Chapter 2
Calculus and Optimisation for Machine Learning

In the fall of 1972, President Nixon announced that the rate of increase of inflation was decreasing. This was the first time a president used the third derivative to advance his case for reelection.

Hugo Rossi

Abstract In this chapter we will look at a very short overview of the necessary calculus and optimisation concepts that are used in machine learning. We will look at the concept of a derivative and of a partial derivative when dealing with multi-dimensional variables. We will discuss briefly the Jacobian (used for example when changing variables in a multiple integral) and the Hessian. Then we will look briefly at the Taylor's expansion and the Taylor's theorem, that we will use, for example, in the proof of the central limit in Chapter 4. After that we will discuss what is optimisation and what will look at the concept of convexity. We will then discuss gradient optimisation methods and their variations.

2.1 DRAFT - DO NOT REDISTRIBUTE

2.2 Motivation

Calculus is the branch of mathematics that concern itself with the study of concepts that involves, between a great number of other things, infinitely small quantities (as the rate of change of a function[1]) or infinitely large number of quantities (for example the sum of infinite terms in a series)[2]. It is one of the key component of deep learning (and in general machine learning), since it is at the core of the algorithms that are

[1] If you do not know why the calculation of the rate of change of a function does involve infinitely small quantities hang on, it will become clear later in this chapter.

[2] That is probably the most inaccurate and generic description that is possible to give. I nonetheless hope that it can give a bit of intuition of its applications.

responsible for the training of models. The most important concepts in calculus for deep learning are listed below.

1. **Derivatives** (in one or many dimensions) are relevant for minimisation algorithms, the ones responsible for training neural networks.
2. **Derivatives properties** are fundamental to understand backpropagation (the most important is the composition property).
3. **Extrema of functions** (minima and maxima) and where to find them[3]. Note that there is an entire branch of applied mathematics that is called optimisation that deals with how to find extrema. We will only scratch the surface of it in this book.

If you want to understand how optimisation algorithms (the ones responsible for minimisation functions) work, you need to understand derivatives. We will start from there, then try to understand what properties extrema of functions have (and how to find them) and finally do a relatively deep and detailed study of the gradient descent algorithm, **the** most famous (but not the most used anymore) and instructive algorithm used to train deep learning neural networks.

2.3 Concept of limit

Let's start with the concept of the limit of a function $f(x)$, necessary to understand derivatives. We will consider for the sake of simplicity only the case where $x \in \mathbb{R}$ and $f(x) \in \mathbb{R}$. An intuitive definition of limit is the following.

Definition 2.1 (intuitive) The limit of a function $f(x)$ at $x = x_0$ is the value L that the function approaches as the input x approaches some value x_0.

The limit L is typically indicated with the following notation.

$$L = \lim_{x \to x_0} f(x) \tag{2.1}$$

In other (and still intuitive) words, Equation (2.1) means that we can make $f(x)$ as close to L as we want by choosing x close enough to x_0. For those of you more mathematically inclined, the formal definition of limit is the following.

Definition 2.2 (formal) ★ L is called the limit of the function $f(x)$ at x_0 if given an arbitrary real $\epsilon > 0$, there is a δ such that for any x that satisfy $|x - x_0| < \delta$, it is true that $|f(x) - L| < \epsilon$.

You may think that it is easy to calculate the limit of a function. Why don't simply calculate $f(x_0)$? In same cases it is really that simple. For example

$$\lim_{x \to x_0} x^2 = x_0^2 \tag{2.2}$$

[3] If you are a fan of Harry Potter you should get the reference.

But what is the value of the following limit[4]?

$$\lim_{x\to 0} \frac{\sin x}{x} \tag{2.3}$$

You cannot simply calculate the value of $\sin x/x$ in 0, since the function is not defined. To calculate such limits there are plenty of methods and tricks, but those go beyond the scope of this book. In deep learning you will never need to calculate a limit. But the concept is important to understand derivatives.

Info ★ Geometric proof that $\lim_{x\to 0} \frac{\sin x}{x} = 1$

Since I do not like to give you any statement without proving it if possible, here is a proof of

$$\lim_{x\to 0} \frac{\sin x}{x} = 1. \tag{2.4}$$

based on geometry. Note that to understand it you need to have a basic understanding of trigonometry.

Proof Consider Figure (2.1).

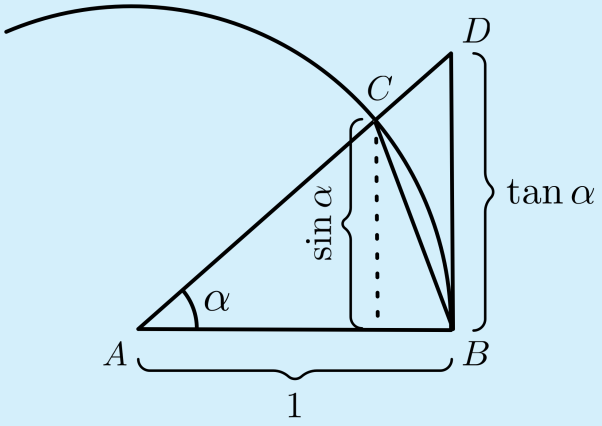

Fig. 2.1 Geometrical construction to justify the proof of $\lim_{x\to 0} \sin x/x = 1$.

The area of the triangle ABC (A(ABC)) is

$$A(\text{ABC}) = \frac{1}{2} \sin \alpha. \tag{2.5}$$

[4] The value of this limit is 1, but it is difficult to calculate, unless you know derivatives and advanced methods as the l'Hopital rule for limits.

The circle wedge limited by the letters ABC is $A_{\text{wedge}}(\text{ABC}) = \alpha/2$. Remember that comes from the fact that the angle α is expressed in radians. The area of the triangle $A(\text{ABD})$ is

$$A(\text{ABD}) = \frac{1}{2}\tan\alpha. \tag{2.6}$$

Now from Figure (2.1) you can see that

$$A(\text{ABD}) < A_{\text{wedge}}(\text{ABD}) < A(\text{ABC}) \tag{2.7}$$

that transaltes into

$$\frac{1}{2}\tan\alpha < \frac{\alpha}{2} < \frac{1}{2}\sin\alpha \tag{2.8}$$

dividing by $1/2\sin\alpha$ and taking reciprocals we get

$$\cos x < \frac{\sin x}{x} < 1 \tag{2.9}$$

Now since

$$\lim_{x\to 0}\cos x = 1 \tag{2.10}$$

from the disequalities in Equation (2.9) and by taking the limit for $x \to 0$ it follows that

$$\lim_{x\to 0}\frac{\sin x}{x} = 1. \tag{2.11}$$

This concludes the proof. □

2.4 Derivative and its Properties

The derivative can be understood intuitively in one dimension. Let's consider a generic function $y = f(x)$ with $x \in \mathbb{R}$ and $y \in \mathbb{R}$. The derivative of $f(x)$ at a point x_0 is indicated with the symbols

$$f'(x_0) \quad \text{or} \quad \frac{df}{dx}(x_0) \tag{2.12}$$

and it can be interpreted geometrically as the slope of the tangent of the function at $x = x_0$. In Figure 2.2 you can see an example. The function $f(x) = x^2$ is plotted in black, and the tangent to $f(x)$ for $x = x_0 = 3$ is plotted in red. The slope of the tangent is the derivative of $f(x)$ at $x = x_0 = 3$. The derivative is defined in terms of a limit. Formally one can give the following definition.

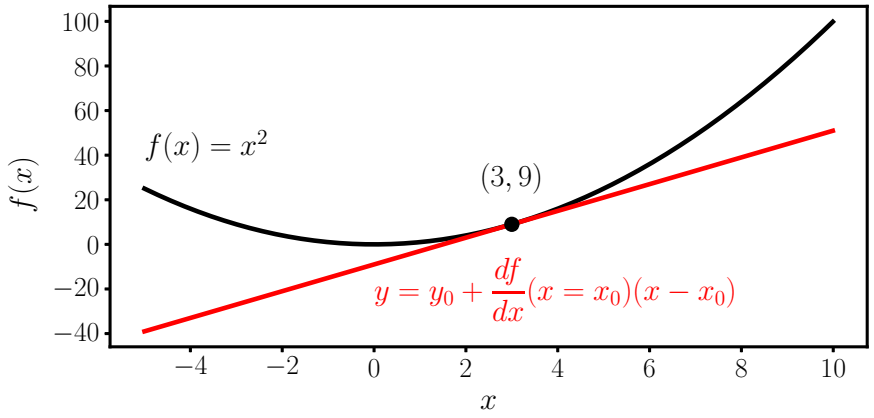

Fig. 2.2 When dealing with a function $f(x)$ of a one-dimensional variable x, the derivative at a given point x_0 is the slope of the tangent to the curve at the point $(x_0, f(x_0))$. The red curve in the figure is the tangent at the point $(3, 9)$ of the function $f(x) = x^2$. The slope of the red line is the derivative of the function (x) evaluated at 3. In this case $f'(x) = 2x$ and $f'(3) = 6$.

Definition 2.3 A function of $f(x) : \mathbb{R} \to \mathbb{R}$ is differentiable at a point x_0 of its domain[5], if its domain contains an open interval I containing x_0, and the limit

$$L = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{2.13}$$

exist. In this case L is called the derivative of $f(x)$ at $x = x_0$ and $L = f'(x_0)$.

Info ★ Existence of the limit

A limit of a function $g(x)$ for $x \to x_0$

$$q = \lim_{x \to x_0} g(x) \tag{2.14}$$

is said to exist if for every positive real number ϵ, there exists a positive real number δ such that $0 < |x - x_0| < \delta$ implies $|g(x) - q| < \epsilon$. In other words, it means that if x is close to x_0, then $g(x)$ will be close to q.
For example let's consider the function x^2 and let's prove the following lemma.

Lemma 2.1 *The limit*

$$L = \lim_{x \to x_0} x^2 \tag{2.15}$$

always exist for all finite x_0 and is $L = x_0^2$.

[5] The domain of a function is the set of all inputs accepted by the function.

Proof Let us consider an $\epsilon \in \mathbb{R}$. Let's find out what δ we need to choose such that from

$$|x - x_0| < \delta \tag{2.16}$$

it follows

$$|x^2 - x_0^2| < \epsilon. \tag{2.17}$$

Let us first note that

$$|x - x_0| < \delta \Rightarrow x < \delta + x_0 \text{ or } x < x_0 - \delta \tag{2.18}$$

By using Equation 2.18 we can write

$$x < \delta + x_0 \Rightarrow x + x_0 < \delta + 2x_0 \tag{2.19}$$

now we are almost there. In fact now we can write

$$|x^2 - x_0^2| = |x - x_0||x + x_0| < \delta|x + x_0| < \delta(2x_0 + \delta) \tag{2.20}$$

so to satisfy $|x^2 - x_0^2| < \epsilon$ we simply need to choose a δ given by the equation $\delta(2x_0 + \delta) = \epsilon$, that have the solution $\delta = \sqrt{x_0^2 + \epsilon} - x_0$. □

Lemma 2.1 can be better understood by looking at Figure 2.3.

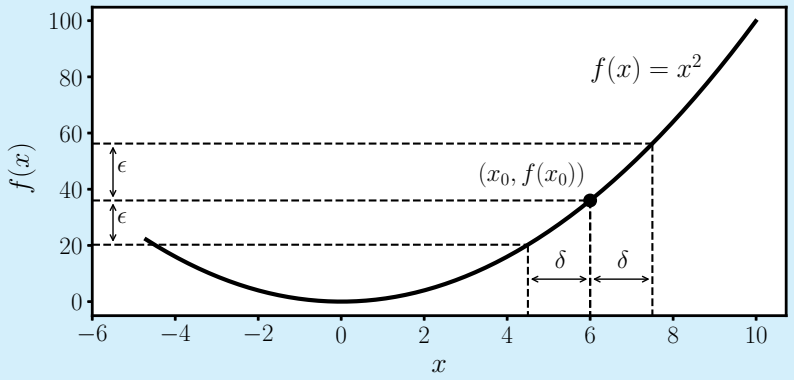

Fig. 2.3 In the figure you can see how, for the function x^2, for any ϵ chosen (the vertical intervals), there is always a δ that correspond to an interval in x. In the Figure we have chosen $x_0 = 6$. This is a graphical explanation of the concept of the existence of a limit. In fact one can show that $\lim_{x \to x_0} x^2$ is always existing for any finite x_0 as we proved in Lemma 2.1.

In this book we will not concern ourselves with the problem of existence of derivatives, but you should be aware that the derivative is not always existing in the domain of a function. Consider for example the function $f(x) = \max\{0, x\}$. This is called the

Rectified Linear Unit (ReLU) and is widely used in deep learning. But this function has no derivative at $x = 0$. In fact if you plot it, you will see how at x_0 the function has not a clear and definite slope. In Figure 2.4 you can see a plot of the ReLU function.

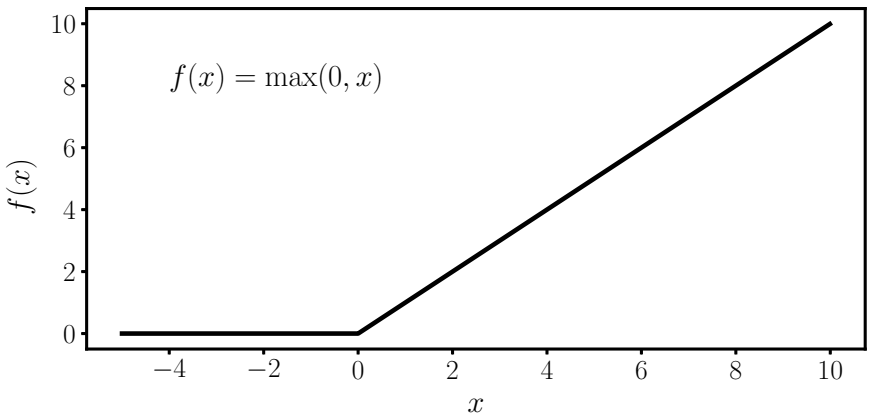

Fig. 2.4 The function ReLU ($\max\{0, x\}$, plotted in this figure, has no derivative at $x = 0$ as can be seen intuitively from the plot.

Warning ReLU function missing derivative at $x = 0$ and deep learning

In this book you will learn how the derivatives are an essential component in training neural networks. But we have just seen that the derivative of ReLU function is not defined at $x = 0$. So how comes, that this function is so used in deep learning? Surely this must be a problem!
In the numerical world, things are slightly different than in theory. When implementing the derivative of the ReLU function, it is enough to simply assign a value to it a specific value at $x = 0$. It is enough to program it in the code as

$$\frac{d\text{ReLU}}{dx} = \begin{cases} 0 & \text{for } x < 0 \\ 0(\text{or } 1) & \text{for } x = 0 \\ 1 & \text{for } x > 0 \end{cases} \tag{2.21}$$

Note that this is, strictly speaking, equal to the derivative of the ReLU function only for $x \neq 0$, but this makes numerical routines work. Additionally due to rounding errors in Python (the same applies to other programming languages) the output of a calculation between floating point variables will, from all practical purposes, never be **equal** to zero, making this discussion moot.

The Google TensorFlow library goes a step beyond that. It simply uses the derivative of the ReLU function only for $x > 0$, since if the derivative is zero, it is useless anyway. But it is important to be aware of such problems in the case, for example, that you want to develop your own activation or loss functions.
If you create in Python a numpy array with one element as `float16` datatype, for example with the line `a = np.array([0.1], dtype = np.float16)`, the output of printing it will be `0.0999755859375`, showing how rounding errors are always present, due to the way numbers are stored in the memory of computers. Some exceptions applies, but we will not spend any more time on this. Just be aware that derivatives are not always defined.

2.4.1 Derivatives' Properties

When you want to calculate derivatives, very often you have combinations of functions. For example you may want to calculate the derivative of

$$e^x + x^2 \tag{2.22}$$

or

$$\cos(\tan(e^x)). \tag{2.23}$$

It is very useful to know how derivatives behave when dealing with composition of functions. Here is a list of properties that you will find useful.

$$\begin{aligned}
&(f(x) + g(x))' = f'(x) + g'(x) \\
&(f(x) - g(x))' = f'(x) - g'(x) \\
&(f(x)g(x))' = f'(x)g(x) + f(x)g'(x) \\
&\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}
\end{aligned} \tag{2.24}$$

There is one additional property that is at the basis on how backpropagation works. Backpropagation is at the core of how it is possible to calculate derivatives of complicate neural network functions (you will find an entire chapter devoted to it in this book). This property tells you how to calculate the derivative of composition of functions.

$$(g(f(x)))' = g'(f(x))f'(x) \tag{2.25}$$

The composition of functions is often indicated with

$$(g \circ f)(x) = g(f(x)) \tag{2.26}$$

Info ★ Proof of the Formula $(g(f(x)))' = g'(f(x))f'(x)$

Proof The formula can be proven by using the definition of derivative.

$$\begin{aligned}
(g \circ f)'(x) &= \lim_{h\to 0} \frac{(g\circ f)(x+h) - (g\circ f)(x)}{h} \\
&= \lim_{h\to 0} \frac{g(f(x+h)) - g(f(x))}{h} \\
&= \lim_{h\to 0} \frac{g(f(x+h)) - g(f(x))}{h} \times \frac{f(x+h)-f(x)}{f(x+h)-f(x)} \\
&= \lim_{h\to 0} \frac{g(f(x+h)) - g(f(x))}{f(x+h)-f(x)} \times \frac{f(x+h)-f(x)}{h} \\
&= \lim_{h\to 0} \frac{g(f(x+h)) - g(f(x))}{f(x+h)-f(x)} \times \lim_{h\to 0} \frac{f(x+h)-f(x)}{h} \\
&= \lim_{h\to 0} \frac{g(f(x+h)) - g(f(x))}{f(x+h)-f(x)} \times f'(x)
\end{aligned} \tag{2.27}$$

by doing the change of variable $k = f(x+h) - f(x)$

$$\begin{aligned}
k &= f(x+h) - f(x) \\
f(x+h) &= f(x) + k
\end{aligned} \tag{2.28}$$

one can see that

$$\lim_{h\to 0} k = \lim_{h\to 0} f(x+h) - f(x) = f(x+0) - f(x) = 0 \tag{2.29}$$

by using k we can write

$$\begin{aligned}
\lim_{h\to 0} \frac{g(f(x+h)) - g(f(x))}{f(x+h)-f(x)} &= \lim_{k\to 0} \frac{g(f(x)+k) - g(f(x))}{k} \\
&= g'(f(x))
\end{aligned} \tag{2.30}$$

and thus finally

$$\begin{aligned}
(g \circ f)'(x) &= \lim_{h\to 0} \frac{g(f(x+h)) - g(f(x))}{f(x+h)-f(x)} \times f'(x) \\
&= g'(f(x)) \cdot f'(x)
\end{aligned} \tag{2.31}$$

That concludes the proof. □

But why this property is so important in deep learning? The reasons lies in the fact that in practically all kind of neural network architectures, the output is obtained by calculating the composition of a large (in some cases up to as much as 50 or more) number of functions. Basically you would need to work with something like

$$f_1(f_2(f_3(...f_N(x))))...) \tag{2.32}$$

with N some large number and $f_i(x)$ some complicated and non-linear function[6]. Even if you still do not know (you will learn exactly why in this book), believe me when I tell you that to train a neural network you have to calculate the derivative of such functions. Therefore property (2.25) is of fundamental importance.

2.5 Partial Derivative

The next concept that you will need in deep learning, is that of a partial derivative. This becomes relevant when you have a function of a variable $\mathbf{x} \in \mathbb{R}^n$, with $n > 1$. Take for example a function $f(x_1, x_2, ..., x_n)$. The partial derivative of f with respect to x_i in $(x_1, x_2, ..., c, ..., x_n)$ (with $x_i = c$) is indicated with

$$\frac{\partial f(x_1, ..., x_n)}{\partial x_i}(c) \tag{2.33}$$

and is defined by (note that the value c is at the i^{th} position)

$$\frac{\partial f(x_1, ..., x_n)}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, ..., c + h, ..., x_n) - f(x_1, ..., c, ..., x_n)}{h} \tag{2.34}$$

if the limit exists of course. This is nothing else that the slope of the function f along the i^{th} direction. It is called *partial* since the derivative is calculated by varying only x_i and keeping all the others inputs constant. In Figure (2.5) you can see the 3-dimensional plot of a function $g(x, y)$. The line indicated with L_x is the tangent to the surface at point P along the x-axis (its slope is the partial derivative of $g(x, y)$ with respect to x), and the line indicated with L_y is the tangent to the surface at point P along the y-axis (its slope is the partial derivative of $g(x, y)$ with respect to y).

2.6 Gradient

Let us consider a scalar function of multiple variables $f(x_1, x_2, ..., x_n)$. The gradient of f is indicated with ∇f and is defined by

$$\nabla f(x_1, ..., x_n) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ ... \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \tag{2.35}$$

[6] In a feed-forward neural network for example, the f_i may be interpreted as the output of layers of neurons in the network.

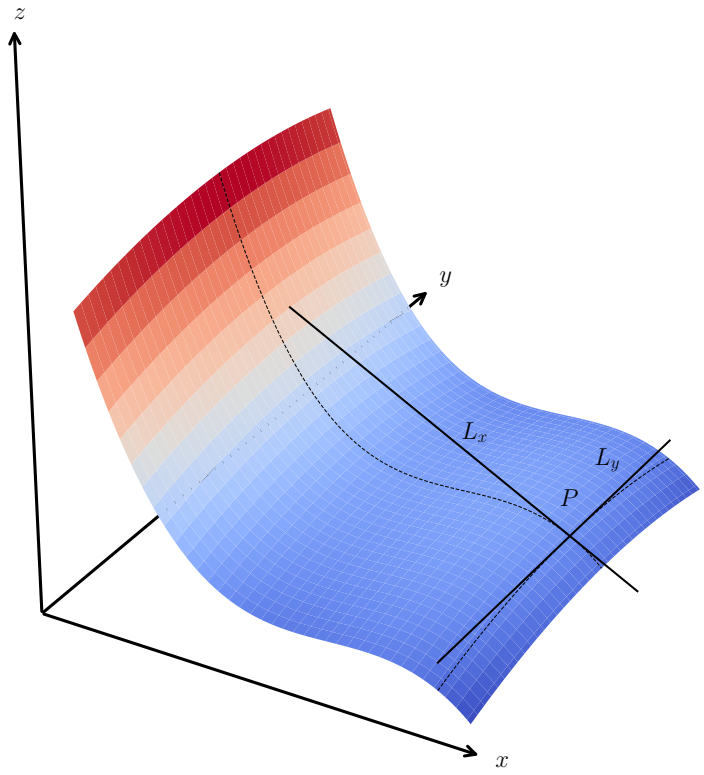

Fig. 2.5 In the figure you can see the 3-dimensional plot of a function $g(x, y)$. The line indicated with L_x is the tangent to the surface at point P along the x-axis, and the line indicated with L_y is the tangent to the surface at point P along the y-axis.

that is a vector with the partial derivatives along the directions x_1, x_2, and so on. We will limit ourselves here to give only the definition and you will see how it is used in this book more extensively.

2.7 An Introduction to Optimisation for Neural Networks

Optimisation is a very complex and wide topic. Since this book is focused on deep learning, it is very useful to start with an (very) intuitive understanding of what a neural network (NN) is and how it learns. For this introductory section, we will

consider only what is called supervised learning[7]. Let us suppose we have a dataset of M tuples (x_i, y_i) with $i = 1, \ldots, M$. The x_i, called input observations or simply inputs, can be anything from images to multidimensional arrays, to one-dimensional arrays or even simple numbers. The outputs y_i (also called target variables or sometime labels) can be multidimensional arrays, numbers (for example the probability of the input observation x_i of being of a specific class) or even images. In the most basic formulation, a NN is a mathematical function (sometime called *network function*) that takes some kind of input (typically multi-dimensional) x_i (the subscript i indicates that we have a dataset of input observations at disposal, and we are now considering only the i^{th} one) and with it calculate some output $\hat{y}_i$. This function depends on a certain number N of parameters, that we will indicate with θ_i. We can write this mathematically as

$$\hat{y}_i \equiv f(\theta, x_i) \tag{2.36}$$

Where we have indicated the parameters in vector form $\theta = (\theta_1, \ldots, \theta_N) \in \mathbb{R}^N$. In Figure 2.6 you can see an intuitive diagram of the idea. The blob in the middle represents the network function f that maps the input x_i to the output. Naturally the output will depend on the parameters θ. The idea behind learning is to change the parameters until $\hat{y}_i$ is as close to y_i as possible. There are two very important undefined concepts in the last sentence: firstly what "close" means, and secondly how can we update the parameters in an intelligent way to make $\hat{y}_i$ and y_i "close". We will answer those exact two questions in depth in this book.

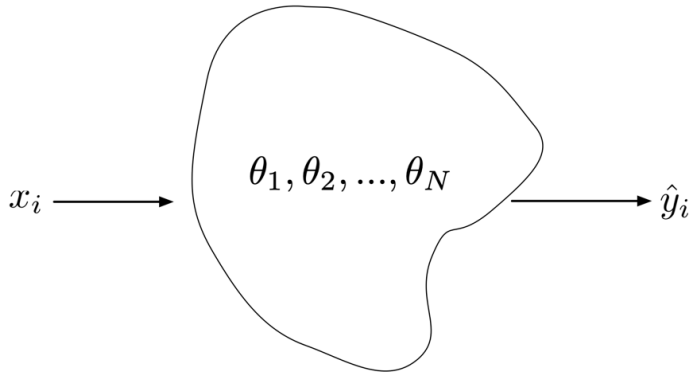

Fig. 2.6 a diagram that gives an intuitive understanding of what is a neural network. x_i are the inputs (for $i = 1, \ldots, n$), θ_i are numbers (or parameters) (for $i = 1, \ldots, N$), and $\hat{y}$ is the output of the network. The network itself is the depicted intuitively as the irregular shape in the middle of the figure.

[7] Supervised learning (SL) is the machine learning task of learning a function that maps an input to an output based on example input-output pairs (Source: Wikipedia).

To summarize a neural network is nothing else than a mathematical function that depends on a set of parameters that are tuned, hopefully in some smart way, to make the network output as close as possible to some expected output. The concept of “close" is not defined here but for the purposes of this section an intuitive understanding will be perfectly enough. At the end of this book the student will have a much more complete understanding of its meaning.

2.7.1 first definition of learning

Let us now gives a more mathematical formulation of “learning" in the context of neural networks. For notation simplicity let’s assume that each input is a mono-dimensional array $x_i \in \mathbb{R}^n$ with $i = 1, \ldots, M$. In the same way we will assume that the output will be a mono-dimensional array $\hat{y}_i \in \mathbb{R}^k$ with k some integer. We will assume to have a set of M input observations with expected target variables $y_i \in \mathbb{R}^k$. We also assume that we have a mathematical function $L(\hat{y}, y) = L(f(\theta, \hat{y}), y)$ called **loss function**, where we have used the vector notation $y = (y_1, \ldots, y_M)$, $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_M)$ and $\theta = (\theta_1, \ldots, \theta_N)$. This function will be a measure of how “close" expected (y) and predicted ($\hat{y}$) values are, given specific values of the parameters θ_i. We will not define yet how this function may be looking like as this is not relevant for this discussion yet. Let us summarize the notation we have defined so far.

- $x_i \in \mathbb{R}^n$: input observations (for this discussion we will assume that they are a mono-dimensional array of dimension $n \in \mathbb{N}$). Examples could be age, weight and height of a person, gray level values of pixels in an image and so on.
- $y_i \in \mathbb{R}^k$: target variables (what we would like the neural network to predict). Examples could be class of an image, what movie to suggest to a specific viewer, the translated version of a sentence in a different language and so on.
- $f(\theta, x_i)$: network function. This function will be built with neural networks and will depend on the specific architecture used (feed-forward, convolutional, recurrent, etc.).
- $\theta = (\theta_1, \ldots, \theta_N)$: a set of real numbers, also called parameters or weights.
- $L(\hat{y}, y) = L(f(\theta, \hat{y}), y)$: loss or cost function. This function is a measure of how “close" y and $\hat{y}$ are. Or in other words how good the neural network’s prediction are.

Those are the fundamentals elements that we will need to understand the basics of learning with neural networks.

Info ★ **Assumption in the formulation**

For the student that already have some experience with neural networks it is important to discuss one important assumption that have silently been made. Note that skipping this short section in a first reading of this book will not impact the understanding of the rest. In case you don’t understand the points

discussed here feel free to skip this part and come back to it later. The most important assumption here can be found in how the loss function $L(\hat{y}, y)$ is written. In fact, as written, it is a function of all the M components of the two vectors $\hat{y}$ and y and this translates in **not** using any mini-batch during the training. The assumption here is that we will measure how good the network's predictions are by considering **all** the inputs and outputs simultaneously. This assumption will be lifted in the following sections and chapters and discussed at length. The experienced reader may notice that this will lead to advanced optimization techniques as stochastic gradient descent and the concept of mini-batch. Using all the M components of the two vectors $\hat{y}$ and y makes the learning generally slower, although in some situations more stable.

2.7.2 definition of learning for neural networks

With the notation defined previously we can now give a formal definition of learning in the context of neural networks.

Definition 2.4 Given a set of tuples (x_i, y_i) with $i = 1, \dots M$, a mathematical function $f(\theta, \hat{y})$ (the network function) and a function (the Loss function) $L(\hat{y}, y) = L(f(\theta, \hat{y}), y)$ the problem of *learning* is equivalent to minimize the loss function with respect to the parameters θ. Or in mathematical notation finding the θ^*

$$\theta^* = \arg\min_{\theta \in \mathbb{R}^N} L(f(\theta, \hat{y}), y) \tag{2.37}$$

The typically used term for *learning* is training and that is the one we will use in this book. Basically, training a neural network is nothing else than minimizing a very complicated function that depends on a very large number of parameters (sometime billions). This presents very difficult technical and mathematical challenges that we will discuss at length in this book. But for now, this understanding will be sufficient to start understanding how we can tackle this problem.

In what follows we will discuss how to solve the problem of minimizing a function in general and discuss the fundamentals theoretical concepts that are necessary to understand more advanced topics. Note that the problem of minimizing a function is called an *optimization problem.*

2.7.3 Constrained vs. unconstrained optimization

The problem of minimizing a function as described in the previous section is what is called an unconstrained optimization problem. The problem of minimizing a function can be generalized by adding constraints to the problem. This can be formulated in

the following way: we want to minimize a generic function $g(x)$ subject to a set of constraints

$$\begin{cases} c_i(x) = 0, \ i = 1, \ldots, C_1 \text{ with } C_1 \in \mathbb{N} \\ q_i(x) \geq 0, \ i = 1, \ldots, C_2 \text{ with } C_2 \in \mathbb{N} \end{cases} \tag{2.38}$$

Where $c_i(x)$ and $q_i(x)$ are constraint functions that define some equations and inequalities that needs to be satisfied. In the context of neural networks, you may have the constraint that the output (suppose for a moment that $\hat{y}_i$ is simply a number) must lie in the interval $[0, 1]$. Or maybe that it must be always greater than zero or smaller than a certain value. Or another typical constraint that we will encounter is when you want the network to output only a finite number of outputs, for example in a classification problem.

Let's make an example. Let's suppose that we want our network output to be $\hat{y}_i \in [0, 1]$. Our learning problem could be formulated as

$$\min_{\theta \in \mathbb{R}^N} L(f(\theta, x), y) \text{ subject to } f(\theta, x_i) \in [0, 1], \ i = 1, \ldots, M \tag{2.39}$$

Or even more generally

$$\min_{\theta \in \mathbb{R}^N} L(f(\theta, x), y) \text{ subject to } f(\theta, x) \in [0, 1] \ \forall \ \theta, x \tag{2.40}$$

This is clearly a *constrained optimization* problem. When dealing with neural networks this problem is typically reformulated by designing the neural network in such a way that the constraint is automatically satisfied, and learning is brought back to an unconstrained optimization problem.

Info ★ Reducing a constrained to an unconstrained optimization problem

The student may be confused by the previous section and wonder how constraints can be integrated in the network architecture design. This happens typically in the output layer of the network. For example, in the examples discussed in the previous section, to ensure that $f(\theta, \hat{y}_\mathbf{i}) \in [0, 1] \, i = 1, \ldots, M$ it is enough to use the sigmoid function $\sigma(s)$ as activation function for the output neuron[a]. This will guarantee that the network output will always be between 0 and 1 since the sigmoid function maps any real number to the open interval $(0, 1)$. If the output of the neural network should always be 0 or greater one could use the ReLU activation function for the output neuron. Building constraint into the network architecture is extremely useful and it typically makes the learning much more efficient. Constraints come typically from a deep knowledge of the data and the problem you are trying to solve. It pays off to find out as many constraints as possible and trying to build them into the network architecture.

Another example of a constrained optimization problem is when you have a classification problem with k classes. Typically, you want your network to output k real numbers p_i with $i = 1, \ldots, k$, where each p_i could be interpreted as the probability of the input observation of being in a specific class. If we want to interpret the p_i as probability the following equation must be satisfied

$$\sum_{i=1}^{k} p_i = 1 \tag{2.41}$$

This is realized by having k neurons in the output layer and use for them the *softmax* activation function. This step reframes the problem into an unconstrained optimization problem since the previous equation will be satisfied by the network architecture. If you don't know what is the softmax activation function, don't fret. We will discuss it in the following chapters but keep this example in mind as it is the key to any classification problem with neural networks.

[a] If you do not understand what this means, come back to this section after having studied the components of neural networks.

2.7.4 Absolute and local minima of a function

Many numerical algorithms that minimize a function are, by design, only able to find what is called a "local" minimum, or in other words a point x_0 at which the function to minimize is smaller than at all other points in any *close* vicinity of x_0. Mathematically speaking x_0 is a local minimum of f if the following is satisfied (in a one-dimensional case)

Definition 2.5 The point $x = x_0$ is called a local minimum of a function $f : \mathbb{R} \to \mathbb{R}$ if

$$\exists \eta \in \mathbb{R} \text{ such that } f(x) \leq f(x_0) \, \forall x \in [x_0 - \eta, x_0 + \eta] \tag{2.42}$$

In principle we would like to find the *global minimum* or in other words the point for which the function value is the smallest between all possible points. In the case of neural networks identifying if the minimum found is a local or a global minimum is impossible, due to the network function complexity and this is one (albeit not the only one) of the reasons because training large neural networks is such a challenging numerical problem. In the next chapters we will discuss at length what factors[8] may make finding the global minimum easier or more challenging.

[8] Factors include things like weight initialization, optimizer algorithm, optimizer parameters (as the learning rate) and so on.

2.8 Optimization algorithms

So far, we have discussed the idea that learning is nothing less than minimizing a specific function, but we have not discussed the issue of how this “minimizing" looks like. This is achieved with what is called an “optimization algorithm", whose goal is to find the location of the (hopefully) absolute minimum. Practically all unconstrained minimization algorithms require the choice of a starting point, that we will denote by x_0. In the example of neural networks this initial point would be the initial values of the weights. Typically, starting from x_0, the optimization algorithms will generate a sequence of iterates $\{x_k\}_{k=0}^{\infty}$ that hopefully will converge toward the global minimum. In all practical applications only a finite number of terms will be generated, since we cannot generate an infinite number of x_k of course. The sequence will stop when no progress can be made anymore (the value of x_k will not change anymore too much[9]) or a specific solution has been reached with sufficient accuracy. Usually, the rule to generate a new x_k will use information about the function f to be minimized and one or more previous values (often properly weighted) of the x_k. In general, there are two main strategies for optimization algorithms: line search and trust regions. Optimizers for neural networks use all a line search approach.

2.8.1 Line search and trust region

In the *line search* approach, the algorithm chooses a direction p_k and searches along this direction for a new value x_{k+1} when trying to minimize a generic function $L(x)$. In general, this approach, once a direction p_k has been chosen consists in solving

$$\min_{\alpha>0} L(x_k + \alpha p_k) \tag{2.43}$$

for each iteration. In other words, one would need to choose the optimal α along the direction p_k. In general, this cannot be solved exactly, thus in practical application (as we will see later) this approach is used by choosing a fixed α, or by reducing it in a way that is easy to calculate (independently of L). α is what is known as *learning rate* when you deal with neural networks and is one of the most important hyper-parameters[10] when training networks. After deciding on a value for α the new x_{k+1} is determined with the equation

$$x_{k+1} = x_k + \alpha p_k \tag{2.44}$$

In the *trust region* approach, the information available on L is used to build a model function m_k (typically quadratic in nature) that approximates f in a sufficiently small

[9] Don’t be annoyed by the intuitive formulation. We will discuss it better later on.

[10] A Hyper-parameter is a parameter that does not change during the training and is not related to the training data. Even if the learning rate will change according to some fixed strategy, is still called an hyper parameter since it does not change due to the training data.

region around x_k. Then this approximation is used to choose a new x_{k+1}. In this book we will not cover trust region approaches, but the interested reader can find a very complete introduction in *Numerical Optimization, 2nd edition* by J. Nocedal and S.J. Wright, published by Springer.

2.8.2 Steepest Descent

The most obvious, and the most used, search direction for line search methods is the steepest direction $p_k = -\nabla L(x_k)$. After all, this is the direction along which the function f decreases more rapidly. To prove it, we can use Taylor expansion[11] for $L(x_k + \alpha p)$ and try to determine along which direction the function decreases the most rapidly. We will stop at the first order and write

$$L(x_k + \alpha p) \approx L(x_k) + \alpha p^T \nabla L(x_k) \tag{2.45}$$

assuming that α is small enough. Our question (along which direction the function L decreases more rapidly?) can be formulated as solving

$$\min_p L(x_k + \alpha p) \text{ subject to } \|p\| = 1 \tag{2.46}$$

Where $\|p\| = 1$ is the norm of the vector p (or in other words $\|p\| = \left(p_1^2 + \ldots + p_n^2\right)$). Using the Taylor expansion and noting that $f(x_k)$ is a constant we simply must solve for

$$\min_p p^T \nabla L(x_k) \tag{2.47}$$

Always subject to $\|p\| = 1$. Now, indicating with θ the angle between the direction p and $\nabla L(x_k)$ we can write

$$p^T \nabla L(x_k) = \|p\| \, \|\nabla L(x_k)\| \cos\theta \tag{2.48}$$

And is easy to see that this is minimized when $\cos\theta = -1$, or in other words choosing

$$p = -\frac{\nabla L(x_k)}{\|\nabla L(x_k)\|} \tag{2.49}$$

as we claimed at the beginning.

To summarise, the steepest descent method is a line search method that search for a better approximation for the minimum along the direction of minus the gradient of the function for every step. This method is at the basis of what is known as **gradient descent algorithm**.

[11] In case you don't know what the Taylor expansion is, you can check Wikipedia to get an idea at https://en.wikipedia.org/wiki/Taylor_series. This is a fundamental tool that is used in calculus for various application and that you should learn.

There are of course other directions that may be used but for neural networks, but those can be neglected (unless you are active in research in optimization algorithms, in that case you probably don't need to read this book). Just to cite an example, possibly the most important is the Newton direction, that can be derived from the second order Taylor expansion of $L(x_k + p)$, but that requires to know the Hessian $\nabla^2 L(x)$.

2.8.3 The Gradient Descent Algorithm

The gradient descent (GD) optimizer finds x_{k+1} by using the gradient of the function L according to the formula

$$x_{k+1} = x_k - \alpha \nabla L(x_k). \tag{2.50}$$

Thus, the GD algorithm is simply a line search algorithm that search for better approximations along the steepest descent direction. We can make a simple one-dimensional example ($x \in \mathbb{R}$) and try the algorithm. Let's suppose we want to minimize the function

$$L(x) = x^2 \tag{2.51}$$

This has a clear minimum at $x = 0$ as this is a simple quadratic form. If you know how to find the minimum of a function with calculus is easy to see that

$$\frac{dL(x)}{dx} = 0 \tag{2.52}$$

implies that $2x = 0 \rightarrow x = 0$. This is indeed a minimum since

$$\frac{d^2 L(x)}{dx^2} = 2 > 0. \tag{2.53}$$

How the GD algorithm looks like in this case? The algorithm will generate a sequence of x_k by using the formula $x_{k+1} = x_k - 2\alpha x_k$ (remember we are trying to minimize $f(x) = x^2$). We need of course to choose an initial value x_0 and a learning rate α. For a first try, let us choose $x_0 = 1$ and $\alpha = 0.1$. The generated x_k sequence can be seen in Table 2.1. From Table 2.1 is evident how, albeit slowly, the GD algorithm converges toward the right answer $x = 0$. That sounds good right? What could go wrong? Not everything is so easy, and in the GD there is a marvellous hidden complexity. Let's re-write the formula that in this case is used to find to generate the sequence x_k:

$$x_{k+1} = x_k - 2\alpha x_k = x_k (1 - 2\alpha) \tag{2.54}$$

Consider for example the value $\alpha = 1$. In this case $x_{k+1} = -x_k$. It is easy to see that this generates an oscillating sequence that never converges. In fact, it is easy to calculate that $x_1 = -1$, $x_2 = 1$, $x_3 = -1$ and so on. An oscillating sequence will

k	x_k for $x_0 = 1$ and $\alpha = 0.1$
0	1
1	0.8
2	0.64
3	0.512
...	...
40	0.00013
...	...
500	10^{-49}

Table 2.1 The sequence x_k generated for the function $L(x) = x^2$ with the parameters $x_0 = 1$ and $\alpha = 0.1$.

always be generated for all values of $1 - 2\alpha < 0$, or for $\alpha > \frac{1}{2}$. In Figure 2.7 you can see a plot of the sequence x_k for various values of the parameter α. When α is

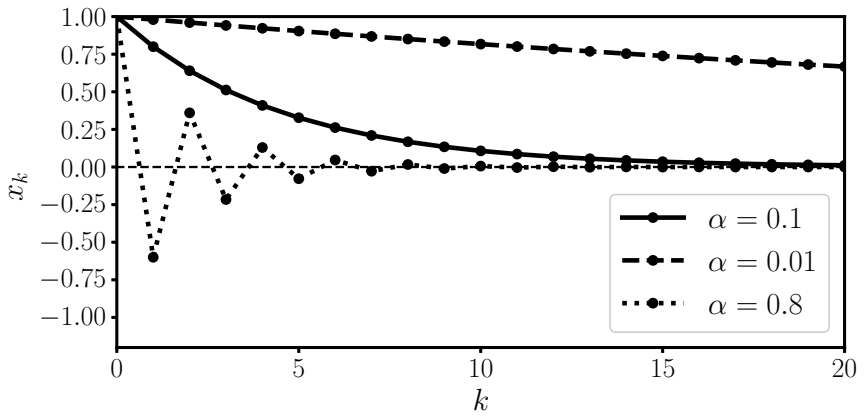

Fig. 2.7 The sequence x_k generated for the function $L(x) = x^2$ for $\alpha = 0.1, 0.01$ and 0.8.

small, the convergence is very slow (dashed line), and as we discussed, for a value $\alpha > \frac{1}{2}$ (dotted line) an oscillating sequence is generated. It is interesting to note how this oscillating sequence converges quite faster than the others. The value $\alpha = 1$ generates a sequence that does not converge but what happens for $\alpha > 1$? This is a very interesting case as it turns out that the sequence diverges (albeit oscillating from positive to negative values). In Figure 2.8 you can see the plot of the sequence for $\alpha = 1.01$.

From a numerical point of view it is easy to get NaN (if you are using Python) or errors. If you are trying neural networks and you get NaN for your loss function (for example) one possible reason may well be a learning rate that is too big.

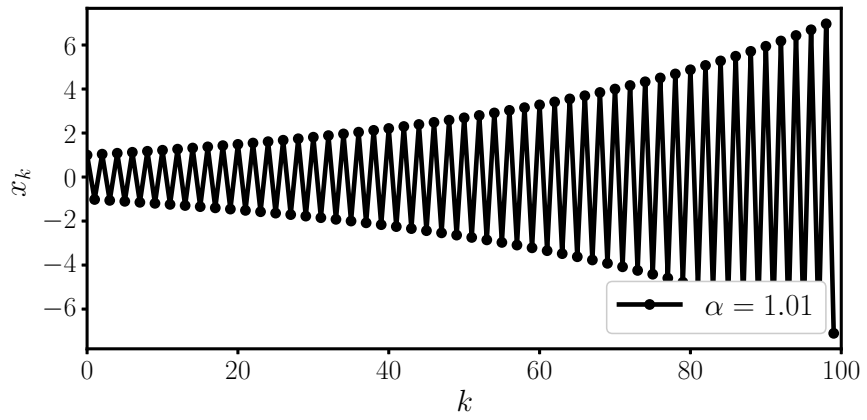

Fig. 2.8 The sequence x_k for the parameter $\alpha = 1.01$. The sequence oscillates from positive to negative values while diverging in absolute value.

Info The importance of the learning rate

The learning rate is possibly one of the most important hyper-parameters that you will have to decide on when training neural network. Choose it too small, and the training will be very slow, but choose it too big and the training will not converge! More advanced optimizers (as Adam for example) try to compensate this shortcoming by effectively varying the learning rate[a] dynamically but the initial value is still important.

[a] To be precise, as we will discuss next chapters, the Adam optimizer does not change the learning rate but uses a different algorithm that is very similar, but not the same, to the gradient descent and therefore sometimes it is said incorrectly that Adam updates the learning rate dynamically.

Now I must admit this is a quite trivial case. In fact, the formula for x_k can also be written explicitly as

$$x_k = x_0 \, (1 - 2\alpha)^k \tag{2.55}$$

And therefore, is easy to see that this sequence converges for $|1 - 2\alpha| < 1$ and diverges for $|1 - 2\alpha| > 1$. For $|1 - 2\alpha| = 1$ it stays at 1 and if $1 - 2\alpha = -1$ it oscillates between 1 and -1. Still, it is quite instructive to see how important the role of the learning rate is when using the gradient descent.

2.8.4 Choosing the right learning rate

You may be wondering how to choose the right α at this point. This is a good question but unfortunately there is no real precise answer, and some clarifications are in order. In all practical cases you will not use the plain gradient descent algorithm. Consider that, for example, in TensorFlow 2.X the gradient descent is not even available out of the box, due to its inefficiency. But in general, to check if the (in some cases only initial) learning rate is the optimal you can follow the steps, assuming you are trying to minimize a function $f(x)$:

1. You choose an initial learning rate. Typical values[12] are 10^{-2} or 10^{-3}.
2. You let your optimizer run for a certain number of iterations saving each time the $f(x_k)$
3. You plot the sequence $f(x_k)$. This sequence should show a convergent behaviour. From the plot you can get an idea if the learning is too small (slow convergence) or too big (divergence). For example, Figure 2.9 shows the sequence $f(x_k)$ for the example we discussed in the previous section. The Figure would tell me that using $\alpha = 0.01$ (dashed line) is very slow. Trying larger values for γ makes clear how convergence can be faster (continuous and dotted). With $\alpha = 0.1$ after 12-13 iterations you already have a good approximation of the minimum, while for $\alpha = 0.01$ you are still very far.

When training neural networks always check the behaviour of your Loss Function. This will give you important information on how the training process is going. This

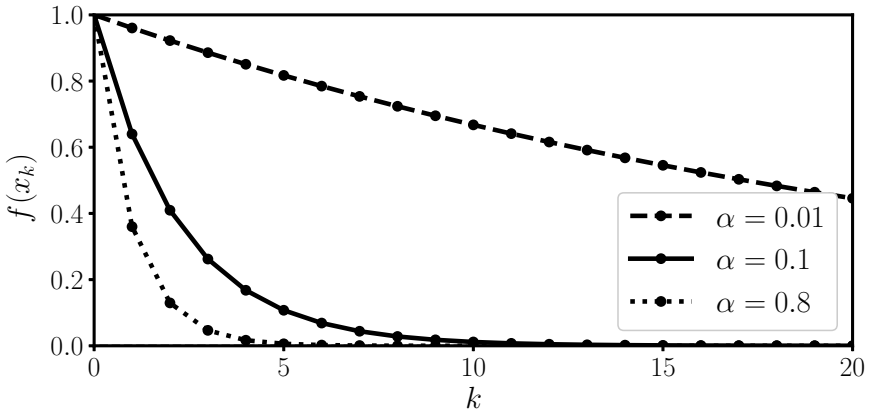

Fig. 2.9 The sequence $f(x_k)$ for the function $f(x) = x^2$ for various values of α.

is the reason why when training neural networks, it is important to always check the

[12] For example, the Adam optimizer in TensorFlow 2.X uses 0.001 as the standard learning rate, unless you specify otherwise.

behavior[13] of the loss function that you are trying to minimize. Never assume that your model is converging without checking the sequence $L(x_k)$.

2.8.5 Variations of GD

To understand variations of GD, the easiest way is to start with the loss function. As we mentioned at the beginning of the chapter, in the *problem of learning* section, our goal is to minimize the loss function $L(f(\theta, \hat{y}), y)$ where we have used the vector notation y=$(y_1, \ldots, y_M)$, $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_M)$ and $\theta = (\theta_1, \ldots, \theta_N)$. In other words, we have at our disposal M input tuples that we can use. In the plain version of GD, the loss function is written as

$$L(f(\theta, \hat{y}), y) = \frac{1}{M} \sum_{i=1}^{M} l_i(f(\theta, \hat{y}_i), y_i) \tag{2.56}$$

Where l_i is the loss function evaluated over **one single** observation. For example, we could have a one-dimensional regression problem where our loss function is the mean square error (MSE). In this case we would have

$$l_i(f(\theta, \hat{y}_i), y_i) = |f(\theta, \hat{y}_i) - y_i|^2 \tag{2.57}$$

And therefore

$$L(f(\theta, \hat{y}), y) = \frac{1}{M} \sum_{i=1}^{M} |f(\theta, \hat{y}_i) - y_i|^2 \tag{2.58}$$

That is the classical formula for the MSE that you may have already seen. In plain GD, we would use this formula to evaluate the gradient that we need to minimize L. Using all M observations have pros and cons.

Info **Advantages and disadvantages of plain gradient descent**

Advantages

- Plain GD shows a stable convergence behavior

Disadvantages

- Usually, this algorithm is implemented in such a way that all the dataset must be in memory, therefore it is computationally quite intensive
- This algorithm is typically very slow for very big datasets

[13] Tools as TensorBoard (from TensorFlow) have been built with exactly this problem in mind, to give a real time check on how the training is going.

Variations of the gradient descent are based on the idea of considering only **some** of the observations in the sum in the previous equation instead of all M. The two most important variations are called *mini-batch GD* (MBGD, where you consider a small number of observations $m < M$) and *Stochastic GD* (SGD, where you consider only one observation at a time). Let us look at both in detail starting with the MBGD.

2.8.5.1 Mini-batch GD

To clarify the idea behind the method, we can write the loss function as

$$L_m(f(\theta, \hat{y}), y) = \frac{1}{M} \sum_{i=1}^{m} |f(\theta, \hat{y}_i) - y_i|^2 \tag{2.59}$$

Where we have introduced $m \in \mathbb{N}$ with $m < M$ called here *batch size*. L_m is defined by summing over m observations sampled from the initial dataset.
Mini-batch GD is implemented according to the following algorithm:

1. A mini-batch size m is chosen. Typical values are 32, 64, 128 or 256 (note that the mini-batch size m does not have to be a power of 2, and could be any number as 137 or 17);
2. $N_b = \left\lfloor \frac{M}{m} \right\rfloor + 1$ subsets of observations are created[14] by sampling each time m observations from the initial dataset S without repetition. We will indicate them with $S_1, S_2, \ldots, S_{N_b}$. Note that in general if M is not a multiple of m the last batch, S_{N_b} may have a number of observations smaller than m;
3. The parameters θ are updated N_b times using the GD algorithm with the gradient of L_m evaluated over the observations in S_i for $i = 1, \ldots, N_b$;
4. Repeat point 3 until the desired result is achieved (for example the loss function does not vary that much anymore).

When training neural networks, you may have heard the term **epoch** instead of iteration. An epoch is *finished* after all the data has been used in the previous algorithm. Let us make an example. Suppose we have $M = 1000$ and we choose $m = 100$. The parameters θ will be updated using each time 100 input observations. After 10 iterations (M/m) the network will have used all M observations for its training. At this point it is said that one epoch is finished. One epoch in this example will consist of 10 parameters update (or 10 iterations). Here are advantages and disadvantages.

Info **Advantages and disadvantages of mini-batch gradient descent**

Advantages

[14] The symbol $\lfloor x \rfloor$ indicates the integer part of x.

- The model update frequency is higher than with plain gradient descent but lower than SGD (see below), therefore allowing for a more robust convergence than SGD.
- This method is computationally much more efficient than plain gradient descent or Stochastic GD since less calculations (as in SGD) and resources (as in Plain GD) are needed
- This variation is by far (as we will see later) the fastest of the three and the most used

Disadvantages

- The use of this variation introduces a new hyperparameter that needs to be tuned: the batch size (number of observations in the mini-batch)

We can define formally an **epoch** as

Definition 2.6 An epoch is *finished* after all the input data has been used to update the parameters of the neural network. Remember that in one epoch the parameters of the network may be updated many times.

2.8.5.2 Stochastic GD

SGD is also a very commonly used version of the GD, and it simply is the mini-batch version with $m = 1$. This means updating the parameters of the network by using one observation at a time for the loss function. This has of course also advantages and disadvantages.

Info **Advantages and disadvantages of Stochastic gradient descent**

Advantages

- The frequent updates allow an easy check on how the model learning is going (you don't need to wait until all the dataset has been considered).
- The convergence process is intrinsically noisy and that may help the model to avoid local minima when trying to find the absolute minimum of the cost function.

Disadvantages

- On large dataset this method is quite slow, since is very computationally intensive due to the continuous updates.
- The fact that the algorithm is noisy can make it hard for the algorithm to settle on a minimum for the cost function, and the convergence may be not so stable as expected.

2.8.6 How to choose the right mini-batch size

Now what is the right mini-batch size m? Typical values used by practitioners are of the order of 100 or less. For example, Google TensorFlow standard value (if you don't specify otherwise) is 32. Why this value? What is so special? To understand the reasons, we need to study the behaviour of MBGD for various choices of m. To make it resembling real cases, we will use the Zalando dataset [1]. You may have already seen it. It is a dataset that contains 70000 images of 10 types of clothing. The images are gray-level 28x28 pixel images. You can see three examples from the dataset in Figure 2.10. We will build a classifier[15] and check its convergence behaviour. I have

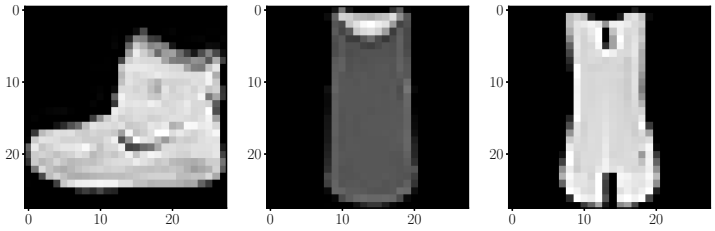

Fig. 2.10 Three images from the Zalando dataset. They are 28×28 pixel, gray level images. The dataset contains 70000 images of 10 types of clothing items.

trained the network for 10 epochs (remember what an epoch is?) on 60000 training images and I have measured the running time[16] needed, the reached value of the loss function and the accuracy at the end of the training. I have used the following values for the mini-batch size m: 5, 10, 100, 1000, 3000, 6000 and 60000 (effectively using all the training data, so no mini-batch). Note that while for $m = 10$ the required time is 2.34 min, when using $m = 1$ 19.18 minutes are needed for 10 epochs! In Figure 2.11 you can see the results of this study.

Let us see what Figure 2.11 tells us. When we use $m = 300$, the running time needed for 100 epochs is the lowest, but the loss function value reached is higher. Decreasing m decreases the reached loss function value quite rapidly until we reach the "elbow". Between $m = 200$ and $m = 50$ the behaviour changes. Decreasing m does not decrease the value of the loss function much, but the running time becomes larger and larger. So, when we reach the elbow decreasing m does not bring many

[15] For those of you interested, the results have been obtained with a neural network with two layers each having 128 neurons with the ReLu activation function, by using the Adam optimizer. Firstly, if you don't exactly what I am talking about you can simply skip those details. The discussion can be followed even without understanding the details of how the network is designed. Secondly, using Adam is only for practical reasons, as in TensorFlow the MBGD is not even available out of the box. But the conclusions continue to be valid.

[16] I have run these tests on a modern macbook pro from 2021, and that means an 2.3 GHz 8-Core Intel Core i9 without GPU.

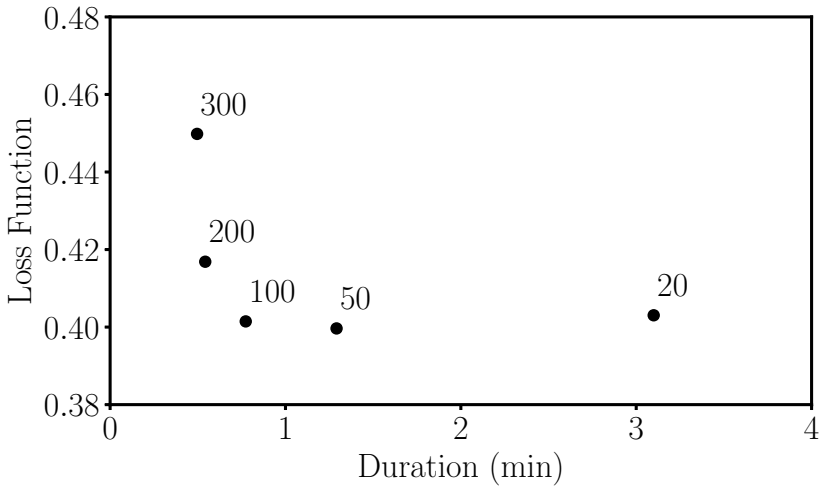

Fig. 2.11 A plot of the loss function value reached after 100 epochs on the MNIST dataset plotted vs. the running time needed. a indicates the accuracy reached.

advantages anymore. As you will notice, around the elbow m is of the order of 100. Figure 2.11 is an explanation why typical values for m are of the order of 100. Of course, the optimal value is dependent on the data and some testing is required, but in most cases a value around 100 or 50 is a very good starting point.

2.8.7 ★ SGD and Fractals

We have discussed in the previous sections how choosing the wrong learning rate can make the convergence slow or even diverge. But the discussion done was for a one-dimensional case and thus was very simple. In this section I want to show you how much complexity is hidden when using SGD. I want you to see how specific ranges of the learning rate make the convergence chaotic (in the mathematical sense of the word), thus showing one of the many hidden gems that you can find when dealing with optimization problems. Let us consider a problem[17] in which our M inputs $x^{[i]}$are bi-dimensional, in other words $x^{[i]} = (x_1^{[i]}, x_2^{[i]}) \in \mathbb{R}^2$. We call our target variables $y^{[i]}$. The optimization problem we are trying to solve involve minimizing the function

$$L = \frac{1}{2} \sum_{i=1}^{M} \left[f(x_1^{[i]}, x_2^{[i]}) - y^{[i]} \right]^2 \tag{2.60}$$

[17] This problem is an adaptation of the one described in Rojas, R. (2013). Neural networks: a systematic introduction. Springer Science & Business Media.

with

$$f(x_1^{[i]}, x_2^{[i]}) = w_1 x_1^{[i]} + w_2 x_2^{[i]} \tag{2.61}$$

A simple linear combination of the inputs. The problem is simple enough. We minimize the MSE (Mean Square Error) and try to find the best parameters w_1 and w_2 that minimize L. Let simply even more the problem. Consider $M = 3$ inputs. In particular, to make it more concrete, consider the following input matrix[18]

$$X = \begin{pmatrix} 0 & 1 \\ 1 & \frac{1}{2} \\ 1 & -\frac{1}{2} \end{pmatrix} \tag{2.62}$$

We write our labels also in matrix form

$$Y = \begin{pmatrix} 0 \\ 4 \\ 0 \end{pmatrix} \tag{2.63}$$

Note that what I will show you here is not dependent on the numerical values. You can reproduce the results with different values without problems. Let's first find the minimum of L exactly (since in this easy case we can do that). To do that we need simply to derive L and solve the two equations

$$\frac{\partial L}{\partial w_1} = 0; \frac{\partial L}{\partial w_2} = 0 \tag{2.64}$$

Calculations are boring but not overly complex. By solving the two above mentioned equations you will find that the minimum is at $x^* = (2, 4/3)$.

Exercises **Minimum of L**

Exercise
Solve the two equations

$$\begin{aligned} \frac{\partial L}{\partial w_1} &= 0 \\ \frac{\partial L}{\partial w_2} &= 0 \end{aligned} \tag{2.65}$$

And prove that L has its global minimum at $x^* = (2, 4/3)$.

To implement a SGD optimizer, the following algorithm can be followed:

1. Choose a learning rate α;
2. Choose a random value between from {1,2,3} and assign it to i
3. Update the parameters w_1, w_2 by using

[18] Note that all the inputs can be written in matrix form for simplicity.

$$l_i = \frac{1}{2}\left(f\left(x_1^{[i]}, x_2^{[i]}\right) - y^{[i]}\right)^2 \tag{2.66}$$

in other words we use l_i to calculate the derivatives to update the weights according to the Gradient Descent rule $w_j \rightarrow w_j - \alpha \partial l_i / \partial w_j$ for $j = 1, 2$; each time save the values w_1, w_2, for example in a python list;

4. Repeat points 2 and 3 a certain number of times N;

By following the previous algorithm, we can plot in the (w_1, w_2) space all the points we have obtained and saved in point 3 above. Those are the all the values that the two parameters w_1 and w_2 will assume during the optimization procedure. In Figure 2.12 you can see the result for $\gamma = 0.65$. The result is nothing short of amazing.

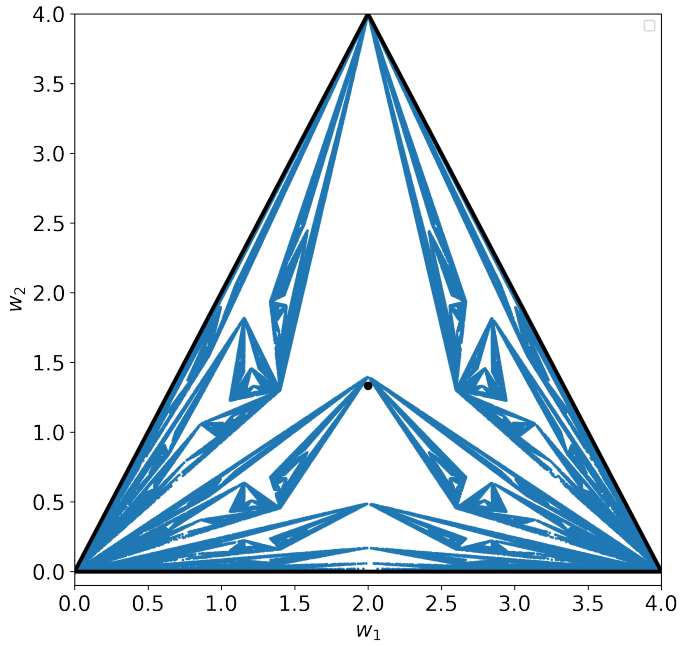

Fig. 2.12 each blue point is a tuple of values (w_1, w_2) that are generated by using SGD as described in the section for a value of the learning rate $\gamma = 0.65$. The plot has been obtained with $4 \cdot 10^5$ iterations.

Exercises **Find the equations of the delimiting lines of the triangle.**

This is difficult question, but can you derive the equations of the lines delimiting the triangle from the input matrix X?

It can be shown that what you see in Figure 2.12 is indeed a fractal. The mathematical proof is way beyond the scope of this book, but in case you are interested you can consult the beautiful book *Fractals Everywhere*, by M.F. Barnsley published by Dover. One of the main property of fractals, is that when you zoom in a detail, you will find the same structure that you observe at a larger scale. To convince you, at least intuitively, that this is what is happening Figure 2.13 shows you a detail of Figure 2.12. In the zoomed area you can observe the same kind of structure that you see at a larger scale.

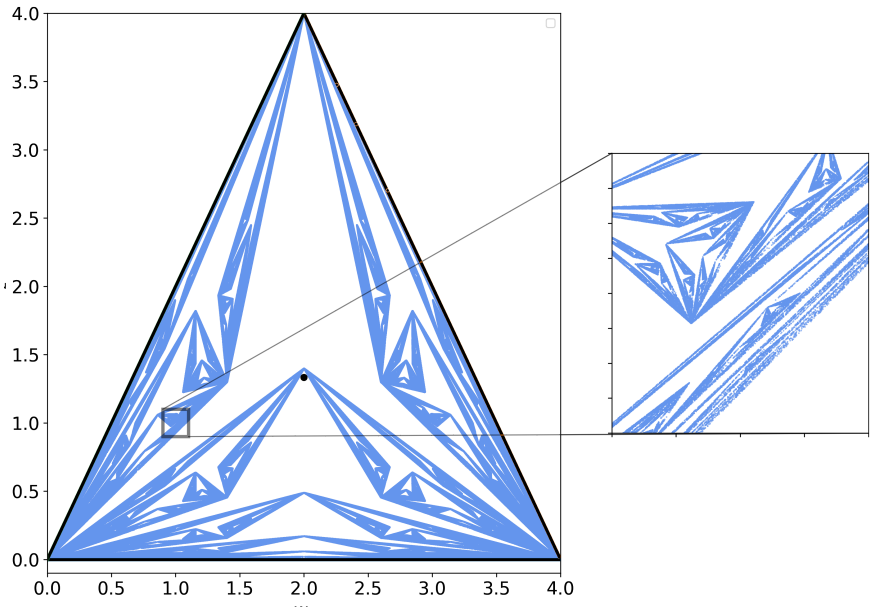

Fig. 2.13 A zoomed region that shows the fractal nature that the SGD algorithm can generate. This picture has been generated with a learning rate of $\gamma = 0.65$ and with 10^7 iterations. In the zoomed region you can clearly see the same kind of structure that you observe at a larger scale on the left. The zoomed region is less sharp than the one on the left since only a fraction of the 10^7 points happen to be in the small region zoomed in.

The particular structure of the fractal depends on the learning rate. In Figure 2.14 you can see the fractal structure for different learning rates, from 0.65 to 1.0. It is quite fascinating to see how the structure change, showing the great complexity that is hidden in using SGD. Now when using smaller learning rates, at a certain point

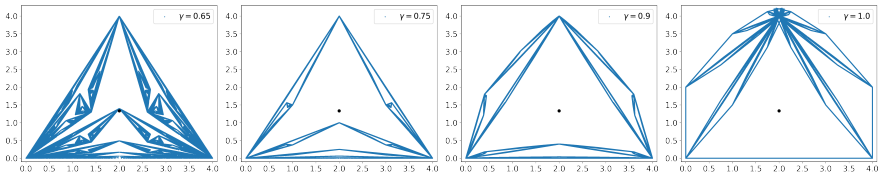

Fig. 2.14 fractal shapes obtained by SGD for different learning rates.

the fractal structure completely disappears quite suddenly, leaving an unstructured cloud of points, as you can see in Figure 2.15. The smaller the learning rate, the smaller the cloud of points.

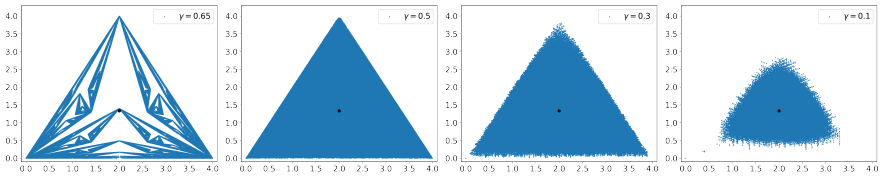

Fig. 2.15 Choosing smaller and smaller learning rates fractal structures completely disappear, leaving an unstructured cloud of points centered on the global minimum x^* of L.

Finally, by choosing a very small learning rate, for example $\gamma = 5 \cdot 10^{-4}$, SGD delivers the behavior we would expect, meaning the algorithm converges and remain close to the expected minimum. You can see how the plot looks like in Figure 2.16. the Gradient Descent algorithm, especially in its Stochastic version, has an incredible hidden complexity, even for a trivial case, as the one described in the previous section. This is the reason why training neural networks can be so difficult and tricky, and why choosing the right learning rate and optimizer is so important.

2.8.8 Conclusions

Now you should have all the ingredients to (at least on an intuitive level) understand what it means learning with neural networks. We have not yet spoken about how to build a neural network, except for the fact that it is a very complicated function f of the inputs that depend on a large number of parameters. We will go into a lot of details on how neurons work, how non-linearity is introduced and so much more.

Let us now summarize what we discussed in this chapter. To train neural networks we need the following major ingredients:

- A neural network architecture, namely a way of getting from an input x an answer $\hat{y}$ (remember the function f we discussed?) that can be tuned by changing a large number of parameters;

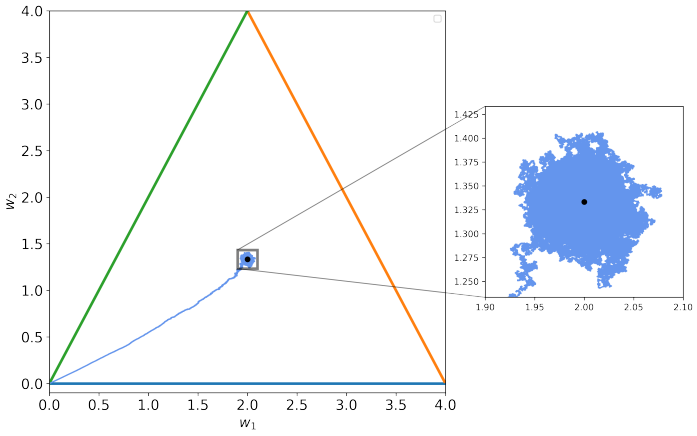

Fig. 2.16 by choosing a very small learning rate $\gamma = 5 \cdot 10^{-4}$ SGD will move into the direction of the expected global minimum x^* and remain in its vicinity, as it can be seen from the zoomed in region. Still, SGD continues to deliver points that remain around x^*, but never converge to it. The smaller the learning rate, the smaller the cloud of points around x^*.

- A set of input observations x_k (possibly a large number of them) with the expected values we want to predict (remember we are dealing with supervised learning);
- An optimizer, or in other words an algorithm that can find the best parameters of the network to get the outputs as close as possible to what you expect.

In this chapter we discussed those points in an almost intuitive way. It is important that you get the main idea behind what training neural networks means. In the next chapters I will discuss each of the three points in a lot of details and I will give you lot of examples to make the discussion as clear as possible. I will build on what we discuss here to bring you to a point where you can use the more advanced techniques for your projects.

References

1. Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.