gpu-cnns.html

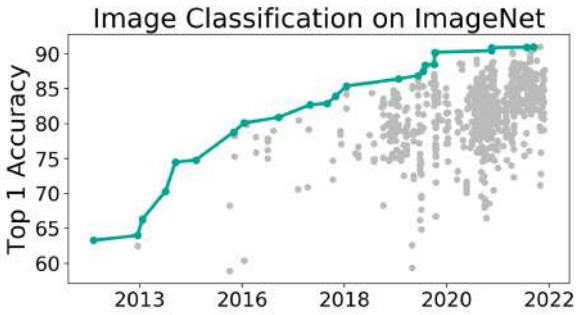


Figure F.9.1: *Top-1 accuracy on the ImageNet dataset. Reproduced from Papers With Code.*[a]

[a] https://paperswithcode.com/

dient descent remained a niche topic in computer vision. Then, In 2012 the winner model (AlexNet, [KSH12]) achieved a top-5% error of 15.3%, 10% lower than all (non-neural) competitors.

This was followed by a veritable "Copernican revolution" (apologies to Copernicus) in the field, since in a matter of a few years almost all submissions turned to convolutional models, and the overall accuracy grew at an unprecedented speed, upward of 95% (leading to the end of the challenge in 2017), as shown in Figure F.9.1. In a span of 5 years, convolutional models trained with gradient descent became the leading paradigm in computer vision, including other subfields we are not mentioning here, from object detection to semantic segmentation and depth estimation.

AlexNet was a relatively simple model consisting of 5 convolutional layers and 3 fully-connected layers, totaling approximately 60M parameters, while the top-performing models in Figure F.9.1 require up to hundreds of layers. This is basic example of a scaling law (Chapter 1): adding layers and compute power for training is proportionally linked to the accuracy of the model up to a saturation point given by the dataset. However, scaling up convolutional models beyond a few layers is non-trivial, as it runs into a number of problems ranging from slow optimization to gradient issues and numerical instabilities. As a consequence, a large array of techniques were developed in 2012-2017 to stabilize training of very large models.

In this chapter we provide an overview of some of these techniques. We focus on ideas and methods that are still fundamental nowadays, even for other architectures (e.g., transformers). We begin by three techniques to improve training that are well-known in machine learning: **weight regularization**, **data augmentation**, and **early stopping**. Then, we describe three of the most influential techniques popularized in 2012-2017: **dropout**, **batch normalization**, and **residual connections**, more or less in chronolog-

ical order of introduction. For each method we describe the basic algorithm along with some variants that work well in practice (e.g., layer normalization).

9.2 Data and training strategies

9.2.1 Weight regularization

One possible way to improve training is to penalize solutions that may seem unplausible, such as having one or two extremely large weights. Denote by $\mathbf{w}$ the vector of all parameters of our model, and by $L(\mathbf{w}, \mathcal{S}_n)$ the loss function on our dataset (e.g., average cross-entropy). We can formalize the previous idea by defining a so-called **regularization term** $R(\mathbf{w})$ that scores solutions based on our preference, and penalize the loss by adding the regularization term to the original loss function:

$$L_{\text{reg}} = L(\mathbf{w}, \mathcal{S}_n) + \lambda R(\mathbf{w})$$

where we assume that a higher value of $R(\mathbf{w})$ corresponds to a worse solution, and $\lambda \geq 0$ is a scalar that weights the two terms. For $\lambda = 0$ the regularization term has no effect, while for $\lambda \to \infty$ we simply select the best function based on our a priori knowledge.

This can also be justified as performing maximum a-priori (instead of maximum likelihood) inference based on the combination of a prior distribution on the weights $p(\mathbf{w})$ and a standard likelihood function on our data (Section 3.3):

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \{\log p(\mathcal{S}_n \mid \mathbf{w}) + \log p(\mathbf{w})\} \tag{E.9.1}$$

where having a regularization term corresponds to a non-uniform prior distribution $p(\mathbf{w})$. We have already seen one example of regularization in Section 4.1.6, i.e., the ℓ_2 norm of the weights:

$$R(\mathbf{w}) = \|\mathbf{w}\|^2 = \sum_i w_i^2$$

For the same unregularized loss, penalizing the ℓ_2 norm will favor solutions with a lower weight magnitude, corresponding to "less abrupt" changes in the output for a small deviation in the input.[5] Consider now the effect of the regularization term on

[5] With respect to (E.9.1), ℓ_2 regularization is equivalent to choosing a Gaussian prior on the weights with diagonal $\sigma^2\mathbf{I}$ covariance.

the gradient term:

$$\nabla L_{\text{reg}} = \nabla L(\mathbf{w}, \mathscr{S}_n) + 2\lambda \mathbf{w}$$

Written in this form, this is sometimes called **weight decay**, because absent the first term, its net effect is to decay the weights by a small proportional factor λ (sending them to 0 exponentially fast in the number of iterations if $\nabla L(\mathbf{w}, \mathscr{S}_n) = 0$). For (S)GD, ℓ_2 regularization and weight decay coincide. However, for other types of optimization algorithms (e.g., momentum-based SGD, Adam), a post-processing is generally applied on the gradients. Denoting by $g(\nabla L(\mathbf{w}, \mathscr{S}_n))$ the post-processed gradients of the (unregularized) loss, we can write a generalized weight decay formulation (ignoring the constant term 2) as:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - g(\nabla L(\mathbf{w}_{t-1}, \mathscr{S}_n)) - \lambda \mathbf{w}_{t-1}$$

Unregularized gradient

Weight decay term

This is different from pure ℓ_2 regularization, in which case the gradients of the regularization term would be inside $g(\cdot)$. This is especially important for algorithms like Adam, for which the weight decay formulation can work better [LH19].

We can also consider other types of regularization terms. For example, the ℓ_1 norm:

$$R(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_i |x_i|$$

can favor *sparse* solutions having a high percentage of zero values (and it corresponds to placing a Laplace prior on the weights). This can also be generalized to group sparse variants to enforce structured sparsity on the neurons [SCHU17].[6] Sparse ℓ_1 penalization is less common than for other machine learning models because it does not interact well with the strong non-convexity of the optimization problem and the use of gradient descent [ZW23]. However, it is possible to re-parameterize the optimization problem to mitigate this issue at the cost of a larger memory footprint. In particular, [ZW23] showed that we can replace $\mathbf{w}$ with two equivalently shaped vectors $\mathbf{a}$ and $\mathbf{b}$, and:

$$\mathbf{w} = \mathbf{a} \odot \mathbf{b}\,, \ \|\mathbf{w}\|_1 \approx \|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 \tag{E.9.2}$$

where $\approx$ means that the two problems can be shown to be almost equivalent under

[6] Training sparse models is a huge topic with many connections also to efficient hardware execution. See [BJMO12] for a review on sparse penalties in the context of convex models, and [HABN+21] for an overview of sparsity and pruning in general differentiable models.

very general conditions [ZW23].

We can gain some geometric insights as to why (and how) regularization works by considering a convex loss function $L(\cdot,\cdot)$ (e.g., least-squares), in which case the regularized problem can be rewritten in an explicitly constrained form as:

$$\begin{array}{ll} \arg\min & L(\mathbf{w}, \mathscr{S}_n) \\ \text{subject to} & R(\mathbf{w}) \leq \mu \end{array} \tag{E.9.3}$$

where μ depends proportionally on λ, with the unconstrained formulation arising by rewriting (E.9.3) with a Lagrange multiplier. In this case, ℓ_2 regularization corresponds to constraining the solution to lie inside a circle centered in the origin, while ℓ_1 regularization corresponds to having a solution inside (or on the vertices) of a regular polyhedron centered in the origin, with the sparse solutions lying at the vertices intersecting the axes.

9.2.2 Early stopping

From the point of view of optimization, minimizing a function $L(\mathbf{w})$ is the task of finding a stationary point as quickly as possible, i.e., a point $\mathbf{w}_t$ such that $\nabla L(\mathbf{w}_t) \approx 0$:

$$\|L(\mathbf{w}_t) - L(\mathbf{w}_{t-1})\|^2 \leq \varepsilon$$

for some tolerance $\varepsilon > 0$. However, this does not necessarily correspond to what we want when optimizing a model. In particular, in a low-data regime training for too long can incur in overfitting and, in general, anything which improves generalization is good irrespective of its net effect on the value on $L(\cdot)$ or the descent direction (e.g., weight decay).

Early stopping is a simple example of the difference between pure optimization and learning. Suppose we have access to a small supervised dataset, separate from the training and test dataset, that we call **validation dataset**. At the end of every epoch, we track a metric of interest on the validation dataset, such as the accuracy or the F1-score. We denote the score at the t-th epoch as a_t. The idea of early stopping is to check this metric to see if it keeps improving: if not, we may be entering an overfitting regime and we should stop training. Because the accuracy can oscillate a bit due to random fluctuations, we do this robustly by considering a window of k epochs (the **patience**):

$$\text{If } a_t \leq a_i, \quad \underbrace{\forall i = t-1, t-1, \ldots, t-k}_{\text{Wait for } k \text{ epochs}} \rightarrow \text{Stop training}$$

For a high value of the patience hyper-parameter k, the algorithm will wait more, but we will be more robust to possible oscillations. If we have a mechanism to store the weights of the model (**checkpointing**) we can also rollback the weights to the last epoch that showed improvement, corresponding to the epoch number $t-k$.

Early stopping can be seen as a simple form of **model selection**, where we select the optimal number of epochs based on a given metric. Differently from the optimization of the model, we can optimize here for any metric of interest, such as the F1-score, even if not differentiable.

Interestingly, for large over-parameterized models early stopping is not always beneficial, as the relation between epochs and validation error can be non-monotone with multiple phases of ascent and descent (a phenomenon called **multiple descents** [RM22]) and sudden drops in the loss after long periods of stasis [PBE+22]. Hence, early stopping is useful mostly when optimizing on small datasets.

9.2.3 Data augmentation

Generally speaking, the most effective method to improve performance for a model is to increase the amount of available data. However, labelling data can be costly and time-consuming, and generating data artificially (e.g., with the help of large language models) requires customized pipelines to work effectively [PRCB24].

In many cases, it is possible to partially mitigate this issue by *virtually* increasing the amount of available data by transforming them according to some pre-specified number of (semantic preserving) transformations. As a simple example, consider a vector input $\mathbf{x}$ and a transformation induced by adding Gaussian noise:

$$\mathbf{x}' = \mathbf{x} + \varepsilon, \; \varepsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})$$

This creates a virtually infinite amount of data comprised in a small ball centered around $\mathbf{x}$. In addition, this data must not be stored in the disk, and the process can be simulated by applying the transformation at runtime every time a new mini-batch is selected. In fact, it is known that training in this way can make the model more robust

and it is connected to ℓ_2 regularization [Bis95]. However, vectorial data is unstructured, and adding noise with too high variance can generate points that are invalid.

For images, we can do better by noting that there is in general a large number of transformations that can change an image while preserving its semantic: zooms, rotations, brightness modifications, contrast changes, etc. Denote by $T(x; c)$ one such transformation (e.g., rotation), parameterized by some parameter c (e.g., the rotation angle). Most transformations include the base image as a special case (in this case, for example, with a rotation angle $c = 0$). **Data augmentation** is the process of transforming images during training according to one or more of these transformations:

$$x' = T(x; c),\ c \sim p(c) \tag{E.9.4}$$

where $p(c)$ denotes the distribution of all valid parameters (e.g., rotation angles between $-20°$ and $+20°$). During training, each element of the dataset is sampled once per epoch, and each time a different transformation (E.9.4) can be applied, creating a (virtually) unlimited stream of unique data points.

Data augmentation is very common for images (or similar data, such as audio and video), but it requires a number of design choices: what transformations to include, which parameters to consider, and how to compose these transformations. A simple strategy called **RandAugment** [CZSL20] considers a wide set of transformations, and for every mini-batch samples a small number of them (e.g., 2 or 3), to be applied sequentially with the same magnitude. Still, the user must verify that the transformations are valid (e.g., if recognizing text, horizontal flipping can make the resulting image invalid). From a practical point of view, data augmentation can be included either as part of the data loading components (see Box C.9.1), or as part of the model.

Data augmentation pipelines and methods can be more complex than simple intuitive transformations. Even for more sophisticated types, the intuition remains that, as long as the model is able to solve a task in a complex scenario (e.g., recognizing an object in all brightness conditions) it should perform even better in a realistic, mild scenario. Additionally, data augmentation can prevent overfitting by avoiding the repetition of the same input multiple times.

As an example of more sophisticated methods, we describe **mixup** [ZCDLP17] for vectors, and its extension **cutmix** [YHO+19] for images. For the former, suppose we sample two examples, $(\mathbf{x}_1, y_1)$ and $(\mathbf{x}_2, y_2)$. The idea of mixup is to create a new, virtual

```python
# Image tensor (batch, channels, height, width)
img = torch.randint(0, 256, size=(32, 3, 256, 256))

# Data augmentation pipeline
from torchvision.transforms import v2
transforms = v2.Compose([
    v2.RandomHorizontalFlip(p=0.5),
    v2.RandomRotation(10),
])

# Applying the data augmentation pipeline: each function
# call returns a different mini-batch starting from the
# same input tensor.
img = transforms(img)
```

Box C.9.1: *Data augmentation pipeline with two transformations applied in sequence, taken from the* `torchvision` *package. In PyTorch, augmentations can be passed to the data loaders or used independently. In other frameworks, such as TensorFlow and Keras, data augmentation can also be included natively as layers inside the model.*

example which is given by their convex combination:

$$\mathbf{x} = \lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2 \tag{E.9.5}$$

$$y = \lambda y_1 + (1-\lambda) y_2 \tag{E.9.6}$$

where λ is chosen randomly in the interval $[0,1]$. This procedure should push the model to have a simple (linear) output in-between the two examples, avoiding abrupt changes in output. From a geometric viewpoint, for two points that are close, we can think of (E.9.6) as slowly moving on the manifold of the data, by following the line that connects two points as λ goes from 0 to 1.

Mixup may not work for images, because linearly interpolating two images pixel-by-pixel gives rise to blurred images. With **cutmix**, we sample instead a small patch of fixed shape (e.g., 32×32) on the first image. Denote by $\mathbf{M}$ a binary mask of the same shape as the images, with 1 for pixels inside the patch, and 0 for pixels outside the patch. In cutmix, we combine two images x_1 and x_2 by "stitching" a piece from the first one on top of the second one:

$$x = \mathbf{M} \odot x_1 + (1 - \mathbf{M}) \odot x_2$$

while the labels are still linearly interpolated as before with a random coefficient λ.

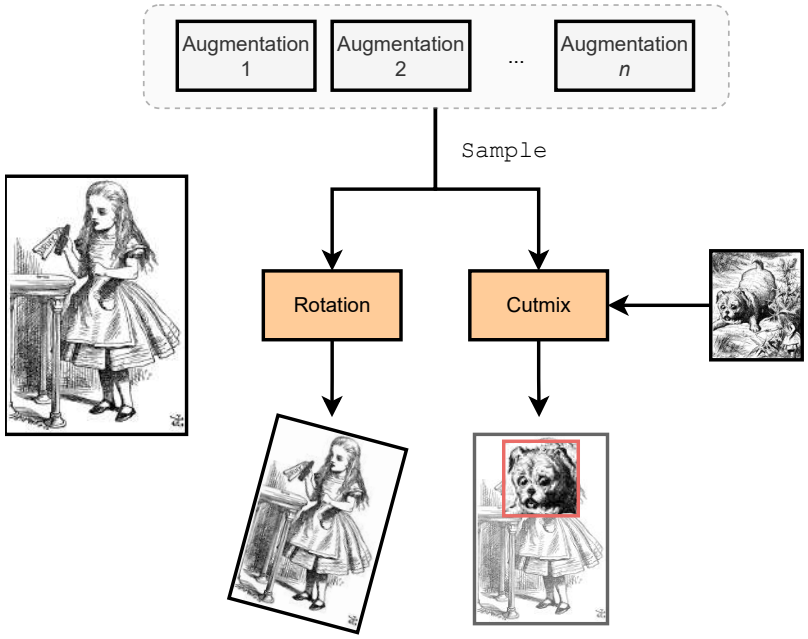


Figure F.9.2: *High-level overview of data augmentation. For every mini-batch, a set of data augmentations are randomly sampled from a base set, and they are applied to the images of the mini-batch. Here, we show an example of rotation and an example of cutmix. Illustrations by John Tenniel, reproduced from Wikimedia.*

See Figure F.9.2 for an example of data augmentation using both rotation and cutmix.

9.3 Dropout and normalization

The strategies we have described in the previous section are very general, in the sense that they imply modifications to the optimization algorithm or to the dataset itself, and they can be applied to a wide range of algorithms.

Instead, we now focus on three ideas that were popularized in the period between 2012 and 2016, mostly in the context of the ImageNet challenge. All three are specific to differentiable models, since they can be implemented as additional layers or connections in the model that simplify training of very deep models. We list the methods in roughly chronological order. As we will see in the remaining sections, these methods remain fundamental also beyond convolutional models.

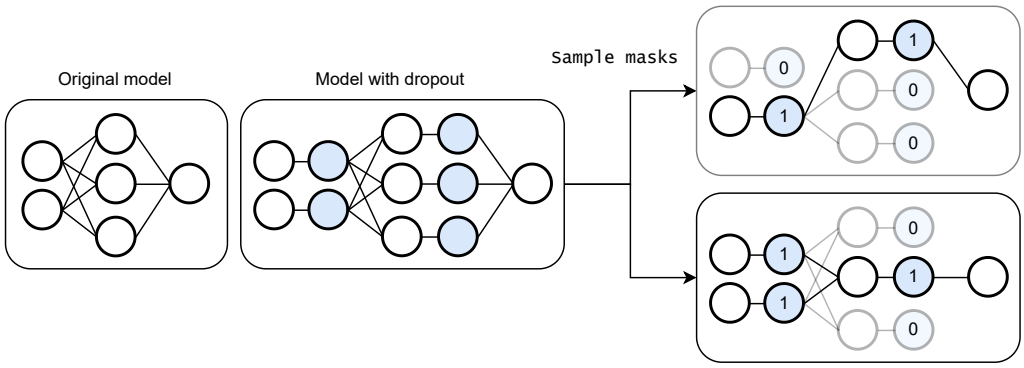


Figure F.9.3: *Schematic overview of dropout: starting from a base model, we add additional units after each layer of interest, shown in blue. At training time, each dropout unit is randomly assigned a binary value, masking part of the preceding layers. Hence, we select one out of exponentially many possible models having a subset of active hidden units every time a forward pass is made. Dropout can also be applied at the input level, by randomly removing some input features.*

9.3.1 Regularization via dropout

When discussing data augmentation, we mentioned that one insight is that augmentation forces the network to learn in a more difficult setup, so that its performance in a simpler environment can improve in terms of accuracy and robustness. **Dropout** [SHK+14] extends this idea to the internal embeddings of the model: by artificially introducing noise during training to the intermediate outputs of the model, the solution can improve.

There are many choices of possible noise types: for example, training with small amounts of Gaussian noise in the activation has always been a popular alternative in the literature of recurrent models. As the name suggests, dropout's idea is to randomly remove certain units (neurons) during the computation, reducing the dependence on any single internal feature and (hopefully) leading to training robust layers with a good amount of redundancy.

We define dropout in the case of a fully-connected layer, which is its most common use case.

Definition D.9.1 (Dropout layer) *Denote by* $\mathbf{X} \sim (n, c)$ *a mini-batch of internal activations of the model (e.g., the output of some intermediate fully-connected layer) with* n *elements in the mini-batch and* c *features. In a dropout layer, we first sam-*

```
model = nn.Sequential(
    nn.Dropout(0.3),
    nn.Linear(2, 3), nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(3, 1)
)
```

Box C.9.2: *The model in Figure F.9.3 implemented as a sequence of four layers in PyTorch. During training, the output of the model will be stochastic due to the presence of the two dropout layers.*

ple a binary matrix $\mathbf{M} \sim Binary(n, c)$ *of the same size, whose elements are drawn from a Bernoulli distribution with probability* p *(where* $p \in [0, 1]$ *is a user's hyper-parameter):*[a]

$$M_{ij} \sim Bern(p) \tag{E.9.7}$$

The output of the layer is obtained by masking the input:

$$\text{Dropout}(\mathbf{X}) = \mathbf{M} \odot \mathbf{X}$$

The layer has a single hyper-parameter, p*, and no trainable parameters.*

[a] The samples from Bern(p) are 1 with probability p and 0 with probability $1-p$.

We call $1-p$ the **drop probability**. Hence, for any element in the mini-batch, a random number of units (approximately $(1-p)$%) will be set to zero, effectively removing them. This is shown in Figure F.9.3, where the additional dropout units are shown in blue. Sampling the mask is part of the layer's forward pass: for two different forward passes, the output will be different since different elements will be masked, as shown on the right in Figure F.9.3.

As the figure shows, we can implement dropout as a layer, which is inserted after each layer that we want to drop. For example, consider the fully-connected model with two layers shown in Figure F.9.3:

$$y = (\text{FC} \circ \text{FC})(\mathbf{x})$$

Adding dropout regularization over the input and over the output of the first layer returns a new model having *four* layers:

$$y = (\text{FC} \circ \text{Dropout} \circ \text{FC} \circ \text{Dropout})(\mathbf{x})$$

See Box C.9.2 for an implementation in PyTorch.

While dropout can improve the performance, the output y is now a random variable with respect to the sampling of the different masks inside the dropout layers, which is undesirable after training. For example, two forward passes of the network can return two different outputs, and some draws (e.g., with a very large number of zeroes) can be suboptimal. Hence, we require some strategy to replace the forward pass with a deterministic operation.

Suppose we have m dropout layers. Let us denote by $\mathbf{M}_i$ the mask in the i-th dropout layer, by $p(\mathbf{M}_1, \dots, \mathbf{M}_m) = \prod_{i=1}^{m} p(\mathbf{M}_i)$ the probability distribution over the union of the masks, and by $f(\mathbf{x}; \mathbf{M})$ the deterministic output once a given set of masks $\mathbf{M} \sim p(\mathbf{M})$ are chosen. One choice is to replace the dropout effect with its expected value during inference:

$$f(\mathbf{x}) = \begin{cases} f(\mathbf{x}; \mathbf{M}), \;\; \mathbf{M} \sim p(\mathbf{M}) & \texttt{[training]} \\ \mathbb{E}_{p(\mathbf{M})}[f(\mathbf{x}; \mathbf{M})] & \texttt{[inference]} \end{cases}$$

We can approximate the expected value via Monte Carlo sampling (Appendix A) by repeatedly sampling masks values and averaging:

$$\mathbf{E}_{p(\mathbf{M})}[f(\mathbf{x}; \mathbf{M})] \approx \frac{1}{k} \sum_{i=1}^{k} f(\mathbf{x}; \mathbf{Z}_i), \;\; \mathbf{Z}_i \sim p(\mathbf{M})$$

which is simply the average of k forward passes. This is called **Monte Carlo dropout** [GG16]. The output is still stochastic, but with a proper choice of k, the variance can be contained. In addition, the outputs of the different forward passes can provide a measure of uncertainty over the prediction.

However, performing multiple forward passes can be expensive. A simpler (and more common) option is to replace the random variables *layer-by-layer*, which is a reasonable approximation. The expected value in this case can be written in closed form:

$$\mathbb{E}_{p(\mathbf{M})}[\text{Dropout}(\mathbf{X})] = p\mathbf{X}$$

which is the input rescaled by a constant factor p (the probability of sampling a 1 in the mask). This leads to an even simpler formulation, **inverted dropout**, where this correction is accounted for during training:

$$\text{Dropout}(\mathbf{X}) = \begin{cases} \dfrac{\mathbf{M} \odot \mathbf{X}}{(1-p)} & \texttt{[training]} \\ \mathbf{X} & \texttt{[inference]} \end{cases}$$

```
x = torch.randn((16, 2))

# Training with dropout
model.train()
y = model(x)

# Inference with dropout
model.eval()
y = model(x)

# Monte Carlo dropout for inference
k = 10
model.train()
y = model(x[:, None, :].repeat(1, k, 1)).mean(1)
```

Box C.9.3: *Applying the model from Box C.9.2 on a mini-batch of* 16 *examples. For layers like dropout, a framework requires a way to differentiate between a forward pass executed during training or during inference. In PyTorch, this is done by calling the* `train` *and* `eval` *methods of a model, which set an internal* `train` *flag on all layers. We also show a vectorized implementation of Monte Carlo dropout.*

In this case, the dropout layer has no effect when applied during inference and can be directly removed. This is the preferred implementation in most frameworks. See Box C.9.3 for some comparisons.

As we mentioned, dropout (possibly with a low drop probability, such as $p = 0.1$ or $p = 0.2$) is common for fully-connected layers. It is also common for attention maps (introduced in the next chapter). It is less common for convolutional layers, where dropping single elements of the input tensor results in sparsity patterns which are too unstructured. Variants of dropout have been devised which take into consideration the specific structure of images: for example, **spatial dropout** [TGJ+15] drops entire channels of the tensor, while **cutout** [DT17] drops spatial patches of a single channel.

Other alternatives are also possible. For example, **DropConnect** [WZZ+13] drops single weights of a fully-connected layer:

$$\text{DropConnect}(\mathbf{x}) = (\mathbf{M} \odot \mathbf{W})\mathbf{x} + \mathbf{b}$$

DropConnect in inference can also be approximated efficiently with moment matching [WZZ+13]. However, these are less common in practice, and the techniques described next are preferred.

9.3.2 Batch (and layer) normalization

When dealing with tabular data, a common pre-processing operation that we have not discussed yet is **normalization**, i.e., ensuring that all features (all columns of the input matrix) share similar ranges and statistics. For example, we can pre-process the data to squash all columns in a $[0,1]$ range (**min-max normalization**) or to ensure a zero mean and unitary variance for each column (called either **standard scaling** or **normal scaling** or **z-score scaling**).

Batch normalization (BN, [IS15]) replicates these ideas, but for the intermediate embeddings of the model. This is non trivial, since the statistics of a unit (e.g., its mean) will change from iteration to iteration after each gradient descent update. Hence, to compute the mean of a unit we should perform a forward pass on the entire training dataset at every iteration, which is unfeasible. As the name implies, BN's core idea is to approximate these statistics using only the data *in the mini-batch itself*.

Consider again the output of any fully-connected layer $\mathbf{X} \sim (n, c)$, where n is the mini-batch size. We will see shortly how to extend the ideas to images and other types of data. In BN, we normalize each feature (each column of $\mathbf{X}$) to have zero mean and unitary variance, based on the mini-batch alone. To this end, we start by computing the empirical column-wise mean $\mu \sim (c)$ and variances $\sigma^2 \sim (c)$:

$$\text{Mean of column } j\text{:} \qquad \mu_j = \frac{1}{n}\sum_i X_{ij} \tag{E.9.8}$$

$$\text{Variance of column } j\text{:} \qquad \sigma_j^2 = \frac{1}{n}\sum_i (X_{ij} - \mu_j)^2 \tag{E.9.9}$$

We then proceed to normalize the columns:

Set the mean of each column to 0

$$\mathbf{X}' = \frac{\mathbf{X} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

Set the variance of each column to 1

where we consider the standard broadcasting rules (μ and σ^2 are broadcasted over the first dimension), and $\varepsilon > 0$ is a small positive term added to avoid division by zero. Differently from normalization for tabular data, where this operation is applied once to

the entire dataset before training, in BN this operation must be recomputed for every mini-batch during each forward pass.

The choice of zero mean and unitary variance is just a convention, not necessarily the best one. To generalize it, we can let the optimization algorithm select the best choice, for a small overhead in term of parameters. Consider two trainable parameters $\alpha \sim (c)$ and $\beta \sim (c)$ (which we can initialize as 1 and 0 respectively), we perform:

$$\mathbf{X}'' = \alpha \mathbf{X}' + \beta$$

with similar broadcasting rules as above. The resulting matrix will have mean β_i and variance α_i for the i-th column. The BN layer is defined by the combination of these two operations.

Definition D.9.2 (Batch normalization layer) *Given an input matrix* $\mathbf{X} \sim (n, c)$*, a* ***batch normalization*** *(BN) layer applies the following normalization:*

$$\text{BN}(\mathbf{X}) = \alpha \left(\frac{\mathbf{X} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \right) + \beta$$

where μ *and* σ^2 *are computed according to* (E.9.8) *and* (E.9.9)*, while* $\alpha \sim (c)$ *and* $\beta \sim (c)$ *are trainable parameters. The layer has no hyper-parameters. During inference,* μ *and* σ^2 *are fixed as described next.*

The layer has only $2c$ trainable parameters, and it can be shown to greatly simplify training of complex models when inserted across each block. In particular, it is common to consider BN placed in-between the linear and non-linear components of the model:

$$\mathbf{H} = (\text{ReLU} \circ \text{BN} \circ \text{Linear})(\mathbf{X})$$

Centering the data before the ReLU can lead to better exploiting its negative (sparse) quadrant. In addition, this setup renders the bias in the linear layer redundant (as it conflates with the β parameter), allowing to remove it. Finally, the double linear operation can be easily optimized by standard compilers in most frameworks.

BN is so effective that is has led to a vast literature on understanding why [BGSW18]. The original derivation considered a problem known as **internal covariate shift**, i.e., the fact that, from the point of view of a single layer, the statistics of the inputs it receives will change during optimization due to the changes in weights of the preceding

layers. However, current literature agrees that the effects of BN is more evident in the optimization itself, both in terms of stability and the possibility of using higher learning rates, due to a combination of scaling and centering effects on the gradients [BGSW18].[7]

Extending BN beyond tabular data is simple. For example, consider a mini-batch of image embeddings $X \sim (n, h, w, c)$. We can apply BN on each channel by considering the first three dimensions together, i.e., we compute a channel-wise mean as:

$$\mu_z = \frac{1}{nhw} \sum_{i,j,k} X_{ijkz}$$

Mean of channel z (all pixels, all images)

Batch normalization during inference

BN introduces a dependency between the prediction over an input and the mini-batch it finds itself in, which is unwarranted during inference (stated differently, moving an image from one mini-batch to another will modify its prediction). However, we can exploit the fact that the model's parameters do not change after training, and we can freeze the mean and the variance to a preset value. There are two possibilities to this end:

1. After training, we perform another forward pass on the entire training set to compute the empirical mean and variance with respect to the dataset [WJ21].

2. More commonly, we can keep a rolling set of statistics that are updated after each forward pass of the model during training, and use these after training. Considering the mean only for simplicity, suppose we initialize another vector $\widehat{\mu} = \mathbf{0}$, corresponding to the "rolling mean of the mean". After computing μ as in (E.9.8), we update the rolling mean with an exponential moving average:

$$\widehat{\mu} \leftarrow \lambda \widehat{\mu} + (1 - \lambda)\mu$$

   where λ is set to a small value, e.g., $\lambda = 0.01$. Assuming training converges, the rolling mean will also converge to an approximation of the average given by

[7] See also https://iclr-blog-track.github.io/2022/03/25/unnormalized-resnets/ for a nice entry point into this literature (and the corresponding literature on developing **normalizer-free** models.

option (1). Hence, after training we can use BN by replacing μ with the (pre-computed) $\widehat{\mu}$, and similarly for the variance.[8]

Variants of batch normalization

Despite its good empirical performance, BN has a few important drawbacks. We have already mentioned the dependence on the mini-batch, which has other implications: for example, the variance of μ during training will grow large for small mini-batches, and training can be unfeasible for very small mini-batch sizes. In addition, training can be difficult in distributed contexts (where each GPU holds a separate part of the mini-batch). Finally, replacing μ with a different value after training creates an undesirable mismatch between training and inference.

Variants of BN have been proposed to address these issues. A common idea is to keep the overall structure of the layer, but to modify the axes along which the normalization is performed. For example, **layer normalization** [BKH16] computes the empirical mean and variance over *the rows* of the matrix, i.e., for each input independently:

$$\text{Mean of row } i: \qquad \mu_i = \frac{1}{c}\sum_j X_{ji} \tag{E.9.10}$$

$$\text{Variance of row } i: \qquad \sigma_i^2 = \frac{1}{c}\sum_j (X_{ji} - \mu_i)^2 \tag{E.9.11}$$

Consider Figure F.9.4, where we show a comparison between BN and LN for tabular and image-like data. In particular, we show in blue all the samples used to compute a single mean and variance. For layer normalization, we can compute the statistics on h, w, c simultaneously (variant A) or for each spatial location separately (variant B). The latter choice is common in transformer models, discussed in the next chapter. Other variants are also possible, e.g., **group normalization** restricts the operation to a subset of channels, with the case of a single channel known as **instance normalization**.[9]

In BN, the axes across which we compute the statistics in (E.9.8) and (E.9.9) are the same as the axes across which we apply the trainable parameters. In LN, the two

[8] $\widehat{\mu}$ is the first example of a layer's tensor which is part of the layer's state, is adapted during training, but is not needed for gradient descent. In PyTorch, these are referred to as *buffers*.

[9] See https://iclr-blog-track.github.io/2022/03/25/unnormalized-resnets/ for a nicer variant of Figure F.9.4.

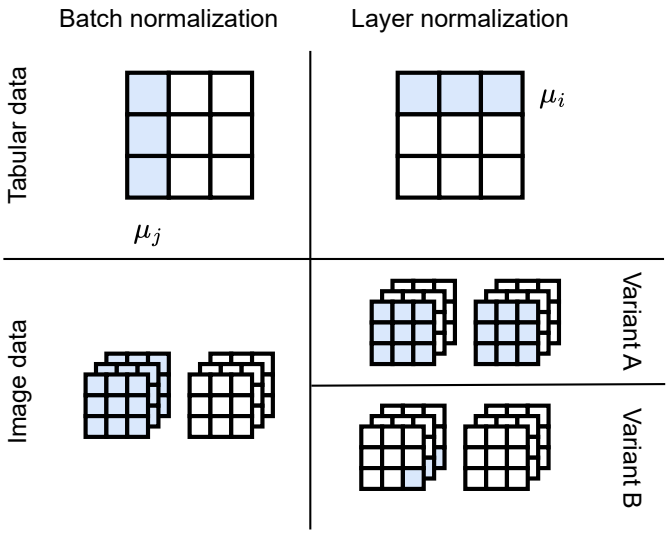


Figure F.9.4: *Comparison between BN and LN for tabular and image data. Blue regions show the sets over which we compute means and variances. For LN we have two variants, discussed better in the main text.*

are decoupled. For example, consider a PyTorch LN layer applied on mini-batches of dimension $(b, 3, 32, 32)$:

```
ln = nn.LayerNorm(normalized_shape=[3, 32, 32])
```

This corresponds to variant A in Figure F.9.4. In this case, α and β will have the same shape as the axes over which we are computing the normalization, i.e., $\alpha, \beta \sim (3, 32, 32)$, for a total of $2 \times 3 \times 32 \times 32 = 6144$ trainable parameters. The specific implementation of LN and BN must be checked for each framework and model.

We close by mentioning another common variant of layer normalization, called **root mean square** normalization (RMSNorm) [ZS19]. It simplifies LN by removing the mean centering and shifting, which for a single input vector $\mathbf{x} \sim (c)$ can be written as:

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{c}\sum_i x_i^2}} \odot \alpha \tag{E.9.12}$$

When $\beta = 0$ and the data is already zero-centered, LN and RMSNorm are identical.

Figure F.9.5: *Bigger models do not always improve monotonically in training error, despite representing larger classes of functions. Reproduced from [HZRS16].*

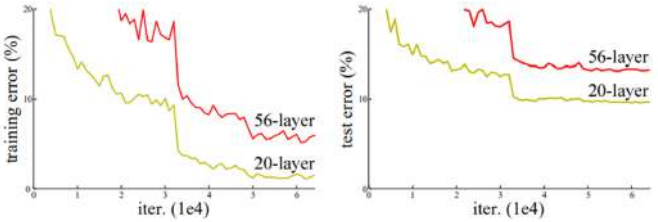


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

9.4 Residual connections

9.4.1 Residual connections and residual networks

The combination of all techniques seen in the previous section is enough to increase significantly the number of layers in our models, but only up to a certain upper bound. Consider three generic sequence of layers f_1, f_2, and f_3, and two models where one is a subset of the other:

$$g_1(x) = (f_3 \circ f_1)(x)$$
$$g_2(x) = (f_3 \circ f_2 \circ f_1)(x)$$

Intuitively, by the universal approximation theorem it should always be possible for the intermediate part, f_2, to approximate the identity function $f_2(x) \approx x$, in which case $g_2(x) \approx g_1(x)$. Hence, there is always a setting of the parameters in which the second (deeper) model should perform at least as well as the first (shallower) one. However, this was not observed in practice, as shown in Figure F.9.5.

We can solve this by biasing the blocks in the network towards the identity function. This can be done easily by rewriting a block $f(x)$ with what is called a **residual (skip) connection** [HZRS16]:

$$r(x) = f(x) + x$$

Hence, we use the block to model deviations from the identity, $f(x) = r(x) - x$, instead of modeling deviations from the zero function. This small trick alone helps in training models up to hundreds of layers. We call $f(x)$ the **residual path**, $r(x)$ a **residual block**, and a convolutional model composed of residual blocks a **residual network** (abbreviated to ResNet).

Residual connections work well with batch normalization on the residual path, which can be shown to further bias the model towards the identity at the beginning of training [DS20]. However, residual connections can be added only if the input and output dimensionality of $f(x)$ are identical. Otherwise, some rescaling can be added to the residual connection. For example, if x is an image and $f(x)$ modifies the number of channels, we can add a 1×1 convolution:

$$r(x) = f(x) + \text{Conv2D}_{1\times 1}(x)$$

The benefit of a residual block can be understood also in terms of its backward pass. Consider the VJP of the residual block:

$$\text{vjp}_r(\mathbf{v}) = \text{vjp}_f(\mathbf{v}) + \mathbf{v}^\top \mathbf{I} = \text{vjp}_f(\mathbf{v}) + \mathbf{v}^\top$$

VJP of f

VJP of the skip connection

Hence, the forward pass lets the input x pass through unmodified on the skip connection, while the backward pass adds the unmodified back-propagated gradient $\mathbf{v}$ to the original VJP, which can help mitigating gradient instabilities.

On the design of the residual block

How to design the block $f(x)$? Consider the batch-normalized block introduced earlier:

$$h = \underbrace{(\text{ReLU} \circ \text{BN} \circ \text{Conv2D})(x)}_{=f(x)} + x$$

Because the output of ReLU is always positive, we have that $h \geq x$ (element-wise). Hence, a stack of residual blocks of this form can only increase the values of the input tensor, or set it to zero. For this reason, the original design proposed in [HZRS16] considered a stack of blocks of this form by removing the last activation function. As an example, for two blocks we obtain the following design:

$$h = (\text{BN} \circ \text{Conv2D} \circ \text{ReLU} \circ \text{BN} \circ \text{Conv2D})(x) + x$$

A series of blocks of this form can be preceded by a small component with non-residual connections to reduce the image dimensionality, sometimes called the **stem**. The specific choice of hyper-parameters for this block has varied significantly over the years.

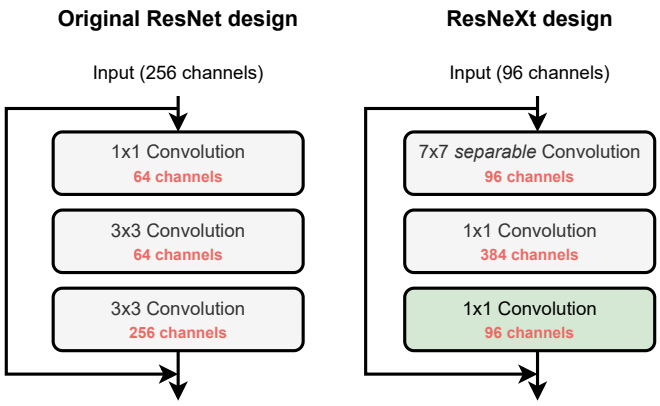


Figure F.9.6: *The original ResNet block [HZRS16], and the more recent ResNeXt [LMW+22] block. As can be seen, the design has shifted from an early channel reduction to a later compression (**bottleneck**). Additional details (not shown) are the switch from BN to LN and the use of GELU activation functions. Adapted from [LMW+22].*

The original ResNet block proposed a compression in the number of channels for the first operation, followed by a standard 3×3 convolution and a final upscaling in the number of channels. Recently, instead, **bottleneck** layers like the v block [LMW+22] (on the right in Figure F.9.6) have become popular. To increase the receptive field of the convolution, the initial layer is replaced by a depthwise convolution. To exploit the reduced number of parameters, the number of channels is *increased* by a given factor (e.g., 3×, 4×), before being reduced by the last 1×1 convolution.

9.4.2 Additional perspectives on residual connections

We close the chapter by discussing two interesting perspectives on the use of residual connections, which have both been explored in-depth in current research. First, consider a network composed of two residual blocks:

$$h_1 = f_1(x) + x \tag{E.9.13}$$
$$h_2 = f_2(h_1) + h_1 \tag{E.9.14}$$

If we unroll the computation:

$$h_2 = f_2(f_1(x) + x) + f_1(x) + x$$

This corresponds to the sum of several *paths* in the network, where the input is either left unmodified, it goes through only a single transformation (f_1 or f_2), or through their combination.

It should be clear that the number of such paths grows exponentially with the number

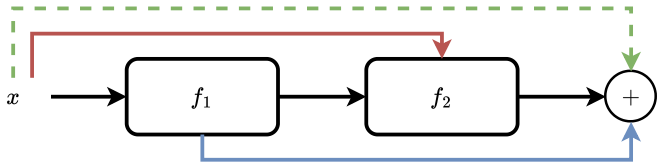


Figure F.9.7: *Residual paths: the black, red, and blue paths are implemented explicitly; the green path is only implicit.*

of residual blocks. Hence, deep residual models can be seen as a combination (an ensemble) of a very large number of smaller models, implemented through weight-sharing. This view can be tested to show, for example, that ResNets tend to be robust to small deletions or modifications of their elements [VWB16]. This is shown visually in Figure F.9.7.

Second, consider the following differential equation, expressed in terms of a continuous parameter t representing time:

$$\partial_t x_t = f(x, t)$$

We are using a neural network with arguments x and t (a scalar) to parameterize the time derivative of some function. This is called an **ordinary differential equation** (ODE). A common problem with ODEs is integrating from a known starting value x_0 up to some specified time instant T:

$$x_T = x_0 + \int_{t=0}^{T} f(x, t) dt$$

Euler's method[10] for computing x_T works by selecting a small step size h and computing iteratively a first-order discretization:

$$x_t = x_{t-1} + hf(x_{t-1}, t)$$

Merging h into f, this corresponds to a restricted form of residual model, where all residual blocks share the same weights, each layer corresponds to a discretized time-instant, and x_T is the output of the network. Under this point of view, we can directly work with the original continuous-time equation, and compute the output by integrating it with modern ODE solvers. This is called a **neural ODE** [CRBD18]. Continuous-time variants of back-propagation can be derived that take the form of another ODE problem. We will see in the next volume an interesting connection between neural ODEs and a class of generative models known as **normalizing flows** [PNR+21].

[10] https://en.wikipedia.org/wiki/Euler_method

From theory to practice

All the layers we discussed in this chapter (batch normalization, dropout, ...) are already implemented in PyTorch, Equinox, and practically every other framework. For what concerns the rest of the techniques we described, it depends on the framework: for example, weight decay in implemented natively in all PyTorch's optimizers, data augmentation can be found as transformations inside `torchvision` (and other corresponding libraries), while early stopping must be implemented manually.[11]

1. Before proceeding to the next chapter, I suggest you try implementing either dropout or batch normalization as a layer using only standard linear algebra routines, comparing the results with the built-in layers.

2. In Chapter 7 you should have implemented a simple convolutional model for image classification. Try progressively increasing its size, adding normalization, dropout, or residual connections as needed.

3. Take a standard architecture, such as a ResNet [HZRS16], or a ResNeXt [LMW+22]. Try implementing the entire model by following the suggestions from the original papers. Training on ImageNet-like datasets can be challenging on consumer's GPU – if you do not have access to good hardware or cloud GPU hours, you can keep focusing on simpler datasets, such as CIFAR-10.

4. At this point, you may have realized that training from scratch very large models (e.g., ResNet-50) on smaller datasets is practically impossible. One solution is to initialize the weights of the model from an online repository using, e.g., the weights of a model trained on ImageNet, and **fine-tuning** the model by modifying the last layer, corresponding to the classification head. By this point of the book, this should come as relatively easy – I suggest using one of the many pre-trained models available on torchvision or on the Hugging Face Hub.[12] We will cover fine-tuning more in-depth in the next volume.

[11] In PyTorch, a common alternative is to use an external library such as PyTorch Lightning to handle the training process. Modifications to the training procedure, such as early stopping, are pre-implemented in the form of *callback* functions.

[12] For an example tutorial: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html.

Part III
Down the rabbit-hole

“It would be so nice if something made sense for a change.”

—Alice in Wonderland, 1951 movie

10 | Transformer models

About this chapter

Convolutional models are strong baselines, especially for images and sequences where local relations prevail, but they are limited in handling very long sequences or non-local dependencies between elements of a sequence. In this chapter we introduce another class of models, called transformers, which are designed to overcome such challenges.

10.1 Long convolutions and non-local models

After the key developments in the period 2012-2016, discussed in the previous chapter, the next important breakthrough in the design of differentiable models came in 2016-2017 with the popularization of the **transformer** [VSP+17], an architecture designed to handle efficiently long-range dependencies in natural language processing. Due to its strong scaling laws, the architecture was then extended to other types of data, from images to time-series and graphs, and it is today a state-of-the-art model in many fields due to its very good scaling laws when trained on large amounts of data [KMH+20, BPA+24].

As we will see, an interesting aspect of the transformer is a decoupling between the data type (through the use of appropriate tokenizers) and the architecture, which for the most part remains data-agnostic. This opens up several interesting directions, such as simple multimodal architectures and transfer learning strategies. We begin by motivating the core component of the transformer, called the **multi-head attention** (MHA) layer. We will defer a discussion on the original transformer model from [VSP+17] to the next chapter.

A bit of history

Historically, this chapter is out of order: in 2015, the most common alternative to CNNs for text were **recurrent neural networks** (RNNs). As an isolated component, MHA was introduced for RNNs [BCB15], before being used as the core component in the transformer model. We cover RNNs and their modern incarnation, linearized RNNs, in Chapter 13. Recently, RNNs have become an attractive competitor to transformers for language modeling.

10.1.1 Handling long-range and sparse dependencies

Consider these two sentences:

> "*The cat is on the table*"

and a longer one:

> "*The cat, who belongs to my mother, is on the table*".

In order to be processed by a differentiable model, the sentences must be tokenized and the tokens embedded as vectors (Chapter 8). From a semantic point of view, the tokens belonging to the red word (cat) and to the green word (table) share a similar dependency in both sentences. However, their relative offset varies in the two cases, and their distance can become arbitrarily large. Hence, dependencies in text can be both **long-range** and **input-dependent**.

Denote by $\mathbf{X} \sim (n, e)$ a sentence of n tokens embedded in e-dimensional vectors and denote by $\mathbf{x}_i$ the ith token. We can rewrite a 1D convolution with kernel size k on token i as follows:

$$\mathbf{h}_i = \sum_{j=1}^{2k+1} \mathbf{W}_j \mathbf{x}_{i+k+1-j} \tag{E.10.1}$$

Each token inside the receptive field is processed with a fixed weight matrix $\mathbf{W}_i$ that only depends on the specific offset i. Modeling long-range dependencies inside the layer requires us to increase the receptive field of the layer, increasing the number of parameters linearly in the receptive field.

One possibility to solve this is the following: instead of explicitly learning the parameter matrices $\mathbf{W}_1, \mathbf{W}_2, \ldots$, we can define them *implicitly* by defining a separate neural block $g(i) : \mathbb{R} \to \mathbb{R}^{e \times e}$ that outputs all weight matrices based on the relative offset i. Hence,

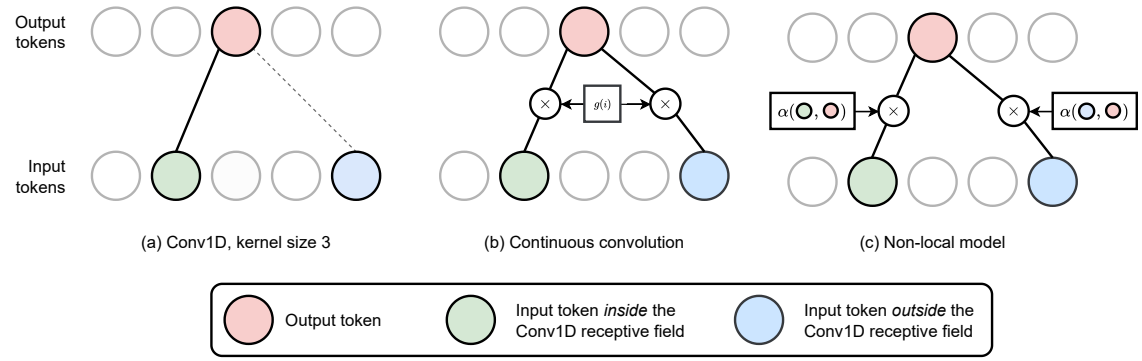


Figure F.10.1: *Comparison between different types of convolution for a 1D sequence. We show how one output token (in red) interacts with two tokens, one inside the receptive field of the convolution (in green), and one outside (in blue). (a) In a standard convolution, the blue token is ignored because it is outside of the receptive field of the filter. (b) For a continuous convolution, both tokens are considered, and the resulting weight matrices are given by $g(-1)$ and $g(2)$ respectively. (c) In the non-local case, the weight matrices depend on a pairwise comparison between the tokens themselves.*

we rewrite (E.10.1) as:

$$\mathbf{h}_i = \sum_{j=1}^{n} g(i-j)\mathbf{x}_j$$

The sum is now on *all* tokens

This is called a **long convolution**, as the convolution spans the entire input matrix $\mathbf{X}$. It is also called a **continuous convolution** [RKG+22], because we can use $g(\cdot)$ to parameterize intermediate positions or variable resolutions [RKG+22]. The number of parameters in this case only depends on the parameters of g, while it does not depend on n, the length of the sequence. Defining g is non-trivial because it needs to output an entire weight matrix. We can recover a standard convolution easily:

$$g(i,j) = \begin{cases} \mathbf{W}_{i-j} & \text{if } |i-j| \leq k \\ 0 & \text{otherwise} \end{cases} \tag{E.10.2}$$

This partially solves the problem of long-range dependencies, but it does not solve the problem of dependencies which are conditional on the input, since the weight given

to a token depends only on the relative offset with respect to the index i. However, this formulation provides a simple way to tackle this problem by letting the trained function g depend on the *content* of the tokens instead of their positions:

$$\mathbf{h}_i = \sum_{j=1}^{n} g(\mathbf{x}_i, \mathbf{x}_j)\mathbf{x}_j \tag{E.10.3}$$

In the context of computer vision, these models are also called **non-local** networks [WGGH18]. We provide a comparison of standard convolutions, continuous convolutions, and non-local convolutions in Figure F.10.1.

10.1.2 The attention layer

The MHA layer is a simplification of (E.10.3). First, working with functions having matrix outputs is difficult, so we restrict the layer to work with scalar weights. In particular, a simple measure of similarity between tokens is their **dot-product**:

$$g(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^\top \mathbf{x}_j$$

As we will see, this results in an easily parallelizable algorithm for the entire sequence. For the following we consider a normalized version of the dot-product:

$$g(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{\sqrt{e}} \mathbf{x}_i^\top \mathbf{x}_j$$

This can be motivated as follows: if we assume $\mathbf{x}_i \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$, the variance of each element of $\mathbf{x}_i^\top \mathbf{x}_j$ is σ^4, hence the elements can easily grow very large in magnitude. The scaling factor ensures that the variance of the dot product remains bounded at σ^2.

Because we are summing over a potentially variable number of tokens n, it is also helpful to include a normalization operation, such as a softmax:[1]

$$\mathbf{h}_i = \sum_{j=1}^{n} \text{softmax}_j(g(\mathbf{x}_i, \mathbf{x}_j))\mathbf{x}_j \tag{E.10.4}$$

In this context, we refer to $g(\cdot, \cdot)$ as the **attention scoring function**, and to the output

[1] The notation softmax_j in (E.10.4) means we are applying the softmax normalization to the set $\{g(\mathbf{x}_i, \mathbf{x}_j)\}_{j=1}^n$, independently for each i. This is easier to see in the vectorized case, described below.

of the softmax as the **attention scores**. Because of the normalization properties of the softmax, we can imagine that each token i has a certain amount of "attention" it can allocate across the other tokens: by increasing the budget on a token, the attention over the other tokens will necessarily decrease due to the denominator in the softmax.

If we use a "dot-product attention", our g does not have trainable parameters. The idea of an attention layer is to recover them by adding trainable projections to the input before computing the previous equation. To this end, we define three trainable matrices $\mathbf{W}_k \sim (k, e)$, $\mathbf{W}_v \sim (v, e)$, $\mathbf{W}_q \sim (k, e)$, where k and v are hyper-parameters. Each token is projected using these three matrices, obtaining $3n$ tokens in total:

$$\text{Key tokens:} \quad \mathbf{k}_i = \mathbf{W}_k \mathbf{x}_i \tag{E.10.5}$$

$$\text{Value tokens:} \quad \mathbf{v}_i = \mathbf{W}_v \mathbf{x}_i \tag{E.10.6}$$

$$\text{Query tokens:} \quad \mathbf{q}_i = \mathbf{W}_q \mathbf{x}_i \tag{E.10.7}$$

These processed tokens are called the **keys**, the **values**, and the **queries** (you can ignore the choice of terminology for now; we will return on this point at the end of the section). The **self-attention** (SA) layer is obtained by combining the three projections (E.10.5)-(E.10.6)-(E.10.7) with (E.10.4):

$$\mathbf{h}_i = \sum_{j=1}^{n} \text{softmax}_j(g(\mathbf{q}_i, \mathbf{k}_j))\mathbf{v}_j$$

Hence, we compute the updated representation of token i by comparing its query to all possible keys, and we use the normalized weights to combine the corresponding value tokens. Note that the dimensionality of keys and queries must be identical, while the dimensionality of the values can be different.

If we use the dot product, we can rewrite the operation of the SA layer compactly for all tokens. To this end, we define three matrices with the stack of all possible keys, queries, and values:

$$\mathbf{K} = \mathbf{X}\mathbf{W}_k \tag{E.10.8}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_v \tag{E.10.9}$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_q \tag{E.10.10}$$

The three matrices have shape (n, k), (n, v), and (n, k) respectively. As a side note, we can also implement them as a single matrix multiplication whose output is chunked in

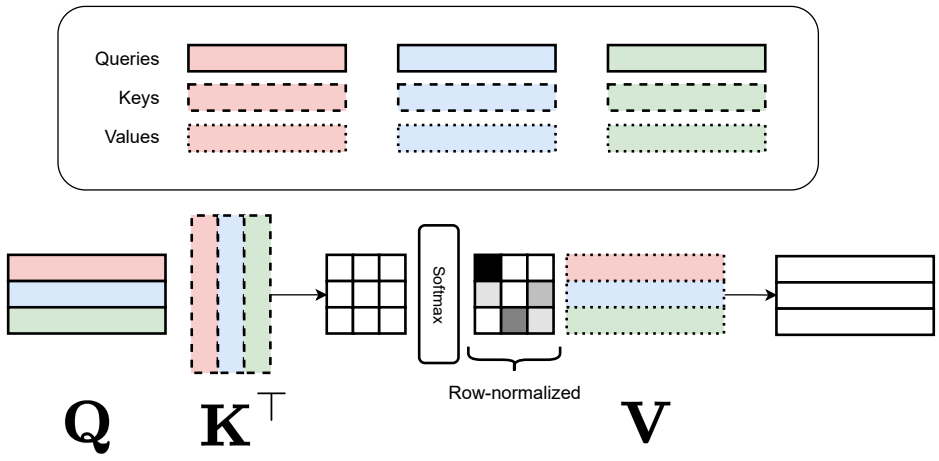


Figure F.10.2: *Visualization of the main operations of the SA layer (excluding projections).*

three parts:

$$\begin{bmatrix} \mathbf{K} \\ \mathbf{V} \\ \mathbf{Q} \end{bmatrix} = \mathbf{X} \begin{bmatrix} \mathbf{W}_k \\ \mathbf{W}_v \\ \mathbf{W}_q \end{bmatrix}$$

The SA layer is then written as:

$$\text{SA}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{k}}\right)\mathbf{V}$$

where we assume the softmax is applied row-wise. We can also make the projections explicit, as follows.

Definition D.10.1 (Self-attention layer) *The **self-attention** (SA) layer is defined for an input $\mathbf{X} \sim (n, e)$ as:*

$$\text{SA}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{X}\mathbf{W}_q\mathbf{W}_k^\top\mathbf{X}^\top}{\sqrt{k}}\right)\mathbf{X}\mathbf{W}_v \tag{E.10.11}$$

The trainable parameters are $\mathbf{W}_q \sim (k, e)$, $\mathbf{W}_k \sim (k, e)$ and $\mathbf{W}_v \sim (v, e)$, where k and v are hyper-parameters. Hence, there are $k(2e+v)$ trainable parameters, independent of n.

We show the operation of the layer visually in Figure F.10.2.

10.1.3 Multi-head attention

The previous layer is also called a **single-head** attention operation. It allows to model pairwise dependencies across tokens with high flexibility. However, in some cases we may have multiple sets of dependencies to consider: taking again the example of "*the cat, which belongs to my mother, is on the table*", the dependencies between "*cat*" and "*table*" are different with respect to the dependencies between "*cat*" and "*mother*", and we may want the layer to be able to model them separately.[2]

A multi-head layer achieves this by running multiple attention operations in parallel, each with its own set of trainable parameters, before aggregating the results with some pooling operation. To this end, we define a new hyper-parameter h, that we call the number of **heads** of the layer. We instantiate h separate projections for the tokens, for a total of $3hn$ tokens ($3n$ for each "head"):

$$\mathbf{K}_e = \mathbf{X}\mathbf{W}_{k,e} \tag{E.10.12}$$
$$\mathbf{V}_e = \mathbf{X}\mathbf{W}_{v,e} \tag{E.10.13}$$
$$\mathbf{Q}_e = \mathbf{X}\mathbf{W}_{q,e} \tag{E.10.14}$$

$\mathbf{W}_{k,e}$ represents the key projection for the e-th head, and similarly for the other quantities. The **multi-head attention** (MHA) layer performs h separate SA operations, stacks the resulting output embeddings, and projects them a final time to the desired dimensionality:

$$\text{MHA}(\mathbf{X}) = \Big[\underbrace{\text{softmax}\Big(\frac{\mathbf{Q}_1\mathbf{K}_1^\top}{\sqrt{k}}\Big)\mathbf{V}_1}_{\text{Individual SA layer}} \parallel \ldots \parallel \text{softmax}\Big(\frac{\mathbf{Q}_h\mathbf{K}_h^\top}{\sqrt{k}}\Big)\mathbf{V}_h \Big] \underbrace{\mathbf{W}_o}_{\text{Output projection}} \tag{E.10.15}$$

Each SA operation returns a matrix of shape (n, v). These h matrices are concatenated across the second dimension to obtain a matrix (n, hv), which is then projected with a matrix $\mathbf{W}_o \sim (o, hv)$, where o is an additional hyper-parameter allowing flexibility in the choice of the output dimensionality.

[2] And everything depends on the cat, of course.

```
d = dict()
d["Simone"] = 2
d["Simone"]        # Returns 2
d["Smone "]        # Returns an error
```

Box C.10.1: *A dictionary in Python: a value is returned only if a perfect key-query match is found. Otherwise, we get an error.*

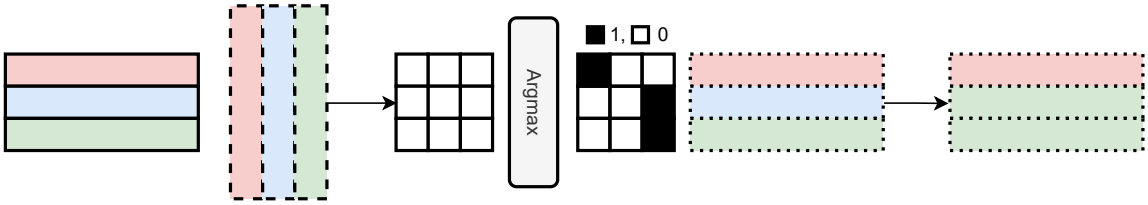


Figure F.10.3: *SA with a "hard" attention is equivalent to a vector-valued dictionary.*

An explanation of the terminology

In order to understand why the three tokens are called queries, keys, and values, we consider the analogy of a SA layer with a standard Python dictionary, which is shown in Box C.10.1.

Formally, a dictionary is a set of pairs of the form (key, value), where the key acts as an univocal ID to retrieve the corresponding value. For example, in the third and fourth line of Box C.10.1 we query the dictionary with two different strings ("*Simone*" and "*Smone* "): the dictionary compares the query string to all keys which are stored inside, returning the corresponding value if a perfect match is found, an error otherwise.

Given a measure of similarity over pair of keys, we can consider a variant of a standard dictionary which always returns the value corresponding to the closest key found in the dictionary. If the keys, queries, and values are vectors, this dictionary variant is equivalent to our SA layer if we replace the softmax operation with an argmax over the tokens, as shown in Figure F.10.3.

This "hard" variant of attention is difficult to implement because the gradients of the argmax operation are zero almost everywhere (we will cover discrete sampling and approximating the argmax operation with a discrete relaxation in the next volume). Hence, we can interpret the SA layer as a soft approximation in which each token is updated with a weighted combination of all values based on the corresponding key/query similarities.

Heads and circuits

We will see shortly that the MHA layer is always combined with a residual connection (Section 9.4). In this case we can write its output for the i-th token as:

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \underbrace{\sum_e}_{\text{Sum over heads}} \underbrace{\sum_j}_{\text{Sum over tokens}} \alpha_e(\mathbf{x}_i, \mathbf{x}_j)\mathbf{W}_e^\top \mathbf{x}_j \tag{E.10.16}$$

where $\alpha_e(\mathbf{x}_i, \mathbf{x}_j)$ is the attention score between tokens i and j in head e, and $\mathbf{W}_e$ combines the value projection of the e-th head with the e-th block of the output projection in (E.10.15). The token embeddings are sometimes called the **residual stream** of the model.[a] Hence, the heads can be understood as "reading" from the residual stream (via the projection by $\mathbf{W}_e$ and the selection via the attention scores), and linearly "writing" back on the streams.

[a] This has been popularized in the context of **mechanistic interpretability**, which tries to retro-engineer the layers' behaviour to find interpretable components called *circuits*: https://transformer-circuits.pub. The linearity of the stream is fundamental for the analisys.

10.2 Positional embeddings

With the MHA layer in hand, we consider the design of the complete transformer model, which requires another component, positional embeddings.

10.2.1 Permutation equivariance of the MHA layer

It is interesting to consider what happens to the output of a MHA layer when the order of the tokens is re-arranged (*permuted*). To formalize this, we introduce the concept of **permutation matrices**.

Definition D.10.2 (Permutation matrix) *A **permutation matrix** of size n is a square binary matrix $\mathbf{P} \sim Binary(n, n)$ such that only a single 1 is present on each row or column:*

$$\mathbf{1}^\top \mathbf{P} = \mathbf{1}, \;\; \mathbf{P}\mathbf{1} = \mathbf{1}$$

If we remove the requirement for the matrix to have binary entries and we only constrain the entries to be non-negative, we obtain the set of ***doubly stochastic*** *matrices (matrices whose rows and columns sum to one).*

The effect of applying a permutation matrix is to rearrange the corresponding rows / columns of a matrix. For example, consider the following permutation:

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Looking at the rows, we see that the second and third elements are swapped by its application:

$$\mathbf{P} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_3 \\ \mathbf{x}_2 \end{bmatrix}$$

Interestingly, the only effect of applying a permutation matrix to the inputs of a MHA layer is to rearrange the outputs of the layer in an equivalent way:

$$\text{MHA}(\mathbf{PX}) = \mathbf{P} \cdot \text{MHA}(\mathbf{X})$$

This is immediate to prove. We focus on the single headed variant as the multi-headed variant proceeds similarly. First, the softmax renormalizes the elements over the columns of a matrix, so it is trivially permutation equivariant across both rows and columns:

$$\text{softmax}(\mathbf{PXP}^\top) = \mathbf{P}\,[\text{softmax}(\mathbf{X})]\,\mathbf{P}^\top$$

From this we can immediately deduce the positional equivariance of SA:

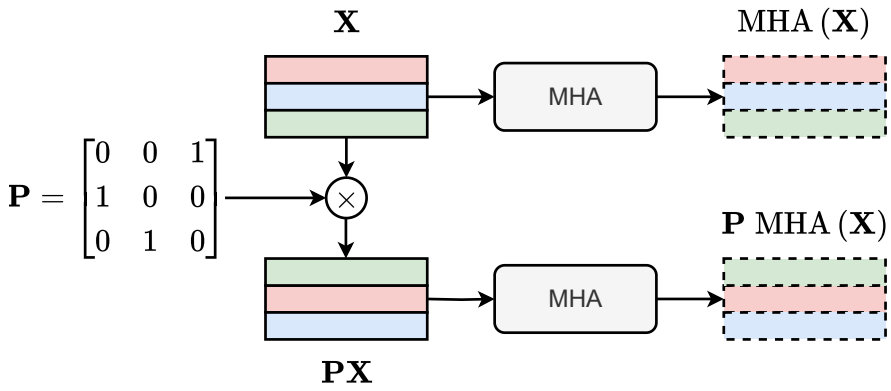


Figure F.10.4: *The output of a MHA layer after permuting the ordering of the tokens is trivially the permutation of the original outputs.*

$$\text{SA}(\mathbf{PX}) = \text{softmax}\left(\mathbf{P}\frac{\mathbf{X}\mathbf{W}_q\mathbf{W}_k^\top\mathbf{X}^\top}{\sqrt{k}}\mathbf{P}^\top\right)\mathbf{P}\mathbf{X}\mathbf{W}_v \tag{E.10.17}$$

$$= \mathbf{P}\text{softmax}\left(\frac{\mathbf{X}\mathbf{W}_q\mathbf{W}_k^\top\mathbf{X}^\top}{\sqrt{k}}\right)\mathbf{X}\mathbf{W}_v = \mathbf{P}\cdot\text{SA}(\mathbf{X}) \tag{E.10.18}$$

where we make use of the fact that $\mathbf{P}^\top\mathbf{P} = \mathbf{I}$ for any permutation matrix. This can also be seen by reasoning on the SA layer for each token: the output is given by a sum of elements, each weighted by a pairwise comparison. Hence, for a given token the operation is **permutation invariant**. Instead, for the entire input matrix, the operation is **permutation equivariant**.

Translational equivariance was a desirable property for a convolutional layer, but permutation equivariance is *undesirable* (at least here), because it discards the valuable ordering of the input sequence. As an example, the only effect of processing a text whose tokens have been reversed would be to reverse the output of the layer, despite the fact that the resulting reversed input is probably invalid. Formally, the SA and MHA layers are set functions, not sequence functions.

Instead of modifying the layer or adding layers that are not permutation equivariant, the transformer operates by introducing the new concept of **positional embeddings**, which are auxiliary tokens that depend only on the position of a token in a sequence

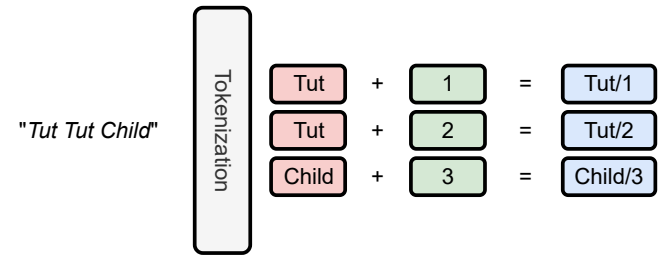


Figure F.10.5: *Positional embeddings (green) added to the tokens' embeddings (red). The same token in different positions has different outputs (blue).*

(**absolute positional embeddings**) or the offset of two tokens (**relative positional embeddings**). We describe the two in turn.

10.2.2 Absolute positional embeddings

Each token in the input matrix $\mathbf{X} \sim (n, e)$ represents the *content* of the specific piece of text (e.g., a subword). Suppose we fix the maximum length of any sequence to m tokens. To overcome positional equivariance, we introduce an additional set of **positional embeddings** $\mathbf{S} \sim (m, e)$, where the vector $\mathbf{S}_i$ uniquely encodes the concept of "being in position i". Hence, the sum of the input matrix with the first rows of $\mathbf{S}$:

$$\mathbf{X}' = \mathbf{X} + \mathbf{S}_{1:n}$$

is such that $[\mathbf{X}']_i$ represents "token $\mathbf{X}_i$ in position i". Because it does not make sense to permute the positional embeddings (as they only depend on the position), the resulting layer is not permutation equivariant anymore:

$$\text{MHA}(\mathbf{PX} + \mathbf{S}) \neq \mathbf{P} \cdot \text{MHA}(\mathbf{X} + \mathbf{S})$$

See Figure F.10.5 for a visualization of this idea.

How should we build positional embeddings? The easiest strategy is to consider $\mathbf{S}$ as part of the model's parameters, and train it together with the rest of the trainable parameters, similarly to the token embeddings. This strategy works well when the number of tokens is relatively stable; we will see an example in the next chapter in the context of computer vision.

Alternatively, we can define some deterministic function from the set of tokens' positions to a given vector that uniquely identifies the position. Some strategies are clearly poor choices, for example:

1. We can associate to each position a scalar $p = i/m$ which is linearly increasing

with the position. However, adding a single scalar to the token embeddings has a minor effect.

2. We can one-hot encode the position into a binary vector of size m, but the resulting vector would be extremely sparse and high-dimensional.

A possibility, introduced in the original transformer paper [VSP+17], is that of **sinusoidal embeddings**. To understand them, consider a sine function:

$$y = \sin(x)$$

The sine assigns a unique value to any input x inside the range $[0, 2\pi]$. We can also vary the frequency of the sine:

$$y = \sin(\omega x)$$

This oscillates more or less rapidly based on the frequency ω, and it assigns a unique value to any input in the range $[0, \frac{2\pi}{\omega}]$. There is an analogy with an (analogical) clock: the seconds' hand makes a full rotation with a frequency of $\frac{1}{60}$ Hz (once every minute). Hence, every "point in time" inside a minute can be distinguished by looking at the hand, but two time instants in general can only be identified modulo 60 seconds. We overcome this in a clock by adding a separate hand (the minute hand) that rotates with a much slower frequency of $\frac{1}{3600}$ Hz. Hence, by looking at the pair of coordinates (second, minute) (the "embedding" of time) we can distinguish any point inside an hour. Adding yet another hand with an even slower frequency (the hour hand) we can distinguish any point inside a day. This can be generalized: we could design clocks with lower or higher frequencies to distinguish months, years, or milliseconds.

A similar strategy can be applied here: we can distinguish each position i by encoding it through a set of e sines (with e an hyper-parameter) of increasing frequencies:

$$\mathbf{S}_i = [\sin(\omega_1 i), \sin(\omega_2 i), \ldots, \sin(\omega_e i)]$$

In practice, the original proposal from [VSP+17] uses only $e/2$ possible frequencies, but adds both sines and cosines:

$$\mathbf{S}_i = \left[\sin(\omega_1 i), \cos(\omega_1 i), \sin(\omega_2 i), \cos(\omega_2 i), \ldots, \sin(\omega_{e/2} i), \cos(\omega_{e/2} i)\right]$$

This can be justified by noting that in this embedding, two positions are related via a simple linear transformation, a rotation, that depends only on the relative offset

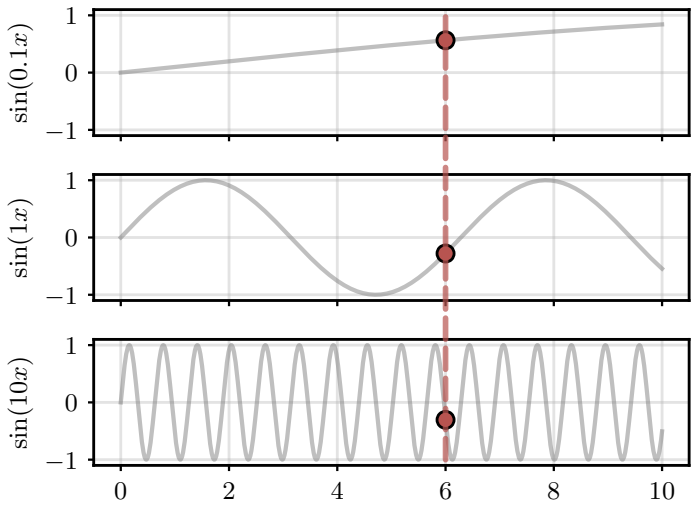


Figure F.10.6: *We show three* sin *functions with $\omega = 0.1$, $\omega = 1$, and $\omega = 10$. The embedding for position $x = 6$ is given by the corresponding values (red circles).*

of the two positions.[3] Any choice of frequency is valid provided they are sufficiently large and increasing at a super-linear rate. The choice from [VSP+17] was a geometric progression:

$$\omega_i = \frac{1}{10000^{i/e}}$$

that varies from $\omega_0 = 1$ to $\omega_e = \frac{1}{10000}$. See Figure F.10.6 for a visualization.

10.2.3 Relative positional embeddings

Trainable positional embeddings and sinuisodal positional embeddings are examples of **absolute** embeddings, because they encode a specific position in the sequence. An alternative that has become common with very long sequences are **relative positional embeddings**. In this case, instead of adding a positional encoding to a token, we modify the attention function to make it dependent on the offset between any two tokens:

$$g(\mathbf{x}_i, \mathbf{x}_j) \rightarrow g(\mathbf{x}_i, \mathbf{x}_j, i - j)$$

This is a combination of the two ideas we introduced at the beginning of this chapter (Figure F.10.1). Note that while absolute embeddings are added only once (at the input), relative embeddings must be added every time an MHA layer is used. As an example, we can add a trainable bias matrix $\mathbf{B} \sim (m, m)$ and rewrite the dot product

[3] See https://kazemnejad.com/blog/transformer_architecture_positional_encoding/ for a worked-out computation.

with an offset-dependent bias:

$$g(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^\top \mathbf{x}_j + B_{ij}$$

A simpler variant, **attention with linear biases** (ALiBi) [PSL22], considers a single trainable scalar in each head which is multiplied by a matrix of offsets. More advanced strategies, such as **rotary positional embeddings** (RoPE), are also possible [SAL+24].

10.3 Building the transformer model

10.3.1 The transformer block and model

A model could be built, in principle, from a stack of multiple MHA layers (with the softmax providing the non-linearity necessary to avoid the collapse of multiple linear projections). Empirically, however, it is found that the MHA works best when interleaved with a separate fully-connected block that operates on each token independently. These two operations can be understood as mixing the tokens (MHA), and mixing the channels (MLP), similarly to the depthwise-separable convolution model.

In particular, for the MLP block it is common to choose a bottleneck architecture composed of two fully-connected layers of the form:

$$\text{MLP}(\mathbf{x}) = \mathbf{W}_2 \phi\left(\mathbf{W}_1 \mathbf{x}\right)$$

where $\mathbf{x} \sim (e)$ is a token, $\mathbf{W}_1 \sim (p, e)$, with p selected as an integer multiple of e (e.g., $p = 3e$ or $p = 4e$), and $\mathbf{W}_2 \sim (e, p)$ reprojecting back to the original embedding dimension. Biases are generally removed as the increased hidden dimension provides sufficient degrees of freedom.

To ensure efficient training of deep models we also need a few additional regularization strategies. In particular, it is common to include two layer normalization steps and two residual connections, respectively for the MHA and MLP blocks. Depending on where the layer normalization is applied, we obtain two variants of the basic transformer block, sometimes denoted as **pre-normalized** and **post-normalized**. These are shown in Figure F.10.7.

While the post-normalized version corresponds to the original transformer block, the pre-normalized variant is generally found to be more stable and faster to train [XYH+20]. The design of the block in Figure F.10.7 is, fundamentally, an empirical choice, and

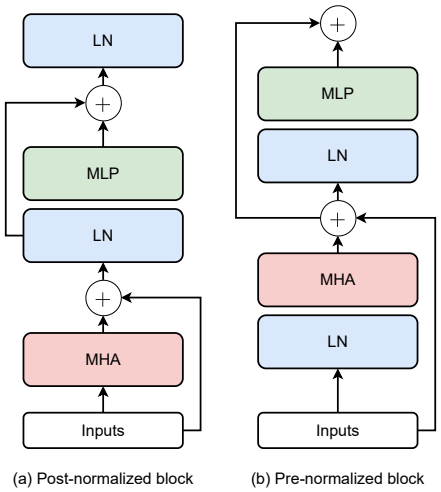


Figure F.10.7: *Schematic view of pre-normalized and post-normalized transformer blocks. In the post-normalized variant the LN block is applied after the MHA or MLP operation, while in the pre-normalized one before each layer.*

many variants have been proposed and tested in the literature. We review some of these later on in Section 11.3.

We can now complete the description of a basic transformer model:

1. Tokenize and embed the original input sequence in a matrix $\mathbf{X} \sim (n, e)$.
2. If using absolute positional embeddings, add them to the input matrix.
3. Apply 1 or more blocks of the form discussed above.
4. Include a final head depending on the task.

The output of step (3) is a set of processed tokens $\mathbf{H} \sim (n, e)$, where neither n nor e are changed by the transformer model (the former because we do not have local pooling operations on sets, the latter because of the residual connections in the block). Considering for example a classification task, we can apply a standard classification head by pooling over the tokens and proceeding with a fully-connected block:

$$y = \text{softmax}\left(\text{MLP}\left(\frac{1}{n}\sum_i \mathbf{H}_i\right)\right)$$

This part is identical to its corresponding CNN design. However, the transformer has a number of interesting properties, mostly stemming by the fact that it manipulates its input as a set, without modifying its dimensionality throughout the architecture. We investigate one simple example next.

10.3.2 Class tokens and register tokens

While up to now we have assumed that each token corresponds to one part of our input sequence, nothing prevents us from adding *additional* tokens to the input of the transformer. This is strictly dependent on its specific architecture: a CNN, for example, requires its input to be precisely ordered, and it is not clear how we could add additional tokens to an image or to a sequence. This is a very powerful idea, and we only consider two specific implementations here.

First, we consider the use of a **class token** [DBK+21], an additional token which is added explicitly for classification in order to replace the global pooling operation above. Suppose we initialize a single trainable token $\mathbf{c} \sim (e)$, which is added to the input matrix:

$$\mathbf{X} \leftarrow \begin{bmatrix} \mathbf{X} \\ \mathbf{c}^\top \end{bmatrix}$$

The new matrix has shape $(n+1, e)$. The class token is identical for all sequences in a mini-batch. After step (3) above, the transformer outputs a matrix $\mathbf{H} \sim (n+1, e)$ of updated representations for all tokens, including the class one. The idea is that, instead of pooling over the tokens, the model should be able to "compress" all information related to the classification task inside the class token, and we can rewrite the classification head by simply discarding all other tokens:[4]

$$y = \text{softmax}(\text{MLP}(\mathbf{H}_{n+1}))$$

Additional trainable tokens can be useful even if not explicitly used. For example, [DOMB24] has shown that adding a few additional tokens (called **registers** in this case) can improve the quality of the attention maps by providing the model with the possibility of using the registers to "store" auxiliary information that does not depend explicitly on a given position.

From theory to practice

We will introduce many important concepts related to transformers in the next chapter. Thus, for this chapter I am suggesting a slightly unconventional exercise which combines a convolutional backbone with a transformer-like head – as depicted in Figure F.10.8.

[4] In the language of circuits and heads from Section 10.1.3, we could say equivalently that the model must learn to move all information related to the classification in the residual stream of the class token.

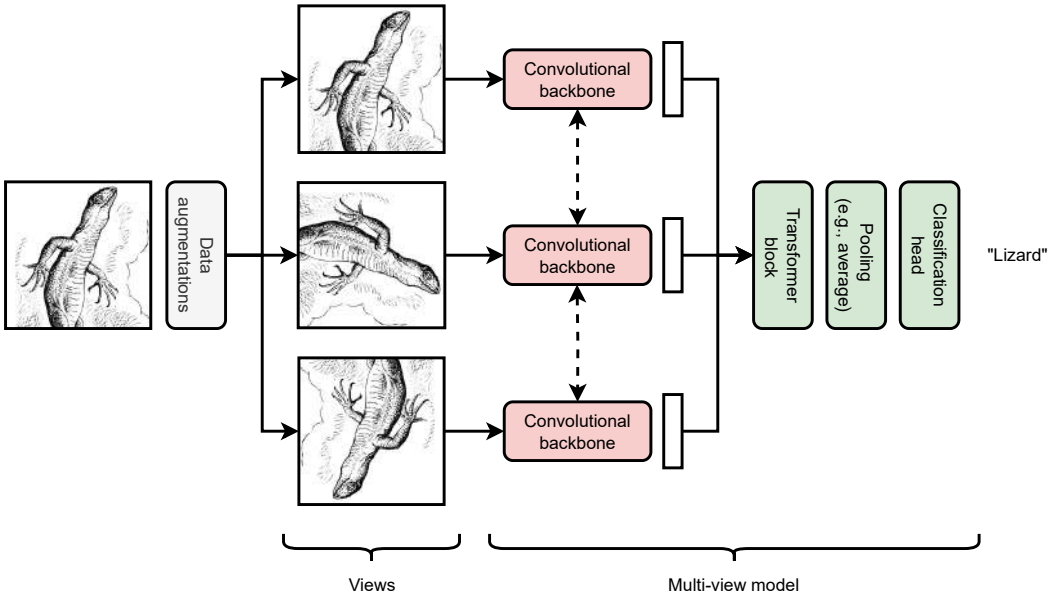


Figure F.10.8: *Multi-view model to be implemented in this chapter. The image is augmented through a set of random data augmentation strategies to obtain a set of **views** of the input (gray). Each view is processed by the same convolutional backbone to obtain a fixed-sized dimensional embedding (red). The set of embeddings are processed by a transformer block before the final classification (green). Illustration by John Tenniel.*

The convolutional models you developed in Chapters 7 and 9 were applied to a *single* image. However, sometimes we have available a *set* of images of the same object to be recognized – for example, in a monitoring system, we may have multiple screenshots of a suspicious person. This is called a **multi-view** system in the literature, and each image is called a **view** of the object. A multi-view model should provide a single prediction for the entire set of views, while being invariant to the order of the views in input. For this exercise we will implement a simple multi-view model – see Figure F.10.8.

1. Using any image classification dataset, you can simulate a multi-view model by applying a fixed number of data transformations to the input (gray block in Figure F.10.8). Ignoring the batch dimension, for each input image of shape $x \sim (h, w, c)$ (height, width, channels), you obtain a *multi-view* input of shape $x' \sim (v, h, w, c)$, where v is the number of views. A single label y is associated to this tensor – the label of the original image. The number of views can also be different from

mini-batch to mini-batch, as no part of the model is constrained to a pre-specified number of view.

2. The multi-view model is composed of three components. Denote by $g(x)$ a model that processes a single view to a fixed-dimensional embedding – for example, this can be any convolutional backbone you trained for the previous exercises. The first part of the full model (red part in Figure F.10.8) applies g in parallel to all views, $\mathbf{h}_i = g(x_i) \sim (e)$, where e is a hyper-parameter (the output size of the backbone).

3. After concatenating the embeddings of the views we obtain a matrix $\mathbf{H} \sim (v, e)$. In order for the full model to be permutation invariant, any component applied on $\mathbf{H}$ must be permutation equivariant.[5] For the purposes of this exercise, implement and apply a single transformer block as per Section 10.3.1. You can implement MHA using basic PyTorch, or you can try a more advanced implementation using `einops`.[6] You can also compare with the pre-implemented version in `torch.nn`.

4. The transformer block does not modify the input shape. To complete the model, perform an average over the views (which represent the tokens in this scenario), and apply a final classification head. You can also experiment with adding a class token (Section 10.3.2). It is easy to show that a model built in this way is permutation invariant with respect to the views.

[5] An average operation over the views is the simplest example of permutation invariant layer. Hence, removing the MHA block from Figure F.10.8 is also a valid baseline. Alternatively, deep sets [ZKR+17] characterize the full spectrum of linear, permutation invariant layers.

[6] See https://einops.rocks/pytorch-examples.html.

11 | Transformers in practice

About this chapter

We now consider a few variations of the basic transformer model, including encoder-decoder architectures, causal MHA layers, and applications to the image and audio domains.

11.1 Encoder-decoder transformers

The model we described in Chapter 10 can be used to perform regression or classification of a given sequence. However, the original transformer [VSP+17] was a more complex model, designed for what are called **sequence-to-sequence** (seq2seq) tasks. In a seq2seq task, both input and output are sequences, and there is no trivial correspondence between their tokens. A notable example is **machine translation**, in which the output is the translation of the input sequence in a different language.

One possibility to build a differentiable model for seq2seq tasks is an **encoder-decoder** (ED) design [SVL14]. An ED model is composed of two blocks: an encoder that processes the input sequence to a transformed representation (possibly of a fixed dimensionality), and a decoder that autoregressively generates the output sequence conditioned on the output of the encoder. The transformer model we described before can be used to build the encoder: transformers of this type for classification are called **encoder-only** transformers. In order to build the decoder we need two additional components: a way to make the model causal (to perform autoregression), and a way to condition its computation to a separate input (the output of the encoder).

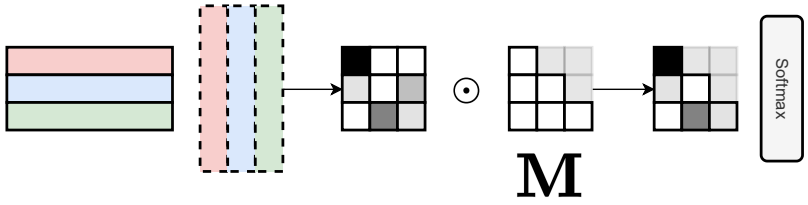


Figure F.11.1: *Visual depiction of causal attention implemented with attention masking.*

11.1.1 Causal multi-head attention

Let us consider first the problem of making the transformer block causal. The only component in which tokens interact is the MHA block. Hence, having a causal variant of MHA is enough to make the entire model causal. Remember that, for convolutions, we designed a causal variant by appropriately masking the weights in the convolutional filter. For MHA, we can mask instead all interactions between tokens that do not satisfy the causality property:

$$\text{Masked-SA}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top \odot \mathbf{M}}{\sqrt{k}}\right)\mathbf{V}$$

It is essential to perform the masking inside the softmax. Consider the following (wrong) variant:

$$\textbf{Wrong:} \quad \left(\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{k}}\right) \odot \mathbf{M}\right)\mathbf{V}$$

Because of the denominator in the softmax, all tokens participate in the computation of each token, irrespective of the later masking. Also note that setting $M_{ij} = 0$ for non-causal links does not work, because $\exp(0) = 1$. Hence, the correct implementation of a masked variant of MHA is to select an upper triangular matrix with $-\infty$ on the upper part, since $\exp(-\infty) = 0$ as desired:

$$M_{ij} = \begin{cases} -\infty & \text{if } i > j \\ 0 & \text{otherwise} \end{cases}$$

Practically, the values can be set to a very large, negative number instead (e.g., -10^9).

11.1.2 Cross-attention

Second, let us consider the problem of making the output of the MHA layer depend on a separate block of inputs. To this end, let us rewrite the MHA operation by explicitly

separating the three appearances of the input matrix:

$$\text{SA}(\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3) = \text{softmax}\left(\frac{\mathbf{X}_1 \mathbf{W}_q \mathbf{W}_k^\top \mathbf{X}_2^\top}{\sqrt{k}}\right) \mathbf{X}_3 \mathbf{W}_v$$

The SA layer corresponds to $\mathbf{X}_1 = \mathbf{X}_2 = \mathbf{X}_3 = \mathbf{X}$ (which, coincidentally, explains the name we gave to it). However, the formulation also works if we consider keys, values, and queries belonging to separate sets. One important case is **cross-attention** (CA), in which we assume that the keys and values are computed from a second matrix $\mathbf{Z} \sim (m, e)$:

Cross-attention between $\mathbf{X}$ and $\mathbf{Z}$

$$\text{CA}(\mathbf{X}, \mathbf{Z}) = \text{SA}(\mathbf{X}, \mathbf{Z}, \mathbf{Z}) = \text{softmax}\left(\frac{\mathbf{X}\mathbf{W}_q \mathbf{W}_k^\top \mathbf{Z}^\top}{\sqrt{k}}\right) \mathbf{Z}\mathbf{W}_v \tag{E.11.1}$$

The interpretation is that the embeddings of $\mathbf{X}$ are updated based on their similarity with a set of external (key, values) pairs provided by $\mathbf{Z}$: we say that $\mathbf{X}$ is *cross-attending* on $\mathbf{Z}$. Note that this formulation is very similar to a concatenation of the two set of input tokens followed by an appropriate masking of the attention matrix.

Comparison with feedforward layers

Consider a simplified variant of the cross-attention operation in (E.11.1), in which we parameterize explicitly the keys and values matrices:[1]

$$\text{NeuralMemory}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{X}\mathbf{W}_q \mathbf{K}}{\sqrt{k}}\right) \mathbf{V} \tag{E.11.2}$$

The layer is now parameterized by a query projection matrix $\mathbf{W}_q$ and by the two matrices $\mathbf{K}$ and $\mathbf{V}$. (E.11.2) is called a **memory layer** [SWF+15], in the sense that rows of the key and value matrices are used by the model to store interesting patterns to be retrieved dynamically by an attention-like operation. If we further simplify the layer by setting $\mathbf{W}_q = \mathbf{I}$, ignoring the normalization by $\sqrt{k}$, and replacing the softmax with a generic activation function ϕ, we obtain a two-layer MLP:

$$\text{MLP}(\mathbf{X}) = \phi\left(\mathbf{X}\mathbf{K}\right)\mathbf{V} \tag{E.11.3}$$

[1] See also the discussion on the perceiver network in Section 11.2.1.

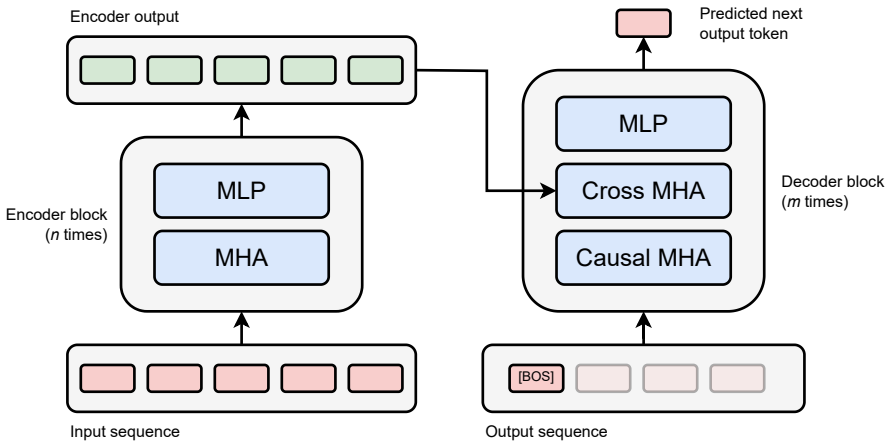


Figure F.11.2: *Encoder-decoder architecture, adapted from [VSP+17]. Padded tokens in the decoder are greyed out.*

Hence, MLPs in transformer networks can be seen as approximating an attention operation over trainable keys and values. Visualizing the closest tokens in the training data shows human-understandable patterns [GSBL20].

11.1.3 The complete encoder-decoder transformer

With these two components in hand, we are ready to discuss the original transformer model, shown in Figure F.11.2.[2] First, the input sequence **X** is processed by a standard transformer model (called the **encoder**), providing an updated embedding sequence **H**. Next, the output sequence is predicted autoregressively by another transformer model (called the **decoder**). Differently from the encoder, the decoder has three components for each block:

1. A masked variant of the MHA layer (to ensure autoregression is possible).
2. A cross-attention layer where the queries are given by the input sequence embedding **H**.
3. A standard token-wise MLP.

Decoder-only models are also possible, in which case the second block of the decoder

[2] A pedantic note: technically, Transformer (upper-cased) is a proper noun in [VSP+17]. In the book, I use transformer (lower-cased) to refer to any model composed primarily of attention layers.

```
def self_attention(Q: Float[Array, "n k"],
                   K: Float[Array, "n k"],
                   V: Float[Array, "n v"]
                 ) -> Float[Array, "n v"]:
        return nn.softmax(Q @ K.T) @ V
```

Box C.11.1: *Simple implementation of the SA layer, explicitly parameterized in terms of the query, key, and value matrices.*

is removed and only masked MHA and MLPs are used. Most modern LLMs are built by decoder-only models trained to autoregressively generate text tokens [RWC+19], as discussed below. In fact, encoder-decoder models have become less common with the realization that many seq2seq tasks can be solved directly with decoder-only models by concatenating the input sequence to the generated output sequence, as described in Section 8.4.3.

11.2 Computational considerations

11.2.1 Time complexity and linear-time transformers

The MHA performance does not come without a cost: since every token must attend to all other tokens, its complexity is higher than a simpler convolutional operation. To understand this, we look at its complexity from two points of view: memory and time. We use a naive implementation of the SA layer for reference, shown in Box C.11.1.

Let us look first at the time complexity. The operation inside the softmax scales as $\mathcal{O}(n^2k)$ because it needs to compute n^2 dot products (one for each pair of tokens). Compare this to a 1D convolutional layer, which scales only linearly in the sequence length. *Theoretically*, this quadratic growth in complexity can be problematic for very large sequences, which are common in, e.g., LLMs.

This has led to the development of several strategies for speeding up autoregressive generation (e.g., speculative decoding [LKM23]), as long as linear or sub-quadratic variants of transformers. As an example, we can replace the SA layer with a cross-attention layer having a *trainable* set of tokens $\mathbf{Z}$, where the number of tokens can be chosen as hyper-parameter and controlled by the user. This strategy was popularized by the Perceiver architecture [JGB+21] to distill the original set of tokens into smaller latent bottlenecks. There are many alternative strategies for designing linearized trans-

formers: we discuss a few variants in Section 11.3 and Chapter 13.

Importantly, an implementation such as the one in Box C.11.1 can be shown to be heavily memory-bound on modern hardware [DFE+22], meaning that its compute cost is dominated by memory and I/O operations. Hence, the theoretical gains of linear-time attention variants are not correlated with actual speedup on hardware. Combined with a possible reduction in performance, this makes them less attractive than a strongly-optimized implementation of MHA, such as the one described next.

11.2.2 Memory complexity and the online softmax

In terms of memory, the implementation in Box C.11.1 has also a quadratic n^2 complexity factor because the attention matrix $\mathbf{QK}^\top$ is fully materialized during computation. However, this is unnecessary and this complexity can be drastically reduced to a linear factor by chunking the computation in blocks and only performing the softmax normalization at the end [RS21].

To understand this, consider a single query vector $\mathbf{q}$, and suppose we split our keys and values into two blocks, which are loaded in turn in memory:

$$\mathbf{K} = \begin{bmatrix} \mathbf{K}_1 \\ \mathbf{K}_2 \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \end{bmatrix} \tag{E.11.4}$$

If we ignore the denominator in the softmax, we can decompose the SA operation, computing the output for each chunk in turn:

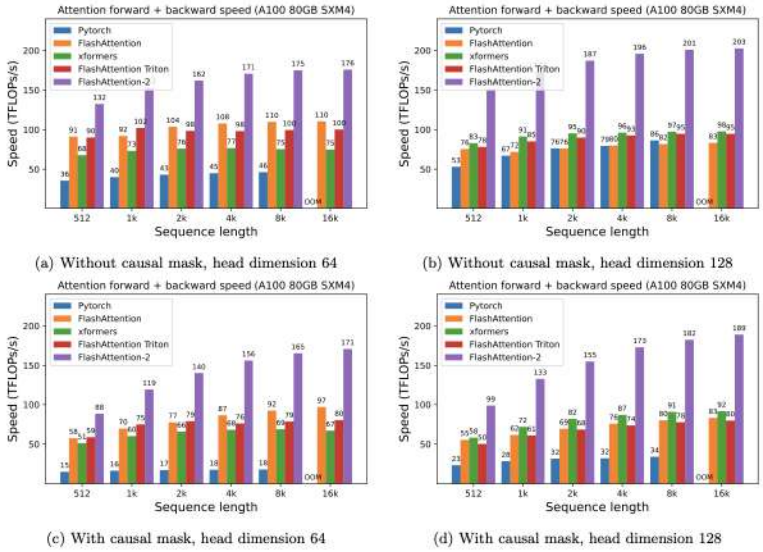
$$\mathrm{SA}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \frac{1}{L_1 + L_2} [\mathbf{h}_1 + \mathbf{h}_2] \tag{E.11.5}$$

where for the two chunks $i = 1, 2$ we have defined two auxiliary quantities:

$$\mathbf{h}_i = \exp(\mathbf{K}_i \mathbf{q}) \mathbf{V}_i \tag{E.11.6}$$

$$L_i = \sum_j [\exp(\mathbf{K}_i \mathbf{q})]_j \tag{E.11.7}$$

Remember we are loading the chunks in memory separately, hence for chunk 1 we compute $\mathbf{h}_1$ and L_1; then we offload the previous chunk and we compute $\mathbf{h}_2$ and L_2 for chunk 2. Note that the operation is not fully-decomposable unless we keep track of



(a) Without causal mask, head dimension 64

(b) Without causal mask, head dimension 128

(c) With causal mask, head dimension 64

(d) With causal mask, head dimension 128

Figure F.11.3: *Official benchmark of FlashAttention and FlashAttention-2 on an NVIDIA A100 GPU card, reproduced from* https://github.com/Dao-AILab/flash-attention.

the additional statistics L_i (which is needed to compute the normalization coefficients of the softmax operation). More in general, for multiple chunks $i = 1, \ldots, m$ we will have:

$$\text{SA}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \frac{1}{\sum_{i=1}^{m} L_i} \left[\sum_{i=1}^{m} \mathbf{h}_i \right] \tag{E.11.8}$$

Hence, we can design a simple iterative algorithm where for every block of keys and values loaded in memory, we update and store the cumulative sum of the numerator and denominator in (E.11.8), only performing the normalization at the end. This trick (sometimes called *online* softmax), combined with an IO-aware implementation and kernel fusion has led to highly memory- and compute- efficient implementations of attention such as FlashAttention-2.[3] Distributed implementations of attention (e.g., RingAttention [LZA23]) can also be devised by assigning groups of queries to different devices and rotating chunks of keys and queries among the devices. Optimizing the operation for specific hardware can lead to some counter-intuitive behaviours, such as *increased* speed for larger sequence lengths - see Figure F.11.3.

[3] https://github.com/Dao-AILab/flash-attention

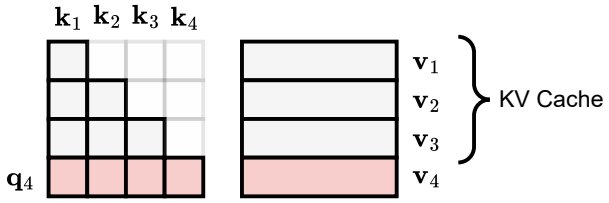


Figure F.11.4: *To compute masked self-attention on a new token, most of the previous computation can be reused (in gray). This is called the **KV cache**.*

11.2.3 The KV cache

An important implementative aspect of MHA occurs when dealing with autoregressive generation in decoder-only models. For each new token to be generated, only a new row of the attention matrix and one value token must be computed, while the rest of the attention matrix and the remaining value tokens can be stored in memory, as shown in Figure F.11.4. This is called the **KV cache** and it is a standard in most optimized implementations of MHA.

The size of the KV cache is linearly increasing in the sequence length. Once again, you can compare this to an equivalent implementation of a causal convolutional layer, where memory is upper-bounded by the size of the receptive field. Designing expressive layers with a fixed memory cost in autoregressive generation is a motivating factor for Chapter 13.

11.2.4 Transformers for images and audio

Transformers were originally developed for text, and they soon became the default choice for language modeling. In particular, the popular GPT-2 model [RWC+19] (and later variants) is a decoder-only architecture which is pre-trained by forecasting tokens in text sequences. Most open-source LLMs, such as LLaMa [TLI+23], follow a similar architecture. By constrast, BERT [DCLT18] is another popular family of pre-trained word embeddings based on an encoder-only architecture trained to predict masked tokens (**masked language modeling**). Differently from GPT-like models, BERT-like models cannot be used to generate text but only to perform text embedding or as the first part of a fine-tuned architecture. Encoder-decoder models for language modeling also exist (e.g., the T5 family [RSR+20]), but they have become less popular.

From a high-level point of view, a transformer is composed of three components: a tokenization / embedding step, which converts the original input into a sequence of vectors; positional embeddings to encode information about the ordering of the original sequence; and the transformer blocks themselves. Hence, transformers for other

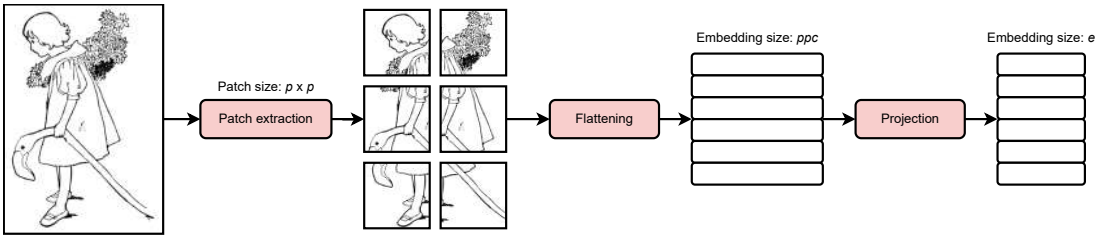


Figure F.11.5: *Image tokenization: the image is split into non-overlapping patches of shape $p \times p$ (with p an hyper-parameter). Then, each patch is flattened and undergoes a further linear projection to a user-defined embedding size e. c is the number of channels of the input image.*

types of data can be designed by defining the appropriate tokenization procedure and positional embeddings.

Let us consider first computer vision. Tokenizing an image at the pixel level is too expensive, because of the quadratic growth in complexity with respect to the sequence length. The core idea of Vision Transformers (ViTs, [DBK+21]) is to split the original input into non-overlapping **patches** of fixed length, which are then flattened and projected to an embedding of pre-defined size, as shown in Figure F.11.5.

The embedding step in Figure F.11.5 can be achieved with a convolutional layer, having stride equal to the kernel size. Alternatively, libraries like einops[4] extend the einsum operation (Section 2.1) to allow for grouping of elements into blocks of pre-determined shape. An example is shown in Box C.11.2.

The original ViT used trainable positional embeddings along with an additional class token to perform image classification. ViTs can also be used for image generation by predicting the patches in a row-major or column-major order. In this case, we can train a separate module that converts each patch into a discrete set of tokens using, e.g., a **vector-quantized variational autoencoder** [CZJ+22], or we can work directly with continuous outputs [TEM23]. For image generation, however, other non-autoregressive approaches such as diffusion models and flow matching tend to be preferred; we cover them in the next volume.

By developing proper tokenization mechanisms and positional embeddings, transformers have also been developed for audio, in particular for speech recognition. In this case, it is common to have a small 1D convolutional model (with pooling) as the tokenization block [BZMA20, RKX+23]. For example, Wav2Vec [BZMA20] is an encoder-

[4] http://einops.rocks

```python
from einops import rearrange
# A batch of images
xb = torch.randn((32, 3, 64, 64))

# Define the operation: differently from standard einsum,
# we can split the output in blocks using brackets
op = 'b c (h ph) (w pw) -> b (h w) (ph pw c)'

# Run the operation with a given patch size
patches = rearrange(xb, op, ph=8, pw=8)
print(patches.shape) # [Out]: (32, 64, 192)
```

Box C.11.2: *einops can be used to decompose an image into patches with a simple extension of the einsum syntax.*

only model whose output is trained with an extension of the cross-entropy loss, called **connectionist temporal classification** loss [GFGS06], to align the output embeddings to the transcription. Because labeled data with precise alignments is scarce, Wav2Vec models are pre-trained on large amounts of unlabeled audio with a variant of a masked language modeling loss. By contrast, Whisper [RKX+23] is an encoder-decoder model where the decoder is trained to autoregressively generate the transcription. This provides more flexibility to the model and reduces the need for strongly labeled data, but at the cost of possible *hallucinations* in the transcription phase. **Neural audio codecs** can also be trained to compress audio into a sequence of discrete tokens [DCSA23], which in turn form the basis for generative applications such as text-to-speech generation [WCW+23].

Transformers can also be defined for time-series [AST+24], graphs (covered in the next chapter) and other types of data. The decoupling between data and architecture is also the basis for **multimodal** variants, which can take as input (or provide as output) different types of data. This is achieved by tokenizing each modality (image, audio, ...) with its appropriate tokenizer, and concatenating the different tokens together into a single sequence [BPA+24]. We show an example for an **image-text** model in Figure F.11.6.

11.3 Variants of the transformer block

We close the chapter by discussing a few interesting variation on the basic transformer block. First, several variants have been devised for very large transformers to slightly

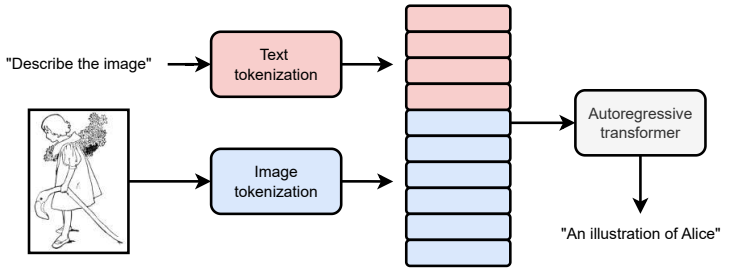


Figure F.11.6: *An example of a **bimodal** transformer that operates on both images and text: the outputs of the two tokenizers are concatenated and sent to the model.*

reduce the computational time or parameter's count. As an example, **parallel blocks** [DDM+23] perform the MLP and MHA operation in parallel:

$$\mathbf{H} = \mathbf{H} + \text{MLP}(\mathbf{H}) + \text{MHA}(\mathbf{H})$$

In this way, the initial and final linear projections in the MLP and MHA layers can be fused for a more efficient implementation. As another example, **multi-query MHA** [Sha19] shares the same key and value projection matrix for each head, varying only the queries.

More in general, we can replace the MHA layer with a simpler (linear complexity in the sequence length) operation, while keeping the overall structure of the transformer block, i.e., alternating token and channel mixing with layer normalization and residual connections. As an example, suppose the sequence length is fixed (e.g., for computer vision, the number of patches can be fixed a priori). In this case, the MHA layer can be replaced by an MLP operating on a single input channel, corresponding to one dimension of the embedding. This type of model is called a **mixer** model [THK+21] - see Figure F.11.7. Ignoring the normalization operations, this can be written as alternating MLPs on transpositions of the input matrix:

$$\mathbf{H} = \text{MLP}(\mathbf{H}) + \mathbf{H} \tag{E.11.9}$$

$$\mathbf{H} = \left[\text{MLP}(\mathbf{H}^\top) + \mathbf{H}^\top\right]^\top \tag{E.11.10}$$

Other variants of the mixer model are also possible using, e.g., 1D convolutions, Fourier transforms, or pooling. In particular, in the S2-MLP [YLC+22] model the token mixing operation is replaced by an even simpler MLP applied on a shifted version of its input. The general class of such models has been called **MetaFormers** by [YLZ+22].

Gated (multiplicative) interactions can also be used in the composition of the block. In this case, several blocks are executed in parallel but their output is combined via

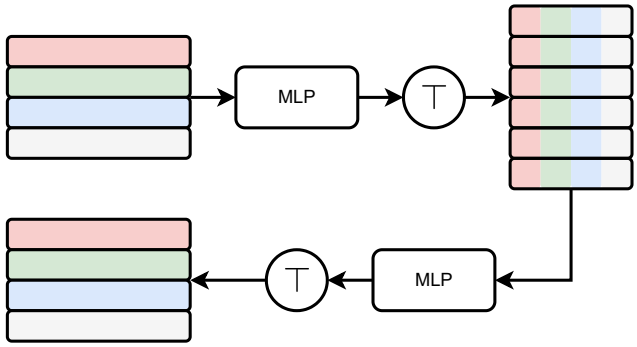


Figure F.11.7: *Mixer block, composed of alternating MLPs on the rows and columns of the input matrix.*

Hadamard multiplication. We can write a generic gated unit as:

$$f(\mathbf{X}) = \phi_1(\mathbf{X}) \odot \phi_2(\mathbf{X}) \tag{E.11.11}$$

where ϕ_1 and ϕ_2 are trainable blocks. For example, with $\phi_1(\mathbf{X}) = \sigma(\mathbf{XA})$ and $\phi_2(\mathbf{X}) = \mathbf{XB}$ we obtain the **gated linear unit** (GLU) described in Section 5.4.

As a few representative examples, the **gMLP** model [LDSL21] uses gated units instead of a channel mixing block in a mixer model; the **LLaMa** family of models [TLI+23] uses GLU-like units instead of the standard MLP block; while the **gated attention unit** (GAU) [HDLL22] uses a simpler attention-like model having a single head for ϕ_1 and a linear projection for ϕ_2. These designs are especially popular in some recent variants of recurrent models, discussed later on in Chapter 13.

To simplify the design even further, the multilinear operator network (MONet) removes all activation functions to define a block which is composed only of linear projections and element-wise multiplications [CCGC24]:

$$\mathbf{H} = \mathbf{E}(\mathbf{AX} \odot \mathbf{BX} + \mathbf{DX})$$

where $\mathbf{E}$ is similar to the output projection in the transformer block, $\mathbf{DX}$ acts as a residual connection, and $\mathbf{B}$ is implemented via a low-rank decomposition to reduce the number of parameters [CCGC24]. In order to introduce token mixing, a token-shift operation is implemented in all odd-numbered blocks in the model.

From theory to practice

There are many interesting exercises that can be done at this point – you are almost a master in designing differentiable models! To begin, using any image classification dataset, you can try implementing from scratch a Vision Transformer as described in Section 11.2.4, following [DBK+21] for choosing the hyper-parameters. Training a ViT from scratch on a small dataset is quite challenging [LLS21, SKZ+21], so be ready for some disappointment unless you have sufficient computational power to consider million-size datasets. You can also try a simpler variant, such as the Mixer model described in Section 11.3. All these exercises should be relatively simple.

1. For tokenizing the image you can use Einops as in Box C.11.2 or other strategies (e.g., a convolution with large stride). For small images you can also try using each pixel as token.

2. For positional embeddings, all strategies described in Section 10.2 are valid. The simplest one for a ViT is to initialize a matrix of *trainable* embeddings, but I suggest you experiment with sinusoidal and relative embeddings as practice.

You can also try implementing a small GPT-like model. There are many sophistications in the tokenization of text data that we do not cover. However, the minGPT repository[5] is a fantastic didactic implementation that you can use as starting point.

[5] https://github.com/karpathy/minGPT

12 | Graph models

About this chapter

In this chapter we consider graph-structured data, i.e., nodes connected by a set of (known) relations. Graph are pervasive in the real world, ranging from proteins to traffic networks, social networks, and recommender systems. We introduce specialized layers to work on graphs, broadly categorized as either message-passing layers or graph transformers architectures.

12.1 Learning on graph-based data

12.1.1 Graphs and features on graphs

Up to now we have considered data which is either completely unstructured (tabular data represented as a vector) or structured in simple ways, including sets, sequences, and grids such as images. However, many types of data are defined by more sophisticated dependencies between its constituents. For example, molecules are composed by atoms which are only sparsely connected via chemical bonds. Networks of many kinds (social networks, transportation networks, energy networks) are composed of millions of units (people, products, users) which interact only through a small set of connections, e.g., roads, feedbacks, or friendships. These are more naturally defined in the language of **graph theory**. The aim of this chapter is to introduce differentiable models to work with data defined in such a way.

In its simplest form, a graph can be described by a pair of sets $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where

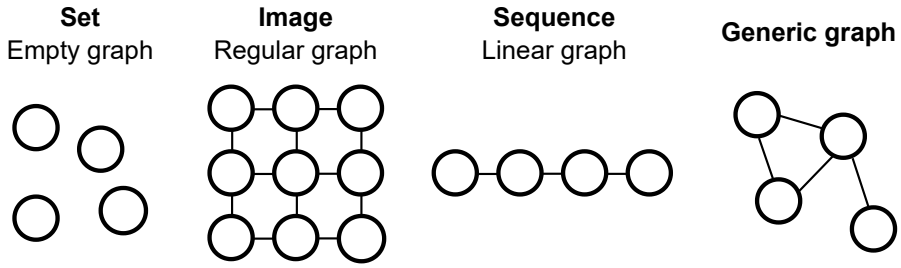


Figure F.12.1: *Graphs generalize many types of data: sets can be seen as empty graphs (or graphs having only self-loops), images as regular graphs, and sequences as linear graphs. In this chapter we look at more general graph structures.*

$\mathcal{V} = \{1, \ldots, n\}$ is the set of **nodes** (**vertices**), while:

$$\mathcal{E} = \Big\{(i,j) \mid \underbrace{i,j \in \mathcal{N}}_{\text{Two nodes of the graph}}\Big\}$$

is the set of **edges** present in the graph. In most datasets, the number of nodes n and the number of edges $m = |\mathcal{E}|$ can vary from graph to graph.

Graph generalize many concepts we have already seen: for example, graphs containing only **self-loops** of the form (i,i) represent sets of objects, while graphs containing all possible edges (**fully-connected graphs**) are connected to attention layers, as we show next. Images can be represented as a graph by associating each pixel to a node of the graph and connecting close pixels based on a regular grid-like structure - see Figure F.12.1.[1]

Connections in a graph can be equivalently represented by a matrix representation called the **adjacency matrix**. This is a binary square matrix $\mathbf{A} \sim \text{Binary}(n,n)$ such that:

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in \mathcal{E} \\ 0 & \text{otherwise} \end{cases}$$

In this format, a set is represented by the identity matrix $\mathbf{A} = \mathbf{I}$, a fully-connected graph by a matrix of all ones, and an image by a Toeplitz matrix. A graph where connections are always bidirectional (i.e., (i,j) and (j,i) are always present as pairs among the

[1] There are many variants of this basic setup, including heterogenous graphs (graphs with different types of nodes), directed graphs, signed graphs, etc. Most of them can be handled by variations of the techniques we describe next.

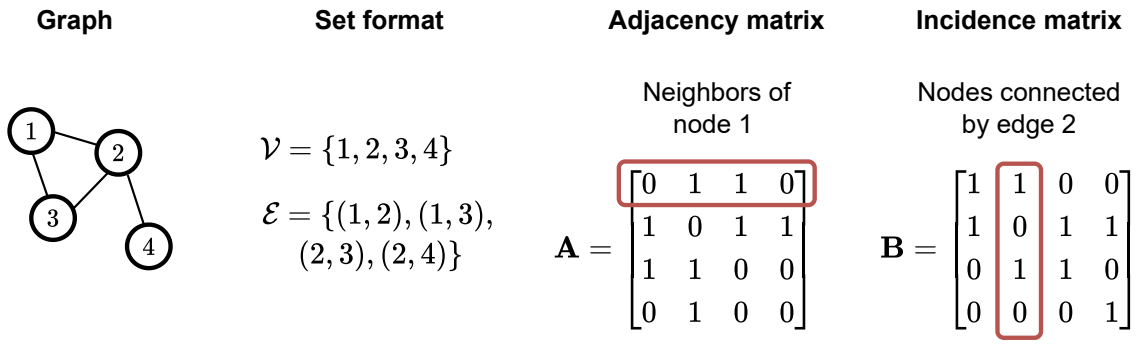


Figure F.12.2: *We can represent the graph connectivity in three ways: as a set $\mathcal{E}$ of pairs (second column); as an (n, n) adjacency matrix (third column); or as an $(n, |\mathcal{E}|)$ incidence matrix (fourth column).*

edges) is called **undirected**, and we have $\mathbf{A}^\top = \mathbf{A}$. We will deal with undirected graphs for simplicity, but the methods can be easily extended to the directed case. We note that there are also alternative matrix representations, e.g., the incidence matrix $\mathbf{B} \sim \text{Binary}(n, |\mathcal{E}|)$ is such that $B_{ij} = 1$ if node i participates in edge j, and we have $\mathbf{B}\mathbf{1}^\top = 2$ because each edge connects exactly two nodes. See Figure F.12.2 for an example.

We will assume our graphs to have self-loops, i.e., $A_{ii} = 1$. If the adjacency matrix does not have self-loops, we can add them by re-assigning it as:

$$\mathbf{A} \leftarrow \mathbf{A} + \mathbf{I}$$

12.1.2 Graph features

Graphs come with a variety of possible features describing them. For example, atoms and bonds in a molecule can be described by categorical features denoting their types; roads in a transportation network can have a capacity and a traffic flow; and two friends in a social networks can be described by how many years they have known each other.

In general, these features can be of three types: **node features** associated to each node, **edge features** associated to each edge, and **graph features** associated to the entire graph. We will begin with the simplest case of having access to only unstructured node features, i.e., each node i has associated a vector $\mathbf{x}_i \sim (c)$. The complete graph can then be described by two matrices $\mathbf{X} \sim (n, c)$, that we call the **feature matrix**, and the adjacency matrix $\mathbf{A} \sim (n, n)$.

In most cases, the ordering of the nodes is irrelevant, i.e., if we consider a permutation matrix $\mathbf{P} \sim \text{Binary}(n, n)$ (see Section 10.2), a graph and its permuted version are fundamentally identical, in other words:

$$(\mathbf{X}, \mathbf{A}) \quad \text{is the same graph as} \quad (\mathbf{PX}, \mathbf{PAP}^\top)$$

Note that the permutation matrix acts by swapping the rows in $\mathbf{X}$, while it swaps both rows and columns in the adjacency matrix.

Some features can also be extracted directly from the topology of the graph. For example, we can associate to each node a scalar value d_i, called the **degree**, which describes how many nodes it is connected to:

$$d_i = \sum_j A_{ij}$$

The distribution of the degrees across the graph is an important characteristic of the graph itself, as shown in Figure F.12.3. We can collect the degrees into a single diagonal matrix called the **degree** matrix:

$$\mathbf{D} = \begin{bmatrix} d_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & d_n \end{bmatrix}$$

We can use the degree matrix to define several types of of *weighted* adjacency matrices. For example, the row-normalized adjacency matrix is defined as:

$$\mathbf{A}' \leftarrow \mathbf{D}^{-1}\mathbf{A}_{ij} \quad \rightarrow \quad A'_{ij} = \frac{1}{d_i} A_{ij}$$

This is normalized in the sense that $\sum_i A'_{ij} = \mathbf{1}$. We can also define a column-normalized adjacency matrix as $\mathbf{A}' = \mathbf{AD}^{-1}$. Both these matrices can be interpreted as "random walks" over the graph, in the sense that, given a node i, the corresponding row or column of the normalized adjacency matrix represents a probability distribution of moving at random towards any of its neighbours. A more general symmetrically normalized

adjacency matrix is given by:

$$\mathbf{A}' = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$$

This is defined by $A'_{ij} = \frac{A_{ij}}{\sqrt{d_i d_j}}$, giving a weight to each connection based on the degree of both nodes it connects to. Both the adjacency matrix and its weighted variants have the property that $A_{ij} = 0$ whenever $(i, j) \notin \mathcal{E}$. In signal processing terms, these are called **graph-shift** matrices.

Sparsity in matrices

Consider a generic adjacency matrix for a 6-nodes graph (try drawing the graph as an exercise):

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

The adjacency is very sparse (many zeros). This is an important property, because sparse matrices have customized implementations and techniques for manipulating them, with better computational complexity than their dense counterparts.[a]

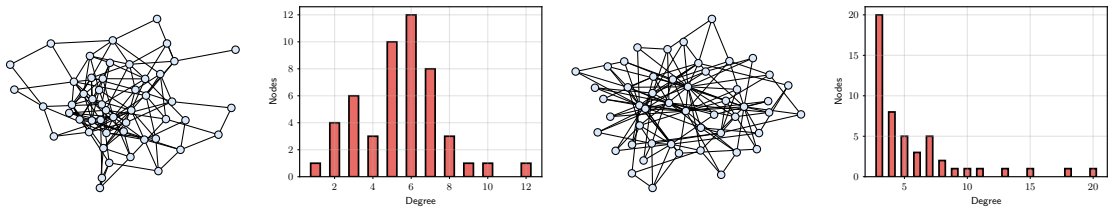
[a] As an example, in JAX: https://jax.readthedocs.io/en/latest/jax.experimental.sparse.html.

12.1.3 Diffusion operations over graphs

The fundamental graph operation we are interested into is something called **diffusion**, which corresponds to a smoothing of the node features with respect to the graph topology. To understand it, consider a scalar feature on each node, that we collect in a vector $\mathbf{x} \sim (n)$, and the following operation over the features:

$$\mathbf{x}' = \mathbf{A}\mathbf{x}$$

where $\mathbf{A}$ can be the adjacency matrix, a normalized variant, or any weighted adjacency matrix. We can re-write this operation node-wise as:



(a) *Erdős–Rényi* **(b)** *Degree* **(c)** *Barabasi-Albert* **(d)** *Degree*

Figure F.12.3: *(a) Random graph generated by drawing each edge independently from a Bernoulli distribution (**Erdős–Rényi model**). (b) These graphs show a Gaussian-like degree distribution. (c) Random graph generated by adding nodes sequentially, and for each of them drawing* 3 *connections towards existing nodes with a probability proportional to their degree (**preferential attachment process** or **Barabasi-Albert model**). (d) These graphs have a few nodes with many connections acting as hubs for the graph.*

$$x_i' = \sum_{j \in \mathcal{N}(i)} A_{ij} x_j$$

where we have defined the 1-hop neighborhood:

All edges with node i as a vertex

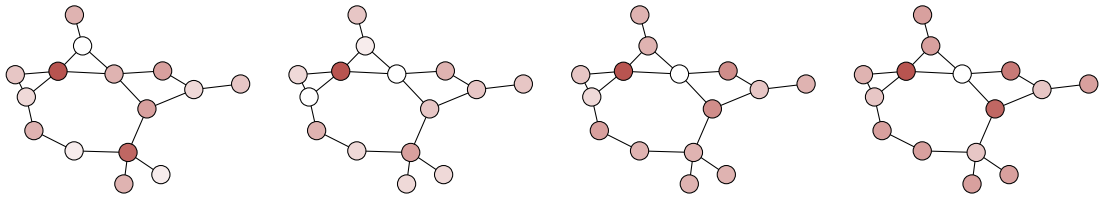
$$\mathcal{N}(i) = \left\{ j \mid (i,j) \in \mathcal{E} \right\}$$

If we interpret the node feature as a physical quantity, projection by the adjacency matrix can be seen as a "diffusion" process which replaces the quantity at each node by a weighted average of the quantity in its neighborhood.

Another fundamental matrix in the context of graph analysis is the **Laplacian matrix**:

$$\mathbf{L} = \mathbf{D} - \mathbf{A}$$

where the degree matrix is computed as $D_{ii} = \sum_j A_{ij}$ irrespective of whether the adjacency matrix is normalized or not. One step of diffusion by the Laplacian can be written as:



(a) *Initial graph* **(b)** *10 steps* **(c)** *20 steps* **(d)** *30 steps*

Figure F.12.4: *(a) A random graph with 15 nodes and a scalar feature on each node (denoted with variable colors). (b)-(d) The result after* 10*,* 20*, and* 30 *steps of diffusion with the Laplacian matrix. The features converge to a stable state.*

$$\mathbf{L}\mathbf{x} = \sum_{(i,j)\in\mathcal{E}} A_{ij}(x_i - x_j) \tag{E.12.1}$$

We can see from here that the Laplacian is intimately linked to the idea of a gradient over a graph, and its analysis is at the core of the field of **spectral graph theory**. As an example, in (E.12.1) $\mathbf{1}$ is always an eigenvector of the Laplacian associated to a zero eigenvalue (in particular, the smallest one). We show an example of diffusion with the Laplacian matrix in Figure F.12.4.

12.1.4 Manifold regularization

From (E.12.1) we can also derive a quadratic form built on the Laplacian:

$$\mathbf{x}^\top \mathbf{L}\mathbf{x} = \sum_{(i,j)\in\mathcal{E}} A_{ij}(x_i - x_j)^2 \tag{E.12.2}$$

Informally, this is a scalar value that measures how "smooth" the signal over the graph is, i.e., how quickly it changes for pairs of nodes that are connected in the graph. To see a simple application of this concept, consider a tabular classification dataset $\mathcal{S}_n = \{(\mathbf{x}_i, y_i)\}$. Suppose we build a graph over this dataset, where each node is an element of the dataset, and the adjacency matrix is built based on the distance between features:

$$A_{ij} = \begin{cases} \exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2) & \text{if } \|\mathbf{x}_i - \mathbf{x}_j\|^2 < \tau \\ 0 & \text{otherwise} \end{cases} \tag{E.12.3}$$

where τ is a user-defined hyper-parameter. Given a classification model $f(\mathbf{x})$, we may want to constrain its output to be similar for similar inputs, where similarity is defined proportionally to (E.12.3). To this end, we can define the features of the graph as the outputs of out model:

$$\mathbf{f} = \begin{bmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_n) \end{bmatrix} \sim (n)$$

The quadratic form (E.12.2) tells us exactly how much similar inputs vary in terms of their predictions:

$$\mathbf{f}^\top \mathbf{L}\mathbf{f} = \sum_{i,j} A_{ij}(f(\mathbf{x}_i) - f(\mathbf{x}_j))^2 \tag{E.12.4}$$

The optimal model can be found by a regularized optimization problem, where the regularizer is given by (E.12.4) :

$$f^*(\mathbf{x}) = \arg\min \left\{ \sum_{i=1}^{n} L(y_i, f(\mathbf{x})) + \lambda \ \mathbf{f}^\top \mathbf{L}\mathbf{f} \right\}$$

where L is a generic loss function and λ is a scalar hyper-parameter:

This is called **manifold regularization** [BNS06] and it can be used as a generic regularization tool to force the model to be smooth over a graph, where the adjacency is either given or is built by the user as in (E.12.3). This is especially helpful in a **semi-supervised** scenario where we have a small labeled dataset and a large unlabeled one from the same distribution, since the regularizer in (E.12.4) does not require labels [BNS06]. However, the prediction of the model depends only on a single element $\mathbf{x}_i$, and the graph is thrown away after training. In the next section, we will introduce more natural ways of embedding the connectivity inside the model itself.

12.2 Graph convolutional layers

12.2.1 Properties of a graph layer

In order to design models whose predictions are conditional on the connectivity, we can augment standard layers $f(\mathbf{X})$ with knowledge of the adjacency matrix, i.e., we consider layers of the form:

$$\mathbf{H} = f(\mathbf{X}, \mathbf{A})$$

where as before $\mathbf{X} \sim (n, c)$ (with n the number of nodes and c the features at each node) and $\mathbf{H} \sim (n, c')$, i.e., the operation does not change the connectivity of the graph, and it returns an updated embedding $\mathbf{H}_i \sim (c')$ for each node i in the graph. For what follows, $\mathbf{A}$ can be the adjacency or any matrix with the same sparsity pattern (a graph-shift matrix), including a weighted adjacency matrix, the Laplacian matrix, and so on.

Since permuting the nodes in a graph should have no impact on the final predictions, the layer should not depend on the specific ordering of the nodes, i.e., for any permutation matrix $\mathbf{P}$ the output of the layer should be **permutation equivariant**:

$$f(\mathbf{PX}, \mathbf{PAP}^\top) = \mathbf{P} \cdot f(\mathbf{X}, \mathbf{A})$$

We can define a notion of "locality" for a graph layer, similar to the image case. To this end, we first introduce the concept of a subgraph. Given a subset of nodes $\mathcal{T} \in \mathcal{V}$ from the full graph, we define the **subgraph** induced by $\mathcal{T}$ as:

$$\mathcal{G}_\mathcal{T} = (\mathbf{X}_\mathcal{T}, \mathbf{A}_\mathcal{T})$$

where $\mathbf{X}_\mathcal{T}$ is a $(|\mathcal{T}|, c)$ matrix collecting the features of the nodes in $\mathcal{T}$, and $\mathbf{A} \sim (|\mathcal{T}|, |\mathcal{T}|)$ is the corresponding block of the full adjacency matrix.

> **Definition D.12.1 (Graph locality)** *A graph layer $\mathbf{H} = f(\mathbf{X}, \mathbf{A})$ is **local** if for every node, $\mathbf{H}_i = f(\mathbf{X}_{\mathcal{N}(i)}, \mathbf{A}_{\mathcal{N}(i)})$, where $\mathcal{N}(i)$ is the 1-hop neighborhood of node i.*

This is similar to considering all pixels at distance 1 in the image case, except that (a) nodes in $\mathcal{N}(i)$ have no specific ordering in this case, and (b) the size of $\mathcal{N}(i)$ can vary a lot depending on i. Hence, we cannot define a convolution like we did in the image case, as its definition requires these two properties (think of the weight tensor in a convolutional layer).

For what follows, note that we can extend our definition of locality beyond 1-hop neighbors. For example, the 2-hop neighborhood $\mathcal{N}^2(i)$ is defined as all nodes at distance at most 2:

$$\mathcal{N}^2(i) = \bigcup_{j \in \mathcal{N}(i)} \mathcal{N}(j)$$

where $\cup$ is the set union operator. We can extend the definition of locality to take higher-order neighborhoods into consideration and design the equivalent of 3×3 filters, 5×5 filters, and so on.

12.2.2 The graph convolutional layer

In order to define a graph layer that mimicks the convolutional layer, we need it to be permutation equivariant (instead of translation equivariant) and local. The MHA layer is naturally permutation equivariant, but it is not local and it does not depend explicitly on the adjacency matrix $\mathbf{A}$. We will see possible extensions to this end in the next section. For now, let us focus on a simpler fully-connected layer:

$$f(\mathbf{X}, \_) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

where $\mathbf{W} \sim (c', c)$ and $\mathbf{b} \sim (c')$. This is also naturally permutation equivariant, but it does not depend on the connectivity of the graph, which is ignored. To build an appropriate differentiable layer, we can alternate the layer's operation with a diffusion step.

Definition D.12.2 (Graph convolution) *Given a graph represented by a node feature matrix $\mathbf{X} \sim (n, c)$ and a generic graph-shift matrix $\mathbf{A} \sim (n, n)$ (the adjacency, the Laplacian, ...), a **graph convolutional** (GC) layer is given by [KW17]:*

$$f(\mathbf{X}, \mathbf{A}) = \phi(\mathbf{A}(\mathbf{X}\mathbf{W} + \mathbf{b}))$$

where the trainable parameters are $\mathbf{W} \sim (c', c)$ and $\mathbf{b} \sim (c')$, with c' an hyperparameter. ϕ is a standard activation function, such as a ReLU.

Note the similarity with a standard convolutional layer: we are performing a "channel mixing" operation via the matrix $\mathbf{W}$, and a "node mixing" operation via the matrix $\mathbf{A}$, the difference being that the former is untrainable in this case (due to, once again, variable degrees between nodes and the need to make the layer permutation equivariant). See Figure F.12.5 for a visualization. The analogy can also be justified more formally by

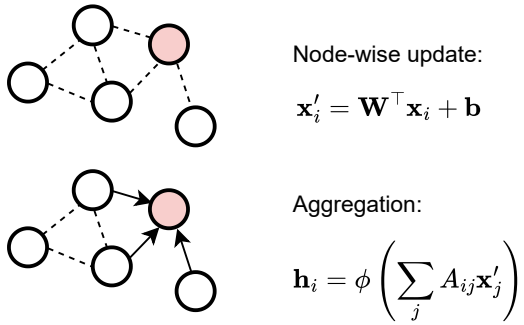


Figure F.12.5: *Two stages of a GC layer: each node updates its embedding in parallel to all other nodes; the output is given by a weighted average of all updated embeddings in the node's neighbourhood.*

leveraging concepts from graph signal processing, which is beyond the scope of this book [BBL+17]. Ignoring the bias, we can rewrite this for a single node i as:

$$\mathbf{H}_i = \phi\left(\sum_{j\in\mathcal{N}(i)} A_{ij}\mathbf{X}_j\mathbf{W}\right)$$

Hence, we first perform a simultaneous update of all node embeddings (given by the right multiplication by $\mathbf{W}$). Then, each node computes a weighted average of the updated node embeddings from itself and its neighbors. Since the number of neighbors can vary from node to node, working with the normalized variants of the adjacency matrix can help significantly in training. It is trivial to show permutation equivariance for the layer:

$$f(\mathbf{PX}, \mathbf{PAP}^\top) = \phi\left(\mathbf{PAP}^\top\mathbf{PXW}\right) = \mathbf{P}\cdot f(\mathbf{X},\mathbf{A})$$

12.2.3 Building a graph convolutional network

A single GC layer is local, but the stack of multiple layers is not. For example, consider a two-layered GC model:

$$f(\mathbf{X},\mathbf{A}) = \phi(\mathbf{A}\ \underbrace{\phi\left(\mathbf{AXW}_1\right)}_{\text{First GC layer}}\ \mathbf{W}_2) \tag{E.12.5}$$

with two trainable weight matrices $\mathbf{W}_1$ and $\mathbf{W}_2$. Similarly to the image case, we can define a notion of receptive field.

Definition D.12.3 (Graph receptive field) *Given a generic graph neural network* $\mathbf{H} = f(\mathbf{X}, \mathbf{A})$*, the* ***receptive field*** *of node* i *is the smallest set of nodes* $\mathcal{V}(i) \in \mathcal{V}$ *such that* $\mathbf{H}_i = f(\mathbf{X}_{\mathcal{V}(i)}, \mathbf{A}_{\mathcal{V}(i)})$.

For a single GC layer, the receptive field is $\mathcal{V}(i) = \mathcal{N}(i)$. For a two-layer network as in (E.12.5), we need to consider neighbors of neighbors, and the receptive field becomes $\mathcal{V}(i) = \mathcal{N}^2(i)$. In general, for a stack of k layers we will have a receptive field of $\mathcal{V}(i) = \mathcal{N}^k(i)$. The smallest number of steps which is needed to move from any two nodes in the graph is called the **diameter** of the graph. The diameter defines the smallest number of layers which is required to achieve a global receptive field for all the nodes.

Polynomial GC layers

Alternatively, we can increase the receptive field of *a single* GC layer. For example, if we remove the self-loops from the adjacency matrix, we can make the layer local with respect to $\mathcal{N}^2(i)$ instead of $\mathcal{N}(i)$ by also considering the square of the adjacency matrix:

$$\mathbf{H} = \phi\left(\mathbf{X}\mathbf{W}_0 + \mathbf{A}\mathbf{X}\mathbf{W}_1 + \mathbf{A}^2\mathbf{X}\mathbf{W}_2\right)$$

where we have three sets of parameters $\mathbf{W}_0$, $\mathbf{W}_1$, and $\mathbf{W}_2$ to handle self-loops, neighbors, and neighbors of neighbors respectively. This is called a **polynomial** GC layer. Larger receptive fields can be obtained with higher powers. More complex layers can be designed by considering ratios of polynomials [BGLA21].

We can combine GC layers with standard normalization layers, residual connections, dropout, or any other operation that is permutation equivariant. Differently from the image case, pooling is harder because there is no immediate way to subsample a graph connectivity. Pooling layers can still be defined by leveraging tools from graph theory or adding additional trainable components, but they are less common [GZBA22].

Denote by $\mathbf{H} = f(\mathbf{X}, \mathbf{A})$ a generic combination of layers providing an updated embedding for each node (without modifying the connectivity). In analogy with CNNs, we call it the **backbone** network. We can complete the design of a generic **graph convolutional network** (GCN) by adding a small head of top of these representations:

$$y = (g \circ f)(\mathbf{X}, \mathbf{A})$$

The design of the head depends on the task we are trying to solve. The most common

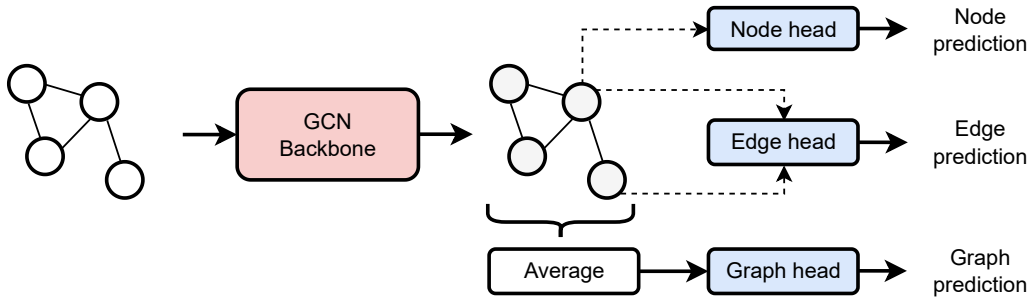


Figure F.12.6: *Different types of graph heads: (a) node tasks need to process the features of a single node; (b) edge tasks require heads that are conditioned on two nodes simultaneously; (c) graph tasks can be achieved by pooling all node representations into a fixed-dimensional vector.*

tasks fall into one of three basic categories: node-level tasks (e.g., node classification), edge-level task (e.g., edge classification), or graph-level tasks (e.g., graph classification). We briefly consider an example for each of them in turn, see Figure F.12.6.

Node classification

First, suppose the input graph describes some kind of social network, where each user is associated to a node. For a given subset of users, $\mathcal{T} \subseteq \mathcal{V}$, we know a label $y_i, i \in \mathcal{T}$ (e.g., whether the user if a real user, a bot, or another kind of automated profile). We are interested in predicting the label for all other nodes. In this case, we can obtain a node-wise prediction by processing each updated node embedding, e.g.:

$$\hat{y}_i = g(\mathbf{H}_i) = \text{softmax}(\text{MLP}(\mathbf{H}_i))$$

Running this operation over the entire matrix $\mathbf{H}$ gives us a prediction for all nodes, but we only know the true labels for a small subset. We can train the GCN by discarding the nodes outside of the training set:

$$\arg\min \frac{1}{|\mathcal{T}|} \sum_{i \in \mathcal{T}} \text{CE}(\hat{y}_i, y_i)$$

where CE is the cross-entropy loss. Importantly, even if we are discarding the output predictions for nodes outside our training set, their input features are still involved in the training process due to the diffusion steps inside the GCN. The rest of the nodes can

then be classified by running the GCN a final time after training. This scenario, where only a subset of the training data is labeled, is called a **semi-supervised** problem.

Edge classification

As a second example, suppose we have a label for a subset of *edges*, i.e., $\mathcal{T}_E \subseteq \mathcal{E}$. As an example, our graph could be a traffic network, of which we know the traffic flow only on a subset of roads. In this case, we can obtain an edge-wise prediction by adding an head that depends on the features of the two connected nodes, e.g., by concatenating them:

$$\hat{y}_{ij} = g(\mathbf{H}_i, \mathbf{H}_j) = \text{MLP}\left(\left[\mathbf{H}_i \parallel \mathbf{H}_j\right]\right)$$

For binary classification (e.g., predicting the affinity of two users with a scalar value between 0 and 1) we can simplify this by considering the dot product between the two features:

$$\hat{y}_{ij} = \sigma(\mathbf{H}_i^\top \mathbf{H}_j)$$

Like before, we can train the network by minimizing a loss over the known edges.

Graph classification

Finally, suppose we are interested in classifying (or regressing) the entire graph. As an example, the graph could be a molecule of which we want to predict some chemical property, such as reactivity against a given compound. We can achieve this by pooling the node representations (e.g., via a sum), and processing the resulting fixed-dimensional embedding:

$$y = \text{MLP}\left(\frac{1}{n}\sum_{i=1}^{n}\mathbf{H}_i\right)$$

The final pooling layer makes the network *invariant* to the permutation of the nodes. In this case, our dataset will be composed of multiple graphs (e.g., several molecules), making it similar to a standard image classification task. For node and edge tasks, instead, some datasets may be composed of a single graph (e.g., a large social network), while other datasets can have more than a single graph (e.g., several unconnected road

networks from different towns). This opens up the question of how to efficiently build mini-batches of graphs.

12.2.4 On the implementation of graph neural networks

As we mentioned, the peculiarity of working with graphs is that several matrices involved in our computations can be very sparse. For example, consider the following adjacency matrix:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

This corresponds to a three-node graph with a single bidirectional edge between nodes 1 and 3. We can store this more efficiently by only storing the indices of the non-zero values, e.g., in code:

```
A = [[0,2], [2,0]]
```

This is called a **coordinate list** format. For very sparse matrices, specialized formats like this one can reduce storage but also significantly improve the runtime of operating on sparse matrices or on combinations of sparse and dense matrices. As an example, pytorch-sparse[2] supports highly-efficient implementations of transposition and several types of matrix multiplications in PyTorch. This is also reflected on the layers' implementation. The forward pass of the layers in PyTorch Geometric[3] (one of the most common libraries for working with graph neural networks in PyTorch) is parameterized by providing as inputs the features of the graph and the connectivity as a list of edge coordinates.

Working with sparse matrices has another interesting consequence in terms of mini-batches. Suppose we have b graphs $(\mathbf{X}_i, \mathbf{A}_i)_{i=1}^{b}$. For each graph we have the same number of node features c but a different number of nodes n_i, so that $\mathbf{X}_i \sim (n_i, c)$ and

[2] https://github.com/rusty1s/pytorch_sparse

[3] https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html#learning-methods-on-graphs

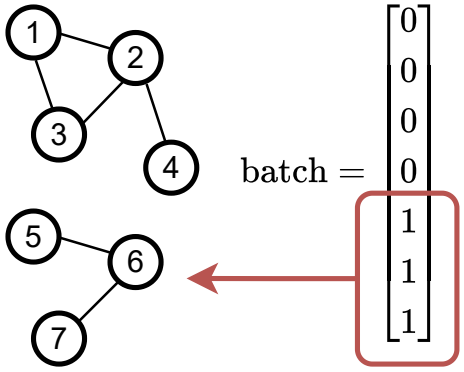


Figure F.12.7: *Two graphs in a mini-batch can be seen as a single graph with two disconnected components. In order to distinguish them, we need to introduce an additional vector containing the mapping between nodes and graph IDs.*

$\mathbf{A}_i \sim \text{Binary}(n_i, n_i)$. In order to build a mini-batch, we can create two rank-3 tensors:

$$X \sim (b, n, c) \tag{E.12.6}$$

$$A \sim \text{Binary}(b, n, n) \tag{E.12.7}$$

where $n = \max(n_1, \dots, n_b)$, and both matrices are padded with zeros to fill up the two tensors. However, a more elegant alternative can be obtained by noting that in a GC layer, two nodes that are not connected by any path (a sequence of edges) will never communicate. Hence, we can build a *single* graph describing the entire mini-batch by simply merging all the nodes:

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_b \end{bmatrix} \tag{E.12.8}$$

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & \dots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \dots & \mathbf{A}_b \end{bmatrix} \tag{E.12.9}$$

where $\mathbf{X} \sim (\sum_i n_i, c)$ and $\mathbf{A} \sim \text{Binary}(\sum_i n_i, \sum_i n_i)$. The adjacency matrix of the mini-batch has a block-diagonal structure, where all elements outside the diagonal blocks are zero (nodes from different graphs are not connected). While seemingly wasteful, this actually increases the sparsity ratio of the graph, making better use of the sparse matrix operations. Hence, for graph datasets in many cases there is no real difference between working with a single graph or a mini-batch of graphs.

In order to keep track which node belongs to each input graph, we can augment the representation with an additional vector $\mathbf{b} \sim (\sum_i n_i)$ such that b_i is an index in $[1, \dots, b]$

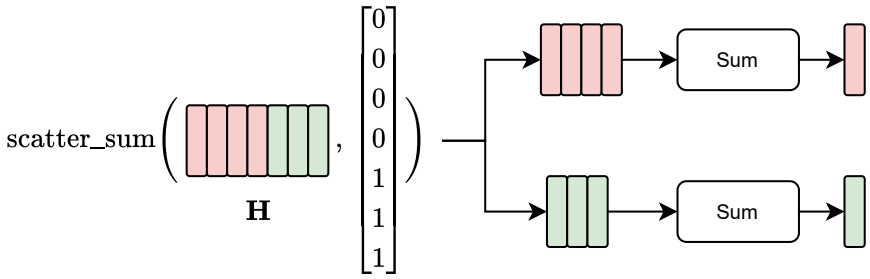


Figure F.12.8: *Example of scattered sum on the graph of Figure F.12.7. In this example nodes (1,2,3,4) belong to graph 1, and nodes (5,6,7) to graph 2. After pooling, we obtain a pooled representation for each of the two graphs.*

identifying one of the b input graphs - see Figure F.12.7. For graph classification, we can exploit $\mathbf{b}$ to perform pooling separately on groups of nodes corresponding to different graphs. Suppose $\mathbf{H} \sim (n, c')$ is the output of the GCN backbone, then:

$$\text{scatter\_sum}\,(\mathbf{H}, \mathbf{b}) = \mathbf{Y} \sim (b, c') \tag{E.12.10}$$

is called a **scattered sum** operation and is such that the $\mathbf{Y}_i$ is the sum of all rows of $\mathbf{H}$ such that $b_j = i$, as shown in Figure F.12.8. Similar operations can be defined for other types of pooling operations, such as averages and maximums.

As a separate problem, sometimes we may have a single graph that does not fit into memory: in this case, mini-batches should be formed by *sampling* subgraphs from the original graph [HYL17]. This is a relatively complex task that goes beyond the scope of this chapter.

12.3 Beyond graph convolutional layers

With the GC layer as template, we now overview a few extensions, either in terms of adaptivity or graph features that can be handled. We close by discussing **graph transformers**, a different family of layers in which the graph is embedded into a structural embedding which is summed to the node features.

12.3.1 Graph attention layers

One issue with GC layers is that the weights that are used to sum up contributions from the neighborhoods are fixed and are given by the adjacency matrix (or a proper normalized variant). Most of the time this is equivalent to assuming that, apart from the relative number of connections, all neighbors are equally important. A graph where nodes are connected mostly with similar nodes is called **homophilic**: empirically, homophily is a good predictor of the performance of graph convolutional layers [LLLG22]. Not all graphs are homophilic: for example, in a dating network, most people will be connected with people from the opposite sex. Hence, in these scenarios we need techniques that can properly adapt the weights given from each node to another node adaptively.

For sufficiently small graphs, we can let the non-zero elements of the weight matrix $\mathbf{A}$ adapt from their starting value through gradient descent. However, the number of trainable parameters in this case increases quadratically with the number of nodes, and this solution does not apply to a scenario with more than a single graph. If we assume that an edge depends only on the features of the two nodes it connects, we can generalize the GC layer with an attention-like operator:

$$\mathbf{h}_i = \phi\left(\sum_{j \in \mathcal{N}(i)} \text{softmax}(\alpha(\mathbf{x}_i, \mathbf{x}_j))\mathbf{W}^\top \mathbf{x}_j\right)$$

where α is some generic MLP block having two inputs and a scalar output, and the softmax is applied, for each node, to the set of outputs of α with respect to $\mathcal{N}(i)$, to normalize the weights irrespective of the size of the neighborhood. Due to the similarity to the attention layer, these are called **graph attention** (GAT) layers [VCC+18]. Seen from the perspective of the entire graph, this is very similar to a MHA layer, where the attention operation is restricted only on nodes having an edge that connects them.

The choice of α is relatively free. Instead of a dot product, the original GAT formulation considered an MLP applied on a concatenation of features:

$$\alpha(\mathbf{x}_i, \mathbf{x}_j) = \text{LeakyReLU}(\mathbf{a}^\top \left[\mathbf{V}\mathbf{x}_i \,\|\, \mathbf{V}\mathbf{x}_j\right])$$

where $\mathbf{V}$ and $\mathbf{a}$ are trainable. This was later found to be restrictive, in the sense that the ordering between elements does not depend on the central node [BAY22]. A less

restrictive variant, called **GATv2** [BAY22] is obtained as:

$$\alpha(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{a}^\top \text{LeakyReLU}(\mathbf{V}\left[\mathbf{x}_i \,\|\, \mathbf{x}_j\right])$$

Both GAT and GATv2 are very popular baselines nowadays.

12.3.2 Message-passing neural networks

Suppose we have available additional **edge features** $\mathbf{e}_{ij}$, e.g., in a molecular dataset we may know a one-hot encoded representation of the type of each molecular bond. We can generalize the GAT layer to include these features by properly modifying the attention function:

$$\alpha(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{a}^\top \text{LeakyReLU}(\mathbf{V}\left[\mathbf{x}_i \,\|\, \mathbf{x}_j \,\|\, \mathbf{e}_{ij}\right])$$

We can further generalize all the layers seen up to now (GC, GAT, GATv2, GAT with edge features) by abstracting away their basic components. Consider a very general layer formulation:

$$\mathbf{h}_i = \psi\left(\mathbf{x}_i, \text{Aggr}\left(\left\{M(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{ij})\right\}_{\mathcal{N}(i)}\right)\right) \tag{E.12.11}$$

where:

1. M builds a feature vector (which we call a **message**) relative to the edge between node i and node j. Contrary to GC and GAT layers, we are not restricting the message to be scalar-valued.

2. Aggr is a generic permutation invariant function (e.g., a sum) to aggregate the messages from all nodes connected to node i.

3. ψ is a final block that combines the aggregated message with the node features $\mathbf{x}_i$. In this way, two nodes with the same neighborhood can still be distinguished.

As an example, in a GC layer the message is built as $M(\_, \mathbf{x}_j, \_) = A_{ij}\mathbf{W}^\top \mathbf{x}_j$, the aggregation is a simple sum, and $\psi(\_, \mathbf{x}) = \phi(\mathbf{x})$. The general layer (E.12.11) was introduced in [GSR+17] with the name of **message-passing** layer, and it has become a very popular way to categorize (and generalize) layers operating on graphs [Vel22].

Let us consider a few examples of using this message-passing framework. First, we may want to give more highlight to the central node in the message-passing phase. We

can do this by modifying the ψ function:

$$\psi(\mathbf{x}, \mathbf{m}) = \phi(\mathbf{V}\mathbf{x} + \mathbf{m})$$

where $\mathbf{V}$ is a generically trainable matrix (this was introduced in [MRF+19] and popularized in PyTorch Geometric as the GraphConv[4] layer). Second, suppose nodes have available more complex features such as a time series per node (e.g., a distributed set of sensors). Note that in the message-passing framework, node-wise operations are decoupled from the way messages are aggregated and processed. Denoting by x_i the time-series at node i, we can generalize the GC layer by simply modifying the message function with a layer working on time series, e.g., a Conv1d layer:

$$h_i = \sum_{j \in \mathcal{N}(i)} A_{ij} \text{Conv1d}(x_i)$$

This is an example of a **spatio-temporal** GC layer [YYZ17]. Furthermore, up to now we have assumed that only node features should be updated. However, it is easy to also update edge features by an additional edge update layer:

$$\mathbf{e}_{ij} \leftarrow \text{MLP}(\mathbf{e}_{ij}, \mathbf{h}_i, \mathbf{h}_j)$$

This can also be seen as a message-passing iteration, in which the edge aggregates messages from its neighbors (the two connected nodes). This line of reasoning allows to further generalize these layers to consider more extended neighborhoods and graph features [BHB+18].

This is a very brief overview that provides a gist of many possible message-passing variants. There are many topics we are not able to cover in detail due to space: among these, we single out building MP layers for higher-order graphs (in which edges connect more that a pair of nodes) [CPPM22] and MP layers for point cloud data, in which we are interested in satisfying additional symmetries (rotational and translational symmetries) [SHW21, EHB23].

12.3.3 Graph transformers

We have seen two techniques to employ the graph structure: one is to add a regularization term that forces the network's outputs to be smooth relative to the graph; the

[4] https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.conv.GraphConv.html

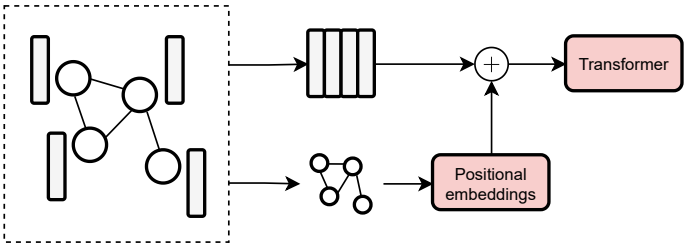


Figure F.12.9: *General idea of a graph transformer: the connectivity is embedded into a set of positional embeddings, which are added to the collected features. The result is then processed by a standard transformer network.*

second is to constrain the operations of the graph to follow the graph connectivity. In particular, in the GAT layer we have used a standard attention operation by properly masking the pairwise comparisons. However, we have also seen in the previous chapter that transformers have become popular because they provide an architecture that is completely agnostic from the type of data. Can we design the equivalent of a **graph transformer** [MGMR24]?

Recall that the two basic steps for building a transformer are tokenization of the input data and definition of the positional embedding. Tokenization for a graph is simple: for example, we can consider each node as a token, or (if edge features are given) each node and each edge as separate tokens after embedding them in a shared space. Let us ignore for now edge features. Consider the generic architecture taking as input the node features:

$$\mathbf{H} = \text{Transformer}(\mathbf{X})$$

This is permutation equivariant but completely agnostic to the connectivity. We can partially solve this by augmenting the node features with some graph-based features, such as the degree of the node, or the shortest path distance to some pre-selected nodes (anchors) [RGD+22, MGMR24]. More in general, however, we can consider an embedding of the graph connectivity into what we call a **structural embedding**:

$$\mathbf{H} = \text{Transformer}(\mathbf{X} + \text{Embedding}(\mathbf{A}))$$

Each row of Embedding($\mathbf{A}$) provides a vectorial embedding of the connectivity of the graph relative to a single node, ignoring all features (see Figure F.12.9). Luckily, embedding the structure of a graph into a vector space is a broad field. As an example, we describe here a common embedding procedure based on random walks [DLL+22]. Recall that the following matrix:

$$\mathbf{R} = \mathbf{A}\mathbf{D}^{-1}$$

can be interpreted as a "random walk", in which R_{ij} is the probability of moving from

node i to node j. We can iterate the random walk multiple times, for a fixed k set a priori by the user:

$$\mathbf{R}, \mathbf{R}^2, \ldots, \mathbf{R}^k$$

Random walk embeddings are built by collecting all the walk probabilities of a node returning on itself, and projecting them to a fixed-dimensional embedding:

$$\text{Embedding}(\mathbf{A}) = \begin{bmatrix} \text{diag}(\mathbf{R}) \\ \text{diag}(\mathbf{R}^2) \\ \vdots \\ \text{diag}(\mathbf{R}^k) \end{bmatrix} \mathbf{W}$$

Under specific conditions on the graph structure, this can be shown to provide a unique representation for each node [DLL+22]. Alternative types of embeddings can be obtained by considering eigen-decompositions of the Laplacian matrix [LRZ+23]. For a fuller exposition of graph transformers, we refer to [MGMR24]. Building graph transformers opens up the possibility of GPT-like foundation models for the graph domain, and also of adding graph-based data as an additional modality to existing language models [MCT+].

From theory to practice

Handling efficiently graph data requires extensions of the basic frameworks, due to the problems described in this chapter (e.g., sparsity). Common libraries include PyTorch Geometric for PyTorch, and Jraph for JAX. Both have ample sets of tutorials, for example for node classification in small citation networks.[5]

If you implemented a Vision Transformer in Chapter 11, I suggest a funny exercise which has (mostly) didactic value, as shown in Figure F.12.10. Suppose we tokenize the image into patches, but instead of adding positional embeddings, we construct an adjacency matrix $\mathbf{A} \sim (p, p)$ (where p is the number of patches) as:

$$A_{ij} = \begin{cases} 1 & \text{if the two patches share a border in the image} \\ 0 & \text{otherwise} \end{cases} \tag{E.12.12}$$

[5] Recommended example in PyTorch Geometric: https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html.

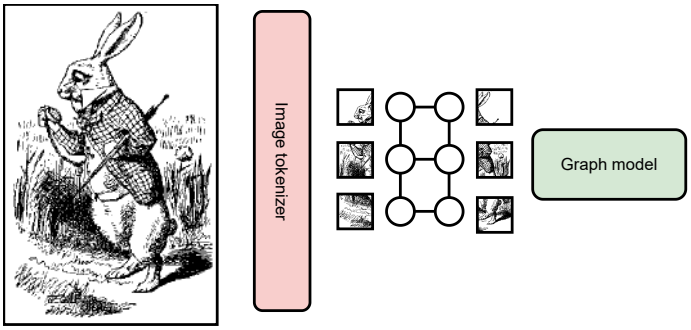


Figure F.12.10: *A GNN for computer vision: the image is tokenized into patches, an adjacency matrix is built over the patches, and the two are propagated through a graph model.*

We now have a graph classification dataset, where the node features are given by the patch embedding, and the adjacency matrix by (E.12.12). Thus, we can perform image classification by adapting the GNN from the previously-mentioned tutorials.

13 | Recurrent models

About this chapter

Transformer models are very effective at processing sequences, but they are hindered by their quadratic complexity in the sequence length. One possibility is to replace them with recurrent layers, having only constant-time for processing each element of a sequence, irrespective of its length. In this final chapter we provide an overview of several recurrent models and their characteristics. The field has been moving very rapidly in the last two years, and we provide a wide overview at the expense of precision – see [TCB+24] for a recent survey.

13.1 Linearized attention models

13.1.1 Replacing the dot product

To provide some intuition on why recurrent neural networks (RNNs) can be useful, we begin with a generalization of the attention layer (called the **linearized attention layer** [KVPF20]) that can be written in a recurrent form. We start by rewriting the SA layer in an abstract form with a generic scalar-valued attention function $\alpha(\cdot,\cdot)$ instead of the dot product:

$$\mathbf{h}_i = \frac{\sum_{j=1}^{n} \alpha\left(\mathbf{q}_i, \mathbf{k}_j\right)\mathbf{v}_j}{\sum_{j=1}^{n} \alpha\left(\mathbf{q}_i, \mathbf{k}_j\right)} \tag{E.13.1}$$

where for the standard SA, $\alpha(\mathbf{x},\mathbf{y}) = \exp(\mathbf{x}^\top\mathbf{y})$. If the elements of the sequence must be processed in order (as in autoregressive generation), (E.13.1) is inconvenient because its cost grows quadratically in the sequence length. Even if a KV cache is used, memory still grows linearly. By comparison, a convolutional layer has fixed time and memory

cost for each element to be processed, but information is lost if a token is outside the receptive field. What we would like, instead, is a mechanism to compress all the information of the sequence into a fixed-size input (which we will call a **memory** or **state** tensor), so that the cost of running the model on our current input token plus the memory is constant. We call models of this form **recurrent**.

To begin, note that any non-negative α is a valid similarity function. In machine learning, this requirement is equivalent to α being what is called a **kernel function** [HSS08]. Many such kernel functions can be written as a generalized dot product:

$$\alpha(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^\top \phi(\mathbf{y}) \tag{E.13.2}$$

for some function $\phi : \mathbb{R}^c \to \mathbb{R}^e$ that performs a feature expansion.

Kernel functions

As an example, the **polynomial kernel function** $\alpha(\mathbf{x}, \mathbf{y}) = (1 + \mathbf{x}^\top \mathbf{y})^d$ can be rewritten as (E.13.2) if $\phi(\bullet)$ explicitly computes all polynomials of its input up to order d [HSS08]. Some kernel functions correspond to infinite-dimensional expansions (e.g., the Gaussian kernel), in which case (2) can still be recovered in terms of an approximated kernel expansion, such as working with random Fourier features [SW17].

Based on (E.13.2) we can rewrite (E.13.1) as:

$$\mathbf{h}_i = \frac{\sum_{j=1}^n \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j) \mathbf{v}_j^\top}{\sum_{j=1}^n \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)}$$

where we have added a transpose operation on $\mathbf{v}_j$ to be consistent with the dimensions. Because $\phi(\mathbf{q}_i)$ does not depend on j we can bring it outside the sum to obtain:

$$\mathbf{h}_i = \frac{\phi(\mathbf{q}_i)^\top \sum_{j=1}^n \phi(\mathbf{k}_j) \mathbf{v}_j^\top}{\phi(\mathbf{q}_i)^\top \sum_{j=1}^n \phi(\mathbf{k}_j)} \tag{E.13.3}$$

This is called a **linearized attention** model [KVPF20]. Computing (E.13.3) for all tokens has complexity $\mathcal{O}(n(e^2 + ev))$, which is linear in the sequence length and advantageous whenever $n < e^2$. ϕ can be chosen freely, e.g., in [KVPF20] they consider a quadratic feature expansion or even a simpler $\phi(\mathbf{x}) = \text{ELU}(\mathbf{x}) + 1$ for short sequences.

13.1.2 A recurrent formulation

We now rewrite the linearized attention model in a recurrent form, by considering what happens for a causal variant of the layer. First, we modify (E.13.3) by constraining the sum only on past input elements to make it causal:

$$\mathbf{h}_i = \frac{\phi(\mathbf{q}_i)^\top \overbrace{\sum_{j=1}^{i} \phi(\mathbf{k}_j)\mathbf{v}_j^\top}^{\text{Attention memory } \mathbf{S}_i}}{\phi(\mathbf{q}_i)^\top \underbrace{\sum_{j=1}^{i} \phi(\mathbf{k}_j)}_{\text{Normalizer memory } \mathbf{z}_i}} \tag{E.13.4}$$

This is our first example of a **recurrent layer**. To understand the name, we note that the attention and normalizer memories can be written recursively as:

$$\mathbf{S}_i = \mathbf{S}_{i-1} + \phi(\mathbf{k}_i)\mathbf{v}_i^\top \tag{E.13.5}$$

$$\mathbf{z}_i = \mathbf{z}_{i-1} + \phi(\mathbf{k}_i) \tag{E.13.6}$$

where the base case of the recurrence is given by their initialization:

$$\mathbf{S}_0 = \mathbf{0} \tag{E.13.7}$$

$$\mathbf{z}_0 = \mathbf{0} \tag{E.13.8}$$

The output is then given by:

$$\mathbf{h}_i = \frac{\phi(\mathbf{q}_i)^\top \mathbf{S}_i}{\phi(\mathbf{q}_i)^\top \mathbf{z}_i} \tag{E.13.9}$$

Equations (E.13.5)-(E.13.9) are particularly interesting for an autoregressive scenario: for any new token to be generated, we update the two memory states (equations (E.13.5) and (E.13.6)), and we use these updated states to compute the output for the i-th element. Importantly, the total computation for generating a new token is constant, and the cost in memory is also fixed since the previous memories $\mathbf{S}_{i-1}$ and $\mathbf{z}_{i-1}$ can be discarded. We can alternate between the two formulations of the layer: we can use a vectorized variant for training (for efficient implementation on GPUs) and the recurrent formulation for inference.

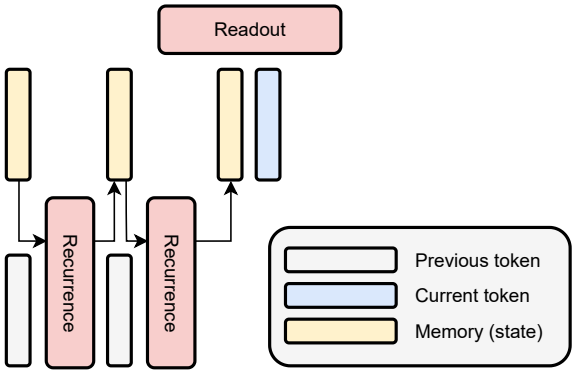


Figure F.13.1: *Overview of a recurrent layer: past tokens are shown in gray, current input token in blue, the memory state in yellow.*

13.2 Classical recurrent layers

13.2.1 General formulation

Let us now abstract away the key components of a recurrent layer, using the previous section as reference. First, we need a **state** of fixed size, which is used to compress all useful information up to the i-th element of the sequence. We denote it generically as $\mathbf{s}_i$, and without lack of generality we assume it is a single vector from now on. Second, we need a **transition function** (recurrence) that updates the state vector based on the previous value and the value of the current token, which we denote as $f(\mathbf{s}_{i-1}, \mathbf{x}_i)$. Third, we need what we call a **readout function** that provides an output for the i-th element of the sequence. We denote it as $g(\mathbf{s}_i, \mathbf{x}_i)$. See also Figure F.13.1 for a visualization.

Definition D.13.1 (Recurrent layer) *Given a sequence of tokens $\mathbf{x}_1, \mathbf{x}_2, \ldots$, a generic recurrent layer can be written as:*

$$\mathbf{s}_i = f(\mathbf{s}_{i-1}, \mathbf{x}_i) \tag{E.13.10}$$

$$\mathbf{h}_i = g(\mathbf{s}_i, \mathbf{x}_i) \tag{E.13.11}$$

*where the **state vector** $\mathbf{s}_i \sim (e)$ is initialized as zero by convention, $\mathbf{s}_0 = \mathbf{0}$. The size of the state vector, e, and the size of the output vector $\mathbf{h}_i \sim (o)$ are hyper-parameters. We call f the **state transition** function and g the **readout** function.*

In this format, a recurrent layer represents a discrete-time, input-driven dynamical system, and it is a causal layer by definition. In control engineering, this is also known as a **state-space model**. For tasks in which causality is unnecessary, **bidirectional**

layers [SP97] can also be defined. In a bidirectional layer we initialize two recurrent layers (with separate parameters), one of which processes the sequence left-to-right, and the second one right-to-left. Their output states are then concatenated to provide the final output.

Recurrent neural networks (RNNs) can be built by stacking multiple recurrent layers on the updated sequence $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_n$ [PGCB14]. Interestingly, a recurrent layer has no requirement on the length of the sequence, which can (in principle) be unbounded. For this reason, RNNs with unbounded precision or growing architectures can be shown to be Turing-complete [CS21].

Implicit layers

What happens if we apply a recurrent layers to a *single* token $\mathbf{x}$?

$$\mathbf{s}_i = f(\mathbf{s}_{i-1}, \mathbf{x}) \tag{E.13.12}$$

If we run the state transition several time starting from a known initialization $\mathbf{s}_0$, this is similar to a model with several layers (one per transition) sharing the same parameters. Suppose we run (E.13.12) an *infinite* number of times. If the dynamic system has a stable attractor, the output will be defined by the fixed-point equation:

$$\mathbf{s} = f(\mathbf{s}, \mathbf{x}) \tag{E.13.13}$$

If we take (E.13.13) as the definition of a layer, we obtain what is called an **implicit layer** [BKK19]. The implementation of implicit layers can be made feasible by using fast solvers for the fixed-point equation and computing the backward pass with the use of the **implicit function theorem** [BKK19]. Implicit graph layers can also be defined by running each diffusion operation to a stable state [GMS05, SGT+08].

13.2.2 "Vanilla" recurrent layers

Historically, recurrent layers were instantiated by considering two fully-connected layers as transition and readout functions:

$$f(\mathbf{s}_{i-1}, \mathbf{x}_i) = \phi(\mathbf{A}\mathbf{s}_{i-1} + \mathbf{B}\mathbf{x}_i) \tag{E.13.14}$$

$$g(\mathbf{s}_i, \mathbf{x}_i) = \mathbf{C}\mathbf{s}_i + \mathbf{D}\mathbf{x}_i \tag{E.13.15}$$

```python
# Input tensor
x = torch.randn(batch_size, sequence_length, features)

# State tensor
s = torch.zeros(batch_size, state_size)

# State update
state_update = nn.RNNCell(features, state_size)
for i in range(x.shape[1]):
  s = state_update(x[:, i, :], s)
```

Box C.13.1: *Vanilla recurrence in PyTorch. It is impossible to parallelize the for-loop with linear algebra because of the dependencies in the recurrence. In PyTorch, the state update is called a* ***recurrent cell****, while the recurrent layers, such as* `torch.nn.RNN`*, wrap a cell and perform the complete for-loop.*

where as always we ignore biases for simplicity, and we have four trainable matrices $\mathbf{A} \sim (e, e)$, $\mathbf{B} \sim (e, c)$, $\mathbf{C} \sim (o, e)$, and $\mathbf{D} \sim (o, c)$, where c is the input dimensionality (the size of each token). A layer in this form is sometimes referred to generically as a "*recurrent layer*", a "*vanilla recurrent layer*", or an **Elman** recurrent layer. When the two matrices **A** and **B** are left untrained and we only have a single layer, these models are called **echo state networks** (ESNs) or **reservoir computers** [LJ09]. ESNs can be a powerful baseline for time series forecasting, especially when the untrained matrices (the reservoir) are initialized in a proper way [GBGB21].

Despite their historical significance, layers of this form are extremely inefficient (and hard) to train. To see this, note that by its design the computation across elements of the sequence cannot be parallelized efficiently, as shown in Box C.13.1. Hence, we need to resort to iterative (for-loops) implementations, and even highly customized CUDA implementations[1] are slower than most alternative sequence layers.

Another issue stems from the gradients involved in the layer's computations. Consider a simplified case having only the transition function. We can unroll the full computation

[1] https://docs.nvidia.com/deeplearning/performance/dl-performance-recurrent/index.html

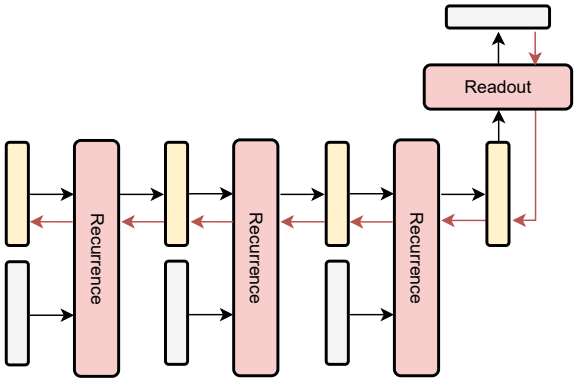


Figure F.13.2: *Backward pass for a recurrent layer: the adjoint values have to be propagated through all the transition steps. Each state then contributes a single term to the full gradient of the parameters.*

as:

$$\begin{aligned}\mathbf{s}_1 &= f(\mathbf{s}_0, \mathbf{x}_1)\\ \mathbf{s}_2 &= f(\mathbf{s}_1, \mathbf{x}_2)\\ &\vdots\\ \mathbf{s}_n &= f(\mathbf{s}_{n-1}, \mathbf{x}_n)\end{aligned}$$

This is similar to a model with n layers, except that the parameters are shared (the same) across the layers. Below we focus on the quantity $\partial_{\mathbf{A}}\mathbf{s}_n$ (the weight Jacobian with respect to $\mathbf{A}$), but similar considerations apply to all gradients. Let us define the following cumulative product:

$$\widetilde{\mathbf{s}}_i = \prod_{j=i+1}^{n} \partial_{\mathbf{s}_{j-1}} f(\mathbf{s}_{j-1}, \mathbf{x}_j) \tag{E.13.16}$$

This represents the gradient of the transition function from the end of the sequence backwards to element i, as shown in Figure F.13.2. Because of weight sharing, the gradient we are looking for has a separate term for each element in the sequence which involves these cumulative products:

$$\partial_{\mathbf{A}}\mathbf{s}_n = \partial_{\mathbf{A}} f(\mathbf{s}_{n-1}, \mathbf{x}_n) + \sum_{i=1}^{n-1} \widetilde{\mathbf{s}}_i \left[\partial_{\mathbf{A}} f(\mathbf{s}_{i-1}, \mathbf{x}_i)\right] \tag{E.13.17}$$

Gradient from element n — Gradient from element i

The first term corresponds to a "standard" weight Jacobian, describing the influence of $\mathbf{A}$ on the last element of the sequence. The terms in the summation are the additional

contributions, one for each element of the sequence, which are weighted by the chained input Jacobian computed over the sequence itself.

Written in this form, reverse mode automatic differentiation is also called **backpropagation through time** (BPTT), and it can be a strong source of instability or gradient problems during gradient descent. To see this, note that each input Jacobian in the inner product in (E.13.17) involves a multiplication by the derivative of the activation function ϕ. Some of the earliest analyses of vanishing and exploding gradients were done in this context [Hoc98]. For long sequences, stability of the layer is guaranteed only when the eigenvalues of the transition matrix are properly constrained [GM17]. Layer normalization was also originally developed to stabilize training in RNNs, by computing statistics over the states' sequence [BKH16].

Several techniques have been developed to partially solve these instabilities in the context of recurrent layers. For example, the sum in (E.13.17) can be truncated to a given interval (**truncated BPTT**), or the gradients can be thresholded if they exceed a predefined upper bound (**clipped gradients**).

13.2.3 Gated recurrent networks

Over the years, several variants of the vanilla layer were proposed to improve its performance. In this section we focus on a popular class of such models, called **gated** RNNs. One issue of RNNs is that the entire state gets overwritten at each transition, which is reflected in the partial products in (E.13.17). However, we can assume that, for many sequences, only a few elements of these transitions are important: as an example, in an audio signal, empty regions or regions with no information are typical. In these cases, we may be interested in *sparsifying* the transition (similarly to how most attention weights tend to be close to zero) and, consequently, setting most elements in $\widetilde{\mathbf{s}}_i$ to 1. This can be achieved with the addition of specialized gating layers.

We consider the simplest form of gated RNN, called **light gated recurrent unit** (LiGRU, [RBOB18]), having a single gate. For our purposes, a gating function is simply a layer that outputs values in the range $[0, 1]$ that can be used to "mask" the input. As an example, a gate over the state can be obtained by a fully-connected layer with a sigmoid activation function:

$$\gamma(\mathbf{s}_{i-1}, \mathbf{x}_i) = \sigma\left(\mathbf{V}\mathbf{s}_{i-1} + \mathbf{U}\mathbf{x}_i\right)$$

where $\mathbf{V}$ and $\mathbf{U}$ have similar shapes to $\mathbf{A}$ and $\mathbf{B}$. We can interpret this as follows: if

$\gamma_i \approx 0$, the i-th feature of the state should be kept untouched, while if $\gamma_1 \approx 1$, we should propagate its updated value as output. Hence, we can rewrite the transition function by properly masking the new and old values as:

$$f(\mathbf{s}_{i-1}, \mathbf{x}_i) = \overbrace{\gamma(\mathbf{s}_{i-1}, \mathbf{x}_i) \odot (\mathbf{A}\mathbf{s}_{i-1} + \mathbf{B}\mathbf{x}_i)}^{\text{New values}} + \underbrace{(1 - \gamma(\mathbf{s}_{i-1}, \mathbf{x}_i)) \odot \mathbf{s}_{i-1}}_{\text{Old values}}$$

This can be seen as a soft (differentiable) approximation to a "real" gate having only binary values, or as a convex combination of the original layer and a skip connection. We can theoretically control the goodness of this approximation by adding an additional regularizer to the loss that constrains the outputs of the gate to lie as close as possible to 0 or 1.

Other gated recurrent layers can be obtained by adding additional gates to this design: the original **gated recurrent unit** (GRU) adds a so-called "reset gate" to the layer [CVMG+14], while **long-short term memory** units (LSTMs) have a third "forget gate" [HS97]. LSTMs were the first gated variant to be introduced in the literature, and for a long time they have been the most successful deep architecture for processing sequences [Sch15]. Because of this, research on LSTM models is still very active [BPS+24].

13.3 Structured state space models

13.3.1 Linear recurrent layers

We now consider a simplified class of recurrent layers, in which we remove the intermediate nonlinearity in the transition function:

$$f(\mathbf{s}_{i-1}, \mathbf{x}_i) = \mathbf{A}\mathbf{s}_{i-1} + \mathbf{B}\mathbf{x}_i \tag{E.13.18}$$

$$g(\mathbf{s}_i, \mathbf{x}_i) = \mathbf{C}\mathbf{s}_i + \mathbf{D}\mathbf{x}_i \tag{E.13.19}$$

Written in this form, (E.13.18)-(E.13.19) are called **state space models** (SSM).[2] Intuitively, an SSM layer is "less expressive" than a standard recurrent layer (because of the

[2] Confusingly, any recurrent layer in the form (E.13.10)-(E.13.11) is an SSM, but in the neural network's literature the term SSM has come to be associated only with the linear variant. Sometimes we refer to them as **structured** SSMs because, as we will see, we need to properly constrain the transition matrix to make them effective.

lack of non-linearities). However, this can be recovered by adding activation functions after the output, or by interleaving these layers with token-wise MLPs [ODG+23].

Interest in this class of models (re)-started in 2020, when [GDE+20] analyzed a theoretical construction for the matrix $\mathbf{A}$ in (E.13.18) that could efficiently compress one-dimensional input sequences according to some pre-defined reconstruction criterion. The result was called the HiPPO (**High-Order Polynomial Projection Operator**) matrix. A family of neural networks built by a stack of SSM layers based on the HiPPO theory soon followed, leading to the **Structured State Space for Sequence Modeling** (S4) layer in 2021 [GGR22] and the simplified S4 model (S5) in 2022 [SWL23].

Because of their roots in HiPPO theory, the proposed SSM layers up to S4 considered a stack of 1D models, one for each channel of the input, with transition matrices initialized as HiPPO matrices. By contrast, S5 introduced a standard multi-input, multi-output model of the form in (E.13.18)-(E.13.19), which is the one we describe here. In particular, we focus our analysis on a simplified variant known as the **linear recurrent unit** (LRU) [OSG+23].

This formulation has a number of interesting properties, mostly stemming from the associativity of the linear transition function. To see this, we start by noting that the recurrence has a closed form solution:

$$\mathbf{s}_i = \sum_{j=1}^{i} \mathbf{A}^{i-j}\mathbf{B}\mathbf{x}_j \tag{E.13.20}$$

We can view this summation from two different points of view. First, we can aggregate all coefficients with respect to the input sequence into a rank-3 tensor:

$$K = \text{stack}\left(\mathbf{A}^{n-1}\mathbf{B}, \mathbf{A}^{n-2}\mathbf{B}, \ldots, \mathbf{A}\mathbf{B}, \mathbf{B}\right)$$

We can compute all outputs via a single 1D convolution of filter size equal to the length of the sequence (a *long* convolution) between the input sequence stacked into a single matrix $\mathbf{X} \sim (n, c)$ and the pre-computed kernel K:

$$\mathbf{S} = \text{Conv1D}(\mathbf{X}, K)$$

Hence, the SSM layer can be interpreted as a convolution [GJG+21]. If the transition matrix is applied on a single channel, this can be exploited to speed-up computations

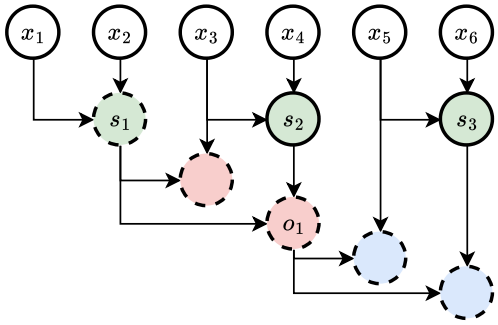


Figure F.13.3: *Parallel scan on a sequence of six elements: circles of the same color can be computed in parallel; dashed circles are the outputs of the parallel scan.*

by operating in a frequency domain, e.g., in the FlashConv implementation.[3] However, a more efficient solution can be found by exploiting a family of algorithms known as **associative (parallel) scans** (or **all-prefix-sums**).

13.3.2 An interlude: associative scans

We introduce parallel scans in their general formulation before seeing their application to linear SSMs. Consider a sequence of elements $(x_1, x_2, \ldots, x_n)$, and an operation $\star$ which is assumed binary (it acts on any two elements of the sequence) and associative. We want to compute all partial applications of this operator to the sequence (using separate colors for readability):

$$x_1, \quad x_1 \star x_2, \quad x_1 \star x_2 \star x_3, \quad \ldots, \quad x_1 \star x_2 \star \cdots \star x_n$$

This can be done trivially by an iterative algorithm which computes the elements one-by-one, adding one element at every iteration (this corresponds to how a standard recurrent layer would be computed). However, we can devise an efficient *parallel* algorithm by exploiting the associativity of the operator $\star$ [Ble90]. The key intuition is that multiple pairs of elements can be computed in parallel and then aggregated recursively.

As a simple example, consider a sequence of 6 elements $x_1, x_2, x_3, x_4, x_5, x_6$ (an in-depth example applied to SSMs can be found in [SWL23]). We will denote by $\hat{x}_i$ the i-th prefix we want to compute. The overall procedure is shown schematically in Figure

[3] https://www.together.ai/blog/h3

F13.3. We first aggregate pairs of adjacent values as:

$$s_1 = x_1 \star x_2 \rightarrow \hat{x}_2$$
$$s_2 = x_3 \star x_4$$
$$s_3 = x_5 \star x_6$$

where we use arrows to denote output values of the algorithm. We now perform a second level of aggregations:

$$s_1 \star x_3 \rightarrow \hat{x}_3$$
$$o_1 = s_1 \star s_2 \rightarrow \hat{x}_4$$

And finally:

$$o_1 \star x_5 \rightarrow \hat{x}_5$$
$$o_1 \star s_3 \rightarrow \hat{x}_6$$

While this looks strange (we made 7 steps instead of 5), the three blocks of computations can be trivially parallelized if we have access to 3 separate threads. In general, by organizing the set of computations in a balanced fashion, we are able to compute the parallel scan in $\mathcal{O}(T \log n)$, where T is the cost of the binary operator $\star$. An example of implementation is the associative scan function in JAX.[4]

It is easy to show that the transition function in a linear SSM is an example of an all-prefix-sums problem. We define the elements of our sequence as pairs $x_i = (\mathbf{A}, \mathbf{B}\mathbf{x}_i)$, and the binary operator as:

$$(\mathbf{Z}, \mathbf{z}) \star (\mathbf{V}, \mathbf{v}) = (\mathbf{V}\mathbf{Z}, \mathbf{V}\mathbf{z} + \mathbf{v})$$

The prefixes of $\star$ are then given by [SWL23]:

$$x_1 \star x_2 \star \ldots \star x_i = (\mathbf{A}^i, \mathbf{s}_i)$$

Hence, running a parallel scan gives us the powers of $\mathbf{A}$ as the first elements of the output, and all the states of the layer as the second element of the output. The complexity of this operation is upper bounded by the complexity of $\mathbf{A}^{i-1}\mathbf{A}$, which scales as $\mathcal{O}(n^3)$. To make the entire procedure viable, we can constrain $\mathbf{A}$ so that its powers can be computed more efficiently. This is the topic of the next section.

[4] https://jax.readthedocs.io/en/latest/_autosummary/jax.lax.associative_scan.html

13.3.3 Diagonal SSMs

A common strategy to make the previous ideas feasible is to work with diagonal transition matrices (or diagonal matrices plus a low-rank term [GGR22]). In this case, powers of $\mathbf{A}$ can be computed easily by taking powers of the diagonal entries in linear time. In addition, as we will see, working with diagonal matrices allows us to control the dynamics of the transition function to avoid numerical instabilities.

In particular, a square matrix $\mathbf{A}$ is said to be **diagonalizable** if we can find another square (invertible) matrix $\mathbf{P}$ and a diagonal matrix $\mathbf{\Lambda}$ such that:

$$\mathbf{A} = \mathbf{P}\mathbf{\Lambda}\mathbf{P}^{-1} \tag{E.13.21}$$

Diagonalizable matrices are (in a sense) "simpler" that generic matrices, For example, if such a decomposition exists, it is easy to show that powers can also be computed efficiently as:

$$\mathbf{A}^i = \mathbf{P}\mathbf{\Lambda}^i\mathbf{P}^{-1}$$

Suppose that the transition matrix is diagonalizable. Then, we can re-write the SSM in an equivalent form having a diagonal transition matrix. We begin by substituting (E.13.21) into the definition of the SSM and multiplying on both sides by $\mathbf{P}^{-1}$:

$$\underbrace{\mathbf{P}^{-1}\mathbf{s}_i}_{\text{New state vector } \bar{\mathbf{s}}_i} = \sum_{j=1}^{i} \mathbf{\Lambda}^{i-j} \underbrace{\mathbf{P}\mathbf{B}}_{\text{New input-state matrix } \bar{\mathbf{B}}} \mathbf{x}_j$$

We now rewrite the readout function in terms of the new variable $\bar{\mathbf{s}}$:

$$\mathbf{y}_i = \underbrace{\mathbf{C}\mathbf{P}}_{\text{New readout matrix } \bar{\mathbf{C}}} \bar{\mathbf{s}}_i + \mathbf{D}\mathbf{x}_i$$

Putting everything together:

$$\bar{\mathbf{s}}_i = \mathbf{\Lambda}\bar{\mathbf{s}}_{i-1} + \bar{\mathbf{B}}\mathbf{x}_i \tag{E.13.22}$$

$$\mathbf{y}_i = \bar{\mathbf{C}}\bar{\mathbf{s}}_i + \mathbf{D}\mathbf{x}_i \tag{E.13.23}$$

Hence, whenever a diagonalization of $\mathbf{A}$ exists, we can always rewrite the SSM into an equivalent form having a diagonal transition matrix. In this case, we can directly train the four matrices $\mathbf{\Lambda} = \text{diag}(\lambda), \lambda \sim (e)$, $\bar{\mathbf{B}} \sim (e, c)$, $\bar{\mathbf{C}} \sim (o, e)$ and $\mathbf{D} \sim (o, c)$, with the diagonal matrix being parameterized by a single vector of dimension e.

Not all matrices can be diagonalized. However, an approximate diagonalization can always be found if one allows for matrices $\mathbf{P}$ and $\mathbf{\Lambda}$ to have complex-valued entries [OSG+23]. Care must be taken to parameterize the values over the diagonal so that the eigenvalues of the transition matrix stay < 1 in absolute value, to avoid diverging dynamics. We refer to [OSG+23] for a description of both points and for a complete analysis of the resulting LRU layer.

13.4 Additional variants

Balancing the different strengths of convolutions, recurrence, and attention is an active research topic. To close the book, we list some recurrent layers (or layers that can be interpreted as recurrent) that have been introduced very recently in the literature.

13.4.1 Attention-free transformers

One issue of the linearized transformer model (Section 13.1.1) is the quadratic complexity in the feature dimension e. The attention-free transformer (ATF) was introduced as a variant of the basic attention layer that is instead linear in both sequence length and in the number of features [ZTS+21].

The core idea is to replace the dot product interactions between keys, query, and values with a simpler *multiplicative interaction* (element-wise):

$$\mathbf{h}_i = \sigma(\mathbf{q}_i) \odot \frac{\sum_j \exp\left(\mathbf{k}_j\right) \odot \mathbf{v}_j}{\sum_j \exp\left(\mathbf{k}_j\right)} \tag{E.13.24}$$

This is similar to the self-attention layer, except that we replace all dot products with element-wise (Hadamard) multiplications. It is also inspired by the linearized attention layer in that the query is only used as a global modulation factor, in this case after normalizing it with a sigmoid operation. In fact, we can recover a standard attention formulation by rewriting (E.13.24) for a single dimension z (exploiting the fact that we only perform element-wise operations):

$$h_{iz} = \frac{\sigma(q_{iz}) \sum_j \exp(k_{jz})}{\sum_j \exp(k_{jz})} v_{jz}$$

Hence, the ATF layer can be re-interpreted as a channel-wise variant of attention, in the sense that for every channel we can rewrite it as an attention operation over the

elements of the sequence. To increase flexibility, [ZTS+21] also considered adding relative embeddings $\mathbf{W} \sim (m, m)$ (where m is the maximum allowed length of the sequences):

$$\mathbf{h}_i = \sigma(\mathbf{q}_i) \odot \frac{\sum_j \exp\left(\mathbf{k}_j + W_{ij}\right) \odot \mathbf{v}_j}{\sum_j \exp\left(\mathbf{k}_j + W_{ij}\right)} \tag{E.13.25}$$

The relative embeddings can also be trained via a low-rank factorization to reduce the number of parameters. See [ZTS+21] for this and for additional variants of the basic ATF layer (e.g., hybridizing it with convolutional operations). We can also convert (E.13.24) to a causal (recurrent) variant by properly restricting the summation.

13.4.2 The Receptance Weighted Key Value (RWKV) model

The RWKV model [PAA+23] extends the ATF layer by incorporating a few additional architectural modifications. At the time of writing, this is one of the only pre-trained RNNs matching transformers at the largest scale, so we describe it in more detail. First, the relative embeddings are simplified by considering a single vector $\mathbf{w} \sim (e)$ which is scaled for each offset:

$$w_{ij} = -(i - j)\mathbf{w}$$

In addition, experiments showed that having a separate offset $\mathbf{u}$ (in place of $\mathbf{w}$) for the current element is beneficial. Written in causal form, this gives:

$$\mathbf{h}_i = \mathbf{W}_o \left(\sigma(\mathbf{q}_i) \odot \frac{\sum_{j=1}^{i-1} \exp\left(\mathbf{k}_j + w_{ij}\right) \odot \mathbf{v}_j + \exp\left(\mathbf{k}_i + \mathbf{u}\right) \odot \mathbf{v}_i}{\sum_{j=1}^{i-1} \exp\left(\mathbf{k}_j + w_{ij}\right) + \exp\left(\mathbf{k}_i + \mathbf{u}\right)} \right) \tag{E.13.26}$$

where we highlight the differences from the basic ATF layer in red. The query is called the **receptance** in [PAA+23], and an additional output projection $\mathbf{W}_o$ is added at the end. Second, the RWKV model modifies the standard MLP in the transformer block with a differently *gated* token-wise block. For a given input token $\mathbf{x}$ this can be written as:

$$\mathbf{y} = \sigma(\mathbf{W}_1\mathbf{x}) \odot \mathbf{W}_2 \max(0, \mathbf{W}_3\mathbf{x})^2 \tag{E.13.27}$$

where $\mathbf{W}_1$, $\mathbf{W}_2$, and $\mathbf{W}_3$ are trainable parameters. This is a standard MLP except for the left-most gate and the use of the squared ReLU. As a final modification, all three projections in the first block (and also the two appearances of $\mathbf{x}$ in (E.13.26)) are

replaced with convex combinations of $\mathbf{x}_i$ and $\mathbf{x}_{i-1}$ to improve performance, which is called *token shift*.

13.4.3 Selective state space models

We have seen three classes of recurrent models: standard recurrent layers (and their gated versions), linearized attention layers, and structured state space models. Although they look different, it is relatively easy to move from one class of models to the other. To see this, let us consider a linearized attention layer where we ignore the denominator:

$$\mathbf{S}_i = \mathbf{S}_{i-1} + \phi(\mathbf{k}_i)\mathbf{v}_i^\top \tag{E.13.28}$$

$$\mathbf{h}_i = \phi(\mathbf{q}_i)^\top \mathbf{S}_i \tag{E.13.29}$$

Apart from the matrix-valued state, we see this has the form of a SSM layer, except that some matrices (e.g., $\mathbf{C} = \phi(\mathbf{q}_i)^\top$) are not fixed but they depend on the specific input token. From the point of view of dynamic systems, we say that standard SSMs describe *time-invariant* systems, while (E.13.28)-(E.13.29) describe a *time-varying* system. This has inspired another class of SSM layers whose matrices are not constrained to be time-invariant, which have been called **selective** SSMs. Most of these models leverage the idea of attention layers of projecting the input multiple times before the layer's computations.

As an example, we focus here on the so-called Mamba layer [GD23] which, at the time of writing, is one of the few SSM layers that was scaled to match the performance of transformer models at very large contexts and parameters' counts. First, in order to make the SSM layer time-varying, a subset of its matrices are made input-dependent:[5]

$$\mathbf{s}_i = A(\mathbf{x}_i)\mathbf{s}_{i-1} + B(\mathbf{x}_i)\mathbf{x}_i \tag{E.13.30}$$

$$\mathbf{h}_i = C(\mathbf{x}_i)\mathbf{s}_i + \mathbf{D}\mathbf{x}_i \tag{E.13.31}$$

where $A(\bullet)$, $B(\bullet)$, and $C(\bullet)$ are linear projections of their input tokens. To make this feasible, the layer is applied to each channel of the input independently, and the transition matrix is selected as diagonal, so that all matrices of the SSM can be represented with a single vector of values. This layer looses a simple parallel scan implementation

[5] The matrix $\mathbf{D}$ can be seen as a simple residual connection and it is left untouched. The original layer has a slightly different parameterization where $\mathbf{A} = \exp(\Delta\bar{\mathbf{A}})$, for some trainable $\bar{\mathbf{A}}$ and input-dependent scalar value Δ. This does not change our discussion.

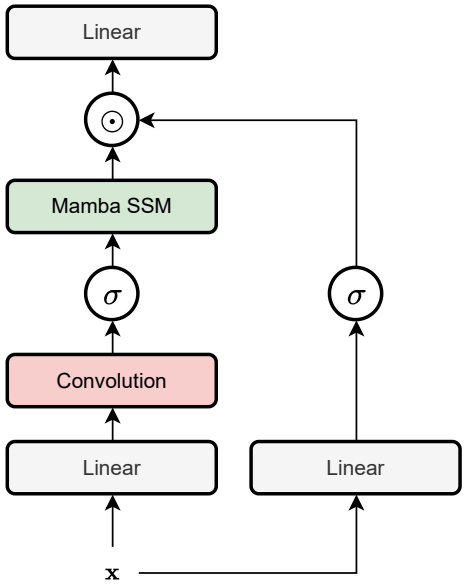


Figure F.13.4: *Mamba block (residual connections around the block and normalization are not shown). σ is the sigmoid function. Adapted from [GD23].*

and requires a customized hardware-aware implementation [GD23]. It can be shown that the Mamba SSM variant and several other SSM layers are degenerate case of a gated recurrent layer [GJG+21, GD23].

To make the overall architecture simpler, Mamba avoids alternating MLPs and SSMs, in favour of a gated archicture (similar to the gated attention unit from Section 11.3) where an MLP is used to weight the outputs from the SSM. An additional depthwise convolution is added for improved flexibility - see Figure F.13.4.

Goodbye (for now)

And so, Alice's first trip in this differentiable wonderland has come (for now) to an end. We only made a very broad tour, with a focus on the many ways layers can be designed and composed to create modern differentiable models (a.k.a., neural networks).

There are many topics we discussed only briefly, including how we can *use* these models in practice: from fine-tuning to generative modeling, explainability, and more. We also skimmed on many engineering aspects: training and serving large models is a huge engineering feat which requires, among other things, distributed training strategies, fast compilers, and DevOps techniques. And the emergence of LLMs has opened up new avenues for their use where knowledge of their inner workings is not even a prerequisite, from prompt engineering to model chaining and agentic behaviours.

This book has a companion website,[6] where I hope to publish additional chapters that touch upon some of these topics. If time allows, some of them may be joined together in a new volume.

I hope you appreciated the journey! For comments, suggestions, and feedback on the book do not hesitate to contact me.

[6] https://sscardapane.it/alice-book

A | Probability theory

About this chapter

Machine learning deals with a wide array of uncertainties (such as in the data collection phase), making the use of probability fundamental. We review here - informally - basic concepts associated with probability distributions and probability densities that are helpful in the main text. This appendix introduces many concepts, but many of them should be familiar. For a more in-depth exposition of probability in the context of machine learning and neural networks, see [Bis06, BB23].

A.1 Basic laws of probability

Consider a simple lottery, where you can buy tickets with 3 possible outcomes: "no win", "small win", and "large win". For any 10 tickets, 1 of them will have a large win, 3 will have a small win, and 6 will have no win. We can represent this with a probability distribution describing the relative frequency of the three events (we assume an unlimited supply of tickets):

$$p(w = \text{'no win'}) = 6/10$$
$$p(w = \text{'small win'}) = 3/10$$
$$p(w = \text{'large win'}) = 1/10$$

Equivalently, we can associate an integer value $w = \{1, 2, 3\}$ to the three events, and write $p(w = 1) = 6/10$, $p(w = 2) = 3/10$, and $p(w = 3) = 1/10$. We call w a **random variable**. In the following we always write $p(w)$ in place of $p(w = i)$ for readability when possible. The elements of the probability distribution must be positive and they

must sum to one:

$$p(w) \geq 0, \sum_w p(w) = 1$$

The space of all such vectors is called the **probability simplex**.

> Remember that we use $\mathbf{p} \sim \Delta(n)$ to denote a vector of size n belonging to the probability simplex.

Suppose we introduce a second random variable r, a binary variable describing whether the ticket is real (1) or fake (2). The fake tickets are more profitable but less probable overall, as summarized in Table T.A.1.

Table T.A.1: *Relative frequency of winning at an hypothetical lottery, in which tickets can be either real or fake, shown for a set of* 100 *tickets.*

	$r = 1$ (real ticket)	$r = 2$ (fake ticket)
$w = 1$ (no win)	58	2
$w = 2$ (small win)	27	3
$w = 3$ (large win)	2	8
Sum	87	13

We can use the numbers in the table to describe a **joint probability distribution**, describing the probability of two random variables taking a certain value jointly:

$$p(r = 2, w = 3) = 8/100$$

Alternatively, we can define a **conditional probability distribution**, e.g., answering the question "*what is the probability of a certain event given that another event has occurred?*":

$$p(r = 1 \mid w = 3) = \frac{p(r = 1, w = 3)}{p(w = 3)} = 0.2$$

This is called the **product rule** of probability. As before, we can make the notation less verbose by using the random variable in-place of its value:

$$p(r, w) = p(r \mid w)p(w) \tag{E.A.1}$$

If $p(r \mid w) = p(r)$ we have $p(r, w) = p(r)p(w)$, and we say that the two variables are

independent. We can use conditional probabilities to **marginalize** over one random variable:

$$p(w) = \sum_r p(w, r) = \sum_r p(w \mid r)p(r) \tag{E.A.2}$$

This is called the **sum rule** of probability. The product and sum rules are the basic axioms that define the algebra of probabilities. By combining them we obtain the fundamental **Bayes's rule**:

$$p(r \mid w) = \frac{p(w \mid r)p(r)}{p(w)} = \frac{p(w \mid r)p(r)}{\sum_{r'} p(w \mid r')p(r')} \tag{E.A.3}$$

Bayes's rule allows us to "reverse" conditional distributions, e.g., computing the probability that a winning ticket is real or fake, by knowing the relative proportions of winning tickets in both categories (try it).

A.2 Real-valued probability distributions

In the real-valued case, defining $p(x)$ is more tricky, because x can take infinitely possible values, each of which has probability 0 by definition. However, we can work around this by defining a probability **cumulative density function** (CDF):

$$P(x) = \int_0^x p(t)dt$$

and defining the probability density function $p(x)$ as its derivative. We ignore most of the subtleties associated with working with probability densities, which are best tackled in the context of measure theory [BR07]. We only note that the product and sum rules continue to be valid in this case by suitably replacing sums with integrals:

$$p(x, y) = p(x \mid y)p(y) \tag{E.A.4}$$

$$p(x) = \int_y p(x \mid y)p(y)dy \tag{E.A.5}$$

Note that probability densities are not constrained to be less than one.

A.3 Common probability distributions

The previous random variables are example of **categorical probability distributions**, describing the situation in which a variable can take one out of k possible values. We can write this down compactly by defining as $\mathbf{p} \sim \Delta(k)$ the vector of probabilities, and by $\mathbf{x} \sim \text{Binary}(k)$ a one-hot encoding of the observed class:

$$p(\mathbf{x}) = \text{Cat}(\mathbf{x}; \mathbf{p}) = \prod_i p_i^{x_i}$$

We use a semicolon to differentiate the input of the distribution from its parameters. If $k = 2$, we can equivalently rewrite the distribution with a single scalar value p. The resulting distribution is called a **Bernoulli distribution**:

$$p(x) = \text{Bern}(x; p) = p^x(1-p)^{(1-x)}$$

In the continuous case, we will deal repeatedly with the **Gaussian** distribution, denoted by $\mathcal{N}(x; \mu, \sigma^2)$, describing a bell-shaped probability centered in μ (the mean) and with a spread of σ^2 (the variance):

$$p(x) = \mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

In the simplest case of mean zero and unitary variance, $\mu = 0$, $\sigma^2 = 1$, this is also called the **normal distribution**. For a vector $\mathbf{x} \sim (k)$, a multivariate variant of the Gaussian distribution is obtained by considering a mean vector $\mu \sim (k)$ and a covariance matrix $\Sigma \sim (k, k)$:

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \mu, \Sigma) = (2\pi)^{-k/2} \det(\Sigma)^{-1/2} \exp\left((\mathbf{x}-\mu)^\top \Sigma^{-1} (\mathbf{x}-\mu)\right)$$

Two interesting cases are Gaussian distributions with a diagonal covariance matrix, and the even simpler **isotropic** Gaussian having a diagonal covariance with all entries identical:

$$\Sigma = \sigma^2 \mathbf{I}$$

The first can be visualized as an axis-aligned ellipsoid, the isotropic one as an axis-aligned sphere.

A.4 Moments and expected values

In many cases we need to summarize a probability distribution with one or more values. Sometimes a finite number of values are enough: for example, having access to **p** for a categorical distribution or to μ and σ^2 for a Gaussian distribution completely describe the distribution itself. These are called **sufficient statistics**.

More in general, for any given function $f(x)$ we can define its **expected value** as:

$$\mathbb{E}_{p(x)}[f(x)] = \sum_x f(x)p(x) \tag{E.A.6}$$

In the real-valued case, we obtain the same definition by replacing the sum with an integral. Of particular interest, when $f(x) = x^p$ we have the **moments** (of order p) of the distribution, with $p = 1$ called the **mean** of the distribution:

$$\mathbb{E}_{p(x)}[x] = \sum_x xp(x)$$

We may want to estimate some expected values despite not having access to the underlying probability distribution. If we have access to a way of sampling elements from $p(x)$, we can apply the so-called **Monte Carlo estimator**:

$$\mathbb{E}_{p(x)}[f(x)] \approx \frac{1}{n} \sum_{x_i \sim p(x)} f(x_i) \tag{E.A.7}$$

where n controls the quality of the estimation and we use $x_i \sim p(x)$ to denote the operation of sampling from the probability distribution $p(x)$. For the first-order moment, this reverts to the very familiar notation for computing the mean of a quantity from several measurements:

$$\mathbb{E}_{p(x)}[x] = \frac{1}{n} \sum_{x_i \sim p(x)} x_i$$

A.5 Distance between probability distributions

At times we may also require some form of distance between probability distributions, in order to evaluate how close two distributions are. The **Kullback-Leibler** (KL) diver-

gence between $p(x)$ and $q(x)$ is a common choice:

$$\mathrm{KL}(p \,\|\, q) = \int p(x) \log \frac{p(x)}{q(x)} dx$$

The KL divergence is not a proper metric (it is asymmetric and does not respect the triangle inequality). It is lower bounded at 0, but it is not upper bounded. The divergence can only be defined if for any x such that $q(x) = 0$, then $p(x) = 0$ (i.e., the support of p is a subset of the support of q). The minimum of 0 is achieved whenever the two distributions are identical. The KL divergence can be written as an expected value, hence it can be estimated via Monte Carlo sampling as in (E.A.7).

A.6 Maximum likelihood estimation

Monte Carlo sampling shows that we can estimate quantities of interest concerning a probability distribution if we have access to samples from it. However, we may be interested in estimating the probability distribution itself. Suppose we have a guess about its functional form $f(x;s)$, where s are the sufficient statistics (e.g., mean and variance of a Gaussian distribution), and a set of n samples $x_i \sim p(x)$. We call these samples identical (because they come from the same probability distribution) and independently distributed, in short, i.i.d. Because of independence, their joint distribution factorizes for any choice of s:

$$p(x_1, \ldots, x_n) = \prod_{i=1}^{n} f(x_i; s)$$

Large products are inconvenient computationally, but we can equivalently rewrite this as a sum through a logarithmic transformation:

$$L(s) = \sum_{i=1}^{n} \log(f(x_i; s))$$

Finding the parameters s that maximize the previous quantity is called the **maximum likelihood** (ML) approach. Because of its importance, we reframe it briefly below.

Definition D.A.1 (Maximum likelihood) *Given a parametric family of probability distributions $f(x;s)$, and a set of n values $\{x_i\}_{i=1}^n$ which are i.i.d. samples from an unknown distribution $p(x)$, the best approximation to $p(x)$ according to the* ***maximum likelihood*** *(ML) principle is:*

$$s^* = \arg\max_s \sum_{i=1}^{n} \log(f(x_i;s))$$

If f is differentiable, we can maximize the objective through gradient descent. This is the core approach we follow for training differentiable models. For now, we close the appendix by describing simple examples of ML estimation in the case of standard probability distributions. We do not provide worked out calculations, for which we refer to [Bis06, BB23].

Maximum likelihood for the Bernoulli distribution

Consider first the case of a Bernoulli distribution with unknown parameter p. In this case, the ML estimator is:

$$p^* = \frac{\sum_i x_i}{n}$$

which is simply the ratio of positive samples over the entire dataset.

Maximum likelihood for the Gaussian distribution

For the Gaussian distribution, we can rewrite its log likelihood as:

$$L(\mu,\sigma^2) = -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2$$

Maximizing for μ and σ^2 separately returns the known rules for computing the empirical mean and variance of a Gaussian distribution:

$$\mu^* = \frac{1}{n}\sum_i x_i \tag{E.A.8}$$

$$\sigma^{2*} = \frac{1}{n}\sum_i (x_i-\mu^*)^2 \tag{E.A.9}$$

The two can be computed sequentially. Because we are using an estimate for the mean inside the variance's formula, it can be shown the resulting estimation is slightly biased. This can be corrected by modifying the normalization term to $\frac{1}{n-1}$; this is known as Bessel's correction.[1] For large n, the difference between the two variants is minimal.

[1] https://en.wikipedia.org/wiki/Bessel%27s_correction

B | Universal approximation in 1D

About this chapter

While formally proving the universal approximation theorem is beyond the scope of this book, it is helpful to get an intuitive feeling for how such proofs can be constructed. In this appendix we follow and extend the visual intuitions from a 2019 online book chapter by M. Nielsen,[a] to which we refer for an extended discussion (and some interactive visualizations), especially for the case of multi-dimensional inputs.

[a] http://neuralnetworksanddeeplearning.com/chap4.html

We focus on the original approximation theorem by Cybenko [Cyb89] which considers models having one hidden layer with sigmoid activation functions. We also restrict the analysis to functions with a single input and a single output, that can be visualized easily. The reasoning can be extended to other activation functions and to higher dimensions.

The outline of this visual proof is relatively simple:

1. As a first step, we show how to manually set the weights of a model with a single neuron in the hidden layer to approximate a step function.

2. Then, we proceed to show how adding another unit in the hidden layer allows to approximate any function which is constant over a small interval, and zero everywhere else (we call these interval functions "bin" functions).

3. Finally, we describe a simple procedure to approximate a generic function by first binning it to the desired accuracy, and then adding as many neurons as needed to approximate all bins in turn. For m bins we obtain a network with $2m$

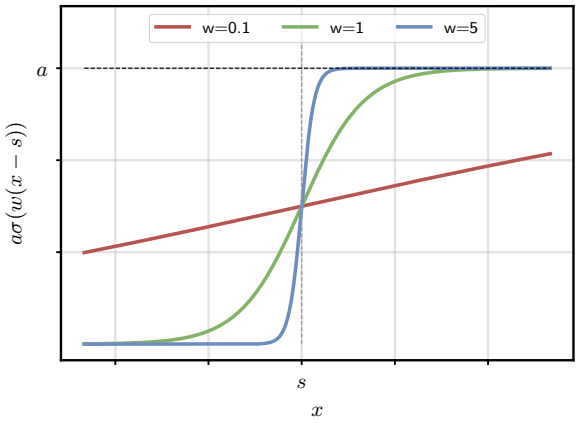


Figure F.B.1: *A network with a single neuron in the hidden layer can be visualized as a sigmoid with controllable slope, center, and amplitude. We show here an example where we fix the amplitude and the center, but we vary the slope.*

neurons. For a generic function with multiple inputs, this number would grow exponentially in the number of dimensions, making the proof non constructive in a practical case.

B.1 Approximating a step function

To begin, let us consider a single neuron in the hidden layer, in which case we can write the network's equation as (ignoring the output bias term, as it is not helpful in our derivation):

$$f(x) = a\sigma(wx + s)$$

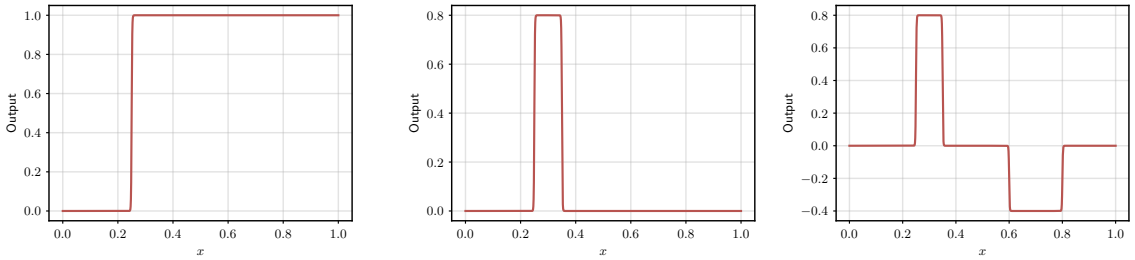
For the purposes of visualization, we rewrite this by adding a minus sign on the bias, and we factor the multiplication term on the entire input of σ (the two variants are clearly equivalent):

$$f(x) = a\,\sigma(\,w\,(x - s\,)) \tag{E.B.1}$$

Amplitude Slope Shift

This is similar to the "tunable" variant of sigmoid we introduce in Section 5.4. In particular, in this formulation a controls the amplitude of the sigmoid, w controls the slope, and s shifts the function by a fixed amount.

We show in Figure F.B.1 several plots of (E.B.1), where we fix a and s while varying w. As can be seen, by increasing w the slope gets steeper. Fixing it to a very large constant (say, $w = 10^4$), we are left with a very good approximation to a step function, of which we can control the location of the step (the s parameter) and the amplitude



(a) *1 neuron* **(b)** *2 neurons* **(c)** *4 neurons*

Figure F.B.2: *(a) A neural network with one input, one hidden neuron, and one output can approximate any step function (here shown with $a = 1$ and $s = 0.3$). (b) With two hidden neurons and one output we can approximate any function which is constant over a small interval. (c) With four neurons, we can approximate any function which is piecewise constant over two non-zero intervals. Note that bins can be negative by defining a negative amplitude.*

(the a parameter), as shown in Figure F.B.2a.

B.2 Approximating a constant function

If we add a second neuron with opposite amplitude (and slightly shifted position), we can approximate a function which is constant over a small interval (we call it a "bin" function). Defining a width Δ we can write:

$$f(x) = a\sigma\left(w\left(x - \left(s - \frac{\Delta}{2}\right)\right)\right) - a\sigma\left(w\left(x - \left(s + \frac{\Delta}{2}\right)\right)\right) \tag{E.B.2}$$

Go up [down] at $s - \frac{\Delta}{2}$ Go down [up] at $s + \frac{\Delta}{2}$

where we recall that w is now a large constant, e.g., 10^4. (E.B.2) describes a function (equivalent to a model with one hidden layer having two neurons) which increases by a at $s - \frac{\Delta}{2}$, is constant with value $f(x) = a$ over the interval $\left[s - \frac{\Delta}{2}, s + \frac{\Delta}{2}\right]$, and then decreases to 0 afterwards. An example is shown in Figure F.B.2b.

For the following, we can rewrite the previous function as $f(x; a, s, \Delta)$ to highlight the dependence on the three parameters a, s, and Δ.

B.3 Approximating a piecewise constant function

Because $f_{a,s,\Delta}(x)$ is effectively 0 outside the corresponding interval, two functions defined over non-intersecting intervals will not influence each other, i.e., the "bin" function we just defined is highly localized. Hence, by adding two additional neurons in the hidden layer we can define a function which is constant over two separate intervals (an example of which is shown in Figure F.B.2c):

$$f(x) = f(x; a_1, s_2, \Delta_1) + f(x; a_2, s_2, \Delta_2)$$

The rest of the proof is now trivial and proceeds by binning the function we want to approximate in many small intervals. Given any (continuous) function $g(x)$ over an interval (which we assume $[0, 1]$ for simplicity), we first bin the input domain into m equispaced intervals, where m controls the accuracy of the approximation (the higher m, the better the approximation). Hence, the i-th bin spans the interval:

$$B_i = \left[\frac{i}{m} - \frac{\Delta}{2}, \frac{i}{m} + \frac{\Delta}{2}\right]$$

where Δ is the size of each bin. For each bin, we compute the average value of $g(x)$ inside the interval itself:

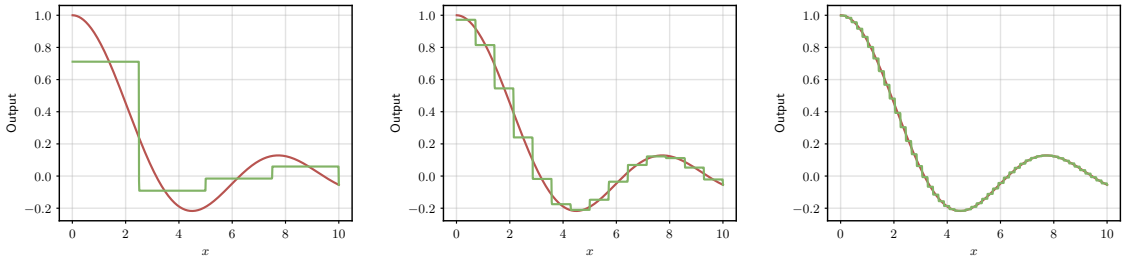
$$g_i = \frac{1}{\Delta}\int_{x \in B_i} g(x)dx$$

Finally, we define a network with $2m$ neurons in the hidden layer, two for each bin. Each bin function is centered in the bin and takes value g_i:

$$f(x) = \sum_{i=1}^{m} f\left(x; g_i, \frac{i}{m}, \Delta\right) \tag{E.B.3}$$

(Approximated) constant value

The i-th bin is centered in $\frac{i}{m}$



(a) *5 bins* **(b)** *15 bins* **(c)** *50 bins*

Figure E.B.3: *Approximating* $g(x) = \frac{\sin(x)}{x}$ *in* $[0, 10]$ *with (a)* $m = 5$*, (b)* $m = 15$*, and (c)* $m = 50$ *bins. The original function is in red, the approximation* (E.B.3) *in green. The average squared error in the three cases decreases exponentially (approximately* 0.02*,* 0.002*, and* 0.00016*).*

We show in Figure E.B.3 an example of such approximation in the case of $g(x) = \frac{\sin(x)}{x}$ for increasing number of bins ($m = 5$, $m = 15$, $m = 50$). It should be clear that the MSE is inversely proportional to m, and we can decrease the error as much as desired by simply increasing the resolution of the approximation.

Similar reasonings can be applied to multi-dimensional inputs and different activation functions.[1]

[1] http://neuralnetworksanddeeplearning.com/chap4.html

Bibliography

[AB21] A. N. Angelopoulos and S. Bates. A gentle introduction to conformal prediction and distribution-free uncertainty quantification. *arXiv preprint arXiv:2107.07511*, 2021. 71

[ADIP21] A. Apicella, F. Donnarumma, F. Isgrò, and R. Prevete. A survey on modern trainable activation functions. *Neural Networks*, 138:14–32, 2021. 85

[AHS23] S. K. Ainsworth, J. Hayase, and S. Srinivasa. Git re-basin: Merging models modulo permutation symmetries. In *ICLR*, 2023. 2

[AHSB14] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi. Learning activation functions to improve deep neural networks. *arXiv preprint arXiv:1412.6830*, 2014. 85

[AJB+17] D. Arpit, S. Jastrzębski, N. Ballas, D. Krueger, E. Bengio, M. S. Kanwal, T. Maharaj, A. Fischer, A. Courville, Y. Bengio, et al. A closer look at memorization in deep networks. In *International conference on machine learning*, pages 233–242. PMLR, 2017. 78

[AR00] J. A. Anderson and E. Rosenfeld. *Talking nets: An oral history of neural networks*. MIT Press, 2000. 6

[ARAA+16] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, pages 1–19, 2016. i, 9

[ASA+23] E. Akyürek, D. Schuurmans, J. Andreas, T. Ma, and D. Zhou. What learning algorithm is in-context learning? investigations with linear models. In *ICLR*, 2023. 1, 2, 40

[AST+24] A. F. Ansari, L. Stella, C. Turkmen, X. Zhang, P. Mercado, H. Shen, O. Shchur, S. S. Rangapuram, S. P. Arango, S. Kapoor, et al. Chronos: Learning the language of time series. *arXiv preprint arXiv:2403.07815*, 2024. 133, 198

[AZLL19] Z. Allen-Zhu, Y. Li, and Y. Liang. Learning and generalization in overparameterized neural networks, going beyond two layers. In *NeurIPS*, 2019. 79

[BAY22] S. Brody, U. Alon, and E. Yahav. How attentive are graph attention networks? In *ICLR*, 2022. 220, 221

[BB23] C. M. Bishop and H. Bishop. *Deep learning: Foundations and concepts*. Springer Nature, 2023. 9, 45, 247, 253

[BBCJ20] M. Biesialska, K. Biesialska, and M. R. Costa-Jussa. Continual lifelong learning in natural language processing: A survey. In *COLING*, pages 6523–6541, 2020. 41

[BBL+17] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017. 118, 213

[BCB15] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015. 170

[BCZ+16] T. Bolukbasi, K.-W. Chang, J. Y. Zou, V. Saligrama, and A. T. Kalai. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. In *NeurIPS*, volume 29, 2016. 40

[Ben09] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends® in Machine Learning*, 2(1):1–127, 2009. i, 5, 79

[BGHN24] J. Blasiok, P. Gopalan, L. Hu, and P. Nakkiran. When does optimizing a proper loss yield calibration? In *NeurIPS*, 2024. 69

[BGLA21] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi. Graph neural networks with convolutional arma filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(7):3496–3507, 2021. 214

[BGMMS21] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *ACM FAccT*, pages 610–623. ACM, 2021. 40

[BGSW18] N. Bjorck, C. P. Gomes, B. Selman, and K. Q. Weinberger. Understanding batch normalization. In *NeurIPS*, 2018. 157, 158

[BHB+18] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018. 222

[Bis95] C. M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural Computation*, 7(1):108–116, 1995. 149

[Bis06] C. Bishop. *Pattern recognition and machine learning*. Springer, 2006. 48, 58, 70, 247, 253

[BJMO12] F. Bach, R. Jenatton, J. Mairal, and G. Obozinski. Optimization with sparsity-inducing penalties. *Foundations and Trends® in Machine Learning*, 4(1):1–106, 2012. 146

[BKH16] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016. 159, 234

[BKK19] S. Bai, J. Z. Kolter, and V. Koltun. Deep equilibrium models. In *NeurIPS*, 2019. 231

[Ble90] G. E. Blelloch. *Prefix sums and their applications*. School of Computer Science, Carnegie Mellon University Pittsburgh, PA, USA, 1990. 237

[BNS06] M. Belkin, P. Niyogi, and V. Sindhwani. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *Journal of Machine Learning Research*, 7(11), 2006. 41, 210

[BP20] J. Bolte and E. Pauwels. A mathematical model for automatic differentiation in machine learning. In *NeurIPS*, pages 10809–10819, 2020. 102

[BPA+24] F. Bordes, R. Y. Pang, A. Ajay, A. C. Li, A. Bardes, S. Petryk, O. Mañas, Z. Lin, A. Mahmoud, B. Jayaraman, et al. An introduction to vision-language modeling. *arXiv preprint arXiv:2405.17247*, 2024. 169, 198

[BPRS18] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018. 87

[BPS+24] M. Beck, K. Pöppel, M. Spanring, A. Auer, O. Prudnikova, M. Kopp, G. Klambauer, J. Brandstetter, and S. Hochreiter. xlstm: Extended long short-term memory. *arXiv preprint arXiv:2405.04517*, 2024. 235

[BR07] V. I. Bogachev and M. A. S. Ruas. *Measure theory*. Springer, 2007. 249

[BR24] M. Blondel and V. Roulet. The elements of differentiable programming. *arXiv preprint arXiv:2403.14606*, 2024. 6, 27, 87, 95

[BZMA20] A. Baevski, Y. Zhou, A. Mohamed, and M. Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. In *NeurIPS*, pages 12449–12460, 2020. 197

[CCGC24] Y. Cheng, G. G. Chrysos, M. Georgopoulos, and V. Cevher. Multilinear operator networks. In *ICLR*, 2024. 200

[CMMB22] F. Cinus, M. Minici, C. Monti, and F. Bonchi. The effect of people recommenders on echo chambers and polarization. In *AAAI ICWSM*, volume 16, pages 90–101, 2022. 41

[CPPM22] E. Chien, C. Pan, J. Peng, and O. Milenkovic. You are allset: A multiset function framework for hypergraph neural networks. In *ICLR*, 2022. 222

[CRBD18] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. In *NeurIPS*, 2018. 164

[CS21] S. Chung and H. Siegelmann. Turing completeness of bounded-precision recurrent neural networks. pages 28431–28441, 2021. 231

[CVMG+14] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP*. ACL, 2014. 235

[CW82] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing*, 11(3):472–492, 1982. 20

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989. 78, 255

[CZJ+22] H. Chang, H. Zhang, L. Jiang, C. Liu, and W. T. Freeman. Maskgit: Masked generative image transformer. In *IEEE/CVF CVPR*, pages 11315–11325, 2022. 197

[CZSL20] E. D. Cubuk, B. Zoph, J. Shlens, and Q. V. Le. Randaugment: Practical automated data augmentation with a reduced search space. In *IEEE/CVF CVPR Workshops*, pages 702–703, 2020. 149

[DBK+21] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021. 185, 197, 201

[DCLT18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, page 4171–4186. ACL, 2018. 196

[DCSA23] A. Défossez, J. Copet, G. Synnaeve, and Y. Adi. High fidelity neural audio compression. *Transactions on Machine Learning Research*, 2023. 198

[DDM+23] M. Dehghani, J. Djolonga, B. Mustafa, P. Padlewski, J. Heek, J. Gilmer, A. P. Steiner, M. Caron, R. Geirhos, I. Alabdulmohsin, et al. Scaling vision transformers to 22 billion parameters. In *ICML*, pages 7480–7512, 2023. 199

[DFAG17] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier. Language modeling with gated convolutional networks. In *ICML*, pages 933–941, 2017. 85

[DFE+22] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *NeurIPS*, pages 16344–16359, 2022. 194

[DLL+22] V. P. Dwivedi, A. T. Luu, T. Laurent, Y. Bengio, and X. Bresson. Graph neural networks with learnable structural and positional representations. In *ICLR*, 2022. 223, 224

[DOMB24] T. Darcet, M. Oquab, J. Mairal, and P. Bojanowski. Vision transformers need registers. In *ICLR*, 2024. 185

[DS20] S. De and S. Smith. Batch normalization biases residual blocks towards the identity function in deep networks. In *NeurIPS*, pages 19964–19975, 2020. 162

[DT17] T. DeVries and G. W. Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017. 155

[DZPS19] S. S. Du, X. Zhai, B. Poczos, and A. Singh. Gradient descent provably optimizes over-parameterized neural networks. In *ICLR*, 2019. 79

[EHB23] F. Eijkelboom, R. Hesselink, and E. J. Bekkers. E (n) equivariant message passing simplicial networks. In *ICML*, pages 9071–9081, 2023. 222

[FAL17] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, pages 1126–1135. PMLR, 2017. 41

[Fle23] F. Fleuret. *The Little Book of Deep Learning*. Lulu Press, Inc., 2023. 9

[GBGB21] D. J. Gauthier, E. Bollt, A. Griffith, and W. A. Barbosa. Next generation reservoir computing. *Nature Communications*, 12(1):5564, 2021. 232

[GD23] A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023. 242, 243

[GDE+20] A. Gu, T. Dao, S. Ermon, A. Rudra, and C. Ré. Hippo: Recurrent memory with optimal polynomial projections. In *NeurIPS*, pages 1474–1487, 2020. 236

[GFGS06] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *ICML*, pages 369–376, 2006. 198

[GG16] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *ICML*, pages 1050–1059, 2016. 154

[GGR22] A. Gu, K. Goel, and C. Ré. Efficiently modeling long sequences with structured state spaces. In *ICLR*, 2022. 236, 239

[GJG+21] A. Gu, I. Johnson, K. Goel, K. Saab, T. Dao, A. Rudra, and C. Ré. Combining recurrent, convolutional, and continuous-time models with linear state space layers. pages 572–585, 2021. 236, 243

[GM17] C. Gallicchio and A. Micheli. Echo state property of deep reservoir computing networks. *Cognitive Computation*, 9:337–350, 2017. 234

[GMS05] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *IEEE IJCNN*, volume 2, pages 729–734. IEEE, 2005. 231

[GOV22] L. Grinsztajn, E. Oyallon, and G. Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? In *NeurIPS*, pages 507–520, 2022. 107

[GPE+23] S. Golkar, M. Pettee, M. Eickenberg, A. Bietti, M. Cranmer, G. Krawezik, F. Lanusse, M. McCabe, R. Ohana, L. Parker, et al. xval: A continuous number encoding for large language models. *arXiv preprint arXiv:2310.02989*, 2023. 130

[GPSW17] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger. On calibration of modern neural networks. In *ICML*, pages 1321–1330. PMLR, 2017. 71

[Gri12] A. Griewank. Who invented the reverse mode of differentiation? *Documenta Mathematica, Extra Volume ISMP*, 389400, 2012. 88

[GSBL20] M. Geva, R. Schuster, J. Berant, and O. Levy. Transformer feed-forward layers are key-value memories. In *EMNLP*, page 5484–5495. ACL, 2020. 192

[GSR+17] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *ICML*, pages 1263–1272, 2017. 221

[GW08] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008. 87, 98

[GWFM+13] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. In *ICML*, pages 1319–1327, 2013. 85

[GZBA22] D. Grattarola, D. Zambon, F. M. Bianchi, and C. Alippi. Understanding pooling in graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2022. 214

[HABN+21] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 22(241):1–124, 2021. 146

[Hac19] B. K. Hackenberger. Bayes or not bayes, is this the question? *Croatian Medical Journal*, 60(1):50, 2019. 49

[HBE+24] A. Ho, T. Besiroglu, E. Erdil, D. Owen, R. Rahman, Z. C. Guo, D. Atkinson, N. Thompson, and J. Sevilla. Algorithmic progress in language models. *arXiv preprint arXiv:1710.05941*, 2024. 1, 2

[HDLL22] W. Hua, Z. Dai, H. Liu, and Q. Le. Transformer quality in linear time. In *ICML*, pages 9099–9117, 2022. 200

[HG16] D. Hendrycks and K. Gimpel. Gaussian error linear units (GELUs). *arXiv preprint arXiv:1606.08415*, 2016. 84

[HHWW14] P. Huang, Y. Huang, W. Wang, and L. Wang. Deep embedding network for clustering. In *ICPR*, pages 1532–1537. IEEE, 2014. 39

[Hoc98] S. Hochreiter. Recurrent neural net learning and vanishing gradient. *International Journal Of Uncertainity, Fuzziness and Knowledge-Based Systems*, 6(2):107–116, 1998. 234

[Hor91] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991. 79

[HR22] M. Hardt and B. Recht. *Patterns, predictions, and actions: Foundations of machine learning*. Princeton University Press, 2022. 9

[HS97] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. 235

[HSS08] T. Hofmann, B. Schölkopf, and A. J. Smola. Kernel methods in machine learning. *The Annals of Statistics*, 36(3):1171–1220, 2008. 78, 228

[HTF09] T. Hastie, R. Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009. 38, 69

[HYL17] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, 2017. 219

[HZC+17] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 121

[HZRS15] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *IEEE ICCV*, pages 1026–1034, 2015. 83

[HZRS16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *IEEE/CVF CVPR*, pages 770–778, 2016. 161, 162, 163, 165

[ICS22] K. Irie, R. Csordás, and J. Schmidhuber. The dual form of neural networks revisited: Connecting test time predictions to training patterns via spotlights of attention. In *ICML*, pages 9639–9659, 2022. 57

[IS15] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, pages 448–456, 2015. 156

[JGB+21] A. Jaegle, F. Gimeno, A. Brock, O. Vinyals, A. Zisserman, and J. Carreira. Perceiver: General perception with iterative attention. In *ICML*, pages 4651–4664, 2021. 193

[JK+17] P. Jain, P. Kar, et al. Non-convex optimization for machine learning. *Foundations and Trends® in Machine Learning*, 10(3-4):142–363, 2017. 32

[JLB+22] L. V. Jospin, H. Laga, F. Boussaid, W. Buntine, and M. Bennamoun. Hands-on bayesian neural networks—a tutorial for deep learning users. *IEEE Computational Intelligence Magazine*, 17(2):29–48, 2022. 47, 48

[KB15] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015. 34

[KG21] P. Kidger and C. Garcia. Equinox: neural networks in JAX via callable PyTrees and filtered transformations. *Differentiable Programming workshop at Neural Information Processing Systems 2021*, 2021. 86

[KL18] S. M. Kakade and J. D. Lee. Provably correct automatic sub-differentiation for qualified programs. In *NeurIPS*, 2018. 102

[KMH+20] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. 1, 169

[KPR+17] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017. 49

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012. 83, 144

[KVPF20] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *ICML*, pages 5156–5165, 2020. 227, 228

[KW17] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017. 212

[Lau19] S. Laue. On the equivalence of automatic and symbolic differentiation. *arXiv preprint arXiv:1904.02990*, 2019. 91

[LBBH98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 6

[LBH15] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. 5

[LCX+23] J. Li, Y. Cheng, Z. Xia, Y. Mo, and G. Huang. Generalized activation via multivariate projection. *arXiv preprint arXiv:2309.17194*, 2023. 85

[LDR23] V. Lialin, V. Deshpande, and A. Rumshisky. Scaling down to scale up: A guide to parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.15647*, 2023. 40

[LDSL21] H. Liu, Z. Dai, D. So, and Q. V. Le. Pay attention to MLPs. In *NeurIPS*, pages 9204–9215, 2021. 200

[LH19] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *ICLR*, 2019. 146

[Lim21] L.-H. Lim. Tensors in computations. *Acta Numerica*, 30:555–764, 2021. 7

[LJ09] M. Lukoševičius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009. 232

[LKM23] Y. Leviathan, M. Kalman, and Y. Matias. Fast inference from transformers via speculative decoding. In *ICML*, pages 19274–19286, 2023. 193

[LLLG22] Y. Li, B. Lin, B. Luo, and N. Gui. Graph representation learning beyond node and homophily. *IEEE Transactions on Knowledge and Data Engineering*, 35(5):4880–4893, 2022. 220

[LLS21] S. H. Lee, S. Lee, and B. C. Song. Vision transformer for small-size datasets. *arXiv preprint arXiv:2112.13492*, 2021. 201

[LMW+22] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie. A ConvNet for the 2020s. In *IEEE/CVF CVPR*, pages 11976–11986, 2022. 163, 165

[LPW+17] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang. The expressive power of neural networks: A view from the width. In *NeurIPS*, 2017. 79

[LRZ+23] D. Lim, J. Robinson, L. Zhao, T. Smidt, S. Sra, H. Maron, and S. Jegelka. Sign and basis invariant networks for spectral graph representation learning. In *ICLR*, 2023. 224

[LTM+22] H. Liu, D. Tam, M. Muqeeth, J. Mohta, T. Huang, M. Bansal, and C. A. Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. In *NeurIPS*, pages 1950–1965, 2022. 2

[LWV+24] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljačić, T. Y. Hou, and M. Tegmark. Kan: Kolmogorov-arnold networks. *arXiv preprint arXiv:2404.19756*, 2024. 85

[LZA23] H. Liu, M. Zaharia, and P. Abbeel. Ring attention with blockwise transformers for near-infinite context. In *Foundation Models for Decision Making Workshop, NeurIPS*, 2023. 195

[MCT+] H. Mao, Z. Chen, W. Tang, J. Zhao, Y. Ma, T. Zhao, N. Shah, M. Galkin, and J. Tang. Position: Graph foundation models are already here. In *Forty-first International Conference on Machine Learning*. 224

[Met22] C. Metz. *Genius makers: the mavericks who brought AI to Google, Facebook, and the world*. Penguin, 2022. 5

[MGMR24] L. Müller, M. Galkin, C. Morris, and L. Rampášek. Attending to graph transformers. *Transactions on Machine Learning Research*, 2024. 223, 224

[MKS+20] J. Mukhoti, V. Kulharia, A. Sanyal, S. Golodetz, P. Torr, and P. Dokania. Calibrating deep neural networks using focal loss. In *NeurIPS*, pages 15288–15299, 2020. 71

[MRF+19] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019. 222

[MRT18] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT Press, 2018. 44

[MSC+13] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NeurIPS*, 2013. 39

[MZBG18] G. Marra, D. Zanca, A. Betti, and M. Gori. Learning neuron non-linearities with kernel-based deep neural networks. *arXiv preprint arXiv:1807.06302*, 2018. 85

[NCN+23] V. Niculae, C. F. Corro, N. Nangia, T. Mihaylova, and A. F. Martins. Discrete latent structure in neural networks. *arXiv preprint arXiv:2301.07473*, 2023. 88

[ODG+23] A. Orvieto, S. De, C. Gulcehre, R. Pascanu, and S. L. Smith. On the universality of linear recurrences followed by nonlinear projections. In *HLD 2023 Workshop, ICML*, 2023. 236

[ODZ+16] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. In *ISCA SSW Workshop*, 2016. 132, 135, 138

[OSG+23] A. Orvieto, S. L. Smith, A. Gu, A. Fernando, C. Gulcehre, R. Pascanu, and S. De. Resurrecting recurrent neural networks for long sequences. In *ICML*, 2023. 236, 240

[PAA+23] B. Peng, E. Alcaide, Q. Anthony, A. Albalak, S. Arcadinho, H. Cao, X. Cheng, M. Chung, M. Grella, K. K. GV, et al. Rwkv: Reinventing rnns for the transformer era. In *EMNLP*. ACL, 2023. 241

[PABH+21] O. Puny, M. Atzmon, H. Ben-Hamu, I. Misra, A. Grover, E. J. Smith, and Y. Lipman. Frame averaging for invariant and equivariant network design. *arXiv preprint arXiv:2110.03336*, 2021. 118

[PBE+22] A. Power, Y. Burda, H. Edwards, I. Babuschkin, and V. Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets. In *1st Mathematical Reasoning in General Artificial Intelligence Workshop, ICLR*, 2022. 2, 148

[PBL20] T. Poggio, A. Banburski, and Q. Liao. Theoretical issues in deep networks. *Proceedings of the National Academy of Sciences*, 117(48):30039–30045, 2020. 44

[PGCB14] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio. How to construct deep recurrent neural networks. In *ICLR*, 2014. 231

[PKP+19] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71, 2019. 41

[PNR+21] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *Journal of Machine Learning Research*, 22(57):1–64, 2021. 164

[PP08] K. B. Petersen and M. S. Pedersen. *The matrix cookbook*. Technical University of Denmark, 2008. 26

[PPVF21] S. Pesme, L. Pillaud-Vivien, and N. Flammarion. Implicit bias of SGD for diagonal linear networks: a provable benefit of stochasticity. 34:29218–29230, 2021. 78

[PRCB24] A. Patel, C. Raffel, and C. Callison-Burch. Datadreamer: A tool for synthetic data generation and reproducible llm workflows. In *ACL*. ACL, 2024. 148

[Pri23] S. J. Prince. *Understanding Deep Learning*. MIT Press, 2023. 9

[PS+03] T. Poggio, S. Smale, et al. The mathematics of learning: Dealing with data. *Notices of the AMS*, 50(5):537–544, 2003. 44

[PSL22] O. Press, N. A. Smith, and M. Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *ICLR*, 2022. 183

[QPF+24] S. Qiu, A. Potapczynski, M. Finzi, M. Goldblum, and A. G. Wilson. Compute better spent: Replacing dense layers with structured matrices. *arXiv preprint arXiv:2406.06248*, 2024. 114

[RBOB18] M. Ravanelli, P. Brakel, M. Omologo, and Y. Bengio. Light gated recurrent units for speech recognition. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(2):92–102, 2018. 234

[RGD+22] L. Rampášek, M. Galkin, V. P. Dwivedi, A. T. Luu, G. Wolf, and D. Beaini. Recipe for a general, powerful, scalable graph transformer. In *NeurIPS*, pages 14501–14515, 2022. 223

[RHM86] D. E. Rumelhart, G. E. Hinton, and J. L. McClelland. A general framework for parallel distributed processing. In *Parallel Distributed Processing Volume 1*, page 26. MIT Press, 1986. 5, 6

[RKG+22] D. W. Romero, D. M. Knigge, A. Gu, E. J. Bekkers, E. Gavves, J. M. Tomczak, and M. Hoogendoorn. Towards a general purpose cnn for long range dependencies in *n*d. *arXiv preprint arXiv:2206.03398*, 2022. 171

[RKX+23] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever. Robust speech recognition via large-scale weak supervision. In *ICML*, pages 28492–28518, 2023. 197, 198

[RM22] J. W. Rocks and P. Mehta. Memorizing without overfitting: Bias, variance, and interpolation in overparameterized models. *Physical Review Research*, 4(1):013201, 2022. 148

[RS21] M. N. Rabe and C. Staats. Self-attention does not need $\mathcal{O}(n^2)$ memory. *arXiv preprint arXiv:2112.05682*, 2021. 194

[RSR+20] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020. 196

[RWC+19] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 2019. 39, 193, 196

[RZL17] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017. 84

[SAL+24] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. 183

[Sch15] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. 235

[SCHU17] S. Scardapane, D. Comminiello, A. Hussain, and A. Uncini. Group sparse regularization for deep neural networks. *Neurocomputing*, 241:81–89, 2017. 146

[SGT+08] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008. 231

[Sha19] N. Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019. 199

[Sha20] N. Shazeer. GLU variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020. 85

[SHK+14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 152

[SHW21] V. G. Satorras, E. Hoogeboom, and M. Welling. E(n) equivariant graph neural networks. In *ICML*, pages 9323–9332, 2021. 222

[SKF+99] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Byte pair encoding: A text compression scheme that accelerates pattern matching. 1999. 129

[SKZ+21] A. Steiner, A. Kolesnikov, X. Zhai, R. Wightman, J. Uszkoreit, and L. Beyer. How to train your vit? data, augmentation, and regularization in vision transformers. *arXiv preprint arXiv:2106.10270*, 2021. 201

[SLJ+15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *IEEE CVPR*, pages 1–9, 2015. 117

[SMDH13] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *ICML*, pages 1139–1147, 2013. 34

[SP97] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997. 231

[SSBD14] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge University Press, 2014. 44

[Sti81] S. M. Stigler. Gauss and the invention of least squares. *The Annals of Statistics*, pages 465–474, 1981. 6

[SVL14] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NeurIPS*, 2014. 189

[SVVTU19] S. Scardapane, S. Van Vaerenbergh, S. Totaro, and A. Uncini. Kafnets: Kernel-based non-parametric activation functions for neural networks. *Neural Networks*, 110:19–32, 2019. 85

[SW17] S. Scardapane and D. Wang. Randomness in neural networks: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(2):e1200, 2017. 228

[SWF+15] S. Sukhbaatar, J. Weston, R. Fergus, et al. End-to-end memory networks. 2015. 191

[SWL23] J. T. Smith, A. Warrington, and S. W. Linderman. Simplified state space layers for sequence modeling. In *ICLR*, 2023. 236, 237, 238

[TCB+24] M. Tiezzi, M. Casoni, A. Betti, M. Gori, and S. Melacci. State-space modeling in long sequence processing: A survey on recurrence in the transformer era. *arXiv preprint arXiv:2406.09062*, 2024. 227

[TEM23] M. Tschannen, C. Eastwood, and F. Mentzer. Givt: Generative infinite-vocabulary transformers. *arXiv preprint arXiv:2312.02116*, 2023. 197

[TGJ+15] J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler. Efficient object localization using convolutional networks. In *IEEE/CVF CVPR*, pages 648–656, 2015. 155

[THK+21] I. O. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, et al. MLP-mixer: An all-MLP architecture for vision. In *NeurIPS*, pages 24261–24272, 2021. 199

[TLI+23] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023. 2, 40, 196, 200

[TNHA24] D. Teney, A. M. Nicolicioiu, V. Hartmann, and E. Abbasnejad. Neural redshift: Random networks are not random functions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4786–4796, 2024. 78

[Unc15] A. Uncini. *Fundamentals of adaptive signal processing*. Springer, 2015. 115

[Vap13] V. Vapnik. *The nature of statistical learning theory*. Springer Science & Business Media, 2013. 44

[VCC+18] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. In *ICLR*, 2018. 220

[Vel22] P. Veličković. Message passing all the way up. *arXiv preprint arXiv:2202.11097*, 2022. 221

[VSP+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *NeurIPS*, 2017. 169, 181, 182, 189, 192

[VWB16] A. Veit, M. J. Wilber, and S. Belongie. Residual networks behave like ensembles of relatively shallow networks. In *NeurIPS*, 2016. 164

[WCW+23] C. Wang, S. Chen, Y. Wu, Z. Zhang, L. Zhou, S. Liu, Z. Chen, Y. Liu, H. Wang, J. Li, et al. Neural codec language models are zero-shot text to speech synthesizers. *arXiv preprint arXiv:2301.02111*, 2023. 198

[WFD+23] H. Wang, T. Fu, Y. Du, W. Gao, K. Huang, Z. Liu, P. Chandak, S. Liu, P. Van Katwyk, A. Deac, et al. Scientific discovery in the age of artificial intelligence. *Nature*, 620(7972):47–60, 2023. 1, 133

[WGGH18] X. Wang, R. Girshick, A. Gupta, and K. He. Non-local neural networks. In *IEEE/CVF CVPR*, pages 7794–7803, 2018. 172

[WJ21] Y. Wu and J. Johnson. Rethinking "Batch" in BatchNorm. *arXiv preprint arXiv:2105.07576*, 2021. 158

[WZZ+13] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus. Regularization of neural networks using DropConnect. In *ICML*, pages 1058–1066, 2013. 155

[XYH+20] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T. Liu. On layer normalization in the transformer architecture. In *ICML*, pages 10524–10533, 2020. 183

[YCC+24] L. Yuan, Y. Chen, G. Cui, H. Gao, F. Zou, X. Cheng, H. Ji, Z. Liu, and M. Sun. Revisiting out-of-distribution robustness in nlp: Benchmarks, analysis, and llms evaluations. In *NeurIPS*, 2024. 38

[YHO+19] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *IEEE/CVF ICCV*, pages 6023–6032, 2019. 149

[YLC+22] T. Yu, X. Li, Y. Cai, M. Sun, and P. Li. S2-MLP: Spatial-shift MLP architecture for vision. In *IEEE/CVF WACV*, pages 297–306, 2022. 199

[YLZ+22] W. Yu, M. Luo, P. Zhou, C. Si, Y. Zhou, X. Wang, J. Feng, and S. Yan. Metaformer is actually what you need for vision. In *IEEE/CVF CVPR*, pages 10819–10829, 2022. 199

[YYZ17] B. Yu, H. Yin, and Z. Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. In *IJCAI*, 2017. 222

[ZBH+21] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021. 1

[ZCDLP17] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz. mixup: Beyond empirical risk minimization. In *ICLR*, 2017. 149

[ZER+23] A. Zador, S. Escola, B. Richards, B. Ölveczky, Y. Bengio, K. Boahen, M. Botvinick, D. Chklovskii, A. Churchland, C. Clopath, et al. Catalyzing next-generation artificial intelligence through neuroAI. *Nature Communications*, 14(1):1597, 2023. 5

[ZJM+21] J. Zbontar, L. Jing, I. Misra, Y. LeCun, and S. Deny. Barlow twins: Self-supervised learning via redundancy reduction. In *ICML*, pages 12310–12320. PMLR, 2021. 39

[ZKR+17] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola. Deep sets. 30, 2017. 187

[ZLLS23] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. *Dive into deep learning*. Cambridge University Press, 2023. 9, 33, 34

[ZS19] B. Zhang and R. Sennrich. Root mean square layer normalization. In *NeurIPS*, 2019. 160

[ZTS+21] S. Zhai, W. Talbott, N. Srivastava, C. Huang, H. Goh, R. Zhang, and J. Susskind. An attention free transformer. *arXiv preprint arXiv:2105.14103*, 2021. 240, 241

[ZW23] L. Ziyin and Z. Wang. spred: Solving l1 penalty with sgd. In *ICML*, pages 43407–43422, 2023. 146, 147