University of Padua

DEPARTMENT OF MATHEMATICS
"TULLIO LEVI-CIVITA"

Master Degree in Mathematics

A Mathematical Approach to Neural Networks Optimization

Supervisor:
Prof. David
Barbato

Candidate:
Andrea Tosi
2029002

23 February 2024

Academic Year 2023/2024

Abstract

The main intent of the thesis is following a path that try to obtain an optimal Neural Network model. At each step we are going to give great importance to formalization of the mathematical aspects. The aim, in fact, is also to mathematically study the problems encounter and, using mathematics again, try to solve them.

The first part (first two chapters) is dedicated to introduce Neural Networks and the greater fields where they are contained. Moreover, here we understand what leads to their invention and how essential the optimization methods are in Machine Learning: learning in its essence means optimizing an error functional.

The earlier optimization algorithms are studied in the second part where we mainly talk about variations of the Gradient Descent Method. We exhibits some of their limits and strengths.

In the last part we present some results on the comparison between the previously shown algorithm. We end the thesis showing a modern method which somehow invert our process: rather than trying to solve a problem, we try to understand why a given solution is indeed working.

Contents

Notation

Acronyms

AI	Artificial Intelligence
GD	Gradient Descent
GOE	Gaussian Orthogonal Ensemble
KL	Kullback-Leibler
MLE	Maximum Likelihood Estimation
MSE	Mean Square Error
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine

Graphical Notation

x	Italic for scalars
$\boldsymbol{x}$	Bold italic for vectors
$\boldsymbol{0}$	Origin of $\mathbb{R}^n$, the vectorial zero: the n-tuple of only zeros
$\boldsymbol{A}$	Uppercase bold italic for matrices
$\boldsymbol{A}_{i,:}$	The vector representing the i-th row of the matrix A
$\boldsymbol{A}_{:,i}$	The vector representing the i-th column of the matrix A
x	Plain text for random variables
$\mathbf{x}$	Bold text for random vectors

Spaces and Subsets

$\mathbb{R}^n$	The cartesian product between n copies of the set of real numbers
$B(C, r)$	The open ball $\subset \mathbb{R}^n$ with center C and radius $r > 0$: $\{\, \boldsymbol{x} \in \mathbb{R}^n \mid \|\boldsymbol{x}\| < r \,\}$
$B(C, r]$	The closed ball $\subset \mathbb{R}^n$ with center C and radius $r > 0$: $\{\, \boldsymbol{x} \in \mathbb{R}^n \mid \|\boldsymbol{x}\| \leq r \,\}$

$C(\mathbb{R}^n)$	The set of continuous functions from $\mathbb{R}^n$ to $\mathbb{R}$
$C_c(\mathbb{R}^n)$	The set of continuous functions with compact support
$C_b(\mathbb{R}^n)$	The set of continuous bounded functions
$C^k(\mathbb{R}^n)$	The set of functions continously differentiable k-times
$C^\infty(\mathbb{R}^n)$	The set of infinitely differentiable functions, also called smooth functions
$C_c^\infty(\mathbb{R}^n)$	The set of infinitely differentiable functions with compact support: $C_c(\mathbb{R}^n) \cap C^\infty(\mathbb{R}^n)$
$L^p(X, \mu)$	The set of functions such that $\int_{\boldsymbol{x}\in X} \lvert f(\boldsymbol{x})\rvert^p d\mu < \infty$. It's a normed vector space

Functions and Distributions

ln	The natural logarithm is the logarithm with base e: $\ln = \log_e$
$\mathbb{1}_A(\boldsymbol{x})$	The indicator function of the set A: is equal to $\mathbb{1}_{\boldsymbol{x}\in A}$, meaning it's 1 when $\boldsymbol{x} \in A$ and 0 otherwise
$\eta(\boldsymbol{x})$	The standard mollifier $\eta(\boldsymbol{x}) = \begin{cases} C\, e^{\frac{1}{\lVert \boldsymbol{x}\rVert^2 - 1}} & \text{if } \lVert \boldsymbol{x}\rVert < 1 \\ 0 & \text{if } \lVert \boldsymbol{x}\rVert \geq 1 \end{cases}$
$\delta(\boldsymbol{x})$	The Dirac delta function is 0 for all $\boldsymbol{x} \neq \boldsymbol{0}$ and $\int_{\mathbb{R}^n} \delta(\boldsymbol{x}) d\boldsymbol{x} = 1$
$\mathcal{N}(\mu, \sigma^2)$	The Gaussian distribution with mean μ and variance σ^2. Its density is $\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$

Operators

$\lVert f\rVert_\infty$	The sup norm: $\lVert f\rVert_\infty = \sup\limits_{x\in\mathbb{R}} \lvert f(x)\rvert$
$\lVert f\rVert_{\infty,K}$	The sup norm of f restricted to K: $\lVert f\rVert_{\infty,K} = \lVert f\vert_K\rVert_\infty = \sup\limits_{x\in K} \lvert f(x)\rvert$
$\lVert f\rVert_p$	The norm of the $f \in L^p$: $\lVert f\rVert_p = \int_{\boldsymbol{x}\in X} \lvert f(\boldsymbol{x})\rvert^p d\mu$

Introduction

In the last years Artificial Intelligence has become a great topic of debate. Whether the discussion is on its real capacities, or on the details of its implementation or on a more ethical point of view, we can't argue that it's not an important modern subject. In fact, it is an actual part of our daily life: it's used in recommendation systems on Netflix or Amazon, it's used to automate machines or industrial processes and we hear about it when there are socially impactful event like the advent of ChatGPT.
Artificial Intelligence models express their real power (and even surpass humans in some cases) when we have an easy task in terms of allowed operations but hard to be formalized in an algorithm or set of rules. For instance we can consider image detection, for a human it's not difficult to know which object is watching in a photo but it's almost impossible to describe a deterministic algorithm that taken a photo it recognize every object in it. Another important example is when we want to automatically extract properties or patterns from raw data, it can be very difficult to a human to see the underline connection while it's often very easy for an Artificial Intelligence model. In particular, these examples and a lot of other ones are the cases where Machine Learning (a subfield of Artificial Intelligence) can help humans.

The increasing power of CPUs and GPUs is one of the reason why Artificial Intelligence it's growing in this years. However the data needed to make a Machine Learning model perform well are increasing, too. In fact, with the improvement of its capacity researchers try to exploit Machine Learning in increasingly complex task. Therefore an important aspect to consider when we are building a model is how efficient and fast it is. That's why we decide to analyse the aspects that can be optimize on a Machine Learning model. In particular we decide to study the case of Neural Networks which are one of the most (if not the most) used type of Machine Learning model.

The main idea and intention of this thesis is try to formalize with a mathematical approach most of the aspects we encounter. Unfortunately, in fact, a lot of works and articles related to this field overlook the mathematical part and they only describe some main intuitions without setting them properly. However it's interesting and challenging to try to understand why some phenomena observed empirically are indeed working. That's why the formalization in mathematical terms is an important part of this thesis. We decide not to proceed directly analysing what we can do to improve Neural Networks, instead we prefer to proceed step by step in some introductory chapters where we understand what mathematically means *"learning"* for a computer.

In the first chapter we briefly present the background which Neural Networks belong to. We explain what it is Artificial Intelligence and the main concepts of its primary subfield, the previously cited Machine Learning. We show the difference between supervised learning, unsupervised learning and reinforcement learning to better understand the different ways and tasks we could encounter. Then we end the chapter introducing and defining Neural Networks mathematically, but also presenting some important historical key steps.

In the second chapter we describe the training phase in all its main aspects. The training phase is the core and most time-consuming part of a Machine Learning model, it's during this step that the machine actually learns. We start the chapter describing what is the training phase in practice: after we fix a performance measure and a set of training examples, in the training phase we try to minimize a cost function defined on the training set assuming that this procedure increases another quantity (the performance). In order to connect the training error and the performances we have to make some assumption and construct the cost function in a way that links the two quantity with some theoretical bounds. We then describe two fundamental steps of the training phase: an example of a building method for the cost function and the basic iterative optimizing algorithm. In particular we show the maximum likelihood method presenting its information theory interpretation. After showing the benefits and drawbacks of the MLE method, we introduce the Gradient Descent algorithm. This last is built on a simple idea, thus is easy to implement. However there are a lot of problem which concerns the basic algorithm and we end the chapter describing them.

In the third chapter we show the first steps towards the optimization of a Neural Networks model. In fact, we analyse how we could improve the basic and raw methods presented in the previous chapter. For example a way to solve the excessive computational cost of each step in the Gradient Descent is to approximate the gradient with an unbiased estimator. In order to manage the learning rate, instead, we see adaptive learning algorithms that calibrate different size step in each axis direction.

In the last chapter we present some results to compare the previously shown algorithm. In particular, we introduce a convergence rate algorithm and a theorem on why SGD usually generalize better than Adam. At the end of the thesis we show a modern method to obtain an optimization algorithm. This recent approach essentially reverse our process: previously we had a problem and we try to solve it, now a Machine Learning model gives us the solution and we have to understand why it works.

Chapter 1

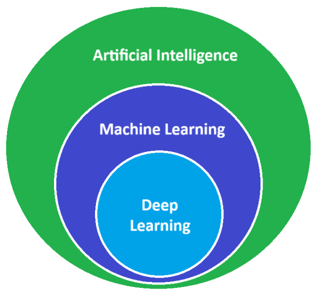
Basic Concepts

At the beginning of this thesis we want to have the most general view on Neural Networks and then in the following chapters we are going to focus on the aspects we want to study and try to improve. Thus, in this chapter we present the information needed to understand the background of Neural Networks. We want to have clear what means to construct a Machine Learning model and what are its components. Then in the second part of the chapter we are going to define Neural Networks and begin to deal with the more technical and mathematical aspects: we explain the notation and mathematical objects we are going to use in the next chapters.

1.1 Overview of AI & Machine Learning

We first want to show the context to which Neural Networks belong: they are one of the model studied by Machine Learning which is a sub-field of Artificial Intelligence.
In particular, within this section we briefly present the definitions and ideas of these subjects to better understand the series of concepts which brings to the notion of reproducing functions through Neural Networks.

Artificial Intelligence



We start with the greater field (AI) then narrowing down to the more specific ones (Machine and Deep Learning). The main idea of Artificial Intelligence is to recreate the human intelligence with machines and (later) softwares. We don't have a practical and unambiguous definition of intelligence that could easily adapt to machines. That's why many definitions of Artificial Intelligence were stated through the decades and they are usually very general. For example, in 1995 (Artificial Intelligence became an academic discipline nearly 40 years earlier, in 1956 with the "Dartmouth workshop") Stuart J. Russel and Peter Norvig published the book "Artificial Intelligence: A Modern Approach" where computer systems were divided by four goals or definitions of Artificial Intelligence "Human approach: systems that think like humans or systems that act like

humans. Ideal approach: systems that think rationally or systems that act rationally."
Another problem could be that we can say that there are different types of intelligence (as stated in the book "Frames of Mind: The Theory of Multiple Intelligences" written in 1983 by the Research Professor at Harvard University and developmental psychologist Howard Gardner). Then, instead of thinking about problem-solving or general intelligence (ability to solve an arbitrary problem, which becomes a long-term aim), the last consideration suggests to divide intelligence into capacities that researchers expect an human-like cognition system should have. Therefore scientists created more sub-fields each focused on different goals. These sub-fields intersect each other and they also tap into different external studies like philosophy, neuroscience, statistics and robotics.
Here there is a list of some of the capacities mentioned above:

- reasoning: like automated theorem proving or proof checking;
- planning and decision making;
- natural processing language: building software which read and respond;
- perception: includes speech recognition and computer vision;
- learning: Machine Learning is the one we are going to deepen.

Machine Learning

Since is one of the most studied and used sub-field, sometimes Machine Learning is understood as synonymous of Artificial Intelligence, but we have just seen that the latter is much wider. The former, in fact, is related "only" to the capacity of learning.
The core idea is to build programs which automatically improve through experience. Unlike the previous paragraphs, now we have a practical and definition of learning that applies well to machines. It was given by Tom M. Mitchell in his book "Machine Learning" (1997) [27]:

Definition 1.1. *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*

The experience is provided by the available data and it can be given at the start or it can be obtained through interactions with the environment (like in online learning). The task essentially describe the type of knowledge to be learned taking advantage of the data. The performance measure shows how well the model is learning and it's chosen depending on the task, keep in mind that it can affect the solution: different measures can lead to different solutions. These three aspects constitute a well-defined learning problem.

Example 1.1. *Task: identify the license plate number of a car in an image;*
Performance measure: percentage of license plate correctly classified;
Experience: a database of license plates with given numbers.

Remark 1.1. *Usually the tasks are associated to making prediction or to classifying objects in different labels.*

Now that we clearly know what is a learning problem, we can present the fundamental components of a Machine Learning model: training data, hypothesis space and learning algorithm.

Training data The training data are usually picked out from a larger set (the remaining samples are the test and validation sets), they must be selected so that they are representative of the prior distribution on the whole database (sometime considered the entire statistical population). Meaning we can't take an unbalanced set or training examples only of one "type", for instance we can't train an automated chess player only with winning games. The importance of every sample "type" (differentiation in types is derived by the data space description) must be proportional to its (prior) probability of happening. Another way is to directly sample the training set with or without replacement from the data space according to the prior probability. Moreover examples in the dataset can be described by different collections of features, we can decide which features to use and how to write them. In this way we are essentially setting the input space X, normally it is a vector space like $\mathbb{R}^n$. It seems easy and natural to construct, but actually studying efficient ways to represent input space is a big part of the first moments when a Machine Learning model is built. Deciding the features to keep, how to scale them, re-balancing unbalanced dataset and some other operations are steps of the so-called *Preprocessing technique.*

Example 1.2. *Suppose that the task is to classify fruits recognizing the different types and dividing them in labels (apple, orange, watermelon, etc.). The database is a set of fruits and the features that describe them could be weight, height, width, volume and colour. So every fruit is represented by a 5-dimensional vector with 4 numerical components and 1 alphabetical one.*

Hypothesis space The task of a learning problem can be thought as looking for a function $f^* : X \to Y$ where X and Y are the chosen input and output space, respectively. It can be a function that associates moves in a certain board state to the probabilities of winning, as output, if we are training an automated chess player, or in the Example 1.1 can be a function from the vector of every pixel's RGB (red-green-blue) values to an alphanumeric word of 7 symbols which represents the license plate number. There exist more type of functions that can be related to the same task, so it's better to choose the f^* type such that it's easy to evaluate and use, for example functions defined recursively are bad for this purpose since they usually are time consuming. Now this ideal target function or an approximation of it are searched in the set of all functions which can be implemented by the model. This set is selected a priori, it's called hypothesis space H and it contains functions $f : X \to Y$. Some of the most employed hypothesis spaces are the set of linear functions, the set of piecewise polynomial functions or the set of particular compositions of a fixed number of linear and non-linear functions (as we are going to see in the Neural Networks model). As we said, H must be chosen in such a way that $f^* \in H$ or it can be approximated by $f \in H$, but doing this choice a tradeoff must be taken into account. A complex and big H give us a better expressive representation since the approximation will be closer to f^*, however it could need a lot of training samples to get a satisfying accuracy or it could be computationally expansive (complex models have more parameter to learn).

Remark 1.2. *Remember that, as we did for X, we must determine a good representation for Y, too. This are the main examples: for binary classification $Y = \{0, 1\}$, for multiclass classification $Y = \{0, 1, ..., k\}$, for multilabel classification $Y = \{0, 1, ..., k\}^d$, for regression $Y = \mathbb{R}$ or for other tasks $Y = \mathbb{R}^d$.*

Learning algorithm The learning algorithm is the search algorithm in the hypothesis space H, it's the procedure that define how we find f^* or its approximation between the elements of H.

One bad example could be looping over all $f \in H$ and taking the one which best approximate f^* in some sense (for example, with respect to a distance between functions). Instead, the usual way is to set an error functional that evaluates the model's predictions f and then optimize it. Thus the mathematical optimization methods are a key point for Machine Learning. They are typically based on repeating the following procedure: evaluate the current prediction f_i with the error functional, autonomously update some parameters to optimize the functional and get another prediction f_{i+1}.

Remark 1.3. *The hypothesis space and the learning algorithm are restrictions on the target function approximation and its selection. The set of these assumptions is called Inductive Bias and it is divided in Representation Bias (restrictions on the hypothesis space H) and Search Bias (restrictions on how we search in H). We need these hypothesis otherwise we would have a huge amount of possibilities, but we have to choose them properly since different assumptions can give different results.*

Most of the Machine Learning models fall under three main categories called learning paradigms: supervised, unsupervised and reinforcement learning.

Supervised learning After we set the input and output space representations, X and Y, we write the training data as a labeled dataset, a set of pairs $\{\, (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \mid i = 1, .., m \,\}$. The "teacher" or "supervisor", which gives the name to this category, is the person or machine (like sensors) that tells us the correct answers (outputs) $\boldsymbol{y}^{(i)}$ of the inputs $\boldsymbol{x}^{(i)}$. This literally means we know $f^*(\boldsymbol{x}^{(i)}) = \boldsymbol{y}^{(i)}$ from the beginning, where f^* is the function we want to find or approximate. The model will then work on an induction reasoning: obtaining a general conclusion f in the hypothesis space H starting from these particular instances (the training examples). The error functional testing $f \in H$ depends on the training set because it computes a kind of distance between the $\boldsymbol{y}_i$ and $f(\boldsymbol{x}_i)$. An example could be the Mean Squared Error, MSE:

$$\text{MSE}_{train} = \frac{1}{m} \sum_{i=1}^{m} \Big(f(\boldsymbol{x}^{(i)}) - y^{(i)} \Big)^2 .$$

Changing the parameters' weights to optimize the functional means decreasing the difference between $f(\boldsymbol{x}^{(i)})$ and $f^*(\boldsymbol{x}^{(i)})$ so the error on training examples $(\boldsymbol{x}^{(i)}$. Thus, in a good supervised model the optimal $\overline{f}$ avoids the problem of underfitting (inability to fit the training set, so having an error on it above the chosen threshold).

This is the scheme supervised learning is based on, but obviously we have to deal with additional considerations when we are building the model. In fact, we must verify if the obtained $\overline{f}$ generalizes well: essentially we look at how $\overline{f}$ behave on new previously unseen inputs. As we already said, we pick only a part of the whole database as training set, the remaining part is convenient for this step: we take the test set $\{\, (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \mid i = m+1, .., m+M \,\}$ and we compute the performance of $\overline{f}$ on it usually with the same error functional type. Following the previous example:

$$\text{MSE}_{test} = \frac{1}{M} \sum_{j=m+1}^{m+M} \Big(f(\boldsymbol{x}^{(j)}) - y^{(j)} \Big)^2 .$$

We are addressing the overfitting problem: the model fits too much the training set but on new inputs it's not able to predict outputs that approximate f^* well. In the case that $\overline{f}$ doesn't generalize well, there are many ways to counter the problem such as adding more training data, reduce the model complexity or bootstrap aggregating which is a technique of ensemble methods. Models under the supervised learning category are linear regression, logistic regression, Perceptron, support vector machine (SVM) and decision tree.

Unsupervised learning In this case the training data aren't labeled: we only have the $\boldsymbol{x}^{(i)}$ (inputs features) but not the $\boldsymbol{y}^{(i)}$ (there is no supervisor to tell us them). The learning algorithm itself has to find a structure and patterns in the data. These last ones can be connected to an association rule which links variables in the dataset creating a sort of lattice between items. For example this protocol is used for products recommendation on Spotify, Netflix or ecommerce websites: it learns the probability of being interested in a product given the usage of other products (based on my habits and those of other consumers), it is the popular "If you bought this you will probably like ..." or "Other people who listen to this also listen to ...".
Otherwise the structure can rely on learned similarities and differences between training examples. Then it's generalized to the whole input domain and new samples are tested seeing if they show these similarities or differences. This type of procedure used to divide data in groups or subsets is called Clustering. An important point of this process is to associate a reasonable error functional with the similarity we want to get on data (such as proximity among points or belonging to a particular distribution) in a way that optimizing the functional means achieving more commonalities within every group and more differences between the different groups (minimize inter-class similarity and maximize intra-class similarity).
Some significant applications are in genetics (helped to map and graph genes), in density estimation in statistics and in dimensionality reduction (decreasing the number of features of a learning model keeping a good amount of information, for example with Principal Component Analysis). However, the most important related field is probably data mining which means, generally speaking, to process large quantities of data to extract useful information usually for businesses (it intersects also with database systems and statistics). Thus, unsupervised learning methods are widely used as data mining techniques.

Example 1.3. *Now we present a particular Clustering method: the k-means clustering. Fixed a positive integer k (hyper-parameter of the model) we want to divide data in k groups (clusters). The model parameters are the clusters centroids* $\{\, \boldsymbol{C}^{(j)} \in X$ (Input space) $\mid j = 1, ..., k \,\}$ *components and the variables* r_{ij} *with* $i = 1, .., m$ *and* $j = 1, ..., k$ *where* m *is the number of training examples. In particular* $r_{ij} = 1$ *when the i-th example* $\boldsymbol{x}^{(i)}$ *belongs to the* j*-th cluster and* $r_{ij} = 0$ *otherwise.*
The similarity metric is associated to the distances between points and clusters centroids: we assume that points are similar (meaning they belong to the same group) if they are closer to the same centroid than to other centroids. In other words inter-point distances of a cluster are smaller than distances to point outside of the cluster, since in this model points of the same cluster should be near. Thus, a possible error functional is

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{m} \sum_{j=1}^{k} r_{ij} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{C}^{(j)} \right\|^2 = \sum_{i=1}^{m} \sum_{j=1}^{k} r_{ij}\, d_{Eucl} \left(\boldsymbol{x}^{(i)}, \boldsymbol{C}^{(j)} \right)^2$$

where θ *is the parameters vector containing all* r_{ij} *and* $\boldsymbol{C}^{(j)}$ *components. In fact, if we keep all* r_{ij} *fixed and we optimize* J *with respect to the components of the centroids* $\boldsymbol{C}^{(j)}$*, then we minimize*

the distance between examples and their cluster's centroid. Instead, if we do the opposite fixing $\boldsymbol{C}^{(j)}$ and adjusting r_{ij}, we choose the nearest centroid for every example implicitly maximizing the distance to other centroids.
Here we use the square of the Euclidean distance, however we can choose a different one if it better applies to our case.

Reinforcement learning In the models under this category a program called agent is in a state S and it acts in a dynamic environment E performing an action A (admissible in the state S). In response, the environment E gives the agent the next state and a reward R as feedback. The latter can be both neutral, positive and negative depending on the learning problem task. For example, if we are training a self-driving vehicle in a track the rewards returned are positive if it arrives at the track's end, negative if it crashes and neutral in other cases. The agent's goal is to understand, by a trial and error strategy, which is the best sequence of action in order to maximize the cumulative reward. We can therefore say, in some way, that we work like in supervised learning but the supervision is delayed after actions. However, we notice that in this category the experience is not a fixed dataset.
Reinforcement learning can be used for learning to play board games (like chess, checkers or Go), to drive vehicles, to control robot or to optimize factories' production steps. It often shows its capacities with tasks which are easy to perform, at least at a basic level, and mechanical for human (playing board games or driving) but we want to make them automated or more efficient. The program AlphaGo beating Lee Sedol (one of the best Go players) proved that machines can achieve to master these tasks.

Remark 1.4. *Neural Networks can follow all three approaches but they are usually used in supervised learning. There exist also models which are a composition of multiple learning paradigms.*

We end this subsection on Machine Learning with a brief observation on when is better to use it. One of the most powerful properties of Machine Learning is that it discovers the algorithm on its own. This becomes really useful when coding the algorithm by human developers would be too hard or in general when the problem is difficult to formalize in order to get a multiple steps procedure. For instance, if we want to recognize faces in images, how can we write down an ad-hoc algorithm for it? Even if we find patterns of pixels to identify them, there are too many cases, too many facial expressions and too many angles to list all patterns.
It's very convenient to employ also when we look for hidden regularities in data (hard to notice by humans) and when the program has to adapt to changing conditions (as in recommendation systems).
It proves to be very useful also when we have noisy data or when the examples' features are incomplete. That's because regular algorithms usually need those features and they can be misdirected by noise. Instead, Machine Learning models can partially compensate the noise and they can approximate missing features.

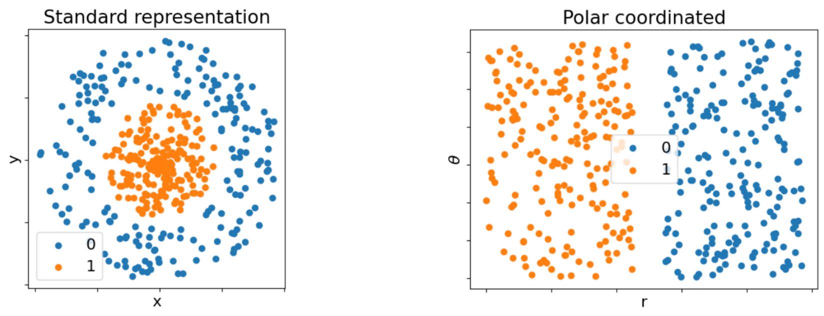
Deep Learning

It is a subfield of Machine Learning primarily built on Neural Networks, for this reason we give a brief presentation of this particular subfield among all possible ones.
As we said before, AI is very interested in problems which are intuitive for human but hard to

formalize. These types of problems are intuitive for humans since we have a huge amount of knowledge about the world. Hard-coding the necessary knowledge in machines is proven to be inefficient, then Deep Learning follows another characteristic of human brain: it builds complex notions using relations to simpler ones. Therefore, at the conceptual level, the "depth" in Deep Learning concerns how many layers of concepts link the starting one with the one we want to obtain. Theoretically, it's the depth of the graph showing how all the concepts connect to each other in a series of nested functions. In a more practical view, it refers to the depth of the Neural Network we are using in the model. However there isn't an established number of layers to classify a network as Deep or non-Deep. In the book "Deep Learning" [13] (by Ian Goodfellow, Yoshua Bengio and Aaron Courville) the authors give the following as an example of concepts' associations seen through Neural Networks: if we are processing an image, the first hidden layer could represent the edges, the second one corners and contours, the third one the object parts and the last one returns the object identity.

An important difference with classical Machine Learning models is that the latter's performance depends a lot on the representation of data (the choice of the input space X): some problems can be easily solved designing the right set of features.

Example 1.4. *Another example from "Deep Learning" [13]: we plot the same data with two different representations and we notice that in the second figure the two categories can be easily divided by a simple linear boundary (here an hyperplane, so a line).*



This suggests the idea of exploiting Machine Learning to learn not only the function from input space X to output space Y but also the representation of data itself. This approach is called Representation Learning and it aims to find a representation of the data with good properties. Deep Learning can be easily used for this purpose thanks to its basic idea: we can express complex representations in terms of other simpler ones. General Representation Learning usually leads to some advantages: it selects features faster, it obtains better performance than hand-designed representations and it allows models to adapt to new tasks. The main purpose, however, is eliminating some of the human interventions in data Preprocessing since choosing the right set of features to extract and how to write them can be very difficult when many factors of variation are high-level abstract features (as the accent in voice recognition or the changes of colours during night and day in object detection). Then, Deep Learning is used to automate features extraction by learning them directly from the raw data. This becomes very helpful when the data are in a large amount or they are unstructured (like images or text). A deep algorithm can easily divides photos of different animals since it can understand on its own which features (like ears, eyes, etc.) are most useful to differentiate each animal and what characterizes every animal. However, this good property is also one of the Deep Learning flaws: a deep model manages very well large amounts of data but it also needs large amounts to understand the connections between concepts and improve its accuracy. In Machine Learning, instead, we need less data since they are structured by Preprocessing and this structure can be exploited by the model.

1.2 Fundamentals of Neural Networks

In this section we give a presentation of Neural Networks going through the definition, some history and a few general notions.

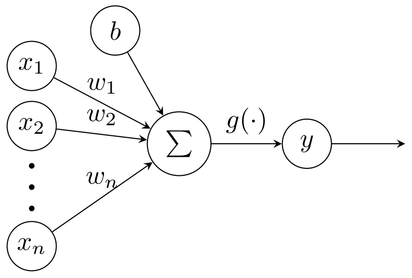
In a lot of the books they adapt the definition to suit the particular types studied or they implicitly give it by presenting an example or figure. Instead, in the "CIRP Encyclopedia of Production Engineering" [23] we can find a more general definition:

Definition 1.2. *"An artificial neural network (ANN), usually called neural network (NN), is a mathematical model or computational model [...]. A neural network consists of an interconnected group of artificial neurons, and it processes information using a connectionist approach to computation. In most cases, an ANN is an adaptive system that changes its structure based on external or internal information that flows through the network during the learning phase. [...]"*

A Neural Network is therefore composed of elementary units (the artificial neurons) that receive informations as inputs, perform some computation on them and send the output to linked units which follow the same procedure (this method is the previously named "connectionist approach"). In this way the Neural Network is able to carry out complex computations essentially composing simpler calculations. Moreover the bond strength among units can be learned thanks to the external inputs and even the ones between neurons.

Artificial Neurons

To understand how the model works overall, we first show the structure of a single artificial neuron (the base computational unit) on its own, also showing what it does in practice: it's essentially a function from the vector $\boldsymbol{x} \in \mathbb{R}^n$ (input of the neuron) to a scalar y. It first calculates a weighted sum of the $\boldsymbol{x}$ components also adding a term b called bias term, then it applies a non-linear function g to the result z eventually obtaining y. We are going to prove in proposition 1.1 the reason why non-linearity is required. Seeing this model as a single node Neural Network, the weights w_i and the variable b will be the parameters to be learned.

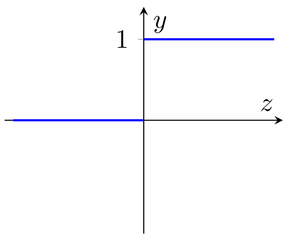


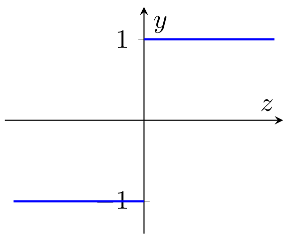
Notation 1.1. Since z is a weighted sum, we can write it using the scalar product between $\boldsymbol{x}$ and the vector of weights $\boldsymbol{w}$. Thus in the vectorial notation we get:

$$z = \sum_{j=1}^{n} w_j x_j + b = \boldsymbol{w}^\top \boldsymbol{x} + b \implies \text{the final result is } y = g(\boldsymbol{w}^\top \boldsymbol{x} + b)$$

The function g is called activation function and can be of different types. We list here some of the most common ones.

Step function These functions make the neurons assume only binary values. If we want the Neural Network to have a bit-like interpretation, we take the values 0 and 1. However we can also consider a small modification which is a well-known function: the sign.



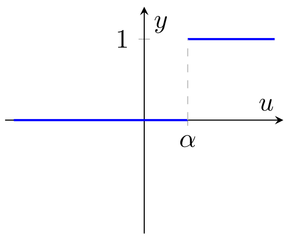


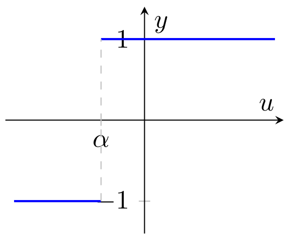
$$g(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \le 0 \end{cases} \qquad \text{or} \qquad g(z) = sgn(z) = \begin{cases} 1 & \text{if } z > 0 \\ -1 & \text{if } z \le 0 \end{cases}$$

Remark 1.5. *Let us set $\alpha = -b$ where b is the bias term of the node. Looking at the whole artificial neuron computation, we see that the previous examples implicitly give us threshold binary functions where α is indeed the threshold to overtake. In fact, defining $u = \boldsymbol{w}^\top \boldsymbol{x}$ we can rewrite the output as:*

$$g(u) = \begin{cases} 1 & \text{if } u > \alpha \\ 0 & \text{if } u \le \alpha \end{cases} \qquad \text{or} \qquad g(u) = sgn(u - \alpha) = \begin{cases} 1 & \text{if } u > \alpha \\ -1 & \text{if } u \le \alpha \end{cases}$$

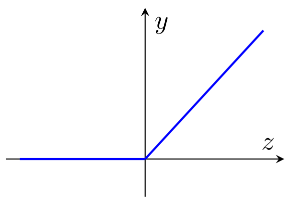
Therefore, the bias term is basically shifting the step functions graphs of $\alpha = -b$.





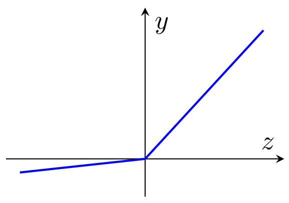
Actually this consideration applies not only to these cases: following the same procedure of defining u and rewriting the composition with the activation function, we can see that "shifting" is the general purpose of b.

ReLU It's a very useful function when we want very easy calculations in the non-binary case but still keeping non-linearity. It essentially is the positive part function and the name ReLU stands for Rectified Linear Unit.



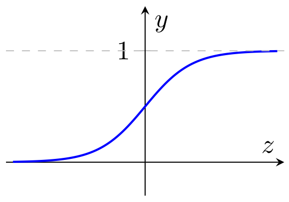
$$g(z) = z^+ = \max(0, z)$$

Remark 1.6. *It also exists a slight modification that doesn't completely cancel the negative part and it is called the Leaky ReLU:*



$$g(z) = z^+ = \max(0.1z, z)$$

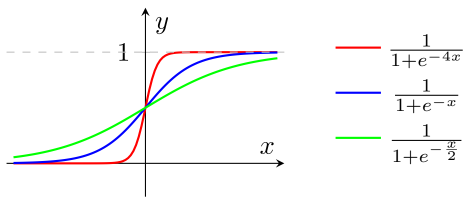
Logistic sigmoid It is usually denoted with $\sigma(z)$ and it can be used as a kind of the step function's approximation. It has a greater computational cost compared with the previous examples, however one of the positive aspects is the smoothness (it is infinitely differentiable). Another important property is the saturation of the logistic sigmoid function: it becomes almost flat for very positive or very negative arguments, in these parts of the domain it is insensitive to small changes in its input z.
Since it takes value in $(0, 1)$, it is helpful when in the model we interpret quantities as probabilities. With this viewpoint we can somehow see the saturation effect as: if an event has probability computed by the logistic sigmoid close to 1 or to 0, other "nearby events" have probabilities close to 1 or 0, respectively.



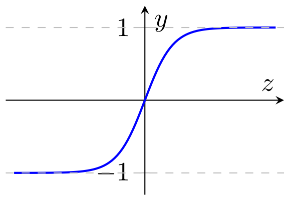
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Remark 1.7. *We already see in remark 1.5 that the bias term shifts the function. Instead, modifying weights in a sigmoid unit (an artificial neuron with a sigmoid activation function) basically means changing the slope of the function.*

For instance, we examine the one-dimensional framework, then we have $z = wx$. As we can see from the figure, a smaller weight $\left(\textit{in the picture } \frac{1}{2}\right)$ *on x decreases the slope and a bigger one* $(\textit{in the picture } 4)$ *increases it. When we talk about "smaller" and "bigger" we have to take into account that the minus sign is already embedded in the function and is not the sign of the weight itself.*



tanh It has a S-shape similar to the logistic sigmoid's one, in fact it is simply rescaled and shifted down: we can write $\tanh(z) = 2\sigma(2z) - 1$. Since it takes values in $(-1, 1)$, then, unlike the previous one, this is an approximation of the sign function.



$$g(z) = \tanh(z)$$

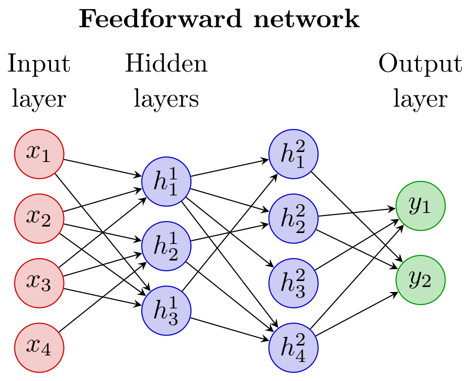
Neural Networks architecture

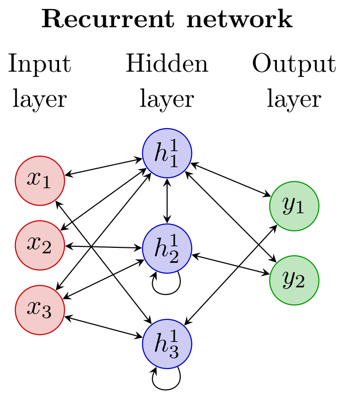
Now that we have presented the elementary components, we go back to general Neural Networks in their entirety. A relevant point of the previous definition is that it doesn't give restriction on the connections. Then, we can apply the structural (also called architectural) categorization of the book "Neural Networks for Identification, Prediction and Control" [32]. In terms of structure, the authors of the book separate Neural Networks into two main categories:

- **Feedforward networks:** The network is usually divided in one input layer, more hidden layers and one output layer. The input layer is the one receiving the data vector $\boldsymbol{x} \in X$, the output layer is the one giving the result vector $\boldsymbol{y} \in Y$ and the hidden layers are the ones carrying out the computations between the previous two. Every layer is connected only to the next one via uni-directional connections in the forward direction, starting from the input layer, going through the hidden layers and ending at the output layer. This gives the name "feedforward" to the model. We stress that with this definition nodes of the same layer can't communicate.
  The models in this category search for a static map from the input space X to the output space Y: at a fixed instant the output depends only on the input at that instant.
- **Recurrent networks:** In this case we have bi-directional Neural Networks. In fact, the signal can go in forward but also in backward direction: some nodes' outputs can be given as input to the previous layers or even to the same nodes. That's the reason why talking

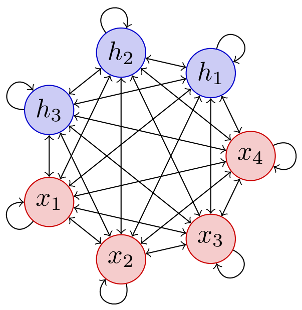
about layers for this category could lose its meaning, however it brings benefit when we want to split the computations in time steps.
It is more difficult to deal with this category since theoretically they could endlessly process the data. We have to set a limit on the iterations or to define the convergence of the global state.
They can exploit a dynamic memory since their internal state depends not only on the current input but also on the previous ones and on the earlier outputs. This becomes more useful when we want to process arbitrary sequences of inputs.

Here we show two examples (both not fully-connected) to visualize better the differences between them:





Example 1.5. *To understand how complicated the structure can get, we show a Fully Recurrent Network: all neurons are connected together in both directions, so outputs of all nodes are inputs to all nodes. It is the most general type of Neural Networks since other ones can be obtained setting desired weights to zero, basically getting the absence of those links. In particular the figure is an example of Boltzmann machine.*



Remark 1.8. *In a mathematical point of view, Neural Networks can be seen as graphs: consider the set of neurons as vertices* $V = \{ v_i \mid i = 1, ..., M \}$ *and the set of synaptic connections as oriented edges* $E = \{ (v_i, v_j) \mid i, j = 1, ..., M \}$ *attached to weights* w_{v_i,v_j}. *Then the Neural Network is the oriented weighted graph* $G(V, E)$. *This graph essentially shows how the used functions are composed in the network.*

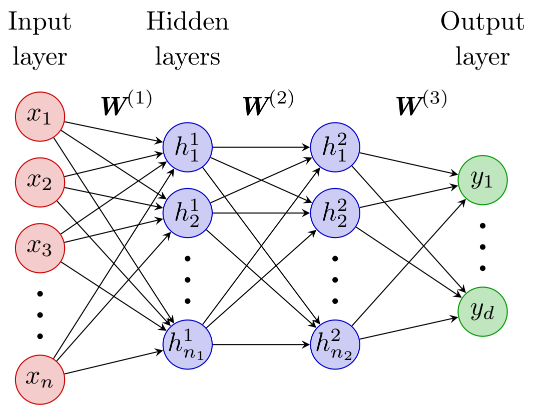
We notice that in feedforward networks the underlying graph can't contain cycles. While in recurrent networks the output of some nodes could influence subsequent inputs of the same nodes, then loops in the graph are possible.

Operations and their notation

In this subsection we introduce the mathematical notation and the operations of a feedforward network which is the category we are going to be focusing on.
First of all, when we talk about the architecture of a Neural Network we don't count the input layer (it's a sort of layer 0), for example in the next figure we have a 3-layer Neural Network. This is important to keep in mind since it influences how we name the layers: the k-th one is the k-th after the input layer and its number of neurons is n_k.
The activation function g could be different for each node. However it would have a high computational cost, that's why it is preferred to employ the same g for every node in the same layer or for all nodes in the network. The activation function of the k-th layer is denoted with $g^{(k)}$. This allows us to use a clean vectorial notation. We can, in fact, extend the function to vectors applying it componentwise: $g^{(k)}(\boldsymbol{z}) = \big(g^{(k)}(z_1), \dots, g^{(k)}(z_{n_k})\big)$.
As regards the parameters to be learned, the connection weight from the neuron i of the $(k-1)$-th layer to the neuron j of the k-th layer is called $w_{ij}^{(k)}$. Together they form the weights matrix $\boldsymbol{W}^{(k)} = \left(w_{ij}^{(k)}\right)_{i \in \text{nodes}^{(k-1)}, j \in \text{nodes}^{(k)}}$. Instead the bias term associated to the neuron j of the k-th layer is $b_j^{(k)}$.
The compositions and operations of the network can be easily shown recursively employing the notation just presented: the output of the neuron j of the k-th layer is denoted by $h_j^{(k)}$, then the output of the k-th (hidden) layer is the vector

$$\boldsymbol{h}^{(k)} = g^{(k)}\left(\boldsymbol{W}^{(k)\top}\boldsymbol{h}^{(k-1)} + \boldsymbol{b}^{(k)}\right). \tag{1.1}$$

In particular the input values are the component of the vector $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ and, if we have an N-layers network, the output values are the components of the vector $\boldsymbol{h}^{(N)} = \boldsymbol{y}$ (the last layer output).



Knowing the notation we can more easily prove in the following proposition why layers with linear activation function are useless.

Proposition 1.1. *Two layers with the first one linear are equivalent to a single layer.*

Proof. Suppose that the k-th layer is linear and its activation function is $g^{(k)}(\boldsymbol{z}) = \boldsymbol{A}^\top \boldsymbol{z} + \boldsymbol{c}$. Then

$$\begin{aligned}\boldsymbol{h}^{(k)} &= g^{(k)}\left(\boldsymbol{W}^{(k)\top}\boldsymbol{h}^{(k-1)} + \boldsymbol{b}^{(k)}\right) = \\ &= \boldsymbol{A}^\top \boldsymbol{W}^{(k)\top}\boldsymbol{h}^{(k-1)} + \boldsymbol{A}^\top \boldsymbol{b}^{(k)} + \boldsymbol{c} = \\ &= \boldsymbol{B}^\top \boldsymbol{h}^{(k-1)} + \boldsymbol{d}\end{aligned}$$

with $\boldsymbol{B} = \boldsymbol{W}^{(k)}\boldsymbol{A}$ and $\boldsymbol{d} = \boldsymbol{A}^\top \boldsymbol{b}^{(k)} + \boldsymbol{c}$. The composition with the next layer give us

$$\begin{aligned}\boldsymbol{h}^{(k+1)} &= g^{(k+1)}\left(\boldsymbol{W}^{(k+1)\top}\boldsymbol{h}^{(k)} + \boldsymbol{b}^{(k+1)}\right) = \\ &= g^{(k+1)}\left(\boldsymbol{W}^{(k+1)\top}\boldsymbol{B}^\top \boldsymbol{h}^{(k-1)} + \boldsymbol{W}^{(k+1)\top}\boldsymbol{d} + \boldsymbol{b}^{(k+1)}\right)\end{aligned}$$

We conclude taking $\widetilde{\boldsymbol{W}} = \boldsymbol{B}\boldsymbol{W}^{(k+1)}$ and $\widetilde{\boldsymbol{b}} = \boldsymbol{W}^{(k+1)\top}\boldsymbol{d} + \boldsymbol{b}^{(k+1)}$, then

$$\boldsymbol{h}^{(k+1)} = g^{(k+1)}\left(\widetilde{\boldsymbol{W}}^\top \boldsymbol{h}^{(k-1)} + \widetilde{\boldsymbol{b}}\right)$$

which represents only one layer from $\boldsymbol{h}^{(k-1)}$ to $\boldsymbol{h}^{(k+1)}$ with weights matrix $\widetilde{\boldsymbol{W}}$ and bias term $\widetilde{\boldsymbol{b}}$. □

The point of the proof is that, if the activation function is linear, going through the two layers we are making the composition of three linear functions which still give us a linear function.

Inductive Bias in Neural Networks

Here we recall the general Machine Learning notions explained in the remark 1.3. We apply those concepts to our case of study (the Neural Networks) to understand them better.
We first show that theoretically there is no Representation Bias if we want to learn a continuous function $f^* : X = \mathbb{R}^n \to Y = \mathbb{R}$. Indeed, on compact sets it can be approximated as much as we want by a Neural Network with only one hidden layer sufficiently large in terms of number of neurons. This result is proven in the article "Approximation Theory of the MLP model in Neural Networks" [33] by Allan Pinkus. The author exploits, in particular, networks composed by one hidden layer of r nodes with an activation function g and a scalar output layer without activation function and bias term. The functions associated to this type of networks can be written as

$$f(\boldsymbol{x}) = y = \sum_{j=1}^{r} c_j g\left(\boldsymbol{w}_j^\top \boldsymbol{x} + b_i\right)$$

with $\boldsymbol{x} \in \mathbb{R}^n$. In our previous notation $c_j = w_{j1}^{(2)}$ and $\boldsymbol{w}_j = \boldsymbol{W}_{:,j}^{(1)}$ which is the i-th column of the first weights matrix. All these functions belong to the function space $\mathcal{M}(g) = \operatorname{span}\left\{ l(\boldsymbol{x}) = g\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right) \mid \boldsymbol{w} \in \mathbb{R}^n, b \in \mathbb{R} \right\}$ and actually they completely form it. Then the formal statement of the result is given in the article by the following theorem:

Theorem 1.1 (Universal Approximation Theorem)**.** *Let $n \in \mathbb{N}$ and $g \in C(\mathbb{R})$, a continuous function from $\mathbb{R}$ to $\mathbb{R}$. Then g is not polynomial if and only if $\mathcal{M}(g)$ is dense in $C(\mathbb{R}^n)$ (the set of continuous function from $\mathbb{R}^n$ to $\mathbb{R}$), in the topology of uniform convergence on compact sets.*

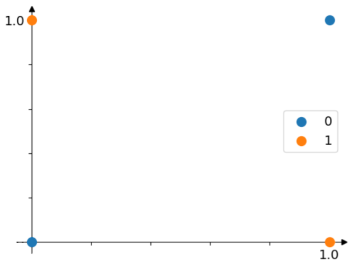
In other worlds, if $g \in C(\mathbb{R})$ is not polynomial, for every $f^* \in C(\mathbb{R}^n)$, every compact subset $K \subset \mathbb{R}^n$ and every $\varepsilon > 0$ there exists a function $h \in \mathcal{M}(g)$ (so a Neural Network) such that $\|f^* - f\|_{\infty,K} < \varepsilon$. This means that, if we use a continuous and not polynomial activation function g (like ReLU or logistic sigmoid functions), a 2-layers Neural Network where the hidden layer has infinitely many neurons can approximate every continuous function on compact sets. However, we can't implement networks with an arbitrary number of neurons, so actually there exists a Representation Bias. The hypothesis space H is, in fact, the set of all possible compositions of functions obtainable by changing weights of a model with fixed number of layers, number of neurons per layer and activation functions. Otherwise the book "Understanding of Machine Learning: From Theory to Algorithm" [37] follows the same graph prospective of remark 1.8 describing H as the set of all Neural Networks that share the same underlying oriented graph structure and vary in the edges' weights.
There can exist a Search Bias, for example, because we choose to employ the Gradient Descent or one of its variations and we can visit only those configurations that are possible to reach with that particular algorithm. Otherwise the Search Bias can occur since we are not guaranteed to get the best parameter (it often depends on the initial parameters and we are going to discuss later the local minima problem).

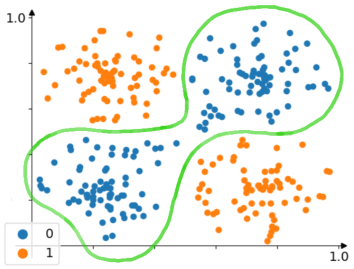
History of Neural Networks

We highlight the crucial moments of the Neural Networks development. This field went through different phases alternating enthusiasm and popularity decline. In particular the authors of the book "Deep Learning" [13] divide the history in three main waves: the first one identified by the name "cybernetics" in the 1940s-1960s, the second one following a connectionist approach in the 1980s-1990s and the last one concerning Deep Learning starting in 2006.
Actually we can interpret Linear Regression as the earliest example of Neural Network: used by Legendre and Gauss at the beginning of 1800 for the prediction of planetary movement, we can see it as a network composed by only one neuron without activation function (1-layer of one node). The inputs and weights produce the output $y = f(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{x}$ and we update the weights trying to minimize the mean square errors between given results and the computed values.

First wave **1943:** In the article "A logical calculus of the ideas immanent in nervous activity" [25] McCulloch and Pitts propose a mathematical model of how biological system process information: the first artificial Neural Network. It is a non-learning model (weights have to be set by a human) built on the binary neurons or also called McCulloch–Pitts (MCP) neurons: nodes can only have two values, $neuron_i = 1$ for the excitatory nodes and $neuron_i = 0$ for the inhibitory ones, the activation functions is then a threshold binary function. In this way the neuron can be used to divide inputs in two categories labelled by 0 and 1. Only years later the Italian physicist Eduardo Renato Caianiello presented in "Outline of a theory of thought-processes and thinking machines" [3] the so-called *mnemonic equation*, based on the Hebb's rule, to update the weights for a specific task.
1958: The American psychologist Frank Rosenblatt published "The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain" [35] where he indeed showed the Perceptron, taking is name from biological mechanism of perception. The model is

basically an artificial neuron: it's a 1-layer feed forward Neural Network with only one node in the output layer, the activation function is the step function. It's a generalized version of the MCP neuron since the input values can be real numbers: the components $x_j \in \mathbb{R}$. Moreover, the most important discovery is the iterative algorithm that, given a set of labeled examples $\{\, (\boldsymbol{x}^{(i)}, y^{(i)}) \mid i = 1, \ldots, m \,\}$, permits to automatically update weights in such a way that the model returns the correct output $y^{(i)}$ for every input $\boldsymbol{x}^{(i)}$ (supervised learning). Rosenblatt proved the convergence of the algorithm but unfortunately it does only for problems solvable by the Perceptron: either the error goes to zero or the algorithm doesn't converge. The problems solvable by this model are a subset of the binary classification problems (the output can be only 0 or 1) where the decision boundary is linear (since where are composing an affine function and a step function). In particular, they are the problem where the two categories can be divided by one single hyperplane.
1960: Widrow and Hoff built the ADALINE (Adaptive Linear Neuron) which learns thanks to an algorithm that essentially is the stochastic gradient descent method for the linear regression.
1969: Single layer Neural Networks such as the Perceptron model can represent a lot of Boolean functions (they map vector with 0 or 1 components to a 0 or 1 value). However in "Perceptrons: An Introduction to Computational Geometry" [26] Minsky and Papert showed that the Perceptron is not able to learn the simple XOR (exclusive-or) function, actually this is a general flaw of linear models. We can see XOR as a simplier version of a classification problem with non-linear boundaries: we plot some training data of both problems with Python





Even if training methods for multilayer Neural Networks were already known (the first multilayer network appeared in 1965), it was stated that computers can't process them efficiently. This fact, added to the statement of Minsky and Papert, made decrease investments and focus in the Neural Networks field.

Remark 1.9. *The XOR is a boolean operation on two variables.* $XOR : \{0,1\}^2 \to \{0,1\}$ *mapping* $(0,0)$ *and* $(1,1)$ *to* 0*,* $(1,0)$ *and* $(0,1)$ *to* 1*.*

Second wave The researches continued, but Neural Networks had a real return only after almost twenty years in the 1980s.
1986: The effective use of back-propagation to train multilayer networks was one of the most important results that led again to the growth of the interest in this field. It is an applitcation of the Leibniz chain rule and it was analysed in "Learning Internal Representations by Error Propagation" [36] by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams.
1990s: In this decade there was a particular attention to employ sequences in Neural Networks, especially in the recurrent ones. The researchers tried to understand mathematical difficulties

and capacities of using a sequence $(x_n)_{n\in\mathbb{N}}$ as input or getting a sequence $(y_n)_{n\in\mathbb{N}}$ as output. This successful period ended at mid-1990s: in those years many unattainable claims were made and, indeed, not satisfied producing delusion in the investors. At the same time other Machine Learning algorithm, such as Support Vector Machine (SVM) or kernel machine, obtained impressive result overwhelming the multilayer Neural Networks. Furthermore, the hardware of that time didn't permit to experimenting with the efficiency of the various algorithms due to high computational costs. Therefore multilayer networks were believed to be too difficult to train.

Third wave Studies on recurrent networks and convolutional Neural Network still proceeded in the period between waves. Moreover in general Neural Networks kept to get excellent performance on some tasks. But the key step for the Neural Networks' resurgence was in 2006.
2006: Geoffrey E. Hinton, Simon Osindero and Yee-Whye Teh published "A Fast Learning Algorithm for Deep Belief Nets" [17] where they presented the deep belief network which could be efficiently trained by the so-called *greedy layer-wise pre-training strategy* (method of unsupervised learning).
2007: Some scientists' groups associated with the Canadian Institute for Advanced Research (CIFAR) discovered that the previous strategy can be applied to many other types of networks. In this years the term "Deep Learning" became popular, mainly to point out that it was now possible to train deeper networks and to highlight the importance of the depth property.
Neural Networks experienced improvements in most of their aspects: the size extremely increased getting to almost 10 thousand of connections per neuron and millions of neurons. It also increased the complexity of tasks solvable by Neural Networks, for example they can now recognize even more than a thousand of different categories of objects. Thus, thanks to these gains and to the fact that larger dataset were accessible (they reach an order of magnitude of tens of millions), they improved their accuracy overtaking the other Machine Learning models. This can be noted looking at the accuracy on famous benchmarks as in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) which evaluates algorithms for objects detection. In the book "Pattern Recognition and Machine Learning" [1] Christopher M. Bishop reported another important point: with multilayer Neural Networks we usually lose the convexity of the error functional (present in model such as SVM) but we obtain a more compact model. This helps in the modern practice where it's preferred to spend more time in the training phase to get a model which is faster to evaluate on new examples.
Additionally, the previously prohibitive hardware were now surpassed and replaced: faster CPUs were available and the new general purpose GPUs permitted to implement parallel computing (more suitable and efficient for Neural Networks with respect to the usual sequential computing).

Chapter 2

Training phase

The most time consuming and probably the most important phase when we are building a model is the training phase. The prediction phase gives us result even in few seconds or less, while the training phase of modern Neural Networks can last for months or years. That's why it is the portion we want to study and analyse how we can make it faster, more efficient or more accurate (sometimes we don't need to decrease the computation time but we want to achieve the best from it).

We previously gave a brief and general presentation of learning and what we optimize in a Machine Leaning model. In this chapter we are going to get more specific looking at the main components of the training phase. We are going to explain what it really means to "learn" and what we want to achieve: we minimize a quantity hoping that another one decrease as well. Then we show an example on how we get and construct, in a clever way, a loss function (and consequently the associated cost function). Lastly, at the end of the chapter we are going to analyse the main starting method to optimize (thus train) our cost function.

As we have already shown, it's easy to see a Neural Network as a function that maps vectors $\boldsymbol{x}$ to other vectors $\boldsymbol{y}$. That is why we are going to primarily talk about supervised learning. However we have to keep in mind that the division between supervised and unsupervised learning is not so strong, in fact, we could use a composition of the former to derive the latter and viceversa. One method for going from supervised to unsupervised is to consider the label vector $\boldsymbol{y}$ as further features of $\boldsymbol{x}$ obtaining an input vector $(\boldsymbol{x}, \boldsymbol{y})$. Instead, we can consider a subset of the features as the label vector to do the inverse. For example, in the density estimation task, finding $p(\boldsymbol{y} \mid \boldsymbol{x})$ is a supervised learning problem while finding $p(\boldsymbol{x})$ is an unsupervised one, but they can be rewrite in terms of the other: by the definition of conditional probability we obtain

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{p(\boldsymbol{y}, \boldsymbol{x})}{p(\boldsymbol{x})} = \frac{p(\boldsymbol{y}, \boldsymbol{x})}{\sum\limits_{\boldsymbol{y}'} p(\boldsymbol{y}', \boldsymbol{x})}$$

and iterating it we get the chain rule of probability

$$p(\boldsymbol{x}) = \prod_{j=1}^{n} p(\boldsymbol{x}_j \mid \boldsymbol{x}_1, \dots, \boldsymbol{x}_{j-1}).$$

In general it will be easy to extend our results and considerations to the unsupervised case (sometimes it's just a matter of excluding the $\boldsymbol{y}$ dependencies).

From now on we are also going to specify the parametrization of the hypothesis space H (Neural Networks are parametric models so we can explicitly do it): we are going to write $f(\cdot\,;\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is the parameters vector that defines the function. In the Neural Network case, $\boldsymbol{\theta}$ is a vector containing the components of all bias vectors $\boldsymbol{b}^{(k)}$ and all weights matrices $\boldsymbol{W}^{(k)}$. It's not often used in practice but it's helpful to contain all parameters in a unique theoretical object.

2.1 Learning

In the first chapter we discussed a paragraph on supervised learning where we also presented what is done in practice during learning: we optimize a function depending on the training set to find the "optimal" hypothesis $\overline{f} \in H$, then we check the performance on the test set. We now describe why we follow this procedure and what it really means to learn, also referring to the definition 1.1.
The main purpose and interest of a Machine Learning model is the ability to perform well on new examples not seen before. This property is called generalization and it's what really matters when we employ our model in real application: the model is very likely to encounter examples not seen in the training phase when it deals with real world data. This is why increasing the performance is almost always equivalent to decrease a generalization error, which is the expectation of the error on a new input over a distribution on all possible inputs. However, as we already know, the model is trained to get an hypothesis $\overline{f} \in H$ gaining experience through the observation of the training set only, and so during the training phase it can measure an error or cost (defined in a specific way which depends on the model) only on this finite set. Therefore, the learning procedure departs from usual optimization because it decreases a cost function depending on some data, the experience E, guessing that this reduction leads to the decrease of another quantity, the generalization error, or equivalently to the improvement of the performance P.
Statistical learning theory gives us a necessary assumption to connect the training error (the error on the training set computed with a chosen cost function) and the generalization error (or really, as we are going to see, its estimator), otherwise the previous idea wouldn't be possible. If we consider collecting an example as a random variable, then the collections of each example are assumed to be independent and identically distributed (i.i.d.). The distribution from which the examples are drawn is called the data generating distribution and it's denoted with P_{data}. We can characterize the whole data generating process using only this example-wise distribution. Then, a key point to link the reduction of the two quantities is that training examples and new examples are drawn from the same probability distribution and are independent.

Now we present the usual process to choose a cost function to obtain an appropriate form of training error. We first set a performance measure derived from what we want to achieve. In fact, there are a lot of considerations taken into account to choose what to measure: we have to decide if we prefer an algorithm that makes small errors frequently or one that makes larger errors but rarely, if we prefer a less precise but faster algorithm or a more precise but slower one, and other properties depending on the application and the task. For example, in computer vision we have to consider if we need to set as correct only when all object are recognized or we can give partial credit for recognizing correctly a subset of them; in fraud detection system we are more concern about minimizing false negative than increasing accuracy.

Example 2.1. *Some examples of possible performance measures for classification tasks are the following (they are seen in the binary case but they can be easily extended to more classes):*

- *Accuracy: it's the percentage of correctly predicted examples. It's useful when the data are balanced (we have almost the same amount of examples for every label), but misleading when they are unbalanced.*

- *Precision: setting one label as the positive and one as the negative, it's the fraction of true positives among all positives predicted:* $\text{Precision} = \frac{\text{TP}}{\text{TP}+\text{FP}} \in [0,1]$*. It's used when we want to be sure that the positives detected are really positive, so we prefer to decrease the number of false positives.*

- *Recall: it's the fraction of true positive prediction among all real positive examples:* $\text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}} \in [0,1]$*. It's used when we don't want to miss some positive instances, so we prefer to decrease the number of false negative.*

- *F1-score: the previous two measures have advantages and flaws, then we use F1-score when we want to consider both recall and precision. It is the harmonic mean of the two quantities* $\text{F1} = \frac{2}{\text{Precision}^{-1}+\text{Recall}^{-1}} = \frac{2(\text{Precision}\cdot\text{Recall})}{\text{Precision} + \text{Recall}}$*. When they are both equal to the same value c then also* $\text{F1} = c$*.*

Instead performance measures for regression tasks give a continuous valued score to each example:

- *Mean Absolute Error:* $\text{MAE} = \frac{1}{M}\sum_{j=1}^{M} \left|f(\boldsymbol{x}^{(j)};\boldsymbol{\theta}) - y^{(j)}\right|$.

- *Mean Square Error:* $\text{MSE} = \frac{1}{M}\sum_{j=1}^{M} \left(f(\boldsymbol{x}^{(j)};\boldsymbol{\theta}) - y^{(j)}\right)^2$.

The second step is to associated the performance to a proper generalization error (also called true risk): it should reflect what we want to obtain in the performance. As we previously say, this is the quantity we would like to reduce and it's the expectation of the error on a new input over P_{data}:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim P_{data}}\Big[L\big(f(\mathbf{x};\boldsymbol{\theta}),\mathbf{y}\big)\Big] \tag{2.1}$$

where L is a per-example loss function and $(\mathbf{x},\mathbf{y}) \sim P_{data}$ means that $(\mathbf{x},\mathbf{y})$ has distribution P_{data}.

Remark 2.1. *We recall the expected value explicit formula for a discrete random vector* $\mathbf{z}$ *with distribution* $P_{\mathbf{z}}$*:*

$$\mathbb{E}_{\mathbf{z}\sim P_{\mathbf{z}}}\big[l(\mathbf{z})\big] = \sum_{\boldsymbol{z}\in X} P_{\mathbf{z}}(\boldsymbol{z})l(\boldsymbol{z})$$

where $P_{\mathbf{z}}(\boldsymbol{z})$ *is the probability of the event* $\{\mathbf{z} = \boldsymbol{z}\}$ *by definition of distribution.*
For absolutely continuous random vectors with probability density function (PDF) $p_{\mathbf{z}}$ *the formula is:*

$$\mathbb{E}_{\mathbf{z}\sim P_{\mathbf{z}}}\big[l(\mathbf{z})\big] = \int_{\boldsymbol{z}\in X} p_{\mathbf{z}}(\boldsymbol{z})l(\boldsymbol{z})d\boldsymbol{z}$$

where $P_{\mathbf{z}}$ *is the distribution obtained with* $p_{\mathbf{z}}$ *as PDF.*

Then, we set the training error as the expected loss on the training set computed with the same L of the previous step. This is an easy and intuitive way to choose a computable quantity that probably makes the generalization error reduce while it decreases, since it's one of generalization error's unbiased estimators. Statistical learning theory gives us many results on the conditions needed to suppose that the true risk will decrease.
In practice, we calculate with L the per-example loss on every training example and then we average them. We notice that it's the explicit formula of the loss expectation with respect to empirical distribution defined by the training set. That is why the training error is also called empirical risk and the training process based on the minimization of this quantity is named empirical risk minimization.

$$\begin{aligned} J(\boldsymbol{\theta}) &= \frac{1}{m}\sum_{i=1}^{m} L\big(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),\boldsymbol{y}^{(i)}\big) = \\ &= \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\widehat{P}_{train}}\Big[L\big(f(\mathbf{x};\boldsymbol{\theta}),\mathbf{y}\big)\Big] \end{aligned} \tag{2.2}$$

where $\widehat{P}_{train}$ is the empirical distribution defined by the training set.

Remark 2.2. *An empirical distribution is precisely called "empirical" because it is determined by a set of samples $\{\, \boldsymbol{v}^{(i)} \in \mathbb{R}^n \mid i = 1,\ldots,l \,\}$ (all drawn from the same distribution) usually observed through experiments and measurements. Therefore, it is one way to associate a probability measure to experimental dataset. It is defined focusing the same probability mass $\frac{1}{l}$ on each sample, where l is the number of observations. The empirical distribution $\widehat{P}_{empirical}$ obtained is:*

$$\widehat{P}_{empirical}(A) = \frac{1}{l}\sum_{i=1}^{l} \mathbb{1}_{\boldsymbol{v}^{(i)}\in A} \quad \forall A \in \mathfrak{B}(\mathbb{R}^n) \tag{2.3}$$

where $\mathbb{1}_{\boldsymbol{x}\in A} = \mathbb{1}_A(\boldsymbol{x})$ is the indicator function of the subset A and $\mathfrak{B}(\mathbb{R}^n)$ is the Borel sigma algebra on $\mathbb{R}^n$. Essentially it counts the fraction of dataset elements contained in the subset A. It is used as an approximation of the real underlying distribution.

At this point we have a learning process to find $\overline{\boldsymbol{\theta}}$ by optimizing J, but we don't know how to check the performance of $f(\cdot\,;\overline{\boldsymbol{\theta}})$ or equivalently its generalization error. Indeed, the true risk is obviously not computable since we don't know P_{data} (otherwise we would already know the solution). Therefore, we use an unbiased estimator instead: the test error, which is the expected loss on the test set (a group of example in the dataset not used during the training phase)

$$\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\widehat{P}_{test}}\Big[L\big(f(\mathbf{x};\boldsymbol{\theta}),\mathbf{y}\big)\Big] \tag{2.4}$$

where $\widehat{P}_{test}$ is the empirical distribution defined by the test set.

One of the main target of statistical learning theory is to find a connection between the training and test errors. It succeeded in this finding some boundaries on the difference between them. These estimates are usually more theoretical rather than practical, since they are often difficult to compute. They justify the aim of Machine Learning proving that it's achievable: we can indeed reduce the test error (estimator of the generalization one which is our goal) optimizing the training error.

Example 2.2. *In the book "The Nature of Statistical Learning Theory" [42] the computer scientist Vladimir N. Vapnik proved that if $VC(H) \ll m$ and we fix $0 < \eta \leq 1$ then the following holds with probability equal to $1 - \eta$ for binary classification models*

$$\text{error}_{\text{test}} \leq \text{error}_{\text{train}} + \sqrt{\frac{1}{m}\left[VC(H)\left(\log\left(\frac{2m}{VC(H)}\right)+1\right)-\log\left(\frac{\eta}{4}\right)\right]}$$

where the errors are computed with the 0-1 loss function as L:

$$L_{0-1}(\widehat{\boldsymbol{y}}, \boldsymbol{y}) = \begin{cases} 0 & \text{if } \widehat{\boldsymbol{y}} = \boldsymbol{y} \\ 1 & \text{otherwise} \end{cases} \tag{2.5}$$

or making explicit the dependence on the parametrized family of functions

$$L_{0-1}(f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y}) = \begin{cases} 0 & \text{if } f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{y} \\ 1 & \text{otherwise} \end{cases}$$

*which means it returns no-cost if the output of f is equal to $\boldsymbol{y}$ otherwise it returns a cost equal to 1. Then, doing the average, the error is the fraction of wrongly predicted outputs (*error = 1−accuracy*). $VC(H)$ is the Vapnik–Chervonenkis dimension and it measures the capacity of the hypothesis space H of a binary classification model. In particular, it is the quantity that makes the previous inequality more theoretical than practical, in fact, it is hard to determine it for a lot of models' types. For example, it can be very difficult in our Neural Networks case. Here we rewrite with our notation the definition from the same book [42] of the previous inequality:*

Definition 2.1. *The VC dimension of an hypothesis space H is the maximum number D of vectors $\{\boldsymbol{v}_1, \dots, \boldsymbol{v}_D\} \subset X$ that can be separated into two classes in all 2^D possible ways using a function of the set H composed with a shifted step function (i.e., the maximum number of vectors that can be shattered by the set of functions). The composition explicitly written is $g(f(\boldsymbol{x};\boldsymbol{\theta});\alpha) = \begin{cases} 0 & \text{if } f(\boldsymbol{x};\boldsymbol{\theta}) < \alpha \\ 1 & \text{if } f(\boldsymbol{x};\boldsymbol{\theta}) \geq \alpha \end{cases}$.*
If for any D there exists a set of D vectors that can be shattered by the set H, then the VC dimension is equal to infinity.

In other words, it's the maximum $D \in \mathbb{N}$ such that there is a collection of points with cardinality D which can be shattered by H, meaning that for every assignment of labels to these points there exists an $f \in H$ correctly predicting the labels without making any errors on this set of points.

In conclusion, the general aim of a Machine Learning model is to reduce the training error (preventing underfitting) and also the gap with the test error (getting rid of the overfitting problem). In this thesis and in particular in the next chapters we are going to focus on the part dealing with the training error: we are going to study optimization methods and their variations to obtain a small training error in fast ways.

Empirical distribution

In this subsection we briefly present the Dirac delta function and how it is used to build the density of a desired empirical distribution.

The Dirac delta is a generalized function that is defined by its properties: it's zero everywhere except at the origin $\boldsymbol{0} \in \mathbb{R}^n$ and its integral on the whole domain $\mathbb{R}^n$ is equal to 1 (so it give us a probability measure on $\mathbb{R}^n$). It can be seen as a linear functional that maps continuous function ϕ to their value at the origin $\phi(0)$:

$$\delta : \phi \mapsto \int_{\mathbb{R}^n} \phi(\boldsymbol{x}) d\delta(\boldsymbol{x}) = \phi(\boldsymbol{0}).$$

Otherwise it can be seen as the limit of a sequence of functions that are zero over most of the domain and have a peak around the origin. An example of this sequence type is one generated by the standard mollifier η, usually used in convolution approximation. We take η definition from "Partial Differential Equations" [11] by Lawrence C. Evans:

$$\eta(\boldsymbol{x}) = \begin{cases} C\, e^{\frac{1}{\|\boldsymbol{x}\|^2 - 1}} & \text{if } \|\boldsymbol{x}\| < 1 \\ 0 & \text{if } \|\boldsymbol{x}\| \geq 1 \end{cases} \tag{2.6}$$

where the constant $C > 0$ is selected such that $\int_{\mathbb{R}^n} \eta d\boldsymbol{x} = 1$. Then, for every $\varepsilon > 0$ we set

$$\eta_\varepsilon(\boldsymbol{x}) := \frac{1}{\varepsilon^n} \eta\left(\frac{\boldsymbol{x}}{\varepsilon}\right). \tag{2.7}$$

Since η is smooth and compactly supported with $\operatorname{supp}(\eta) \subset B(\boldsymbol{0}, 1)$ ($\eta \in C_c^\infty(\mathbb{R}^n)$), the elements of a sequence defined by $(\varepsilon_k)_{k\in\mathbb{N}}$ are smooth, compactly supported with $\operatorname{supp}(\eta_{\varepsilon_k}) \subset B(\boldsymbol{0}, \varepsilon_k)$ and $\int_{\mathbb{R}^n} \eta_{\varepsilon_k} d\boldsymbol{x} = 1$. So they keep their integral on the entire domain equal to 1 and shrink their support around the origin. Moreover, if $\varepsilon_k \xrightarrow{k\to\infty} 0$ then $\lim_{k\to\infty} \eta_{\varepsilon_k}(\boldsymbol{x}) = \delta(\boldsymbol{x})$. We also notice that these are positive mollifiers: $\eta_\varepsilon \geq 0 \ \forall \varepsilon > 0$.

When we have a continuous random vector, but we want to focus the probability mass of a distribution in a single point $\widetilde{\boldsymbol{v}}$, we employ $\delta(\boldsymbol{x} - \widetilde{\boldsymbol{v}})$ as probability density function. In fact, the Dirac delta as probability density function puts all the mass in the origin and the term $-\widetilde{\boldsymbol{v}}$ shifts the function so that all the mass moves to $\widetilde{\boldsymbol{v}}$. Using this idea we can spread the probability mass in more points (but still a finite set) giving to each point $\boldsymbol{v}^{(i)}$ a probability p_i obviously such that $\sum_{i=1}^{l} p_i = 1$. The obtained distribution has density

$$\widehat{p}(\boldsymbol{x}) = \sum_{i=1}^{l} p_i \delta(\boldsymbol{x} - \boldsymbol{v}^{(i)}).$$

This is basically what we did for the empirical distribution in remark 2.2 giving, in particular, $p_i = \frac{1}{l}$ to each observed sample. That is why the density of the distribution defined in the formula 2.3 is:

$$\widehat{p}_{empirical}(\boldsymbol{x}) = \frac{1}{l} \sum_{i=1}^{l} \delta(\boldsymbol{x} - \boldsymbol{v}^{(i)}). \tag{2.8}$$

This can be easily seen also following another approach to determine the empirical distribution that is setting it's cumulative distribution function (CDF). This procedure is presented for univariate variables in "A Modern Introduction to Probability and Statistics: Understanding Why and How" [8] (by Frederik Dekking, Cor Kraaikamp, Hendrik Lopuhaä and Ludolf Meester) where they define

$$F_l(x) = \frac{\text{number of } \{\text{elements} \le x\} \text{ in the dataset}}{l} = \frac{1}{l}\sum_{i=1}^{l} \mathbb{1}_{v^{(i)} \le x}.$$

This type of CDF can be clearly obtained by a PDF of the type in the formula 2.8.

Remark 2.3. *The case of discrete random vectors is easier since we can simply use the categorical distribution, also called multinoulli distribution to stress that it's a generalization of the Bernoulli distribution. In the Bernoulli we have two possible values and we assign a probability p to one of them, the other one will automatically have a probability $1-p$. Instead, in the multinoulli we have a finite number K of different values (or categories) as results of the random vector, then the distribution is described by a vector $\boldsymbol{p} \in [0,1]^{K-1}$ such that $\sum_{i=1}^{K-1} p_i \le 1$. In this way the last value will have probability $1 - \boldsymbol{1}^T\boldsymbol{p} \in [0,1]$. The Bernoulli distribution is a particular case where $K = 2$.*

Getting back to our particular case, the empirical distribution defined by the training set $\{\, (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \in \mathbb{R}^{n+d} \mid i = 1, \dots, m \,\}$ has density

$$\widehat{p}_{train}(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{m}\sum_{i=1}^{m} \delta\big((\boldsymbol{x}, \boldsymbol{y}) - (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\big). \tag{2.9}$$

And now that we know the density we can justify the explicit formula of the expectation value in the equation 2.2:

$$\begin{aligned}
&\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\widehat{P}_{train}}\Big[L\big(f(\mathbf{x};\boldsymbol{\theta}), \mathbf{y}\big)\Big] = \\
&= \int_{\mathbb{R}^n} L\big(f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y}\big)\frac{1}{m}\sum_{i=1}^{m} \delta\big((\boldsymbol{x}, \boldsymbol{y}) - (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\big)d\boldsymbol{x} = \\
&= \frac{1}{m}\sum_{i=1}^{m}\int_{\mathbb{R}^n} L\big(f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y}\big)\delta\big((\boldsymbol{x}, \boldsymbol{y}) - (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\big)d\boldsymbol{x} = \\
&= \frac{1}{m}\sum_{i=1}^{m} L\big(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)}\big) = J(\boldsymbol{\theta})
\end{aligned}$$

Surrogate loss

The test error serves as an unbiased estimator of the generalization error J^*, so it must be defined by the same loss function L chosen for J^* in order to reflect the performance. However many practical and valuable functions L are unsuitable for the Gradient Descent and its variations, which are the most employed optimization methods to train a model. The reason may be the intractability of the function and therefore of its gradient, or it may be the presence of gradients with very small norm almost everywhere.
For example, the 0-1 loss function (already defined in the equation 2.5) is simple and interpretable since it returns us the percentage of errors on the test set, but its partial derivatives are zero

almost everywhere and they are even not defined where they aren't zero. In general, this problem affects all functions L that can't be optimized in an efficient way due to their properties.

This is the reason why sometimes we prefer to optimize a different cost function J to train the model and, in order to achieve this, we define the training error with a loss function L different from the one used in the test error. In these cases, the new one is called surrogate loss function. This process, obviously, requires further considerations to select a proper surrogate loss function that matches the intent of the generalized error. Nevertheless, it has other advantages in addition to the ones related to optimization.

In classification problems, for example, we can take the negative log-likelihood as surrogate of the 0-1 loss. As we are going to see in the next section, this new L can make the model estimate a conditional probability on the classes y given the input $\boldsymbol{x}$. Therefore, the substitution is reasonable: if the model can approximate well the conditional distribution $P_{data}(y \mid \boldsymbol{x})$, then it can also predict well the single output class of an input $\boldsymbol{x}$ taking, for instance, the y with highest probability given $\boldsymbol{x}$. In this way it's reasonable that we are minimizing the underlying 0-1 cost function, too. With the negative log-likelihood we can use Gradient Descent variations, but we have another main advantage. The log-likelihood cost can keep decreasing on the training set even when the 0-1 cost is at zero on the same set. In fact, even if it classifies perfectly the training samples, the model can still improve the distribution estimate by dividing the classes even more: we can get a better and more reliable classifier. It is basically obtaining more information by the samples than the 0-1 cost minimization. This can increase the training phase time but it's a good property to get better generalization.

2.2 Construct L with MLE

The loss function L can be manually designed often requiring a lot of time and human effort. Otherwise, a way to automatically derive it can be employing a well-known method to find point estimators: the maximum likelihood estimation (MLE). It's one of the most used method to build L for Neural Networks and in general for Machine Learning. In this section we are going to present this principle and see how the choice of output units affects the cost function when we are using the maximum likelihood estimation.

Remark 2.4. *Whether it's manually selected or it's automatically derived by some procedure, from now on we are going to assume that L is defined so that $J^*(\boldsymbol{\theta})$ exists for every $\boldsymbol{\theta}$. This means that $\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim P_{data}}\Big[L\big(f(\mathbf{x};\boldsymbol{\theta}),\mathbf{y}\big)\Big] < \infty$ for every $\boldsymbol{\theta}$.*

Maximum likelihood procedure

The idea of using this method comes from the fact that, in general supervised learning, we often redirect our aim to probability mass (or density) function estimation. Even when the task is to predict the output $\boldsymbol{y}$ of an input $\boldsymbol{x}$, most of the time we prefer to predict a conditional distribution on $\boldsymbol{y}$ given $\boldsymbol{x}$ because it gives us more information and we can use the probability mass (or density) function for further computations and other tasks. Formally stating the new task, we want to learn a conditional probability mass (or density) function $p_{model} : \mathbb{R}^{n+d} \to \mathbb{R}$ which maps $(\boldsymbol{x}, \boldsymbol{y}) \mapsto p_{model}(\boldsymbol{y} \mid \boldsymbol{x})$. Essentially, instead of associating $\boldsymbol{x}$ to a single output $\boldsymbol{y}$, we map $\boldsymbol{x}$ to a whole conditional probability mass (or density) function on the possible outputs

given $\boldsymbol{x}$: $p_{model}(\cdot \mid \boldsymbol{x})$.
The model now specifies a parametrized family of conditional probability mass (or density) functions $p_{model}(\boldsymbol{y} \mid \boldsymbol{x};\boldsymbol{\theta})$. However the Neural Network used in the model still represents a function $f(\boldsymbol{x};\boldsymbol{\theta})$. Then, to shift the focus from these functions to the probability interpretation we can exploit this family of functions to indirectly parametrize $p_{model}(\boldsymbol{y} \mid \boldsymbol{x};\boldsymbol{\theta})$. For example, the output $f(\boldsymbol{x};\boldsymbol{\theta})$ can be the mean of Gaussian distributions with fixed variance: the distribution generated by $p_{model}(\boldsymbol{y} \mid \boldsymbol{x};\boldsymbol{\theta})$ is the Gaussian $\mathcal{N}\big(f(\boldsymbol{x};\boldsymbol{\theta}),\sigma^2\big)$. In other word the parameters $\boldsymbol{\theta}$ define a Neural Network as a function $f(\boldsymbol{x};\boldsymbol{\theta})$ that is used to determine the conditional probability mass (or density) function $p_{model}(\boldsymbol{y} \mid \boldsymbol{x};\boldsymbol{\theta})$ specified in the model (they compose the new hypothesis space H).

After this reworking of the model with a probabilistic point of view, we can define the maximum likelihood estimation for our models. What we want to obtain is the point estimator $\widehat{\boldsymbol{\theta}}_{MLE}$ such that $p_{model}(\boldsymbol{y} \mid \boldsymbol{x};\widehat{\boldsymbol{\theta}}_{MLE})$ coincides with or approximates the probability mass (or density) function $p_{data}(\boldsymbol{y} \mid \boldsymbol{x})$ defined by the data generating distribution P_{data} (presented at the beginning of the previous sections).
The maximum likelihood principle derives a good estimator by this simple procedure: it selects the $\widehat{\boldsymbol{\theta}}_{MLE}$ that maximizes the probability of the training set being drawn.

$$\widehat{\boldsymbol{\theta}}_{MLE} = \arg\max_{\boldsymbol{\theta}} \; p_{model}\big(\boldsymbol{y}^{(1)},\dots,\boldsymbol{y}^{(m)} \mid \boldsymbol{x}^{(1)},\dots,\boldsymbol{x}^{(m)};\boldsymbol{\theta}\big). \tag{2.10}$$

In the next steps we present how to rewrite the principle to get a suitable cost function. In the first part of the previous section we show the necessary i.i.d. assumption of the samples. We take advantage of this property to rewrite the earlier formula in a product:

$$\widehat{\boldsymbol{\theta}}_{MLE} = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{model}\big(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)};\boldsymbol{\theta}\big). \tag{2.11}$$

Remark 2.5. *We can be derived the equation 2.11 from 2.10 by induction using the following procedure: if* $(\mathrm{v}_1,\mathrm{w}_1)$ *and* $(\mathrm{v}_2,\mathrm{w}_2)$ *are two independent random vectors, then*

$$p(\boldsymbol{w}_1,\boldsymbol{w}_2 \mid \boldsymbol{v}_1,\boldsymbol{v}_2) = \frac{p(\boldsymbol{w}_1,\boldsymbol{w}_2,\boldsymbol{v}_1,\boldsymbol{v}_2)}{p(\boldsymbol{v}_1,\boldsymbol{v}_2)} = \frac{p(\boldsymbol{w}_1,\boldsymbol{v}_1)}{p(\boldsymbol{v}_1)}\frac{p(\boldsymbol{w}_2\boldsymbol{v}_2)}{p(\boldsymbol{v}_2)} = p(\boldsymbol{w}_1 \mid \boldsymbol{v}_1)p(\boldsymbol{w}_2 \mid \boldsymbol{v}_2).$$

However, the formulation 2.11 is affected by numerical underflow: if a lot of the factors are very small, then the entire product can become so small that it's approximated to zero. We notice that the logarithm (in particular we take the natural one) is a strictly increasing function, hence composing with it changes the maximum and minimum values but not the arguments where they are achieved. With this composition we obtain a sum formulation that overtakes the underflow problem:

$$\begin{aligned}\widehat{\boldsymbol{\theta}}_{MLE} &= \arg\max_{\boldsymbol{\theta}} \; \ln\left(\prod_{i=1}^{m} p_{model}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)};\boldsymbol{\theta})\right) = \\ &= \arg\max_{\theta} \; \sum_{i=1}^{m} \ln p_{model}\big(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)};\boldsymbol{\theta}\big).\end{aligned}$$

One could argue that this formulation is affected by numerical overflow but we can say it's less likely to happen than underflow: suppose we have a fixed amount of examples m and the previous

probabilities are all equal to p, then p^m goes faster to 0 compared to how fast $\ln(p^n) = n \ln p$ goes to $-\infty$ as $p \to 0$, since $\lim_{p\to 0} p^n(n \ln p) = 0$.
Multiplying by a positive constant (which is rescaling) is an increasing function, too. Then we can proceed with the composition, as previously done, and essentially multiply by $\frac{1}{m}$. This new version can be seen as the expected value with respect to the empirical distribution defined by the training set $\widehat{P}_{train}$ (with PDF as in the formula 2.9):

$$\begin{aligned}\widehat{\boldsymbol{\theta}}_{MLE} &= \arg\max_{\boldsymbol{\theta}} \ \frac{1}{m}\sum_{i=1}^{m} \ln p_{model}\big(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\big) \\ &= \arg\max_{\boldsymbol{\theta}} \ \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\widehat{P}_{train}} \Big[\ln p_{model}\big(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta}\big)\Big].\end{aligned}$$

We change the function sign (it exchanges the minimum and maximum arguments) to consider it as a cost function and we look at the minimum points:

$$\widehat{\boldsymbol{\theta}}_{MLE} = \arg\min_{\boldsymbol{\theta}} \ -\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\widehat{P}_{train}} \Big[\ln p_{model}\big(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta}\big)\Big]. \tag{2.12}$$

Therefore, recalling the quantities named at the previous section, the training error or cost function to optimize is in this case

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\widehat{P}_{train}} \Big[-\ln p_{model}\big(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta}\big)\Big]. \tag{2.13}$$

Then the actual loss function per example is $L\big(f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y}\big) = -\ln p_{model}\big(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}\big)$.

Information theory interpretation

Information theory was initially introduced as mathematical field by Claude E. Shannon in the paper "A Mathematical Theory of Communication" [38] taking inspirations also from statistical mechanics. The title is due to the fact that the author started presenting it with a prospective on information brought by messages composed from discrete alphabets. That's why the initial considerations and definitions in the article are on discrete random variables or vectors.
One of information theory's main topics is the quantification of information. The basic and core tool for this purpose is the self-information (also called Shannon information, information content or surprisal) of an event $\{\mathbf{z} = \boldsymbol{z}\}$:

$$I(\boldsymbol{z}) = -\log_b P_{\mathbf{z}}(\boldsymbol{z}) \tag{2.14}$$

where $b > 1$ and $P_{\mathbf{z}}$ is the distribution of the discrete random vector $\mathbf{z}$. The base b of the logarithm is chosen depending on the context, for example, the base $b = 2$ is useful when we deal with bits. However they are all rescaling of the same information quantity since $\log_b = \frac{1}{\log_c b}\log_c$, then we can say that the general formulation of self-information is $I(\boldsymbol{z}) = -C \ln P_{\mathbf{z}}(\boldsymbol{z})$ with $C > 0$.

This mathematical object reflects some important intuitions that can be extracted from some observations and the context of the paper. Indeed, these intuitions and the self-information itself are not explicitly defined and they are shown as parts of wider concepts (for example in the formulation of entropy). Instead, in "An Introduction to Information Theory and Entropy" [4] the author Tom Carter formally listed these intuitions as axioms and proved that the previous definition 2.14 of the self-information can be derived by them since it's the only function with these properties. The uniqueness is up to a multiplicative factor, since as we already said we can use different bases to rescale a logarithm chosen as starting point.

We describe the axioms and the complete derivation of the formula following Carter's procedure. The aim is measuring with a function $I(\omega)$ the information contained in the occurrence of an event ω with probability $P(\omega) = p \in (0,1]$. Before considering the actual axioms and the proof we have to show an important simplification applied by Carter: $I(\cdot)$ should depend only on the probability p and not on the event ω or its characteristics, therefore we have $I(\omega) = f(P(\omega))$ for some f to be found. For simplicity, we write $I(p)$ since we are interested in the properties of $I(\cdot)$ with respect to p. The axioms required are the next four:

1. $I(p) \geq 0$: the information brought by a certain event is non-negative.
2. $I(1) = 0$ and $I(\cdot)$ is non-decreasing: an event with probability equal to 1 doesn't give us information; while the smaller the probability, the greater the information.
3. $I(p_1 p_2) = I(p_1) + I(p_2)$: additivity of independent events. If two events ω_1 and ω_2 are independent, then the information gained at the occurrence of the intersection $\omega_1 \cap \omega_2$ is equal to the sum of the information of the two events. Recall that if they have probabilities p_1 and p_2 respectively, then the joint probability is $P(\omega_1 \cap \omega_2) = p_1 p_2$.
4. $I(\cdot)$ is a continuous function of the probability p.

Proposition 2.1. *$I(p) = -C \ln p$ with $C > 0$ is the unique (obviously up to a multiplicative factor) function that verifies the previous four properties.*

Proof. We can prove by induction that $I(p^n) = nI(p)$ for every $n \in \mathbb{N}$. For $n = 2$ we have the base case

$$I(p^2) = I(pp) = I(p) + I(p) = 2I(p)$$

from the third axiom. For $n > 2$ we have

$$I(p^n) = I(pp^{n-1}) = I(p) + I(p^{n-1}) = I(p) + (n-1)I(p) = nI(p)$$

using the inductive hypothesis in the third equation. Moreover it holds that $I\left(p^{\frac{1}{m}}\right) = \frac{1}{m}I(p)$ for every $m \in \mathbb{N}$:

$$I(p) = I\left(\left(p^{\frac{1}{m}}\right)^m\right) = mI\left(p^{\frac{1}{m}}\right)$$

We can combine the previous two to obtain $I\left(p^{\frac{n}{m}}\right) = \frac{n}{m}I(p)$, which is the property $I(p^q) = qI(p)$ for positive rationals $q \in \mathbb{Q}_+$. By continuity (required by the fourth axiom), we can extend this property to all positive real number: for $a \in \mathbb{R}_+$ we take a sequence of rational numbers $(q_k)_{k\in\mathbb{N}} \subset \mathbb{Q}_+$ such that $\lim_{k\to\infty} q_k = a$, then

$$I(p^a) = \lim_{q_k \to a} I(p^{q_k}) = \lim_{q_k \to a} q_k I(p) = aI(p).$$

Now suppose that we have a function $I(p) = f(p)$ with this property. For $p \in (0,1]$ we take $y = -\ln p \in \mathbb{R}_+$ and get

$$f(p) = f\big((e^{-1})^y\big) = yf(e^{-1}) = -(\ln p)f(e^{-1}) = -C \ln p$$

where $C = f(e^{-1})$. Then we have just proven that the only family of functions with this property is $I(p) = -C \ln p$. We conclude noticing that the first axiom (positivity) imposes that $C > 0$ since $p \in (0,1]$. □

The definition was extended to continuous random vectors in the last sections of the Shannon's article, however it loses a lot of the previous properties and intuitions:

$$I(\boldsymbol{z}) = -\log_b p_{\mathrm{z}}(\boldsymbol{z}) = -C \ln p_{\mathrm{z}}(\boldsymbol{z}) \tag{2.15}$$

where $p_{\mathbf{z}}$ is the density function of $\mathbf{z}$ and $b > 1$. We clearly see that it preserves additivity and monotonicity, but it can become negative for points with high density. Furthermore, the values with density 1 bring zero information, even if they are not sure. Nevertheless, we have to introduce it since it's needed to define the next mathematical objects in the continuous case, too. For our purpose, the continuous version of these objects is going to be necessary in the case where $P_{model;\boldsymbol{\theta}}$ are continuous distributions. From now on we will use $C = 1$ for both discrete and continuous case.

The self-information is a quantity related to a single event. To measure the information given by the entire random variable or vector we use the Shannon entropy, which is the expected value of the information given by the outcomes of the random variable or vector. In the discrete case is

$$H(\mathbf{z}) = \mathbb{E}_{\mathbf{z}\sim P_{\mathbf{z}}}[I(\mathbf{z})] = \mathbb{E}_{\mathbf{z}\sim P_{\mathbf{z}}}[-\ln P_{\mathrm{z}}(\mathbf{z})] = -\sum_{\boldsymbol{z}\in Z} P_{\mathrm{z}}(\boldsymbol{z}) \ln P_{\mathrm{z}}(\boldsymbol{z}). \tag{2.16}$$

Remark 2.6. *By convention, we set* $0 \ln 0 = 0$ *for the addends with* $P_{\boldsymbol{z}}(\boldsymbol{z}) = 0$ *to be consistent with the limit* $\lim\limits_{p\to 0^+} p \ln p = 0$.

While in the continuous case is named differential entropy and it's

$$H(\mathbf{z}) = \mathbb{E}_{\mathbf{z}\sim P_{\mathbf{z}}}[-\ln p_{\mathbf{z}}(\mathbf{z})] = -\int_{\boldsymbol{z}\in Z} p_{\mathbf{z}}(\boldsymbol{z}) \ln p_{\mathbf{z}}(\boldsymbol{z}) d\boldsymbol{z}. \tag{2.17}$$

If we have different random variables or vectors with the same distribution P, then they all have the same entropy. That's why we can also define the entropy related to a particular distribution P: $H(P) = H(\mathbf{z})$ for any $\mathbf{z} \sim P$.
If we take a discrete distribution with $P(\widetilde{\boldsymbol{z}}) = 1$ for one vector and equal to zero for all other outcomes, then the entropy of this distribution is zero. This means that, as expected, the average information is null when the outcome is sure. In general the entropy is very low when we have a distribution nearly constant (in some sense); for example a Gaussian distribution with very low variance. Instead, the uniform distribution over a set Z gives us the highest value of all possible entropies related to distributions over that set. Generalizing this fact, the distributions near to uniform (in same sense) have high entropy.

We can exploit the information and entropy ideas to compare two distributions Q and P: we use the Kullback-Leibler divergence as statistical distance between P and Q. The KL divergence, also called relative entropy, was presented in the article "On information and sufficiency" [22] by the two mathematicians after whom it's named. In its main interpretation, the KL divergence of P from Q is the expected additional self-information due to using Q when the actual underlying distribution is P (so the expectation is with respect to P)

$$\begin{aligned} D_{KL}(P \parallel Q) &= \mathbb{E}_{\mathbf{z}\sim P}\big[I_Q(\mathbf{z}) - I_P(\mathbf{z})\big] = \\ &= \mathbb{E}_{\mathbf{z}\sim P}\big[-\ln Q(\mathbf{z}) - \big(-\ln P(\mathbf{z})\big)\big] = \\ &= \mathbb{E}_{\mathbf{z}\sim P}\big[\ln P(\mathbf{z}) - \ln Q(\mathbf{z})\big] = \\ &= \mathbb{E}_{\mathbf{z}\sim P}\left[\ln \frac{P(\mathbf{z})}{Q(\mathbf{z})}\right] \end{aligned} \tag{2.18}$$

Remark 2.7. *A statistical distance is function which measures the difference between two statistical objects, such as random variables, distribution, samples and other quantities. A usual distance, also called metric, has the following properties: non-negativity, indiscernibility, symmetry and sub-additivity. A statistical distance, instead, is a more general concept since it hasn't always all the previous property. For instance, the statistical distances with only the first two properties are called divergences and the KL one is an example. Indeed, we can prove that it's non-negative and that it's equal to zero if and only if P and Q are the same distribution in the discrete case or are equal almost everywhere in the continuous case.*

We need an additional tool to get the MLE association with information theory: the cross-entropy

$$H(P,Q) = \mathbb{E}_{\mathbf{z}\sim P}\big[-\ln Q(\mathbf{z})\big] \tag{2.19}$$

It can be written also in terms of entropy and KL divergence: $H(P,Q) = H(P) + D_{KL}(P \parallel Q)$. Then it's equal to the divergence without a term, which is canceled by $H(P)$.

Lastly let $\mathbf{z} = (\mathbf{x}, \mathbf{y})$, $P = \widehat{P}_{train}$ and $Q = P_{model;\boldsymbol{\theta}}$, the J derived by the MLE procedure (formula 2.13) is equal to the cross-entropy between the empirical distribution defined by the training set $\widehat{P}_{train}$ and the distribution parametrized by the model $P_{model;\boldsymbol{\theta}}$

$$J(\boldsymbol{\theta}) = H\left(\widehat{P}_{train}\,, P_{model;\boldsymbol{\theta}}\right).$$

The MLE maximization correspond to the cost J minimization, or rather, to the cross-entropy minimization with respect to $P_{model;\boldsymbol{\theta}}$. Given that

$$H\left(\widehat{P}_{train}\,, P_{model;\boldsymbol{\theta}}\right) = H\left(\widehat{P}_{train}\right) + D_{KL}\left(\widehat{P}_{train} \parallel P_{model;\boldsymbol{\theta}}\right),$$

it's also equivalent to minimize the KL divergence between $\widehat{P}_{train}$ and $P_{model;\boldsymbol{\theta}}$ keeping the first fixed. This equivalence comes from the fact that the first term $H\left(\widehat{P}_{train}\right)$ depends only on $\widehat{P}_{train}$ and thus it's constant varying $\boldsymbol{\theta}$.

Remark 2.8. *The equivalence is true in general: for every fixed P, minimizing $H(P,Q)$ is equivalent to minimizing $D_{KL}(P \parallel Q)$ both with respect to Q.*

In conclusion, with the MLE procedure we change $\boldsymbol{\theta}$ trying to decrease the difference (determined by the KL divergence) between the model distribution $P_{model;\boldsymbol{\theta}}$ and the (fixed) empirical distribution defined by the training set $\widehat{P}_{train}$, which in particular can be considered as an approximation of the true data generating distribution P_{data}. In this way we hope that the model distribution will match the true data generating distribution and, consequently, it will generalize well.

Advantages and disadvantages

When we are manually deciding the loss function, we take into account some useful properties that appear in the associated cost function:

- Positivity: sometimes we prefer not to deal with negative cost.

- Existence of a global minimum: even if we are almost sure in advance that we won't reach a global minimum, the non-existence can cause time-inefficiency. Indeed, without an early stopping criterion, the algorithm may be iterated infinitely. On the other hand, even if we set a bound on the iterations, there's an high probability that the algorithm still reaches the fixed bound every time we use it. The bound is defined as the worst case scenario, then it's a case we would rather avoid.

These two properties are beneficial when they hold, however they aren't necessary: it's not always an adverse situation when they aren't obtained. In fact, we first construct and think the loss function so that it's practical and suitable to the model purpose, then we check if we can adapt it to get those two additional properties in the correspondent cost function (but it's not always achievable). This is why we often choose the MLE method to construct L even if these two properties aren't achieved: its benefits are more significant than the drawbacks.

Let us first discuss the fact that these two properties doesn't hold. In the discrete case (meaning that $\mathbf{y}$ has discrete values) p_{model} of the formula 2.13 (defining J) is a probability mass function. Then it takes value in $[0, 1]$ and its logarithm is negative, consequently the cost is always positive. In the continuous case, instead, p_{model} is a density function. Then it can be negative for $\boldsymbol{y}$ with high conditional density, losing the positivity property.
The existence of a global minimum depends on the explicit formula of J, so on the p_{model} we choose. However, in most of the models used in practice there isn't a minimum value of J. In the discrete case, we usually parametrize the conditional mass functions so that we haven't points with probability 0 or 1 (no sure or impossible events), but we can approach these values as we desire. This type of parametrization can lead to the non-existence of a minimum.

Example 2.3. *In the logistic regression model we can use the logistic sigmoid to parametrize Bernoulli distributions so that, if we fix $\boldsymbol{x}$, their parameters $p_{\boldsymbol{\theta}}(\boldsymbol{x}) = P(1 \mid \boldsymbol{x}; \boldsymbol{\theta}) = \frac{1}{1+e^{-\boldsymbol{\theta}\cdot\boldsymbol{x}}}$ and $(1 - p_{\boldsymbol{\theta}}(\boldsymbol{x}))$ can be arbitrarily close to 0 and 1 changing $\boldsymbol{\theta}$. For instance, if we have only one training example $\left(\boldsymbol{x}^{(1)}, 1\right) \in \mathbb{R}^{n+1}$, the cost function is $J(\boldsymbol{\theta}) = -\ln P\left(1 \mid \boldsymbol{x}^{(1)}; \boldsymbol{\theta}\right) = -\ln p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(1)})$. In this way $J(\boldsymbol{\theta}) \to 0$ as $p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(1)}) \to 1$ but never reaches it: it doesn't exist a vector $\boldsymbol{\theta}$ such that $J(\boldsymbol{\theta}) = 0$.*

In the continuous case, problems arise when we have a significant control on the density; in particular, when for each training example we can set an arbitrarily high peak of conditional density on the correct output and, at the same time, low conditional density on all other points.

Example 2.4. *The previous situation can happen when we are able to control also the variance (or standard deviation) parameter of a family of Gaussian distributions: suppose we have only one training example $\left(\boldsymbol{x}^{(1)}, y^{(1)}\right) \in \mathbb{R}^{n+1}$, then the cost is*

$$J(\boldsymbol{\theta}) = -\ln p\left(y^{(1)} \mid \boldsymbol{x}^{(1)}; \boldsymbol{\theta}\right) = \ln \sigma\big(\boldsymbol{x}^{(1)}\big) + \frac{1}{2}\left(\frac{y^{(1)} - \mu\big(\boldsymbol{x}^{(1)}\big)}{\sigma\big(\boldsymbol{x}^{(1)}\big)}\right)^2 + \ln\sqrt{2\pi}$$

since we took

$$p\left(y \mid \boldsymbol{x}; \boldsymbol{\theta}\right) = \frac{1}{\sigma(\boldsymbol{x})\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y-\mu(\boldsymbol{x})}{\sigma(\boldsymbol{x})}\right)^2}.$$

The mean $\mu(\boldsymbol{x})$ and the standard deviation $\sigma(\boldsymbol{x})$ are functions defined by the parameters vector $\boldsymbol{\theta}$ (for example $\mu(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{\theta}$). We can construct $\mu(\boldsymbol{x})$ so that it keeps $\mu\big(\boldsymbol{x}^{(1)}\big) = y^{(1)}$ while we *change $\sigma\big(\boldsymbol{x}^{(1)}\big)$: one possible way is to make $\mu(\boldsymbol{x})$ depend on some components of $\boldsymbol{\theta}$ and $\sigma(\boldsymbol{x})$ on the remaining ones, then fix the components associated to $\mu(\boldsymbol{x})$ so that they return $\mu(\boldsymbol{x}^{(1)}\big) = y^{(1)}$. Calling $\boldsymbol{\theta}^*$ this vector of parameters with some fixed components, the cost function become*

$$J\big(\boldsymbol{\theta}^*\,\big) = \ln \sigma\big(\boldsymbol{x}^{(1)}\big) + C$$

since the second term is zero and where $C = \ln\sqrt{2\pi}$. Decreasing the standard deviation we get an increasing peak of conditional density at $y^{(1)}$ (the correct output of the training example). As we said before, this condition can lead to the non-existence of the minimum, in fact in our case, $J\big(\boldsymbol{\theta}^*\,\big) \to -\infty$ as $\sigma\big(\boldsymbol{x}^{(1)}\big)$.

Both previous scenarios are ad hoc examples to keep the cost function J simple, however the absence of an actual global minimum can appear even in more complex cases with more training examples and more elaborate J.

Now that we've seen some MLE drawbacks, we show a few benefits it can bring. One of the major conveniences of the MLE procedure is very easy to see: it ensures us an automatic way of implementing the cost function. As we already said, choosing the right cost function given the task can be a difficult and slow work, it can need even months of testing. Using instead the MLE method, the explicit formula of the cost function comes directly from the model specifications $p_{model}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$. This restricts the designing problem only to selecting the right set of probability mass (or density) functions.

The main theoretical advantage of the MLE over other estimators is that it's been proven to be the best one as $m \to \infty$. This property is called asymptotic efficiency and it holds when an estimator is consistent (property presented a subsection of the appendix) and it has the best asymptotic covariance matrix for unbiased estimators. When m is fixed, the efficiency is usually measured with the Mean Square Error of the estimator (MSE, defined in the appendix) and efficient estimators are the ones with the lowest MSE possible. In order to check if an estimator achieves the lowest value, we usually exploit the Cramér-Rao lower bounds. In these last, the variance or covariance matrix is bounded below with a function of the bias' Jacobian and the Fisher information. For example, the Cramér-Rao lower bound for unbiased estimators (the case we are going to need) states that the lowest possible covariance matrix is

$$\operatorname{Cov}\left(\widehat{\boldsymbol{\theta}}_m\right) \geq \mathcal{I}^{-1}(\boldsymbol{\theta}) \tag{2.20}$$

where $\mathcal{I}(\boldsymbol{\theta})$ is the Fisher information matrix defined by

$$\mathcal{I}(\boldsymbol{\theta})_{ij} = \mathbb{E}_{\boldsymbol{\theta}}\left[-\frac{\partial^2 \ln p_{model}\big(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta}\big)}{\partial\theta_i \partial\theta_j}\right].$$

Remark 2.9. *We say that a matrix $\boldsymbol{A}$ is greater than $\boldsymbol{B}$ ($\boldsymbol{A} \geq \boldsymbol{B}$) if $\boldsymbol{A}-\boldsymbol{B}$ is positive semidefinite.*

In this case it's easy to see why this inequality helps us to check the efficiency. In fact, if no unbiased estimator has lower covariance matrix, then the one with that covariance matrix has the smallest MSE. This is given by the decomposition of the MSE of an estimator in bias and covariance terms (formula A.9): for unbiased estimators minimizing the their MSE is equivalent to minimizing the covariance matrix.
Going back to the asymptotic case, consistency is a usually required property when we analyse

the behaviour for $m \to \infty$. That's why in the definition of asymptotic efficiency we ask that the estimator is consistent and that its limit achieves the equality in the Cramér-Rao lower bound for unbiased estimator. In particular in our case, if there exists a unique parameters vector $\boldsymbol{\theta}$ such that $P_{model;\boldsymbol{\theta}} = P_{data}$, then the MLE estimator is consistence. Moreover, in "Chapter 36 Large sample estimation and hypothesis testing" [30] the authors prove that the following limit holds in distribution:

$$\sqrt{m}\left(\widehat{\boldsymbol{\theta}}_{MLE} - \boldsymbol{\theta}\right) \xrightarrow{d} \mathcal{N}\big(0, \mathcal{I}^{-1}(\boldsymbol{\theta})\big). \tag{2.21}$$

This proves that, between consistent estimators, the MLE one asymptotically achieves the lowest MSE possible. In other words, we can say that for m large enough $\widehat{\boldsymbol{\theta}}_{MLE}$ behaves like an unbiased estimator with the best covariance matrix $\mathcal{I}^{-1}(\boldsymbol{\theta})$.

Remark 2.10. *The previous convergence implies also that the MLE estimator is $\sqrt{m}$-consistence. This means that $\sqrt{m}\left(\widehat{\boldsymbol{\theta}}_{MLE} - \boldsymbol{\theta}\right)$ is a tight sequence and adds an important information to consistency: we already know that $\widehat{\boldsymbol{\theta}}_{MLE} - \boldsymbol{\theta} \xrightarrow{P} 0$, but now we can say that it convergence with rate $O(\frac{1}{\sqrt{m}})$.*

Another important benefit, which anticipates a topic of the next section, is that it de-saturates many cost functions. We are going to see that learning algorithms need the cost function's gradient to have a norm not too small. When the cost function saturates it becomes almost flat in certain regions, then its gradient in those regions has a small norm. The saturation problem usually comes from the activation functions, which saturate themself, of some hidden or output units. Moreover, in many cases it's caused by en exponential function, which is almost flat for very negative arguments. For instance, it occurs with sigmoid or softmax output units. The logarithm in MLE undoes the exponential providing us gradients with a reasonable norm.

2.3 Gradient Descent Introduction

In the first section of the chapter we show what learning really means: minimizing a cost function assuming that this procedure increases the performance. Now that we also know a practical approach to construct a cost function suitable for our purpose, we can study the starting point of the most used (especially in Neural Networks training) methods to optimize the cost function: the Gradient Descent method. In this section we present the basic algorithm and its core concepts, while in the next one we show the issue we can find and that we want to limit.

Basic idea

Gradient-Based methods are first-order (they use, indeed, gradient) iterative algorithm to optimize (both maximize and minimize) a function F. The Gradient Descent belongs to them and, in particular, it's employed for minimization. At every point, in fact, the method follows the direction that makes F decrease faster: the direction of steepest descent.

Remark 2.11. *It can obviously be used to maximize a function since conceptually the two optimization problems are equivalent: to obtain one from the other we simply analyse $-F$ instead of F.*

We prove the next result, often taken for granted, to understand which is the direction of fastest decrease and why:

Proposition 2.2. *If the gradient isn't null (is null when all components equal to zero), the direction of steepest descent is the negative gradient.*

Proof. We know that the usual derivative $F'(x)$ of an univariate function $F(x)$ is the slope of the tangent to the function at that point x. Then it gives us how the value of the first approximation changes when we vary the input x: $F(x+\varepsilon) \approx F(x) + \varepsilon F'(x)$, so in the first approximation if we vary the input of $+\varepsilon$ the output changes of $+\varepsilon F'(x)$. It essentially quantify how fast the function increases or decreases at that point x for small variation in the domain.
When the domain is $X \subset \mathbb{R}^n$ (we are in the multivariate case which is the more likely scenario in Neural Networks algorithm), we define the directional derivative to study each possible direction one at a time. It is a particular case of the Gateaux derivative where $F : X \subset D \to Y$ with D and Y generic locally convex topological vector spaces. The idea is to build the computation going back to the univariate case: at every fixed point $\boldsymbol{x}$ we want to make the function F vary only in one direction $\boldsymbol{v}$, so we set a function depending only on one variable $\tilde{F}(\alpha) = F(\boldsymbol{x} + \alpha\boldsymbol{v})$. Then we define the directional derivative in direction v as the usual derivative of $\tilde{F}(\alpha)$ computed at $\alpha = 0$. Given the derivative's interpretation at the beginning of the proof, we understand that the directional derivative of F in direction $\boldsymbol{v}$ is the slope of the function F at $\boldsymbol{x}$ when we make the input vary only in the direction $\boldsymbol{v}$, it measure how fast F decreases (when negative) or increases (when positive) at $\boldsymbol{x}$ for small variation in that direction.
Therefore, to get the direction of steepest descent (or direction of fastest decrease), we simply search for the $\boldsymbol{v}$ that returns the minimum between all possible directional derivatives. Since the gradient isn't null, we know that there exists at least one negative directional derivative. Then the minimum is negative, confirming us that we decrease in that direction (we are going to prove this and explain better what we mean in the next proposition). Now we first compute the directional derivative formula using the chain rule:

$$\begin{aligned} \tilde{F}'(0) &= \frac{\partial}{\partial \alpha} F(\boldsymbol{x} + \alpha\boldsymbol{v})\bigg|_{\alpha=0} = \\ &= \nabla_{\boldsymbol{x}} F(\boldsymbol{x} + \alpha\boldsymbol{v})|_{\alpha=0} \cdot \boldsymbol{v} = \nabla_{\boldsymbol{x}} F(\boldsymbol{x}) \cdot \boldsymbol{v}. \end{aligned}$$

If we take the minimum for any $\boldsymbol{v} \in \mathbb{R}^n$ of the previous quantity, then it always is $-\infty$ (if the gradient isn't null). That's why we have to compute the minimum for the vectors $\boldsymbol{v}$ with unit norm. This doesn't lead to problem for our purpose since we are interest only on the direction. Therefore we are looking for $\boldsymbol{v}$ that minimize

$$\begin{aligned} &\min_{\|\boldsymbol{v}\|=1} \nabla_{\boldsymbol{x}} F(\boldsymbol{x}) \cdot \boldsymbol{v} = \\ &= \min_{\|\boldsymbol{v}\|=1} \|\boldsymbol{v}\| \|\nabla_{\boldsymbol{x}} F(\boldsymbol{x})\| \cos(\beta) = \\ &= \min_{\|\boldsymbol{v}\|=1} \|\nabla_{\boldsymbol{x}} F(\boldsymbol{x})\| \cos(\beta) \end{aligned}$$

where β is the angle between the gradient and $\boldsymbol{v}$. Since $\|\nabla_{\boldsymbol{x}} F(\boldsymbol{x})\|$ doesn't depend on $\boldsymbol{v}$ the $\arg\min$ doesn't change is we consider:

$$\min_{\|\boldsymbol{v}\|=1} \cos(\beta).$$

The minimum is -1 and it's given by $\beta = \pi$, which means that $\boldsymbol{v}$ is in the opposite direction of the gradient: $\boldsymbol{v} = -\frac{\nabla_{\boldsymbol{x}} F(\boldsymbol{x})}{\|\nabla_{\boldsymbol{x}} f(\boldsymbol{x})\|}$. □

Remark 2.12. *If the gradient is null $\nabla_{\boldsymbol{x}} F(\boldsymbol{x}) = \boldsymbol{0}$, then we are in a critical point which can be a minimum or maximum (local or global) or a saddle point. As in the univariate case, the first derivatives don't give us enough information to know where we have to move in the next step. In fact, we can clearly see that the directional derivatives are all equal to zero: $\nabla_{\boldsymbol{x}} F(\boldsymbol{x}) \cdot \boldsymbol{v} = 0$ for every $\boldsymbol{v} \in \mathbb{R}^n$. Then all directions return us the minimum value and we don't even have a negative directional derivative to decrease the function.*

In summary, the idea is to take at each iteration a step in the negative gradient direction because we know it's the direction where f decreases faster. In fact, following it we get the minimum directional derivative (which is negative). As we said in the proof, the fact that is negative ensures us that, if we take a small enough step, F decreases:

Proposition 2.3. *If the gradient isn't null, then there exists an $\bar{\varepsilon} > 0$ such that $F\big(\boldsymbol{x} - \varepsilon \nabla_{\boldsymbol{x}} F(\boldsymbol{x})\big) < F(\boldsymbol{x})$ for every $0 < \varepsilon < \bar{\varepsilon}$.*

Proof. Assume for sake of contradiction that it doesn't exist such $\bar{\varepsilon}$. Then for every $\bar{\varepsilon} = \frac{1}{k}$ there is an $\varepsilon_k < \frac{1}{k}$ such that $F\big(\boldsymbol{x} - \varepsilon_k \nabla_{\boldsymbol{x}} F(\boldsymbol{x})\big) - F(\boldsymbol{x}) \geq 0$. Since $\varepsilon_k > 0$ we also have that

$$\frac{F\big(\boldsymbol{x} - \varepsilon_k \nabla_{\boldsymbol{x}} F(\boldsymbol{x})\big) - F(\boldsymbol{x})}{\varepsilon_k} \geq 0$$

and taking the limit we get

$$0 \leq \lim_{k \to \infty} \frac{F\big(\boldsymbol{x} - \varepsilon_k \nabla_{\boldsymbol{x}} F(\boldsymbol{x})\big) - F(\boldsymbol{x})}{\varepsilon_k} = \nabla_{\boldsymbol{x}} F(\boldsymbol{x}) \cdot \big(- \nabla_{\boldsymbol{x}} F(\boldsymbol{x})\big) = -\|\nabla_{\boldsymbol{x}} F(\boldsymbol{x})\|^2 < 0$$

obtaining the contradiction. □

Basic algorithm

The $\varepsilon > 0$ it's called learning rate since it determines the step size. In the simplest version of the Gradient Descent method it's a fixed positive constant, then at each iteration the algorithm moves from $\boldsymbol{x}$ to the point $\boldsymbol{x} - \varepsilon \nabla_{\boldsymbol{x}} F(\boldsymbol{x})$.
In our case $F = J$ and $\boldsymbol{x} = \boldsymbol{\theta}$, therefore the actual algorithm can be written as this pseudocode:

Algorithm 1 Basic Gradient Descent

```
Require: Learning rate $\varepsilon > 0$
Require: Randomly initialized parameters vector $\boldsymbol{\theta}$
  while $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \neq 0$ do
      Update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
  end while
  return $\boldsymbol{\theta}$
```

This version is very simple to understand and implement, but it has a lot of theoretical and numerical issues. We present some of them in the next subsection.

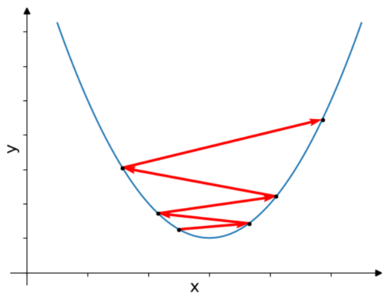
Remark 2.13. *Due to the non-differentiability of some activation functions it could be argued that this method isn't consistent. However many motivations come in support to the common practice of neglecting this problem. For instance, the set of non-differentiable points of activation functions used in practical model is always a subset of measure zero. Moreover, because of the machine precision it's almost impossible to get exactly the point of non differentiability. In real applications, if we don't want to encounter errors, we can also decide to take the left or right derivative when we are computing gradients on a non-differentiable point. Furthermore theoretical results manage to exploit other tools like subgradients or boundedness to deal with non-differentiability.*
This are the reasons why from now on we are going to suppose that the cost function $J(\boldsymbol{\theta})$ is differentiable independently from the training set chosen to define it.

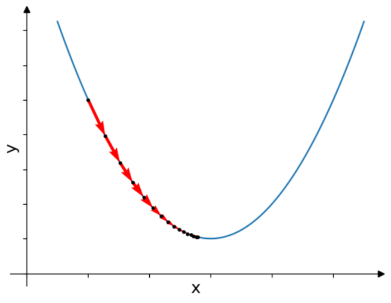
2.4 Gradient Descent Main Issues

In this section we present some of the problems we can encounter using the basic Gradient Descent algorithm. We analyse if they are a significant problem or if it can be proven that there is a small probability of them to happen. Some considerations are going to be more empirical than practical since in this field the studies often move mainly in that way. For some of them we don't provide a solution, in fact, we are going to see how the Gradient Descent variations try to solve some of them in the next chapter.

Learning rate size

We first notice that the proposition 2.3 ensures us only that in the negative gradient direction exists a segment of points near $\boldsymbol{\theta}$ that have smaller J values. However, we don't know $\bar{\varepsilon}$ a priori, so the fixed ε of the algorithm could be greater. This can lead to passing the minimum (local or global), obtaining an higher J value. Actually, if the ε is too large, it can even make the algorithm diverge as we can see in the figure on the left (plotted using Python). On the other hand, if we take a too small ε, the algorithm can become very slow performing a lot of iterations to get to the minimum. This problem is worsened by the fact that the gradient approaches the null vector as it gets closer to a critical point. Then $-\varepsilon\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$ becomes very small due to both ε and the gradient as we can see in the figure on the right.





A way to solve this problem is to change ε at each iteration. For example, we can employ the so-called *line search*: before the actual step we compare the values of $J\big(\boldsymbol{\theta} - \varepsilon\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})\big)$ for different ε and we select the one corresponding to the smallest J value.

Local minima

Even when we are sure that the algorithm decreases the function J at each step, we encounter another problem: it's not guaranteed that we reach a global minimum. Our algorithm stops when we get to a critical point but it can be a local minimum or, even worse situation, a saddle point. This and other issues (that we are going to present later) are due also to the lack of convexity in the Neural Networks models, which is one of the reasons why we have to use iterative method like Gradient Descent. In fact, we know that every local minimum of a convex function must be also a global minimum. We can still have a flat region at the bottom, but this only gives us more than one global minimum. In Neural Networks models, instead, the presence of the activation functions leads to non-linearity that can cause the loss of convexity. This obviously becomes more probable the more we add hidden layers, we can even say it's substantially sure to have many of them in deep Neural Networks.
The presence of multiple local minima can be easily seen noticing the weight space symmetry: if we switch, maintaining the order of the inputs, all the incoming weights of a node j_1 with all the ones of another node j_2 in the same layer (or we switch the incoming weights as a vector to directly keep the order), meaning w_{ij_1} swapped with w_{ij_2} for every node i in the previous layer, and if we perform the same for the bias terms and the outgoing weights, then we obtain an equivalent Neural Network. In particular, if we have an N-layers network, we get $\prod_{k=1}^{N-1} n_k!$ equivalent ways of disposing the weights, where n_k is the number of nodes in the k-th layer. Thus, for every local minimum we have $\prod_{k=1}^{N-1} n_k!$ equivalent local minima where our algorithm can get stuck. What we are essentially doing with this procedure is exchanging latent variables. The definition in the book "A Dictionary of Statistics" [41] published by Oxford University states that:

Definition 2.2. *A latent variable is "An unobserved variable that may account for variation in the data and/or for apparent relations between observed variables."*

Then every hidden unit is a latent variable of our framework. The characteristic of latent variable is that if we have an observed example $(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)})$ we can't know directly how the latent variables behave in the underlying structure: they must be inferred/learned by the algorithm. Using the example of representation learning (presented in the subsection on Deep Learning in the previous chapter), we don't know how simpler concepts represented by the hidden layers are linked and weighted to construct the complex representation of the output layer (in image recognition the hidden layers represent edges, corners, contours, objects part, while the output layer represent the entire image). How latent variables affect the output must be inferred using the observable variables, which are quantifiable. At the end of the process we can clarify explicitly the relations (for example, we can write $p_{model}(\boldsymbol{y} \mid \boldsymbol{h}, \boldsymbol{x}; \boldsymbol{\theta})$) or it's possible to describe the distribution of observable variables even without explaining the latent variables contribution (continuing the example, $p_{model}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$). Swapping equivalently parametrized latent variables to obtain different local minima is a general Machine Learning's problem not specific to Neural Networks. In particular, the ability to build different equivalent models is one of the main causes of a larger issue called model unidentifiability. Identifiability is a Statistics concept, we show the definition from the book "Theory of Point Estimation" [24]:

Definition 2.3. *A statistical model $\mathcal{P} = \{ P_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \Theta \}$ is identifiable if for every $P \in \mathcal{P}$ there exists only $\boldsymbol{\theta} \in \Theta$ such that $P_{\boldsymbol{\theta}} = P$. In other words, $\boldsymbol{\theta} \to P_{\boldsymbol{\theta}}$ is a one-to-one correspondence.*

In Machine Learning, this translate into a unique parametrization for every model in our framework. It can be proven that it is equivalent to the fact that with an infinite amount of observed examples we can identify the true parameters vector $\boldsymbol{\theta}$ of the real underlying model. In the case of absence of this property (unidentifiability), instead, the parameters vector isn't always uniquely determined even if we have all input-output relations. As we have seen, the unidentifiability can result into many local minima.
Neural Networks can cause unidentifiability in many ways other than latent variables swapping. For instance, if we have a ReLU unit we can multiply the incoming weights vector and the bias term by a factor $\alpha \neq 0$ and the outgoing weight vector by a factor $\frac{1}{\alpha}$. In this way, if the cost function depends only on the network output and not directly on the weights (it hasn't regularization terms like weight decay $\lambda \|\boldsymbol{\theta}\|^2$, where λ is an hyperparameter to be set), we obtain an equivalent model. Suppose we have a local minimum at $\boldsymbol{\theta}^*$, the incoming vector and bias term of a node i in the k-th layer form a vector $v_i^* = \left(\boldsymbol{W}_{:,i}^{(k-1)*}, b_i^{(k-1)*} \right)$ (we assume it different from the zero vector) and the outgoing vector is $u_i^* = \boldsymbol{W}_{i,:}^{(k)*}$. While the former changes as $v_i = \alpha_i v_i^*$, the latter varies as $u_i = \frac{1}{\alpha_i} u_i^*$. Then they keep their directions modifying only their norms. Writing the norm of the latter as a function of the first we get:

$$\|u_i\| = f(\|v_i\|) \implies \left\| \frac{1}{\alpha_i} u_i^* \right\| = f(\|\alpha_i v_i^*\|)$$
$$\implies \frac{\|u_i^*\|}{|\alpha_i|} = f(|\alpha_i| \|v_i^*\|) \implies \|u_i\| = \frac{\|u_i^*\| \|v_i^*\|}{\|v_i\|}$$

where the last implication is given by $|\alpha_i| = \frac{\|v_i\|}{\|v_i^*\|}$. We've just obtained an equation of the type $y = \frac{c}{x}$. Therefore, when we make one α_i vary, we move on an hyperbole of equivalent local minima. Since we can make vary α_i for every hidden node, we can modify $\sum_{k=1}^{N-1} n_k$ parameters. This shows that the hyperboles are all contained in a $\sum_{k=1}^{N-1} n_k$-dimensional submanifold.

Remark 2.14. *We assumed v_i^* not null, since otherwise we would have been in a more general situation where we can change the outgoing vector as we want because that hidden node would have value equal to zero. Notice that it makes $v_i = \alpha_i v_i^*$ not null, thus it allows us to perform the previous computation putting $\|v_i^*\|$ and $\|v_i\|$ at the denominator.*

As we have just seen we can get a very large number of local minima when we have model unidentifiability (almost always occurs with Neural Networks). However the problem represented by this type of local minima is downsized by the fact that they are all equivalent models, so they return the same cost function value. In other words we have many local minima but all with the same value, therefore in the event that the value is very low it could be a benefit to have such amount of points with a cost reasonable for our purpose. This shows us that the real problem isn't the presence of many local minima but rather the difference between their cost and the global minimum one. Many researchers believe that most local minima have a low cost value when the network is large enough. But since it's not been proven yet, it's still heavily discussed whether there exist or not a large amount of high cost local minima in the types of networks which are most used in practice (or at least researchers give priority to them). We can also deal with the problem checking whether the most common optimization algorithms run into them or not, nevertheless the most used and practical way is to conduct specific test on the employed model: we want to analyse the behaviour with the training set, Neural Network, cost function and optimization algorithm of that particular framework.

Saddle points

There exist critical points that could establish more problematic issues than local minima: the saddle points. This type of points isn't neither a minimum nor a maximum (it has both lower and higher points nearby) but it's still a critical point. While the maxima are avoided by Gradient Descent and its variations, so aren't a problem, saddle points may be encounter instead. When this happens, the algorithm stops at a point where we can still decrease the cost function moving with a sufficiently small step.

In this paragraph we try to derive some properties on saddle points looking at the Hessian matrix. First of all, to check the second derivative in a given direction we use twice the directional derivative in that direction: let $\partial_{\boldsymbol{v}}$ denote the directional derivative operator in direction $\boldsymbol{v}$, then we notice

$$\begin{aligned}\partial_{\boldsymbol{v}}^2 F(\boldsymbol{x}) &= \partial_{\boldsymbol{v}}(\partial_{\boldsymbol{v}} F(\boldsymbol{x})) = \partial_{\boldsymbol{v}}\big(\nabla_{\boldsymbol{x}} F(\boldsymbol{x}) \cdot \boldsymbol{v}\big) = \\ &= \big(\boldsymbol{J}(\nabla_{\boldsymbol{x}} F)(\boldsymbol{x}) \cdot \boldsymbol{v}\big)^\top \cdot \boldsymbol{v} = \boldsymbol{v}^\top \cdot \boldsymbol{H}(F)(\boldsymbol{x}) \cdot \boldsymbol{v}\end{aligned}$$

where $\boldsymbol{J}(\nabla_{\boldsymbol{x}} F)(\boldsymbol{x})$ is the Jacobian of the gradient of F, which is the Hessian matrix $\boldsymbol{H}(F)(\boldsymbol{x})$ of F, both computed at $\boldsymbol{x}$. The directional second derivative in direction $\boldsymbol{v}$ is computed using the Hessian as a bilinear form with both arguments equal to $\boldsymbol{v}$. Therefore, to test the type of a critical point we look at the Hessian matrix signature. In particular, at the local minima the Hessian is positive semi-definite (all $+1$ and/or zero in the signature matrix), while in the local maxima it's negative semi-definitive (all -1 and/or zero in the signature matrix). In the former case, in fact, all directional second derivatives are non-negative providing the minimality of the critical point, while in the latter case all directional second derivatives are non-positive providing the maximality. At saddle points we can get both $+1$ and -1 in the signature matrix. This property helps us prove that for large dimension n of the domain (of the function we are analysing) the expected fraction between number of saddle points and minima approximately grows exponentially with respect to n.
We now propose some ideas and key steps for the proof of this claim. We follow the article "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization" [6] that summarizes a few results on the topic. In particular the authors cite "Statistics of critical points of Gaussian fields on large-dimensional spaces" [2] for this approach looking at the Hessian matrix. As stated in the title, the article specifically studies critical points of Gaussian fields. General random fields are often defined as a generalization of stochastic process (even if in some books this wider definition is the process one) where the time index lives in an Euclidean space or a generic topological space:

Definition 2.4. *A random field F is a collection of random variables $F_{\boldsymbol{x}}$ all with values in the same space Y. The indeces $\boldsymbol{x}$ belong to a topological space X (usually contained in $\mathbb{R}^n$).*

The idea behind this definition is to construct a random function $F(\boldsymbol{x})$ with a random value at each point $\boldsymbol{x}$ of its domain X. Depending on the random variables type we get different classes of random field/function. One of the most used is obviously the one of the article: the Gaussian random field, where the random variables are all described by Gaussian distributions. It can be seen as a generalization of Brownian motions. The importance of this type of fields is stressed by the Central Limit Theorem, informally giving us the intuition that random fields tend to this type in some sense.

With a formal definition of random function we can study the average behaviour of the Hessian and its eigenvalues. Before doing so, we recall one of the main theorems used in the article: the Wigner's semicircle law for symmetric Gaussian random matrices (also called Gaussian Orthogonal Ensemble, GOE), which are matrices with normal independent entries. The theorem is well explained in "Wigner Semicircle Law for Gaussian Random Matrices" [19] by Tianchong Jiang. The author defines the eigenvalues distribution ν_{A_n} of a GOE $A_n \in M_{n\times n}(\mathbb{R})$ as a measure on $\mathbb{R}$ using the Dirac delta: the density of the measure is

$$d\nu_{A_n}(\lambda) = \frac{1}{n}\sum_{i=1}^{n} \delta(\lambda - \lambda_i) \tag{2.22}$$

where $\lambda_1, \dots, \lambda_n$ are the eigenvalues counted with their multiplicity. Since A_n is a random variable (with value in the space of symmetric matrices), we can set the average eigenvalues distribution for that particular dimension n:

$$\nu_n = \mathbb{E}\big[\nu_{A_n}\big]. \tag{2.23}$$

Then we have the tools to state the theorem:

Theorem 2.1 (Wigner's semicircle law for GOE)**.** *The expected eigenvalues distribution ν_n weakly converges to the standard semicircle distribution ν_{SC} as $n \to \infty$.*

The semicircle distribution ν_{SC} is the one defined by the density

$$d\nu_{SC}(\lambda) = \frac{2}{\pi R^2}\sqrt{R^2 - \lambda^2}\,\mathbb{1}_{[-R,R]}(\lambda) \tag{2.24}$$

and the standard one conventionally has $R = 2$.

Remark 2.15. *The weak convergence reported in the theorem is the convergence in distribution:*

$$\nu_k \xrightarrow{weak} \nu \quad \text{if} \quad \int_{\mathbb{R}} h(x)d\nu_k(x) \to \int_{\mathbb{R}} h(x)d\nu(x) \;\; \forall h \in C_b(\mathbb{R})$$

or if we consider Z_k with distribution ν_k and Z with distribution ν

$$\nu_k \xrightarrow{weak} \nu \quad \text{if} \quad \mathbb{E}\big[h(X_k)\big] \to \mathbb{E}\big[h(X)\big] \;\; \forall h \in C_b(\mathbb{R})$$

Now that we have recalled the Wigner's semicircle law, we go back to the article "Statistics of critical points of Gaussian fields on large-dimensional spaces", which was cited before. Here the authors prove that, with some modifications, the theorem applies also to Hessian matrices computed at critical points of Gaussian random fields. For now we don't take into account the theorem changes in order to not complicate the notation, we can proceed anyway with similar

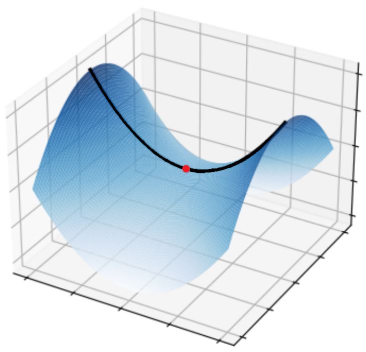
arguments adjusting the considered quantities. The two main consequences to consider for our goal are that at any fixed critical point, for large Hessian matrices (i.e. great n dimension), the zero eigenvalue has zero probability and the probability to get a positive eigenvalue is $\frac{1}{2}$ (equal to the probability for negative eigenvalues).
With the former consequence we can use the characterization of critical points: since the Hessian is invertible with probability equal to 1, if it's positive definite we are in a local minimum, if it's negative definite in a local maximum, otherwise we are in a saddle point. In other words we also have the viceversa of the previously stated property of the signature.
With the latter consequence we know that, if we are at a critical point, we have a probability of $\frac{1}{2^n}$ to get a signature matrix with all $+1$ elements, which is equivalent to being in a local minimum. The probability of being in a saddle point, instead, it's equal to the probability of not being neither in a local minimum nor in a local maximum: $1 - \frac{1}{2^n} - \frac{1}{2^n} = \frac{2^{n-1}-1}{2^{n-1}}$.
Therefore, for large n, the ratio between the probability of being in a saddle and the one of being in a local minimum approximately grows exponentially with respect to the dimension n: $ratio = \frac{2^{n-1}-1}{2^{n-1}} / \frac{1}{2^n} = 2^n - 2$. This reasoning on probabilities can be actually transposed to the numbers of saddle points and local minima, obtaining the previously stated result: the expected fraction between number of saddle points and minima approximately increases exponentially.

The modifications we mention before are primarily in the theorem's thesis and they provide us properties useful for our purpose. In fact we have that: at a fixed critical point with a fixed function value E, the eigenvalues distribution of the Hessian given E weakly convergences to a shifted semicircle distribution depending on E. Therefore, the mean value given E isn't 0 but it's a quantity depending on E, which is the value of function at that point. It's denoted with E since the article follows also a physical interpretation and the analysed function represents a potential energy. With very low values of E, as at the global minimum, the mean is so high that the distribution has a density of zero on all negative eigenvalues. Thus all eigenvalues are almost surely positive, showing us that we are at a minimum. As E increases the semicircle spectrum moves to the left decreasing its mean value and leading to a larger number of negative eigenvalues.
Then, if we look at the level set of the function, while we increase its given value E we encounter more and more saddle points and much less local minima, obtaining the previous exponential relation. At a certain point we obviously get the inverse result seeing more maxima and less saddle points since the semicircle spectrum is going to the negative part. Using this considerations, we can say that if our Gradient Descent algorithm is stuck in a local minima, then it probably has low cost value.

We can show this correspondences also looking at how critical points are plotted in a Cartesian plan with E on one axis and α on the other. In the article α is used to denote the fraction of negative eigenvalues of the Hessian at the critical point. They are nearly all placed on a curve which increases as α varies between 0 and 1. In other words, it confirms us that for higher value of the function we see a larger number of negative eigenvalues (passing from a lot of local minima, to a lot of saddle points, to a lot of local maxima). The exponential relation can be rewritten with this different prospective: if n is large enough and we fix a constant D, the probability of finding a critical point at a distance from the curve greater than D decreases exponentially with respect to n.

To summarize, we have just proven that for Gaussian random functions with high-dimensional domains (which is usually the Neural Networks case) saddle points are much more than local minima and local minima are most likely to have a low function value. These properties extend also to other types of random functions. However, it's still an open question whether they apply to most types of Neural Networks or not. Some studies theoretically show that all local minima have the same value of the global one when we train an autoencoder with only one hidden layer and no non-linear activation functions. Further generalizations are more heuristic assumptions. The previous type of Neural Networks, in fact, can provide qualitative information on non-linear Neural Networks even if it is essentially a composition of linear function, so it's linear in the input but it is non-convex in the weights and parameters. We can find some considerations in literature to extend the properties to deeper Neural Networks.

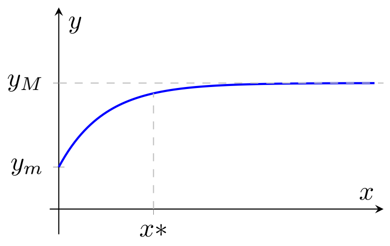
In conclusion, there is a big chance of having a lot of saddle points with high cost value in a Neural Networks model. However we still haven't enough information to state if it really represents a problem in common networks used in practice. In fact, empirically, it seems that, even if the gradient is small near this type of points, the Gradient Descent is usually rejected by these critical points. The reason could be that the steepest descent direction rarely directs the algorithm towards these points. In other words there may not be many points with negative gradient direction pointing at the saddle points. In the next figure we plot the graph of the usual hyperbolic paraboloid $F(x,y) = x^2 - y^2$ as an example. We can easily notice that most of the points (even very close to the saddle point (0,0)) have a negative gradient not pointing towards (0,0). The reason is that most of the time there are better direction to decrease the function: saddle points aren't local minimum, then even really close to them there exist lower points and the algorithm is able to recognize that it's better to go towards one of those ways. The only points where the steepest descent direction leads to (0,0) are represented by the black parabola in the figure. In the (x,y) Cartesian plane they all belong to the line of points with $y = 0$: $\{(x,0) \mid x \in \mathbb{R}\}$, which is a set of Lebesgue measure zero.



Remark 2.16. *When we don't get our desired results using a particular model, we can try to exclude problems associated to local minima and saddle points using a simple test. We can, in fact, compute or plot if the norm of the gradient become very small with the last iterations: the gradient should approach the null vector near critical points. This test can be employed not only for Neural Networks but in general for models using gradient-based optimization methods.*

Plateaus and Cliffs

By their formal definition, plateaus are regions of the graph where the function output is constant. However, most of the times the definition is, not formally, extended to regions where the function value presents very little changes when we vary the function input. One of the most famous example is the so-called Exponential plateau: the one shown after a certain point x^* in the graph of the function $F(x) = y_M - (y_M - y_m)e^{-kx}$ with $y_M \in \mathbb{R}$, $y_m < y_M$ and $k > 0$.

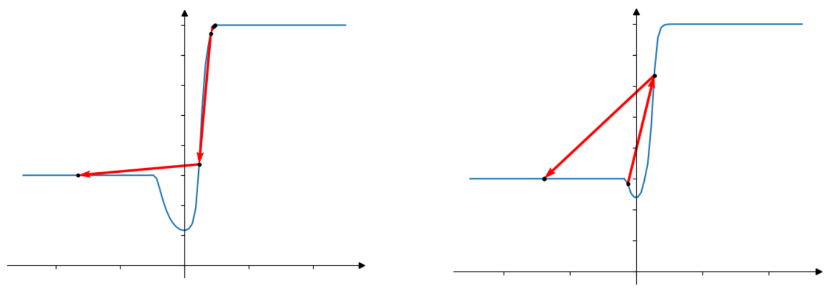


Totally flat regions are a significant problem for all numerical optimization algorithms: for gradient-based methods, for second order methods which use also the Hessian information and for many other types of methods. In fact, not only the gradient is null but also the Hessian is the zero matrix (so all eigenvalues are zero).
Even the second type of plateaus can bring us a lot of issues. In fact, even if the plateau is slightly decreasing (like in the previous figure), the gradient is almost null there, then the algorithm becomes very slow when it passes through this type of regions.
Moreover, considering again the results of the previous subsection, we can notice that if the value E is large enough, then the shifted semicircle distribution has positive density on eigenvalues near zero. This implies that if we have a critical point with high value E, so most probably a saddle point, then we encounter a plateau around it. Therefore, saddle points seem to imply an important problem again.

In contrast with plateaus, we present now another issue for gradient-based methods: the cliffs. They are, in fact, associated with the presence of gradient with too large norm. Contrary to the previous issue, in these regions the algorithm doesn't slow down but, instead, it accelerates too much. This brings to an excessive step in the iteration starting from a point on a cliff. Then, the opposition between the properties of these two types of regions obviously is reflected in the optimization problems they are associated to: plateaus generate the vanishing gradient problem, while cliffs generate the exploding gradient problem.

Remark 2.17. *Cliffs are more probable when we have numerous layers since they can be caused by the multiplication and composition of many great weights.*

We can see two examples built with Python: both graphs are obtained bumping down the logistic sigmoid function subtracting the standard mollifier η defined in 2.6 (or one of its reparametrization defined in 2.7).



In the first figure we approach the cliff from above while in the second one from below. Both paths lead to the same result: an iteration starts from a point where the gradient has a big norm, then the step $-\varepsilon\nabla_{\boldsymbol{x}}F(\boldsymbol{x})$ becomes too large even with small learning rate (it fails to balance the gradient norm).
A way to solve the problem is the gradient clipping heuristic. More methods fall in this category and they are shown to be very similar empirically. A simple and intuitive approach is to make the step depending only on the learning rate ε: we normalize the gradient dividing it by its norm. The negative gradient is, in fact, the direction of steepest descent in the local area near the point, however its norm doesn't imply anything on the distance we can travel to be sure to decrease the function value (i.e. on the step size). The step is then defined by $-\varepsilon\frac{\nabla_{\boldsymbol{x}}F(\boldsymbol{x})}{\|\nabla_{\boldsymbol{x}}F(\boldsymbol{x})\|}$. Theoretically it doesn't ensure us that we don't take a too large step, but it reduces the probability of being in such a case.

Remark 2.18. *Both plateaus and cliffs are common in recurrent Neural Networks. This occurs since we keep computing with the same function (processing as input the output of the previous iteration). We get to the last element of the sequence composing multiple times this function. Therefore if it has a small gradient in some points, then the gradient of the composition becomes smaller and smaller as we extend the sequence. It behaves analogously if the gradient of the starting function is large and it usually reflects on the cost function. Then, in conclusion, the cost function computed for long temporal sequences will have many plateaus and cliffs.*

Convergence

In general Machine Learning, linear models can use convex optimization algorithm to grant global convergence like in SVM or in logistic regression, or they can even have close form solution directly given by solving $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}) = 0$ like in linear regression. In the case of Neural Networks with Gradient Descent, instead, the absence of convexity and/or a too large ε don't ensure convergence starting from any initial parameters vector $\boldsymbol{\theta}$, like in convex optimization. Even when we have convergence it's not sure whether we arrive at a local minimum, global minimum or saddle point and we also don't know if the critical point found is much higher than the global minimum. This suggests us that Neural Networks optimization is strongly dependent on initial parameters.

Remark 2.19. *Actually, convex optimization convergence starting from any point isn't sure, too. Or rather, it is sure from a theoretical point of view, but even if it's stable and robust we could have to deal with numerical problems.*

The are multiple parameter initialization strategies, but the general key point is to randomly initialize many parameters vectors and then choose the one that leads to the lowest test error. The entire procedure is:

1. randomly initialize a parameters vector $\boldsymbol{\theta}_i$;
2. use the Gradient Descent starting from it, obtaining $\bar{\boldsymbol{\theta}}_i$;
3. compute the test error associated to $\bar{\boldsymbol{\theta}}_i$;
4. repeat from point 1 multiple times and select $\bar{i}$ such that $\bar{\boldsymbol{\theta}}_{\bar{i}}$ gives the minimum between the test errors.

The random initialization of the parameters can have many distribution prioritizing different aspects or including some a priori knowledge of the researchers. We often prefer to initialize all weights to small values different from zero. In other situation (for example when we have vast number of parameters) we prefer the so-called sparse initialization that assigns the zero value to a lot of weights. For example, we can impose that for each unit a fixed number of outgoing weights is equal to zero.

Regarding the convergence, a way to attack the problem is to set some early stopping criteria which stop the algorithm before getting to a critical point. In our basic algorithm, presented in the previous section, we could be stuck in the "while" loop for infinite iterations.
The basic stopping criterion is obviously to set a fixed amount of iterations that the algorithm must not exceed. In this way we have a last control to break the "while" loop if all other specific stopping criterion keep failing.
Now, we notice that the presence of local minima, high saddle points surrounded by plateaus and the gradient explosion problem can make the training phase hard if we want to achieve optimization in a general sense (especially since we almost always are in the high-dimensional domain case). What we can accept is that for practical purposes we don't need an actual minimum (not even a local one), we only need a cost function value which is low in a sense reasonable for our aim. Therefore an admissible stopping criterion can be: setting a tolerance tol that stops the algorithm if we get to a parameters vector $\boldsymbol{\theta}^*$ such that $J(\boldsymbol{\theta}^*) < tol$. Another stopping criterion based on the cost function J behaviour is to stop the algorithm if we don't decrease the value for fixed quantity of iterations. Observe that when for practical reasons we are employing a surrogate loss function, we usually tend to set stopping criterion on the real underlying loss function.
Being near a critical point is another situation that can cause an excessive number of iterations without really decrease J by a good amount. In fact, as we have already said before, near critical points the gradient become very small. A good stopping criterion, then, is to stop the algorithm whenever the gradient stays small for fixed amount of iteration.

Remark 2.20. *Early stopping can be useful to limit overfitting, too. As we said at the beginning of the chapter, we follow the empirical risk minimization to train our Neural Networks models. This method is subject to overfitting. In fact, minimizing the training error too much it could occur that the model simply memorizes the training set without extracting information for not seen examples. Then it's difficult that the learned $\bar{\boldsymbol{\theta}}$ achieves such performances on the test error. It's more probable that the model perfectly predicts almost every training example but it doesn't generalize well, leading to overfitting. With early stopping criteria we can make the algorithm halt before this happens.*

Ill-conditioning

Gradient Descent, but also other gradient-based methods, essentially follows the information given by the first order approximation (or Taylor expansion). Therefore it lacks of the information given by higher order approximation. One of the main problems caused by this flaw is related to the ill-conditioning of the Hessian matrix: an high condition number of the Hessian often slows down the algorithm.

The condition number is a general property of non-singular matrix. Let A be a non-singular matrix belonging to $M_{n\times m}(\mathbb{R})$ and $b \in \mathbb{R}^n$ a vector, the solution $x \in \mathbb{R}^m$ of the linear equation

$Ax = b$ can be computed as $x = A^{-1}b$ (b becomes our input). When we have an error e in the input b (e.g., error in measurements), the error in the solution is $A^{-1}e$. Let $\|\cdot\|_\beta$ a norm on the solutions space $\mathbb{R}^m$ and $\|\cdot\|_\alpha$ one on the inputs space $\mathbb{R}^n$. Then, the norm of the relative error in the solution x is $\frac{\|A^{-1}e\|_\beta}{\|A^{-1}b\|_\beta}$, while the norm of the relative error in the input b is $\frac{\|e\|_\alpha}{\|b\|_\alpha}$. The condition number is the ratio between these two norms:

$$k(A) = \max_{e,b\neq \boldsymbol{0}} \frac{\frac{\|A^{-1}e\|_\beta}{\|A^{-1}b\|_\beta}}{\frac{\|e\|_\alpha}{\|b\|_\alpha}}. \tag{2.25}$$

We can rewrite the condition number to obtain the dependence on the induced matrix norms:

$$\begin{aligned} k(A) &= \max_{e,b\neq \boldsymbol{0}} \frac{\|A^{-1}e\|_\beta}{\|e\|_\alpha} \frac{\|b\|_\alpha}{\|A^{-1}b\|_\beta} = \\ &= \max_{e\neq \boldsymbol{0}} \frac{\|A^{-1}e\|_\beta}{\|e\|_\alpha} \max_{b\neq \boldsymbol{0}} \frac{\|b\|_\alpha}{\|A^{-1}b\|_\beta} = \\ &= \max_{e\neq \boldsymbol{0}} \frac{\|A^{-1}e\|_\beta}{\|e\|_\alpha} \max_{x\neq \boldsymbol{0}} \frac{\|Ax\|_\alpha}{\|x\|_\beta} = \\ &= \|A^{-1}\|_{\beta,\alpha}\|A\|_{\alpha,\beta} \end{aligned}$$

In particular, if $\|\cdot\|_{\alpha,\beta}$ and $\|\cdot\|_{\beta,\alpha}$ are the norm induced by the Euclidean norm $\|\cdot\|_2$, then we obtain

$$k(A) = \frac{s_{max}(A)}{s_{min}(A)} \tag{2.26}$$

where $s_{max}(A)$ is the highest singular value and $s_{min}(A)$ is the lowest one. Moreover, if the matrix A is normal (the case of our interest), we get

$$k(A) = \max_{i,j} \left|\frac{\lambda_j}{\lambda_i}\right| = \frac{|\lambda_{max}(A)|}{|\lambda_{min}(A)|} \tag{2.27}$$

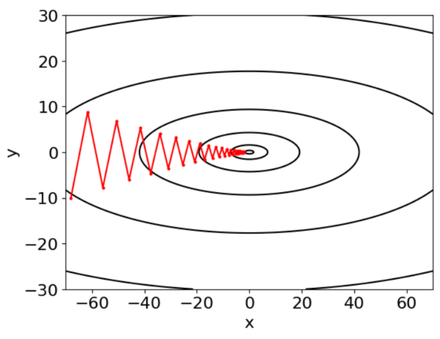
where λ_{max} is the eigenvalue with the maximum absolute value and λ_{min} is the one with the minimum absolute value.

A real matrix A is normal if $A^\top A = AA^\top$. Therefore, when the second derivatives are continuous, the Hessian matrix is normal since it's symmetric. That's why we can use the formula 2.27 to see if the Hessian is ill-conditioned or not. With that formulation we can show an easy interpretation on what the condition number means in our context. If we consider only unit vectors $\boldsymbol{u}$, then the eigenvalues are the second directional derivatives in the directions $\boldsymbol{u}$ of the corresponding eigenvectors: $\lambda_{\boldsymbol{u}} = \boldsymbol{u}^\top H(F)\boldsymbol{u}$. Hence the condition number measures how much two second directional derivates can be different in proportions. If there exists second directional derivatives with absolute value much higher than others, then we face ill-conditioning since the condition number is clearly large.

We understand what ill-conditioning means for Hessian matrices, now we see why it leads to problem in both convex and non-convex optimization. First we recall that the second directional derivative in a direction $\boldsymbol{u}$ basically quantify how much the first directional derivative changes with small modifications of the input in that direction $\boldsymbol{u}$. Then, while the negative gradient gives us the direction of steepest descent, the Hessian shows us important information on how fast this negative gradient could change. We don't encounter big issues if the second directional

derivatives are similar enough in all directions. Instead, in the ill-conditioned case, the negative gradient at some points could be almost aligned to the direction with larger curvature, not moving much in other directions. However it isn't the most efficient direction that the algorithm should follow. In fact, it still is the direction of steepest descent at those particular points, nevertheless there are better directions to decrease the function faster overall. The problem is caused by the fact that the Gradient Descent can't see the information given by the Hessian: the algorithm can't detect how fast the first derivative change in a desired direction. Therefore the algorithm doesn't know in which direction the first directional derivative stay negative for longer.
For instance, it doesn't detect that in the direction with the lowest curvature the first directional derivative increase slowly, then the function decreases for longer since the first derivative remains negative for longer. Instead in the direction with highest curvature, even if the first directional derivative is a very low negative number (meaning the direction of steepest descent is nearly align to this one), it increases fast leading quickly to a positive derivative. Then, in that direction the function value goes down and up again rapidly and this makes the algorithm overtake the minimum in that direction when we have a not small enough learning rate. On the other hand, a smaller learning rate makes even tinier the movements in the direction with lower curvature.

We show here an example where we plot the level sets of the function $F(x, y) = x^2 + 20y^2$ with Python and we use Gradient Descent on it. The Hessian is $H(F)(x, y) = \begin{pmatrix} 2 & 0 \\ 0 & 40 \end{pmatrix}$ then the direction corresponding to the highest eigenvalue is $(0, 1)$ while the one corresponding to the lowest is $(1, 0)$. The condition number is $\frac{40}{2} = 20$ which is large, causing ill-conditioning. At every step the direction of steepest descent prioritize the direction $(0, 1)$ but it overtakes the minimum of the parabola obtained by the section of the graph with a plane parallel to the gradient and the z-axis. Then we get a zig-zag path since the algorithm can't know that it should move mainly in direction $(0, 1)$.



Computational cost

When we implement an algorithm, we have to analyse practical issues in addition to the theoretical ones. In fact, we want the algorithm to be correct but also efficient. That's the reason why we discuss about the computational cost of Gradient Descent. As we can see from its definition 2.2, the cost function needs all the training examples to be computed. This affects the gradient since it can be seen as a sum on the training examples, too. It can be shown with some simple computations and interchanging differentiation with the finite sum:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L\big(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\big) \right) = \\ &= \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\big) = \\ &= \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim \widehat{P}_{train}} \Big[\nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y}\big) \Big].\end{aligned} \tag{2.28}$$

In other words, the gradient is the expected value of the per-example loss' gradient with respect to the empirical distribution defined by the training set. At each step of the algorithm we have to compute the gradient and it becomes more and more expensive in computational cost as we increase the number of training examples. In particular, if the operations to determine the per-example loss' gradient are a fixed amount $C \in \mathbb{N}$, then the computational cost (defined as the total number of operations) to compute the complete gradient is $O(Cm) = O(m)$ as a function of m in the big-O notation. Notice that this is the computational cost for each step, to obtain the total operations cost we have to multiply it by the number of iterations needed in order to get convergence. In general, optimization algorithms that exploit the entire training set are called batch (literally "quantity produced at one time") or deterministic algorithms.

In real life applications the database are significantly enlarging as years pass. Therefore, this is becoming a wider problem not only for Neural Networks models but also for general Machine Learning. We need this large dataset since modern models are more complex and so they need more examples to generalize well.

Remark 2.21. *Given the previous computation, we notice that the considerations made in remark 2.13 can be shifted to requiring properties on L: from now on, we are going to assume that we don't encounter problems on L differentiability or that they are solvable.*

Chapter 3

Gradient Descent Improvements

In the previous chapter we explain what *learning* (or equivalently, *training*) means, how to construct a loss function (hence a cost function) and describe the starting point of all optimization iterative methods for Neural Networks. The first step in order to optimize or improve a Neural Networks model is obviously to solve the problems of the basic version of the model. In particular we want to focus on the issues related to the Gradient Descent and not on how we can design a loss function with the best properties (instead of just using MLE).

In this chapter we are going to present some of the main variations of the Gradient Descent algorithm. We try to analyse them in order to show their improvements but also their limits. In particular, we start from the Stochastic Gradient Descent that is the actual basic algorithm from which the implementations of other methods start. Then we study the momentum (a very useful tool to implement new methods) and algorithms which adjust their learning rate differently in each direction. To show these variations, we are going to follow mainly the notions in the book "Deep Learning" [13].

3.1 Stochastic Gradient Descent

Most of the algorithms applied in practice are an extension of this one. In fact, there are very few which directly derive from the basic Gradient Descent. That's because even if CPU and GPU are getting faster, the computational cost issue is still a problem. We have to consider, in fact, that even database, so training set size, are increasing a lot. The Stochastic Gradient Descent (SGD) try to solve the problem resulting by the usage of all training examples.

The most intuitive idea to fix this problem is to not compute the exact gradient but to approximate it. The clever approach taken by the Stochastic Gradient Decent is to estimate the gradient simply reducing the elements over which the sum is calculated. In this way we keep a gradient cost of the same type and very similar to the exact gradient in its "shape" (the formula remain basically the same). Doing so we can convey most of the exact gradient's properties to its approximation. Furthermore we keep the previous structure of the algorithm, we only have to make a little adjustment in the gradient computation.
We now explain what we perform in practice. At each iteration we uniformly sample a random subset of the training set with a fixed size m'. We call this subset *minibatch* and we denote it

with $B = \{ \big(\boldsymbol{x}^{(i_1)}, \boldsymbol{y}^{(i_1)}\big), \ldots, \big(\boldsymbol{x}^{(i_{m'})}, \boldsymbol{y}^{(i_{m'})}\big) \}$ or with the subset of indexes $B = \{ i_1, \ldots, i_{m'} \}$ (when we want to lighten the notation). Then we average the per-example loss' gradients of these examples obtaining this estimate:

$$\widehat{\boldsymbol{g}} = \frac{1}{m'} \sum_{h=1}^{m'} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i_h)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i_h)}\big). \tag{3.1}$$

Lastly, we upgrade the parameters using the Gradient Descent rule replacing the exact gradient with its approximation: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \widehat{\boldsymbol{g}}$.

Remark 3.1. *As stated in "Stochastic Approximation in Online Steady State Optimization Under Noisy Measurements" [16], Stochastic approximation algorithms are iterative methods usually used to find roots or extrema when we have noisy observations of an unknown function. In these methods we exploit the observed random samples to approximate the actual function that we can't evaluate.*
SGD belongs to this type of algorithms since it can be seen as a stochastic approximation of the Gradient Descent. In particular, it recalls the Robbins-Monro Stochastic Approximation (RMSA) algortihm. Therefore, the word "Stochastic" in its name comes from the fact that at each iteration we don't know the real gradient (it's too expensive to be computed) so we approximate it using noisy observations. In our case these noisy observations are the sums over different random subsets of the training set.

Minibatch size

The size m' of the subsets B can be fixed a priori or set as hyperparameter. With the hyperparameter strategy we further divide the dataset in training set, test set and validation set. We train the model with different values of the hyperparameter m' and evaluate the errors on the validation set (usual sum of per-example losses over this set). We choose the m' with the lowest validation error, then we typically retrain the model with that m' using the union of training and validation set as complete training set. Eventually, we get the optimal $\bar{\boldsymbol{\theta}}$ and we check its performance with the test error.

Instead, if we don't use the hyperparameter procedure and we have to manually fix or adjust the size m' we have to take into account multiple factors. For example, increasing m' obviously improves the estimate of the gradient but it slows down the algorithm. We have to compare the improvement in accuracy with the computational cost in order to select the correct m' (we are going to see later how much the estimate precision vary when we modify the size).
Another feature we have to consider was experimentally shown in the article "The general inefficiency of batch training for gradient descent learning" [44]: smaller minibatch sizes seem to lead to better generalization errors. The authors even state that the best generalization errors are often obtained through the limit case of $m' = 1$. The reason is probably that the less the size m' the more the noise inserted during training phase and this partially prevents overfitting (it's harder to only memorize training examples if there is noise). On the other hand they point out that little minibatches lead to high variance in the gradient approximation, then we have to decrease the learning rate to overcome this problem and to provide more stability and possibly convergence.

We should look also at limitations caused by the hardware architecture, for instance the ones associated to parallel computing. In fact, most of the times we run in parallel the operations for each example in the batch. For each sample we occupy a part of the memory for the computations, then the total amount of storage required is directly proportional to the size m'. However, we shouldn't go under a minimum minibatch size which depends on the hardware used: multi-core machines are, in fact, underexploited by too small minibatches. Therefore, all size m' are equivalent under this lower bound, since the machine simply doesn't use part of its cores in order to lower its computational capacity to the strictly necessary.
Moreover it's better to work with particular sizes associated to the capacity of parallel computations: for example when we use GPUs, we prefer to set the size as a power of 2 ($m' = 2^h$ for some $h \in \mathbb{N}$).

Remark 3.2. *Notice that the size m' depends on the information needed by the algorithm, too. Not only because the information could be tough to estimate with a small amount of example, but also because some algorithm are more responsive to errors in the approximations. Gradient-based methods (which is our case) work well even with small minibatch size. Second-order methods, instead, can present problems when the estimate error is amplified by the algorithm: for instance, if we use $\boldsymbol{H}^{-1}\boldsymbol{g}$ where $\boldsymbol{H}$ has a large condition number, then the approximation error in $\boldsymbol{g}$ is propagated and increased by $\boldsymbol{H}^{-1}$. That's why we often require larger minibatch m' size for second-order methods.*

SGD effectiveness

In this subsection we want to understand why the SGD works even if the algorithm basically oscillates around the actual cost function's gradient. The main reason is that at each iteration SGD follows (approximately) the gradient of true generalization error $J^*(\boldsymbol{\theta})$, defined at 2.1, if examples aren't reused in different minibatches. We have to keep in mind that the cost function or empirical risk is an estimator of the generalization error (which is intractable due to the unknown data generating distribution P_{data}). Therefore with SGD we are simply shifting the interest in exploiting at every step an unbiased estimator of the generalization error's gradient.

We show that this idea actually works proving the previous statements. In order to do so, we need an auxiliary result on the gradient of generalization error. In fact, we want to adjust it as expected value of the per-example loss's gradient. Essentially we have to prove that we can transfer the differentiation under the expected value. In the proof we are going to need the three following theorems from general Mathematical Analysis.

Theorem 3.1. *Let $(F_k)_{k\in\mathbb{N}}$ a sequence of differentiable functions, if there exists a finite pointwise limit for at least a point ($\exists \lim_{k\to\infty} F_k(\boldsymbol{\theta}^*) < \infty$ for at least one $\boldsymbol{\theta}^*$) and if all partial derivatives $\partial_{\theta_i} F_k$ uniformly converge on compacts, then F_k uniformly converge to a function F on compacts and $\partial_{\theta_i} F(\boldsymbol{\theta}) = \lim_{k\to\infty} \partial_{\theta_i} F_k(\boldsymbol{\theta})$ for every $\boldsymbol{\theta}$ in the domain.*

Theorem 3.2. *Suppose that the following assumptions on $F : \Theta \times T \to \mathbb{R}$ hold:*

1. *For every $\boldsymbol{\theta}$ we have $F(\boldsymbol{\theta}, \boldsymbol{t}) \in L^1(T)$, which means it's an integrable function with respect to $\boldsymbol{t}$.*

2. *For almost every $\boldsymbol{t} \in T$, all partial derivatives $\partial_{\theta_i} F(\boldsymbol{\theta}, \boldsymbol{t})$ exist for all $\boldsymbol{\theta} \in \Theta$.*

3. *There exist an integrable function $l : T \to \mathbb{R}$ such that for all $\boldsymbol{\theta}$ we have $|\partial_{\theta_i} F(\boldsymbol{\theta}, \boldsymbol{t})| \le l(\boldsymbol{t})$ for almost every $\boldsymbol{t} \in T$.*

Then we get

$$\partial_{\theta_i} \int_T F(\boldsymbol{\theta}, \boldsymbol{t}) d\boldsymbol{t} = \int_T \partial_{\theta_i} F(\boldsymbol{\theta}, \boldsymbol{t}) d\boldsymbol{t}. \tag{3.2}$$

With these three tools we can prove the actual required property:

Proposition 3.1. *If $\mathbf{z} = (\mathbf{x}, \mathbf{y})$ takes values on a compact space $X \times Y$ and $\partial_{\theta_i} L\big(f(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{y}\big)$ is bounded on compact as function of $\big((\boldsymbol{x}, \boldsymbol{y}), \boldsymbol{\theta}\big)$. Then we obtain the following equation:*

$$\nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) := \nabla_{\boldsymbol{\theta}} \Big(\mathbb{E}_{\mathbf{z} \sim P_{data}} \left[L\big(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y}\big) \right] \Big) = \mathbb{E}_{\mathbf{z} \sim P_{data}} \left[\nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y}\big) \right]. \tag{3.3}$$

Proof. We start with the case where both $\mathbf{x}$ and $\mathbf{y}$ are discrete, then also $\mathbf{z} = (\mathbf{x}, \mathbf{y})$ is a discrete random variable with value in $Z = X \times Y$. We redefine the cost function in terms of $\mathbf{z}$ in order to simplify the notation in the next steps: $\widetilde{L}(\mathbf{z}, \boldsymbol{\theta}) = L\big(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y}\big)$. Given the new notation we write explicitly the generalization error as a countable sum:

$$\begin{aligned} J^*(\boldsymbol{\theta}) &= \sum_{\boldsymbol{x} \in X} \sum_{\boldsymbol{y} \in Y} p_{data}(\boldsymbol{x}, \boldsymbol{y}) L\big(f(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{y}\big) \\ &= \sum_{\boldsymbol{z} \in Z} p_{data}(\boldsymbol{z}) \widetilde{L}(\boldsymbol{z}, \boldsymbol{\theta}). \end{aligned} \tag{3.4}$$

Now we index the elements of Z: meaning $Z = (\boldsymbol{z}_j)_{j \in \mathbb{N}}$. Then, we define the sequence of function $J_k(\boldsymbol{\theta}) = \sum_{j=1}^{k} p_{data}(\boldsymbol{z}_j) \widetilde{L}(\boldsymbol{z}_j, \boldsymbol{\theta})$. Since its a finite sum, its partial derivatives are well-defined and easy to compute: $\partial_{\theta_i} J_k(\boldsymbol{\theta}) = \sum_{j=1}^{k} p_{data}(\boldsymbol{z}_j) \partial_{\theta_i} \widetilde{L}(\boldsymbol{z}_j, \boldsymbol{\theta})$. Moreover we notice that the sequence has a pointwise limit for every $\boldsymbol{\theta}$ (it's even more than what the theorem 3.1 requires to be applied):

$$J_k(\boldsymbol{\theta}) \xrightarrow{k \to \infty} J^*(\boldsymbol{\theta}) = \sum_{\boldsymbol{z} \in Z} p_{data}(\boldsymbol{z}) \widetilde{L}(\boldsymbol{z}, \boldsymbol{\theta}) \tag{3.5}$$

This limit is finite since it's exactly the generalization error $J^*(\boldsymbol{\theta})$.
It remains to show that all partial derivatives uniformly converge on compacts. Let $K \subset \Theta$ be a compact subset, then $\big(\boldsymbol{z}, \boldsymbol{\theta}\big) \in Z \times K$ are in a compact space. Since $\sum_{\boldsymbol{z} \in Z} p_{data}(\boldsymbol{z}) = 1 < \infty$, for every $\varepsilon > 0$ exists $\bar{k} \in \mathbb{N}$ such that $\sum_{j=k+1}^{\infty} p_{data}(\boldsymbol{z}) \le \varepsilon$ for every $k \ge \bar{k}$. Then for every $\varepsilon > 0$ exists $\bar{k} \in \mathbb{N}$ such that

$$\begin{aligned} &\left| \partial_{\theta_i} f_k(\boldsymbol{\theta}) - \sum_{\boldsymbol{z} \in Z} p_{data}(\boldsymbol{z}) \partial_{\theta_i} \widetilde{L}(\boldsymbol{z}, \boldsymbol{\theta}) \right| = \\ &= \left| \sum_{j=k+1}^{\infty} p_{data}(\boldsymbol{z}) \partial_{\theta_i} \widetilde{L}(\boldsymbol{z}, \boldsymbol{\theta}) \right| \le \\ &\le C_K \left| \sum_{j=k+1}^{\infty} p_{data}(\boldsymbol{z}) \right| \le C_K \varepsilon \end{aligned} \tag{3.6}$$

for every $k \geq \bar{k}$.
Therefore, we have a sequence of differentiable function J_k with pointwise limit at least in one point and partial derivatives $\partial_{\theta_i} J_k$ uniformly convergence on compacts. We can apply the theorem 3.1 to obtain that $\partial_{\theta_i} J^*(\boldsymbol{\theta}) = \lim_{k\to\infty} \partial_{\theta_i} J_k(\boldsymbol{\theta})$. In particular we can write:

$$\begin{aligned}\partial_{\theta_i} J^*(\boldsymbol{\theta}) &= \partial_{\theta_i}\left(\sum_{\boldsymbol{z}\in Z} p_{data}(\boldsymbol{z})\widetilde{L}(\boldsymbol{z},\boldsymbol{\theta})\right) = \\ &= \lim_{k\to\infty}\sum_{j=1}^{k} p_{data}(\boldsymbol{z}_j)\partial_{\theta_i}\widetilde{L}(\boldsymbol{z}_j,\boldsymbol{\theta}) = \\ &= \sum_{\boldsymbol{z}\in Z} p_{data}(\boldsymbol{z})\partial_{\theta_i}\widetilde{L}(\boldsymbol{z},\boldsymbol{\theta})\end{aligned} \tag{3.7}$$

In the divided starting notation:

$$\begin{aligned}\partial_{\theta_i} J^*(\boldsymbol{\theta}) &= \partial_{\theta_i}\left(\sum_{\boldsymbol{x}\in X}\sum_{\boldsymbol{y}\in Y} p_{data}(\boldsymbol{x},\boldsymbol{y})L\big(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}\big)\right) = \\ &= \sum_{\boldsymbol{y}\in Y}\sum_{\boldsymbol{y}\in Y} p_{data}(\boldsymbol{x},\boldsymbol{y})\partial_{\theta_i}L\big(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}\big) = \mathbb{E}_{\mathbf{z}\sim P_{data}}\left[\partial_{\theta_i}L\big(f(\mathrm{x};\boldsymbol{\theta}),\mathbf{y}\big)\right].\end{aligned} \tag{3.8}$$

The continuous case directly comes from the application of theorem 3.2: we set $F(\boldsymbol{\theta},\boldsymbol{t}) = p_{data}(\boldsymbol{x},\boldsymbol{y})L\big(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}\big)$ where $\boldsymbol{t} = (\boldsymbol{x},\boldsymbol{y})$. F is differentiable and it has a dominator $l(\boldsymbol{t}) = l(\boldsymbol{x},\boldsymbol{y})$ by hypothesis. Then

$$\begin{aligned}\partial_{\theta_i} J^*(\boldsymbol{\theta}) &= \partial_{\theta_i}\left(\int_{X\times Y} p_{data}(\boldsymbol{x},\boldsymbol{y})L\big(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}\big)d\boldsymbol{x}d\boldsymbol{y}\right) = \\ &= \partial_{\theta_i}\left(\int_T F(\boldsymbol{\theta},\boldsymbol{t})d\boldsymbol{t}\right) = \int_T \partial_{\theta_i}F(\boldsymbol{\theta},\boldsymbol{t})d\boldsymbol{t} = \\ &= \int_{X\times Y} p_{data}(\boldsymbol{x},\boldsymbol{y})\partial_{\theta_i}L\big(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}\big)d\boldsymbol{x}d\boldsymbol{y} = \mathbb{E}_{\mathbf{z}\sim P_{data}}\left[\partial_{\theta_i}L\big(f(\mathrm{x};\boldsymbol{\theta}),\mathbf{y}\big)\right]\end{aligned} \tag{3.9}$$

concluding the proof of both cases. □

Remark 3.3. *The exchange of expectation and differentiation can be achieved also with different hypothesis on the loss function L and the probability mass (or density) function p_{data}. However, for the sake of simplicity, we try to keep some of the most general and reasonable assumptions. Moreover they are often encounter in practical model, for instance, keep in mind that computing on machine we can't never deal with $X \times Y$ not compact.*

Now that we have the needed auxiliary result (the previous proposition) we can demonstrate our claim on minibatch gradient being an unbiased estimator of general error's gradient.

Proposition 3.2. *Let m' be the fixed size. The average of the per-example loss' gradients $\frac{1}{m'}\sum_{i\in B}\nabla_{\boldsymbol{\theta}}L\big(f(\mathbf{x}^{(i)};\boldsymbol{\theta}),\mathbf{y}^{(i)}\big)$ over a randomly taken minibatch B of cardinality m' is an unbiased estimator of the generalization error's gradient.*

Proof. We denote with U the random variable which uniformly takes as values the subsets of $I_m = \{\, 1, \ldots, m \,\}$ with cardinality m'. In the next formulas we are going to name its distribution $Unif$. Moreover we consider the training data as samples from random variables $\mathbf{z}^{(i)} = \big(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\big)$ for $i \in I_m$, all with distribution P_{data}.

$$\begin{aligned}
&\mathbb{E}_{\mathrm{U}\sim Unif,\, \mathbf{z}^{(i)}\sim P_{data}\forall i}\left[\frac{1}{m'}\sum_{i\in \mathrm{U}} \nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x}^{(i)};\boldsymbol{\theta}), \mathbf{y}^{(i)}\,\big)\right] = \\
&= \sum_{B\subseteq I_m, |B|=m'} \left(P(\mathrm{U}=B)\, \mathbb{E}_{\mathbf{z}^{(i)}\sim P_{data}\forall i}\left[\frac{1}{m'}\sum_{i\in B} \nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x}^{(i)};\boldsymbol{\theta}), \mathbf{y}^{(i)}\,\big)\right]\right) = \\
&= \sum_{B\subseteq I_m, |B|=m'} \left(P(\mathrm{U}=B)\, \frac{1}{m'}\sum_{i\in B} \mathbb{E}_{\mathbf{z}^{(i)}\sim P_{data}\forall i}\left[\nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x}^{(i)};\boldsymbol{\theta}), \mathbf{y}^{(i)}\,\big)\right]\right) = \\
&= \sum_{B\subseteq I_m, |B|=m'} \left(P(\mathrm{U}=B)\, \frac{1}{\cancel{m'}}\cancel{m'}\, \mathbb{E}_{\mathbf{z}\sim P_{data}}\left[\nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x};\boldsymbol{\theta}), \mathbf{y}\,\big)\right]\right) = \\
&= \sum_{B\subseteq I_m, |B|=m'} \left(\binom{m}{m'}^{-1} \mathbb{E}_{\mathbf{z}\sim P_{data}}\left[\nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x};\boldsymbol{\theta}), \mathbf{y}\,\big)\right]\right) = \\
&= \cancel{\binom{m}{m'}}\cancel{\binom{m}{m'}^{-1}}\, \mathbb{E}_{\mathbf{z}\sim P_{data}}\left[\nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x};\boldsymbol{\theta}), \mathbf{y}\,\big)\right] = \\
&= \mathbb{E}_{\mathbf{z}\sim P_{data}}\left[\nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x};\boldsymbol{\theta}), \mathbf{y}\,\big)\right] = \nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}).
\end{aligned} \tag{3.10}$$

The fourth equality is obtained by the following formula

$$P(\mathrm{U}=B) = \frac{1}{\#\text{ subsets of cardinality } m' \text{ in } I_m} = \frac{1}{\binom{m}{m'}},$$

since U is uniform on the set of subsets with cardinality m'. The last equality, instead, is simply the thesis of the auxiliary result: the previous proposition 3.1. □

The problem is that we are trying to implement an iterative algorithm and, actually, what we have just proven is that at the first iteration we have an unbiased estimator of the true gradient of the generalization error. However in the subsequent iterations we have to take into account that we already know the values of the random variables in the previous minibatches. In terms of probability theory, the expected value of the estimators of subsequent steps has to be computed with the conditional probability given the occurrences of previous minibatches.
For example, if at the first step we took the first m' elements of the training set: $B_1 = \{\, 1, \ldots, m' \,\}$. Then at the second iteration we already know the outputs of the random variables representing the first m' training examples: $\mathbf{z}^{(1)} = \big(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}\big) = \big(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\big) = \boldsymbol{z}^{(1)}, \ldots, \mathbf{z}^{(m')} = \big(\mathbf{x}^{(m')}, \mathbf{y}^{(m')}\big) = \big(\boldsymbol{x}^{(m')}, \boldsymbol{y}^{(m')}\big) = \boldsymbol{z}^{(m')}$. Therefore the expected value of the estimator at the second step is computed in the following way:

$$\mathbb{E}_{\mathrm{U}\sim Unif,\, \mathbf{z}^{(i)}\sim P_{data}\forall i}\left[\frac{1}{m'}\sum_{i\in \mathrm{U}} \nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x}^{(i)};\boldsymbol{\theta}), \mathbf{y}^{(i)}\,\big)\,\middle|\, \mathbf{z}^{(1)} = \boldsymbol{z}^{(1)}, \ldots, \mathbf{z}^{(m')} = \boldsymbol{z}^{(m')}\right].$$

This expectation could be different from the true gradient. In fact, computing with the conditional probability given the event $A = \{\, \mathbf{z}^{(1)} = \boldsymbol{z}^{(1)}, \ldots, \mathbf{z}^{(m')} = \boldsymbol{z}^{(m')} \,\}$, not all random variables representing the training examples still have the distribution P_{data}. In particular, we are talking about the first m' elements:

$$P(\mathbf{z}^{(i)} = \boldsymbol{z} \mid A) \neq P_{data}(\boldsymbol{z}) \text{ for } i = 1, \ldots, m'.$$

This affects the expected value since $U \cap \{ 1, \ldots, m' \}$ could be not empty. So it's the case when we reselect examples of the previous minibatches, the problem can be seen in the fact that we are trying to estimate a quantity resampling observed values instead of drawing new ones from the data generating distribution.

To overcome this problem we need that the average of the per-example loss' gradients $\frac{1}{m'} \sum_{i \in U} \nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)}\big)$ keeps its randomness even when the previous minibatches are given. We already know that the random variables representing different training examples are independent (presented at the beginning of the "Learning" section in chapter 2), so samples in the same minibatch are independent. Nevertheless we need an additional feature: we want the minibatches to be independent of each other in order to get subsequent independent gradient estimates. The easiest theoretical method to obtain this property is to exclude training examples once they are used. In other words, the uniform distribution choosing the indexes changes at each step: at the k-th step, U_k randomly picks a subset of cardinality m' contained in $I_m \setminus \left(\bigcup_{h=1}^{k-1} B_k \right)$ where B_k are the previously selected minibatches. This discussion explain why, at the beginning of the section, we say that SGD algorithm approximately follows the true gradient of generalization error only if the example aren't reused in different minibatches.

Remark 3.4. *The previous concepts show that Gradient Descent on the cost function J doesn't follow the gradient of generalization error in the previous sense: at each step we are essentially reusing the same minibatch of size $m' = m$ (all training set), then we have an unbiased estimator of the true gradient of J^* from the second iteration onwards.*

We call this method *theoretical* since it's different from what it's done in the actual Machine Learning models. When we have very large training sets, it could be difficult to draw a minibatch perfectly uniformly at each step. Implementations of randomness are also limited by the deterministic operations executable by machines. These last ones, in fact, can perform only pseudorandom algorithms. We can't even use the datasets as they are, since most are sorted so that subsequent examples are extremely correlated. To solve these problems in practical models we shuffle the entire dataset once and then we take minibatches consisting of elements which are consecutive in the new order: $B_k = \{ (k-1)m' + 1, \ldots, km' \}$. It doesn't lead to actual uniform randomness but it's an efficient method to apply SGD on more models since the shuffled dataset can be exploited by different models. Furthermore, empirically, it doesn't seem to bring relevant negative aspects.

In real life models we also have to take into account the training set size m and the performance we want to obtain. That's one of the reasons why sometimes we deviate from the theory and abandon the property of unbiased estimator. In some models where the training set isn't too large, in fact, it's preferred to pass through the shuffled dataset multiple times, even if we get a biased estimator of the true gradient from the second iteration onward. Using an unbiased estimator leads to a greater gap between training and test error since we are not following the gradient of the generalization error, therefore we aren't decreasing the generalization error in the

best way: it still decreases but it may do it faster using an unbiased estimator. Instead, going through the dataset multiple times results in a reduction of the training error and, when this reduction is large enough, it overcomes the increased gap leading to a lower test error (so better performance overall).
Keep in mind that, conversely, when the training set is very large (a lot of modern cases) it can happen that we don't even use all training examples and we perform an incomplete pass through the shuffled dataset.

In conclusion, we have just proven that, when we don't reuse examples, the SGD follows an unbiased estimator of generalization error's true gradient. In other words, SGD on the cost function J is equivalent to a stochastic approximation of the Gradient Descent on the generalization error J^*. This gives us the idea on why the SGD on J works and effectively decreases the test error.

Remark 3.5. *An easy way to summarize some of the intuitions in this subsection is to analyse online learning. The latter is a method where we don't have a fixed training set, instead the examples are taken from a stream of constantly generated examples. It's named online since it's the one used when we train a model on examples obtained by users of a web service. For example, recommendation algorithms of Amazon or Netflix get a user's feedback and use it as a training example.*
Since examples are usually taken one by one from the stream, we use minibatches of size $m' = 1$, then at each iteration we use the gradient computed on one example. The idea is that the model receives a new example drawn from the actual data generating distribution P_{data} at each iteration. Therefore we can say that every example is a true sample from P_{data}, so they are never repeated. In this situation it seems reasonable that we are minimizing the generalization error since we don't get through an empirical distribution $\widehat{P}_{train}$ as approximation but, instead, we directly estimate the true gradient using only examples drawn by true data generating distribution P_{data}.

Additional benefits

We can notice other benefits in addition to the reduction of computational cost at each iteration. In particular we are going to show four of them: the lower redundancy, the non-linear returns when using more samples, the faster convergence in the sense of total time spent and the $O(1)$ behaviour of the updates' number.

Redundancy lowering This is an obvious consequence of taking smaller sets to compute the gradient. In the training set we can encounter examples which give the same or very similar contribution to the gradient approximation. Since the estimate is an average, this type of examples doesn't improve the approximation. In fact, we get the same or very similar result deleting them from the training set. Instead, the SGD approach decreases the probability of taking redundant examples when we are performing a step.
This scenario is easy to see with an extreme worst case (it's very unlikely that it happens): if in the training set we have the exact same example repeated m times, then with SGD and size $m' = 1$ we obtain the same gradient but executing m times less operations.

Non-linear returns with more samples More than a benefit, this is a drawback of the standard Gradient Descent. As we've previously said, the gradient of the generalization error is

an expectation $\nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim P_{data}}\left[\nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x};\boldsymbol{\theta}),\mathbf{y}\big)\right] = \boldsymbol{\mu}$. Both cost gradient and gradient over the minibatch are the sample mean estimator of that expectation, they are only using a different number of observation to compute the estimate. Therefore we can quantify the standard error of these two estimator with the formula A.12 in the Appendix: $\mathrm{SE}\big(\widehat{\boldsymbol{\mu}}_{cost}\big) = \frac{\boldsymbol{\Sigma}^{\frac{1}{2}}}{\sqrt{m}}$ and $\mathrm{SE}\big(\widehat{\boldsymbol{\mu}}_{minibatch}\big) = \frac{\boldsymbol{\Sigma}^{\frac{1}{2}}}{\sqrt{m'}}$. This proves that using all training examples decreases the oscillations in the estimate by a less than linear factor.
For example, if we have $m = 100^2$ training examples and minibatches of size $m' = 100$, then at each iteration of the Gradient Descent we perform 100 times more operations to compute the total cost gradient, but we obtain an improvement in the standard error only of a factor $\sqrt{100} = 10$.

Faster convergence in time It has been shown empirically that, in terms of total time cost, we have faster convergence when we follow a quickly computed approximation of the gradient, even if are performed more steps than the Gradient Descent ones. This means that, usually, many but rapid steps are better than less but slower steps. The computational cost saved by the minibatch estimation is too significant compared to the additional steps taken. Or rather from another prospective, using all training example to compute the cost gradient is too slow with the large amount of data usually available.

$\boldsymbol{O(1)}$ updates' number We saw some properties related to the computational cost of each step and of the total algorithm. However all the considerations were stated with a fixed amount of training example m. Now we want to present some observation on the asymptotic behaviour of the number of iterations. When the training set's cardinality is increased, it's good practice to raise the model size and capacity (for instance in Neural Networks, more nodes or layers are added) but it's not necessary. In our discussion, in fact, we are going to keep the model size fixed. Not only in the standard Gradient Descent but also in many other optimization algorithms, the number of updates to get convergence increases with m. In the SGD case, instead, for very large m we obtain the best possible test error without even exploiting all the elements of the training set. Therefore, even if we increase m even further, it seems that the computational cost to get the best possible test error doesn't grow. In conclusion it can be believed that the convergence time of the SGD doesn't depend on m and asymptotically it has the same behaviour of a constant: the number of steps is a $O(1)$ function of m. Nevertheless this an asymptotic behaviour and we need a too large m to actually notice it. That's why most of the time the Gradient Descent perform less updates to converge compared to SGD.

Remark 3.6. *This represents a huge improvement in the optimization of non-linear models. Before Neural Networks, non-linear models were trained combined the kernel trick with linear models. However, the kernel trick often needs to calculate a matrix indexed with all training examples: $K_{ij} = k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})$ with $i, j = 1, \ldots, m$. Therefore this operation takes a computational cost of time complexity $O(m^2)$, which is a significant problem for big training set.*

Learning rate size

In this subsection we want to discuss if a constant learning rate $\varepsilon > 0$ can cause problems as in the Gradient Descent algorithm. Suppose we are at a critical point of the cost function

J. Then its gradient is the zero vector, but the estimator of the true gradient used in the SGD algorithm could be different from the zero vector:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)}\big) = \boldsymbol{0} \quad \text{but} \quad \widehat{\boldsymbol{g}} = \frac{1}{m'}\sum_{h=1}^{m'} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i_h)};\boldsymbol{\theta}), \boldsymbol{y}^{(i_h)}\big) \neq \boldsymbol{0}$$

Therefore, even if we are at a critical point, the SGD still moves from what should be an ending point of the algorithm. The same effect happens near that critical point: the gradient used to update the parameters vector could be large even if we are near a critical point. The reason is that the SGD algorithm introduce noisy estimation to provide faster steps. However this noise due to the random selections of the minibatches exists at every steps, then also in the ones at a critical point and near it.

In order to tackle this problem and achieve the right convergence, it's required to reduce the learning rate as the iterations pass. In particular we prefer to choose a sequence of learning rates $(\varepsilon_k)_{k\in\mathbb{N}}$ tending to zero: $\lim\limits_{k\to\infty} \varepsilon_k = 0$. Some theoretical sufficient conditions to ensure convergence are:

$$\sum_{k=1}^{\infty} \varepsilon_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty.$$

Nevertheless, we don't demand these conditions in practice, instead, we achieve convergence combining a reasonable non-increasing sequence with the stopping criteria. A possible simple example is a sequence of polynomial rate: $\varepsilon_k = \frac{1}{k^a}$ where $a \in \mathbb{R}_+$ is a positive constant. Another one is a sequence of exponential rate: $\varepsilon_k = \frac{1}{\varepsilon_0^k}$ where $\varepsilon_0 \in \mathbb{R}_+$ is a positive initial constant used to derive all subsequent rates. This idea of computing the later rates using a starting quantity appears also in one of the most common type of sequence: the learning rate sequence with linear decay. The sequence is defined by three hyperparameters: a starting rate $\varepsilon_0 > 0$, a fixed iteration τ and the last rate $\varepsilon_\tau > 0$. The learning rates start from ε_0 and linearly decrease at each iteration until they get to ε_τ at the τ-th iteration. Then we usually keep the rate small and constant in the following iterations. The formula to obtain each rate is:

$$\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha\varepsilon_\tau$$

for $k = 1, \ldots, \tau$ and with $\alpha = \frac{k}{\tau}$.

In general, finding the correct sequence of learning rates $(\varepsilon_k)_{k\in\mathbb{N}}$ is a difficult assignment. In fact, we have to use empirical methods as trial and error or more structured strategy, since it's very difficult to obtain theoretical results that can help us. A general approach is to plot the cost function progression through the iterations to see how much it oscillates and how fast it decreases. There are some heuristic rules of thumb that have been tested for many years, but there isn't a sure path to follow to get the right sequence.
In the linear decay case, for example, we have to set ε_0, τ and ε_τ. Generally we take τ so that the algorithm go through the training set multiple times (in the order of hundreds). The "ending" rate ε_τ is often defined in terms of ε_0: the most used proportion is around one percent of ε_0. It remains to choose the starting rate, which is the most difficult to select. It causes problems similar to Gradient Descent ones: a large rate can make J vary a lot and even increasing the function at some initial steps, while a small rate can make the algorithm converge with a very slow rate. Moreover, if the initial rate ε_0 is too small, since it reduces in the subsequent iteration,

the algorithm could be so slow that J doesn't improve for many iterations and the algorithm stops with an high cost. The usual procedure is to check the behaviour of the cost function in the first 100 iterations for different initial rates. Then we set ε_0 slightly larger than the rate which returns the best test error after those 100 iterations.

We have to keep in mind that when we talk about convergence we are considering the behaviour in the long term. However we don't have to improve the convergence rate to the excessive detriment of the first iterations. In fact, it's at these steps that the SGD algorithm gives its best result in terms of speeding up the process and improving J fast.

Complete algorithm

In conclusion we show the complete algorithm and we summarize the main properties of the Stochastic Gradient Descent algorithm. The SGD helps us to reduce the computational cost of every iteration and, actually, often even the total time cost. In fact, it moves for more but faster steps, especially in the first ones where the algorithm demonstrate its power decreasing J really quick. However, depending on the size m' it can be unstable and the oscillation can make the algorithm very slow in the final convergence. Given the instability, the learning rate can't be constant and it's preferable tending to zero. Moreover, the actual sequence of learning rate $(\varepsilon_k)_{k\in\mathbb{N}}$ can be difficult to set.

Algorithm 2 Stochastic Gradient Descent (SGD)

```
Require: Learning rate sequence $(\varepsilon_k)_{k\in\mathbb{N}}$
Require: The size $m'$ of the minibatches
Require: Randomly initialized parameters vector $\boldsymbol{\theta}_0$
  $k \leftarrow 1$
  while All stopping criterion aren't achieved do
    Randomly select a minibatch $\{\,(\boldsymbol{x}^{(i_1)},\boldsymbol{y}^{(i_1)}),\,\ldots,\,(\boldsymbol{x}^{(i_{m'})},\boldsymbol{y}^{(i_{m'})})\,\}$ from the training set
    Gradient estimator: $\widehat{\boldsymbol{g}}_k \leftarrow \frac{1}{m'}\sum_{h=1}^{m'} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i_h)};\boldsymbol{\theta}_{k-1}),\boldsymbol{y}^{(i_h)}\big)$
    Update: $\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \varepsilon_k\widehat{\boldsymbol{g}}_k$
    $k \leftarrow k+1$
  end while
  return $\boldsymbol{\theta}_{k-1}$
```

3.2 Momentum

As we previously said, the SGD can be slow since it can have a large number of updates. This is due to the fact that we use noisy gradients which oscillate around the true gradient. The problem can be limited by the momentum strategy presented for the first time in the Polyak's paper "Some methods of speeding up the convergence of iteration methods" [34]. Moreover, we are going to see that this strategy is implemented to solve also the ill-conditioning problem.

The basic idea is to take advantage of the previous directions of steepest descent giving a weight which decay exponentially as the direction becomes "old" moving forward in the iterations. In order to exploit them, we compute an exponentially decaying moving (at each step we add a

term) average of previous negative gradients. Then we use the average in the parameters updates to keep going in a similar direction. Keep in mind that, even if it's called "*average*", in our case the weights don't add up to 1 since the sum isn't normalized at each step.
We are going to see in the algorithm subsection that this exponentially decaying moving average denoted with $\boldsymbol{v}$ is calculated adding the previous average multiplied by an hyperparameter $\alpha > 0$ to the usual SGD step $-\varepsilon\widehat{\boldsymbol{g}}$: the update is $\boldsymbol{v}_{new} = \alpha\boldsymbol{v}_{previous} - \varepsilon\widehat{\boldsymbol{g}}$.

Physical interpretation

The word "*momentum*" clearly comes from a physical interpretation. We think $\boldsymbol{\theta}$ as the position of a particle in the parameters space Θ and it is a function of the iterations (representing time) if this particle is moving. Then, the introduced average $\boldsymbol{v}$ is considered the velocity of this moving particle. $\boldsymbol{v}$ is a vector, then, it contains information on both direction and absolute value of the velocity. In Physics, the momentum (or translational momentum) is the product of the mass and the velocity. In the momentum strategy we assume that the particle has unit mass, therefore the momentum is precisely equal to the velocity $\boldsymbol{v}$.

We now proceed to explain the general and complete physical framework: we are going to basically describe a continuous-time Newtonian dynamics. We suppose that the particle at $\boldsymbol{\theta}(t)$ moves in the parameters space Θ subject to a conservative force $\boldsymbol{F}(\boldsymbol{\theta})$. The potential energy of this force is proportional to the cost function $U(\boldsymbol{\theta}) = \varepsilon J(\boldsymbol{\theta})$, then the actual force is the negative gradient multiplied by the learning rate $\boldsymbol{F}(\boldsymbol{\theta}) = -\varepsilon\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. Notice that in this type of scenario the particle naturally take a path which passes through the minimal potential energy and the acceleration follows a force equal to the negative gradient. In other words, the solution of this physical system given the initial state is a good starting point to develop the momentum strategy. However, if we have only $\boldsymbol{F}(\boldsymbol{\theta})$ that drives the particle down on the cost function surface, we are in the ideal scenario with no friction and therefore the particle may keep moving in a never-ending motion. To solve this problem we can add a friction term to the force, essentially supposing that the particle moves in a viscous medium. The additional viscous drag is proportional to the negative velocity with a positive factor: in particular we write it as $-(1-\alpha)\boldsymbol{v}(t)$ where $\alpha \in [0, 1)$, so $(1 - \alpha) > 0$.

Using the Newton's laws of motion we can write down the formula defining the system: denoting the acceleration with $\boldsymbol{a}(t)$ and knowing that it's the second derivative with respect to time t of the position $\boldsymbol{\theta}(t)$, we get

$$m\boldsymbol{a}(t) = \frac{\partial^2}{\partial t^2}\boldsymbol{\theta}(t) = \boldsymbol{F}\big(\boldsymbol{\theta}(t)\big) = -\varepsilon\nabla_{\boldsymbol{\theta}} J\big(\boldsymbol{\theta}(t)\big) - (1-\alpha)\frac{\partial}{\partial t}\boldsymbol{\theta}(t) \tag{3.11}$$

since the mass is unitary, $m = 1$. It's a second order differential equation which can be rewritten as a first order differential system if we introduce the velocity $\boldsymbol{v}(t)$:

$$\begin{cases} \frac{\partial}{\partial t}\boldsymbol{v}(t) = \boldsymbol{F}(t) = -\varepsilon\nabla_{\boldsymbol{\theta}} J\big(\boldsymbol{\theta}(t)\big) - (1-\alpha)\boldsymbol{v}(t) \\ \frac{\partial}{\partial t}\boldsymbol{\theta}(t) = \boldsymbol{v}(t) \end{cases} . \tag{3.12}$$

We can solve this system using numerical methods, in particular, we implement the Euler's methods (backward and forward). In these methods we discretize the time in steps t_k where $t_k = t_{k-1} + h = t_0 + kh$ with $h > 0$ fixed time step. We denote our approximations of the particle's position at those time steps with $\boldsymbol{\theta}_k \approx \boldsymbol{\theta}(t_k)$. Then we approximate the derivatives with the difference quotients on those steps: in the general univariate case where $y(t)$ is the unknown function, we get $y_{k+1} - y_k = hy'_k$ or $y_{k+1} - y_k = hy'_{k+1}$. Since the algorithm we want to obtain works with iterations as time, we can take $t_k \in \mathbb{N}$ starting from $t_0 = 0$ and taking $h = 1$. With these considerations we can write the numerical system:

$$\begin{cases} \boldsymbol{v}_{k+1} = \boldsymbol{v}_k + \boldsymbol{F}_k \\ \boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \boldsymbol{v}_{k+1} \end{cases} . \tag{3.13}$$

Substituting the approximation of the total force we get

$$\begin{cases} \boldsymbol{v}_{k+1} = \boldsymbol{v}_k + -\varepsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k) - (1-\alpha)\boldsymbol{v}_k = \alpha \boldsymbol{v}_k - \varepsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k) \\ \boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \boldsymbol{v}_{k+1} \end{cases} . \tag{3.14}$$

Therefore we have just obtained the update rule in the momentum algorithms: we compute the gradient g at the current position of the parameters, we set the new velocity as the sum of the weighted previous velocity and the gradient multiplied by the learning rate, then we update the parameters vectors following the current velocity

$$\begin{aligned} &\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\ &\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \boldsymbol{g} \\ &\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v} \end{aligned} \tag{3.15}$$

The previous is update is in the Gradient Descent with momentum (since we compute the total gradient), to get the SGD algorithm with momentum we simply substitute $\boldsymbol{g}$ with its stochastic approximation $\widehat{\boldsymbol{g}}$.

Algorithm features

In this subsection we show the characteristic of the momentum strategy: we start analysing the effect of α variations, then we make some considerations on the maximum speed achieved, we talk about the ill-conditioning problem recalling the previous example and at the end we describe the entire algorithm.

The rate of exponential decay is defined by the hyperparameter $\alpha \in [0, 1)$. This means that it affects how fast the previous negative gradient directions become less significant in the current iteration. Its effect is quantified comparing α with the learning rate ε: the larger α compared to ε, the more previous negative gradients control the parameters update. As all other hyperparameters we can find the optimal one with the validation set procedure.
We notice that in this new version the step size doesn't depend only on the gradient norm and ε, but also on previous gradients norms and how much they are aligned: the more they are aligned the larger the step. Moreover, in the extreme case where all gradient are equal to $\boldsymbol{g}^*$, there exists a limit step size. In fact, we have $\boldsymbol{v}_1 = \boldsymbol{v}_0 - \varepsilon \boldsymbol{g}^*$, $\boldsymbol{v}_2 = \alpha\big(\boldsymbol{v}_0 - \varepsilon \boldsymbol{g}^*\big) - \varepsilon \boldsymbol{g}^*$, $\boldsymbol{v}_3 = \alpha^2\big(\boldsymbol{v}_0 - \varepsilon \boldsymbol{g}^*\big) + \alpha\big(-\varepsilon \boldsymbol{g}^*\big) + \alpha^0\big(-\varepsilon \boldsymbol{g}^*\big)$, etc. Then proceeding by induction we get the relation

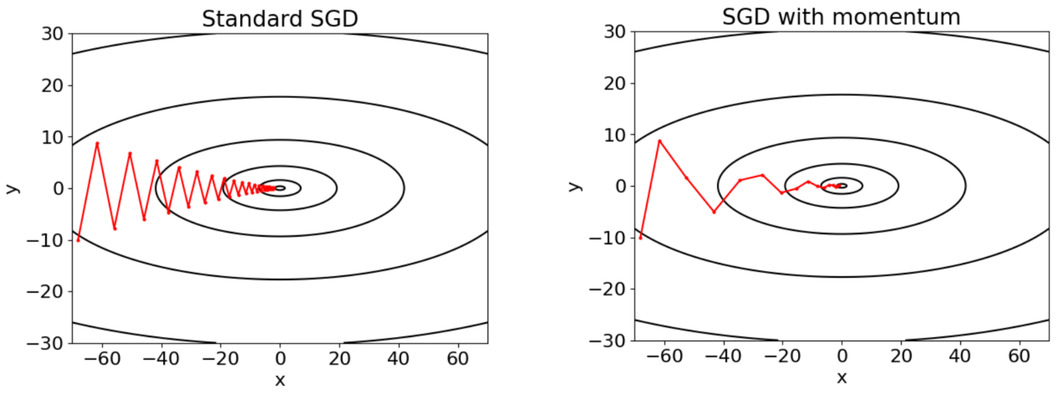
$$\boldsymbol{v}_k = \alpha^{k-1} \boldsymbol{v}_0 - \sum_{i=0}^{k-1} \alpha^i \varepsilon \boldsymbol{g}^* .$$

This is given by the fact that we always have $\boldsymbol{v}_k = \alpha \boldsymbol{v}_{k-1} - \varepsilon \boldsymbol{g}^*$. Then the limit speed v_∞ is obtained by

$$\begin{aligned}\boldsymbol{v}_\infty &= \lim_{k\to\infty} \boldsymbol{v}_k = \\ &= \lim_{k\to\infty} \left(\alpha^{k-1}\boldsymbol{v}_0 - \varepsilon \sum_{i=0}^{k-1} \alpha^i \boldsymbol{g}^*\right) = \\ &= \varepsilon \boldsymbol{g}^* \sum_{i=0}^{\infty} \alpha^i = \frac{\varepsilon}{1-\alpha}\boldsymbol{g}^*.\end{aligned}$$

With the limit speed we automatically obtain the limit step size since it is the norm of the step: $\frac{\varepsilon\|\boldsymbol{g}^*\|}{1-\alpha}$. This helps us understand another intuition on the effect of the momentum hyperparameter: choosing a specific α means to increase the maximum speed by a factor $\frac{1}{1-\alpha}$ with respect to the standard Gradient Descent algorithm.

In this paragraph we recall the example shown in the "Ill-conditioning" subsection to see how much faster the algorithm convergences if we add the momentum in the parameter updates. As we can see in the figure, even if we start with a null velocity $\boldsymbol{v}_0 = \boldsymbol{0}$, we get a significant improvement: in the figure on the right we achieve better result being closer to the minimum $(0, 0)$ with only 15 iterations, compared to the 35 of the figure on the left. This is due to the fact that adding the previous gradients together (even if they are weighted) at least partially cancels opposite components of gradients in subsequent iterations. In particular the projections on the direction with more curvature cancel and it mainly remains the components along the direction with less curvature. This limits the back and forth movement overtaking the minimum and following the direction with more curvature. In our case the SGD with momentum correctly moves on a path mostly in the direction $(1, 0)$ removing the oscillations in direction $(0, 1)$.



Now we show how we can modify an algorithm to include the momentum method. In particular we show the changes with respect to the SGD algorithm: the implementation is very simple since it sufficient to add one line in the pseudocode of the algorithm and it's the same also for other algorithms such as the Gradient Descent.

Algorithm 3 Stochastic Gradient Descent (SGD) with momentum

```
Require: Learning rate sequence $(\varepsilon_k)_{k\in\mathbb{N}}$
Require: The size $m'$ of the minibatches
Require: Momentum parameter $\alpha$ and initial velocity $\boldsymbol{v}_0$
Require: Randomly initialized parameters vector $\boldsymbol{\theta}_0$
  $k \leftarrow 1$
  while All stopping criterion aren't achieved do
      Randomly select a minibatch $B$
      Gradient estimator: $\widehat{\boldsymbol{g}}_k \leftarrow \frac{1}{m'} \sum_{h=1}^{m'} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i_h)}; \boldsymbol{\theta}_{k-1}), \boldsymbol{y}^{(i_h)}\big)$
      Velocity: $\boldsymbol{v}_k \leftarrow \alpha \boldsymbol{v}_{k-1} - \varepsilon_k \widehat{\boldsymbol{g}}_k$          ▷ New step, compared to SGD 2
      Update: $\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} + \boldsymbol{v}_k$
      $k \leftarrow k+1$
  end while
  return $\boldsymbol{\theta}_{k-1}$
```

Nestorov variation

In 1963 the classic momentum strategy was introduced in the general context of iteration methods and only later it was implemented for Neural Networks. The same happens for this variation which was first shown by Yurii Nesterov in "A method for solving the convex programming problem with convergence rate $O\big(1/k^2\big)$" [29] in 1983 with the purpose of speeding up gradient-based methods. Before the resurge of Neural Networks in its "third wave" (presented in "History of Neural Networks" subsection in the first chapter) it was believed that deep networks couldn't be trained efficiently with Gradient Descent or SGD. That's why we see an application of Nesterov momentum in Neural Networks only in 2013 when the article "On the importance of initialization and momentum in deep learning" [40] was published. This article and other researches of the same period show that deep and recurrent Neural Networks can be trained fast with the correct initialization of the parameters and a well-designed schedule of the momentum parameter. In his article Nesterov proves that this method improves the rate of convergence when we use a batch method with a convex function. In particular, the rate of convergence of the Gradient Descent in convex optimizatiom is $O\big(\frac{1}{k}\big)$ and it's improved to $O\big(\frac{1}{k^2}\big)$ with the Nesterov momentum.

The learning rate ε and the momentum parameter α have effects similar to the ones in the standard momentum algorithm. However, in the Nesterov variation we first move in the parameter space with the current velocity, then we compute the gradient at the new point to update the parameters again with the usual gradient step. We show the actual update rule as we did for classical momentum at 3.15:

$$\begin{aligned} \boldsymbol{g} &\leftarrow \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \alpha \boldsymbol{v}) \\ \boldsymbol{v} &\leftarrow \alpha \boldsymbol{v} - \varepsilon \boldsymbol{g} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \varepsilon \boldsymbol{g} \end{aligned} \tag{3.16}$$

which can be easily modified to get the SGD version.

We conclude the subsection with the complete algorithm of the SGD variation. We comment the new lines and explain the modifications:

Algorithm 4 Stochastic Gradient Descent (SGD) with Nesterov momentum

```
Require: Learning rate sequence $(\varepsilon_k)_{k\in\mathbb{N}}$
Require: The size $m'$ of the minibatches
Require: Momentum parameter $\alpha$ and initial velocity $\boldsymbol{v}_0$
Require: Randomly initialized parameters vector $\boldsymbol{\theta}_0$
  $k \leftarrow 1$
  while All stopping criterion aren't achieved do
      Randomly select a minibatch $B$
      Intermediate update: $\widetilde{\boldsymbol{\theta}}_k \leftarrow \boldsymbol{\theta}_{k-1} + \alpha \boldsymbol{v}_{k-1}$    ▷ New step, compared to algorithm 3
      Gradient estimator: $\widehat{\boldsymbol{g}}_k \leftarrow \frac{1}{m'} \sum_{h=1}^{m'} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i_h)}; \widetilde{\boldsymbol{\theta}}_k), \boldsymbol{y}^{(i_h)}\big)$    ▷ Gradient on intermediate $\widetilde{\boldsymbol{\theta}}_k$
      Velocity: $\boldsymbol{v}_k \leftarrow \alpha \boldsymbol{v}_{k-1} - \varepsilon_k \widehat{\boldsymbol{g}}_k$
      Update: $\boldsymbol{\theta}_k \leftarrow \widetilde{\boldsymbol{\theta}}_k - \varepsilon_k \widehat{\boldsymbol{g}}_k$    ▷ Only the gradient since $\alpha \boldsymbol{v}_{k-1}$ has already been added
      $k \leftarrow k + 1$
  end while
  return $\boldsymbol{\theta}_{k-1}$
```

3.3 Adaptive Learning Rates

As we have previously seen at least in part, setting the right learning rate or learning sequence is a difficult assignment. Moreover, a single learning rate equal for all directions can result into poor performances. For example in the ill-conditioned case, a large learning rate makes the algorithm oscillate back and forth in the direction with high curvature, but a small learning rate decreases too much the magnitude of the steps in the direction with less curvature, which are already small. The momentum strategy limits the problem adding another hyperparameter α that we have to tune, however there exist methods which automatically adjust the learning rate differently in each axis direction. These methods allow us to build solutions to a wider situation. The ill-conditioning, in fact, isn't the only case where giving the same learning rate to each direction leads to issues: more in general we encounter similar problems when the cost function is more sensitive to some directions than to others.

The procedures that automatically modify ε are called algorithms with adaptive learning rates. To understand better the concept behind them, we first show one of the earliest: the delta-bar-delta algorithm presented by Robert A. Jacobs in "Increased rates of convergence through learning rate adaptation" [18] (1988). The algorithm follows a simple idea: at the k-th step, if the sign of a partial derivative $\partial_{\theta_i} J(\boldsymbol{\theta}_{k-1})$ is the same of the previous step then increase the learning rate ε in that direction (parallel to the i-th axis), otherwise decrease it. Since the stochastic approximation of the gradient $\widehat{\boldsymbol{g}}$ using a minibatch oscillates a lot around the true gradient, the delta-bar-delta algorithm is always employed as a part of a batch algorithm (we must use all the training examples to have less noise).

Remark 3.7. *Keep in mind that all these algorithms can be extended with momentum.*

AdaGrad

AdaGrad was invented more than 20 years later, in 2011, after the resurge of Neural Networks. It's the algorithm with adaptive learning rates that brings the main intuition to construct the most important ones: in "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization" the authors decide to exploit the second moment of the partial derivatives approximations, in other words the square of the partial derivatives. In particular for every partial derivative $\widehat{g}_i = \partial_{\theta_i} \widehat{J}(\boldsymbol{\theta})$ the algorithm keeps track of the sum of all historical squared values of that partial derivative. Then it scales the learning rate in the i-th axis direction (the associated direction) with the inverse of the square root of that sum. In this way the learning rate related to the i-th axis decreases faster when the historical sum of the partial derivatives with respect to θ_i is large. Conversely it increases if the sum is small. This is reasonable since we prefer to make small steps in the directions with high derivatives and large steps where the cost surface has more soft slope.
We can write the computations in vectorial notation using elementwise operations like the product component by component: $\boldsymbol{u} = \boldsymbol{v} \odot \boldsymbol{w}$ is such that $u_i = v_i w_i$. At the k-th step, let $\boldsymbol{r}_k = \boldsymbol{r}_{k-1} + \widehat{\boldsymbol{g}}_k \odot \widehat{\boldsymbol{g}}_k$ and $\delta > 0$. Then the actual adjustment is

$$\varepsilon_k = \frac{\varepsilon}{\delta + \sqrt{\boldsymbol{r}_k}}$$

where the square root, the sum with δ and the fraction are elementwise, hence ε_k is a vector. Notice that $\delta > 0$ is a numerical constant to provide stability: we usually initialize $\boldsymbol{r}_0 = 0$ or $\boldsymbol{r}_k$ could be so small that it's approximated to 0.

Algorithm 5 AdaGrad

```
Require: Starting earning rate $\varepsilon$, stability constant $\delta$
Require: The size $m'$ of the minibatches
Require: Randomly initialized parameters vector $\boldsymbol{\theta}_0$
  Initial accumulation vector $\boldsymbol{r}_0 \leftarrow 0$
  $k \leftarrow 1$
  while All stopping criterion aren't achieved do
      Randomly select a minibatch $B$
      Gradient estimator: $\widehat{\boldsymbol{g}}_k \leftarrow \frac{1}{m'} \sum_{h=1}^{m'} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i_h)}; \boldsymbol{\theta}_{k-1}), \boldsymbol{y}^{(i_h)}\big)$
      Accumulation update: $\boldsymbol{r}_k \leftarrow \boldsymbol{r}_{k-1} + \widehat{\boldsymbol{g}}_k \odot \widehat{\boldsymbol{g}}_k$
      Compute step: $\Delta\boldsymbol{\theta}_k \leftarrow -\frac{\varepsilon}{\delta+\sqrt{\boldsymbol{r}_k}} \odot \widehat{\boldsymbol{g}}_k$
      Update: $\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} + \Delta\boldsymbol{\theta}_k$
      $k \leftarrow k + 1$
  end while
  return $\boldsymbol{\theta}_{k-1}$
```

RMSProp

The previous type of algorithm was designed for convex optimization and it usually works well in that scenario converging quickly. However if it doesn't converge fast enough (for instance in the non-convex case), taking the sum of the all historical squares from the first iteration can lead to an excessive reduction of the learning rate and step size. Therefore the steps could be so small that the algorithm stops too early. To overcome this problem Geoffrey E. Hinton and his research group change the accumulation term from a mere sum to an exponentially weighted moving average which, then, gives more importance to the current and last gradients. Moreover it gives less and less weight to the older squared values of the partial derivatives: $\boldsymbol{r}_k = \rho \boldsymbol{r}_{k-1} + (1-\rho)\widehat{\boldsymbol{g}}_k \odot \widehat{\boldsymbol{g}}_k$. At the expense of having to tune another hyperparameter $\rho > 0$, with this procedure we obtain an algorithm that can cross efficiently different types of regions (even non-convex ones) but when it encounters a locally convex region it behaves similarly to the AdaGrad initialized at that current point, converging rapidly. In other words the weight on old values are so low when the algorithm arrives at the convex region that it's like initializing AgaGrad in that convex region.

Algorithm 6 RMSProp

```
Require: Starting earning rate $\varepsilon$, stability constant $\delta$, decay rate $\rho$
Require: The size $m'$ of the minibatches
Require: Randomly initialized parameters vector $\boldsymbol{\theta}_0$
  Initial accumulation vector $\boldsymbol{r}_0 \leftarrow 0$
  $k \leftarrow 1$
  while All stopping criterion aren't achieved do
      Randomly select a minibatch $B$
      Gradient estimator: $\widehat{\boldsymbol{g}}_k \leftarrow \frac{1}{m'} \sum_{h=1}^{m'} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i_h)}; \boldsymbol{\theta}_{k-1}), \boldsymbol{y}^{(i_h)}\big)$
      Accumulation update: $\boldsymbol{r}_k \leftarrow \rho \boldsymbol{r}_{k-1} + (1-\rho)\widehat{\boldsymbol{g}}_k \odot \widehat{\boldsymbol{g}}_k$
      Compute step: $\Delta\boldsymbol{\theta}_k \leftarrow -\frac{\varepsilon}{\delta + \sqrt{\boldsymbol{r}_k}} \odot \widehat{\boldsymbol{g}}_k$
      Update: $\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} + \Delta\boldsymbol{\theta}_k$
      $k \leftarrow k + 1$
  end while
  return $\boldsymbol{\theta}_{k-1}$
```

Adam

The last variation we want to show is probably the most used one: the Adam algorithm. In 2014 Diederik P. Kingma and Jimmy Ba introduce it in "Adam: A Method for Stochastic Optimization" [21]. It uses exponentially weighted moving average to estimate the first and second moment of the gradient, also combining aspects of both momentum and RMSProp. The two estimates are multiplied by a correction term which removes the bias in the estimates. The momentum is the exponentially weighted moving average of the minibatch gradient which estimates the first moment, however it is rescaled with the inverse square root of $\boldsymbol{r}$ and not directly added after the rescaling as in the usual momentum strategy. The estimate of the second moment is the same $\boldsymbol{r}$ as in RMSProp, but in this case it also has the bias correction factor.

Remark 3.8. *If we substitute recursively till we don't have* $\boldsymbol{v}$ *terms on the right, knowing that* *we start from $\boldsymbol{v}_0 = 0$:*

$$\boldsymbol{v}_k = (1-\rho_1)\sum_{h=1}^{k} \rho_1^{k-h}\widehat{\boldsymbol{g}}_h.$$

Then the expected value is

$$\begin{aligned}\mathbb{E}\left[\boldsymbol{v}_k\right] &= (1-\rho_1)\sum_{h=1}^{k} \rho_1^{k-h}\mathbb{E}\big[\widehat{\boldsymbol{g}}_h\big] = \\ &= (1-\rho_1^k)\mathbb{E}\Big[\nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x};\boldsymbol{\theta}),\mathbf{y}\,\big)\Big]\end{aligned}$$

since we have seen in the SGD section that $\mathbb{E}\big[\widehat{\boldsymbol{g}}_h\big] = \nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) = \mathbb{E}\Big[\nabla_{\boldsymbol{\theta}} L\big(f(\mathbf{x};\boldsymbol{\theta}),\mathbf{y}\,\big)\Big]$ for every $h \in \mathbb{N}$. That's why we need the bias correction factor $\frac{1}{1-\rho_1^k}$ and we can prove it in the exact same way for the second moment.

Algorithm 7 Adam

```
Require: Starting earning rate $\varepsilon$, stability constant $\delta$, decay rates $\rho_1$ and $\rho_2$ in $[0,1)$
Require: The size $m'$ of the minibatches
Require: Randomly initialized parameters vector $\boldsymbol{\theta}_0$
  Initial first and second moment estimate $\boldsymbol{v}_0 \leftarrow 0$ and $\boldsymbol{r}_0 \leftarrow 0$
  $k \leftarrow 1$
  while All stopping criterion aren't achieved do
      Randomly select a minibatch $B$
      Gradient estimator: $\widehat{\boldsymbol{g}}_k \leftarrow \frac{1}{m'}\sum_{h=1}^{m'} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i_h)};\boldsymbol{\theta}_{k-1}),\boldsymbol{y}^{(i_h)}\big)$
      First moment estimate: $\boldsymbol{v}_k \leftarrow \rho_1\boldsymbol{v}_{k-1} + (1-\rho_1)\widehat{\boldsymbol{g}}_k$
      Second moment estimate: $\boldsymbol{r}_k \leftarrow \rho_2\boldsymbol{r}_{k-1} + (1-\rho_2)\widehat{\boldsymbol{g}}_k \odot \widehat{\boldsymbol{g}}_k$
      First moment correction: $\boldsymbol{v}_k^* \leftarrow \frac{\boldsymbol{v}_k}{1-\rho_1^k}$
      Second moment correction: $\boldsymbol{r}_k^* \leftarrow \frac{\boldsymbol{r}_k}{1-\rho_2^k}$
      Compute step: $\Delta\boldsymbol{\theta}_k \leftarrow -\frac{\varepsilon}{\delta+\sqrt{\boldsymbol{r}_k^*}} \odot \boldsymbol{v}_k^*$
      Update: $\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} + \Delta\boldsymbol{\theta}_k$
      $k \leftarrow k+1$
  end while
  return $\boldsymbol{\theta}_{k-1}$
```

Chapter 4

Choosing and Building Variations

In the previous chapter we presented better versions of the Gradient Descent to solve its problems with different methods, however all variations have their benefits and drawbacks. Even if some are better than others, there isn't a best one which works better in all situation since, depending on what we want to prioritise in our model, we could prefer one or another. The two main characteristics that we analyse when we are selecting an optimization algorithm for Neural Networks is how quick it converges (convergence rate) and how well it generalizes. In this chapter we collect some mathematical results that help us compare the various algorithms.
We conclude the chapter briefly introducing a different method to build new variations. This modern approach somehow reverses the procedure we've used so far: previously we tried to develop a solution through mathematical reasoning starting from the issues, while with this new method a Machine Learning model suggests us a new version of Gradient Descent then we have to formalize it and understand why it may work better. In some sense we can say that we start observing the solution (the new version) and only after we apply mathematical reasoning to analyse it. At the end of the chapter, we are going to show the actual algorithm suggested by this method.

4.1 Variations Comparison

Here we mainly compare the most applied methods that are SGD, SGD with momentum and Adam. SGD is, in fact, the most simple to implement and it's the base version for all other. Even it's core idea of using an unbiased estimator of the true gradient is very simple. This general simplicity often results in non-refined steps which means that the algorithm usually doesn't go through the very optimal path. It leads to a point which has a low but not lowest training error but also a low test error. Instead, the very optimal path and too precise algorithms can obtain point with overfitting problem since we try too hard to memorize training examples with complex algorithm.
Regarding the Adam algorithm instead, we can probably consider it the best adaptive learning rate algorithm since it executes strategy similar to the momentum one and also takes features from RMSProp (a better version of AdaGrad).

Convergence Results

As we previously said, SGD speed up the computations per iterations, however it could perform a lot of steps to converge due to its noisy aspect. When we study the excess error $J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ in convex case, it can be proven that this error is a $O\left(\frac{1}{\sqrt{K}}\right)$ as a function of the number of performed iterations K. The rate of convergence to a point with minimal value can be improved in the strongly convex case, getting an asymptotic behaviour of $O\left(\frac{1}{K}\right)$. We need stronger assumption to improve more the convergence rate or to obtain relevant result in the non-convex case, moreover this estimates don't get better using momentum.

That's why we use adaptive learning algorithms (especially Adam) when we need to obtain good performance faster. In order to show the differences we exhibit a result on the convergence rate of adaptive learning algorithms. The theorem is presented in the pretty recent article "A Simple Convergence Proof of Adam and Adagrad" [10] and it takes AdaGrad and Adam as case of study. We focus on the Adam case, since the AdaGrad one is similar and even with some easier inequalities.

We first rewrite their framework in our context. We want to minimize the function $J(\boldsymbol{\theta})$ with $\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^D$ whose gradient is an expected value of the gradient of a random function $\widehat{L} : \Theta \to \mathbb{R}$. In our case of Neural Networks we have $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\widehat{P}_{train}}\left[\nabla_{\boldsymbol{\theta}} \widehat{L}(\boldsymbol{\theta})\right]$ where the random function is $\widehat{L}(\boldsymbol{\theta}) = L\big(f(\mathbf{x};\boldsymbol{\theta}), \mathbf{y}\,\big)$. Using the training examples we can create a set of i.i.d. samples of the random function: we denote with $\widehat{L}_i(\,\cdot\,) = L\big(f(\boldsymbol{x}^{(i)};\,\cdot\,), \boldsymbol{y}^{(i)}\,\big)$. Notice that proceeding in this way we are implicitly using minibatches of size $m' = 1$, this makes the notation not too heavy but it can be easily adjust to get the same result with $m' > 1$.
Since the article concerns both AdaGrad and Adam, they describe the parameters update in a unique framework where we can obtain both algorithms changing the hyperparameters:

$$\begin{aligned} \boldsymbol{v}_k &= \beta_1 \boldsymbol{v}_{k-1} + \nabla_{\boldsymbol{\theta}} \widehat{L}_k(\boldsymbol{\theta}_{k-1}) \\ \boldsymbol{r}_k &= \beta_2 \boldsymbol{r}_{k-1} + \nabla_{\boldsymbol{\theta}} \widehat{L}_k(\boldsymbol{\theta}_{k-1}) \odot \nabla_{\boldsymbol{\theta}} \widehat{L}_k(\boldsymbol{\theta}_{k-1}) \\ \boldsymbol{\theta}_k &= \boldsymbol{\theta}_{k-1} - \varepsilon_k \frac{\boldsymbol{v}_k}{\sqrt{\delta + \boldsymbol{r}_k}} \end{aligned} \tag{4.1}$$

where we obtain the AdaGrad algorithm with $\beta_1 = 0$, $\beta_2 = 1$ and $\varepsilon_k = \varepsilon$, instead we obtain (actually nearly obtain) Adam with

$$\varepsilon_k = \varepsilon \frac{1-\beta_1}{\sqrt{1-\beta_2}} \frac{\sqrt{1-\beta_2^k}}{1-\beta_1^k}.$$

It's easy to see that in the previous formula we insert the first fraction to get the actual exponentially weighted average of the real Adam and the second fraction contains the bias correction terms. Actually, in order to make the sequence ε_k monotonic and thus to facilitate the proof, the authors decided to get rid of the correction term $(1-\beta_1^k)$ of the first moment. It is a reasonable simplification since with the most used setting ($\beta_1 = 0.9$ and $\beta_2 = 0.999$) that correction terms tends to 1 more rapidly.

Remark 4.1. *Moreover notice that there is another slight modification in the algorithm which doesn't affect its behaviour: in this version the stabilization term δ is put under the square root.*

The authors exploit an idea shown in the article "Stochastic First- and Zeroth-order Methods for Nonconvex Stochastic Programming" [12] that then became the standard in optimization of

non-convex function. It can be proven that in the deterministic case we can study convergence looking at how $\min_{k=1,\dots,K} \|\nabla F(\boldsymbol{x}_k)\|^2$ behaves as function of the number of iterations performed K. In a similar way Saeed Ghadimi and Guanghui Lan show that in the stochastic context (we are using stochastic approximation of the gradient) we can study convergence analysing the expectation $\mathbb{E}\big[\|\nabla F(\boldsymbol{x}_\tau)\|^2\big]$ where τ is a random variable with a specific distribution over the set of indexes $\{0,\dots,K-1\}$.
In our case $\nabla F(\boldsymbol{x}_\tau) = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_\tau)$ and we assume

$$P(\tau = k) \propto 1 - \beta_1^{K-k} \quad \text{for } k < K. \tag{4.2}$$

Therefore, when $\beta_1 = 0$ we have that τ has a uniform distribution, while for $\beta_1 \to 1$ the last iterations have much less probability. The intuition behind this parametrization is that when $\beta_1 = 0$ the approximation of the gradient $\widehat{\boldsymbol{g}}_k$ of each step has the same weight in the momentum. Instead, when we compute the current momentum with $\beta_1 \to 1$, the current gradient approximation has small weight $(1 - \beta_1)$ compared to the weight β_1 of the previous momentum. Then a gradient approximation really influences the learning step only when it has been part of the momentum for a few iterations. In the distribution of τ we set the probability of an iteration h considering the influence of the gradient approximation at that step $\widehat{\boldsymbol{g}}_k$.

To declare the actual result we need some usual assumptions take to obtain a lot of theoretical results even if in some real cases they could be too restrictive.

1. J is bounded below by J_*: $J(\boldsymbol{\theta}) \geq J_*$ for every $\boldsymbol{\theta} \in \Theta$.

2. $\widehat{L}(\boldsymbol{\theta})$ has uniformly a.s. bounded gradient: meaning that for every $\boldsymbol{\theta} \in \Theta$ it holds the following $\left\|\nabla_{\boldsymbol{\theta}} \widehat{L}(\boldsymbol{\theta})\right\|_\infty \leq R - \sqrt{\delta}$ a.s., where δ is added just to simplify the computations later and $\sqrt{\delta} \leq R$.

3. The gradient of J is Lipschitz continuous: $\|\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_1) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_2)\|_2 \leq L\|\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2\|_2$ for every $\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2 \in \Theta$.

Then the theorem states:

Theorem 4.1. *Suppose that the previous assumptions are verified. Let $\boldsymbol{\theta}_k$ a sequence following the discrete dynamic 4.1 with $0 < \beta_2 < 1$, $0 \leq \beta_1 < \beta_2$, $\varepsilon > 0$, $\varepsilon_k = \varepsilon(1-\beta_1)\sqrt{\frac{1-\beta_2^k}{1-\beta_1^k}}$, τ defined by 4.2. Then for every $K \in \mathbb{N}$ such that $K > \frac{\beta_1}{1-\beta_1}$ we have*

$$\mathbb{E}\big[\|\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_\tau)\|^2\big] \leq 2R\frac{J(\boldsymbol{\theta}_0) - J_*}{\varepsilon\widetilde{K}} + C\left(\frac{1}{\widetilde{K}}\ln\left(1 + \frac{R^2}{(1-\beta_2)\delta}\right) - \frac{K}{\widetilde{K}}\ln\beta_2\right) \tag{4.3}$$

where $\widetilde{K} = K - \frac{\beta_1}{1-\beta_1}$ and

$$C = \frac{\varepsilon DRL(1-\beta_1)}{\left(1 - \frac{\beta_1}{\beta_2}\right)(1-\beta_2)} + \frac{12DR^2\sqrt{1-\beta_1}}{\left(1 - \frac{\beta_1}{\beta_2}\right)^{\frac{3}{2}}\sqrt{1-\beta_2}} + \frac{2\varepsilon^2 DL^2\beta_1}{\left(1 - \frac{\beta_1}{\beta_2}\right)(1-\beta_2)^{\frac{3}{2}}}.$$

Generalization Result

As we have just seen, adaptive learning rate algorithms converge fast to a critical point. However, after the quick initial decrease of the training error, they rapidly reach a plateau in the performance on test set. In other words, they seem to be stuck in minimal point of the training error which correspond to low, but not always satisfying, point of the test error. This property is probably related to specific characteristic of the minima they usually encounter. In this context the regions containing the minima are, indeed, divided in "sharp", flat and asymmetric basins: the first have high curvature in all directions, the second have low curvature in all directions and the third have both high and low curvature directions. It is largely believed and it has a lot of empirical evidence to support it that usually the flat and asymmetric basin make the model generalize better, while the sharp ones often lead to poor generalization. Really, it was first believed that we can make this type of empirical assumption only on sharp and flat basin. That's why researchers thought that the generalization gap between SGD and adaptive algorithms was theoretically connected to a curvature argument, as we can see in "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima" [20]. It was only after the publication of "Asymmetric Valleys: Beyond Sharp and Flat Local Minima" [15] that researchers start to suppose the well generalization property also on the asymmetric basins, losing the intuition on curvature arguments to understand the generalization gap between SGD and adaptive algorithms. However in the article "Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning" [45] the authors provide a new interesting viewpoint related to the volume of the basins and we are going to analyse it in this subsection. In particular they compare SGD with the Adam algorithm.

We first rewrite the updates of both algorithm in terms of the noise caused by the gradient approximation. Let $\boldsymbol{u}_k = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k) - \widehat{\boldsymbol{g}}_{k+1}$ be the noise, recalling that $\widehat{\boldsymbol{g}}_{k+1} = \nabla_{\boldsymbol{\theta}} J_B(\boldsymbol{\theta}_k)$ is the gradient computed on the sum over B of the per-example costs $J_i(\boldsymbol{\theta}_k)$, then $J = \sum_{i=1}^m J_i$. It was previously believed that it was distributed as a Gaussian vector $\mathcal{N}(\boldsymbol{0}, \boldsymbol{\Sigma}_k)$. In particular, it holds the following:

Proposition 4.1. *If examples are reused in subsequent minibatches, then the covariance matrix is*

$$\boldsymbol{\Sigma}_k = \frac{1}{m'}\left(\frac{1}{n}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}_k) \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}_k)^\top - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k)^\top\right)$$

Proof. Let $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k)$ and $\boldsymbol{g}_i = \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}_k)$.

$$\begin{aligned}\boldsymbol{\Sigma}_k &= \mathbb{E}\left[\left(\frac{1}{m'}\sum_{h=1}^{m'} \boldsymbol{g}_{i_h} - \boldsymbol{g}\right)\left(\frac{1}{m'}\sum_{h=1}^{m'} \boldsymbol{g}_{i_h} - \boldsymbol{g}\right)^\top\right] = \\ &= \mathbb{E}\left[\frac{1}{m'}\sum_{h=1}^{m'} (\boldsymbol{g}_{i_h} - \boldsymbol{g}) \frac{1}{m'}\sum_{h=1}^{m'} (\boldsymbol{g}_{i_h} - \boldsymbol{g})^\top\right] = \\ &= \frac{1}{m'^2}\sum_{h=1}^{m'} \mathbb{E}\left[(\boldsymbol{g}_{i_h} - \boldsymbol{g})(\boldsymbol{g}_{i_h} - \boldsymbol{g})^\top\right]\end{aligned} \tag{4.4}$$

since for $j \neq h$ by independence of the indexes chosen by the minibatch we have

$$\mathbb{E}\left[(\boldsymbol{g}_{i_j} - \boldsymbol{g})(\boldsymbol{g}_{i_h} - \boldsymbol{g})^\top\right] = \mathbb{E}\big[\boldsymbol{g}_{i_j} - \boldsymbol{g}\big]\mathbb{E}\big[\boldsymbol{g}_{i_h} - \boldsymbol{g}\big]^\top = 0.$$

Continuing the previous series of equality, by the identical distribution of the random indexes i_h we get

$$\begin{aligned}\boldsymbol{\Sigma}_k &= \frac{1}{m'}\mathbb{E}\left[(\boldsymbol{g}_h - \boldsymbol{g})(\boldsymbol{g}_h - \boldsymbol{g})^\top\right] = \\ &= \frac{1}{m'}\frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{g}_i - \boldsymbol{g})(\boldsymbol{g}_i - \boldsymbol{g})^\top = \\ &= \frac{1}{m'}\left(\frac{1}{m}\sum_{i=1}^{m}\boldsymbol{g}\boldsymbol{g}_i^\top - \boldsymbol{g}\boldsymbol{g}^\top\right)\end{aligned} \tag{4.5}$$

since $\frac{1}{m}\sum\limits_{i=1}^{m}\boldsymbol{g}_i\boldsymbol{g}^\top = \frac{1}{m}\sum\limits_{i=1}^{m}\boldsymbol{g}\boldsymbol{g}_i^\top = \boldsymbol{g}\boldsymbol{g}^\top$. $\square$

In "A Tail-Index Analysis of Stochastic Gradient Noise in Deep Neural Networks" [39] it was shown that the tails of the distribution are heavier than the Gaussian ones. Therefore the noise is better approximated by a symmetric alpha-stable distribution, denoted with $S\alpha S$. The Gaussian distribution is a particular case of $S\alpha S$ with $\alpha = 2$. Knowing that we can rewrite the SGD update:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \varepsilon\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}_k) + \varepsilon^{(1-\frac{1}{\alpha})}\varepsilon^{\frac{1}{\alpha}}\boldsymbol{u}_k. \tag{4.6}$$

We can see it as the discretization of a continuous time stochastic differential equation driven by a Lévy process, then for a small enough learning rate we can approximate the update dynamics of SGD with

$$d\boldsymbol{\theta}_t = -\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}_t)dt + \varepsilon^{\frac{\alpha-1}{\alpha}}\boldsymbol{\Sigma}_t^{\frac{1}{2}}d\boldsymbol{L}_t \tag{4.7}$$

where $\boldsymbol{L}_t$ is a stochastic Lévy process with independent components obeying $S\alpha S(1)$ distributions and $\boldsymbol{\Sigma}_t$ is defined as in the previous proposition.

Remark 4.2. *We say that a random variable* x *has an* $S\alpha S(\sigma)$ *distribution if* $\mathbb{E}\big[e^{iu\mathrm{x}}\big] = e^{-\sigma^\alpha|u|^\alpha}$.

We can do the same with the Adam's update rule. Notice that in this case the noise of the gradient is contained in the momentum terms: let $\boldsymbol{v}'_k = \beta_1\boldsymbol{v}'_{k-1} + (1-\beta_1)\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}_{k-1})$ be the sequence of first moment with the real gradient, then the noise of its estimate is

$$\boldsymbol{v}'_k - \boldsymbol{v}_k = (1-\beta_1)\sum_{h=0}^{k}\beta_1^{k-h}\boldsymbol{u}_h. \tag{4.8}$$

Since it's an exponentially weighted average of the noises, it keeps the heavy tails and the same behaviour. Therefore with the bias correction terms it has an alpha stable distribution: $\frac{1}{1-\beta_1}(\boldsymbol{v}'_k - \boldsymbol{v}_k)$ is $S\alpha S$. Consequently we can approximate Adam's updates dynamics with a continuous time SGD

$$\begin{aligned}d\boldsymbol{\theta}_t &= -\mu_t\boldsymbol{Q}_t^{-1}\boldsymbol{v}_t + \varepsilon^{\frac{\alpha-1}{\alpha}}\boldsymbol{Q}_t^{-1}\boldsymbol{\Sigma}_t^{\frac{1}{2}}d\boldsymbol{L}_t \\ d\boldsymbol{v}_t &= \beta_1(\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})_t - \boldsymbol{v}_t) \\ d\boldsymbol{r}_t &= \beta_2\big(\nabla_{\boldsymbol{\theta}}J_B(\boldsymbol{\theta}_t)\odot\nabla_{\boldsymbol{\theta}}J_B(\boldsymbol{\theta}_t) - \boldsymbol{r}_t\big)\end{aligned} \tag{4.9}$$

where $\boldsymbol{Q}_t = \operatorname{diag}\left(\sqrt{\omega_t \boldsymbol{r}_t} + \delta\right)$ and the two bias correction constant are $\mu_t = \frac{1}{1-e^{-\beta_1 t}}$ and $\omega_t = \frac{1}{1-e^{-\beta_2 t}}$.

Now we study the behaviour of the trajectories generated by the previous stochastic differential equation. Suppose that both $\boldsymbol{\theta}_t$ start from a point $\boldsymbol{\theta}_0$ in a basin Ω with minimum $\boldsymbol{\theta}^*$: $\boldsymbol{\theta}_0 \in \Omega$. To simplify the notation, from now on let $\eta = \varepsilon^{\frac{\alpha-1}{\alpha}}$. We set $\Omega^{-\eta^\gamma} = \{\, \boldsymbol{y} \in \Omega \mid d(\partial\Omega, \boldsymbol{y}) \geq \eta^\gamma \,\}$, a sort of inner part of Ω, with the constant $\gamma > 0$. Then we define two important object for the next theorem: the escaping time Γ from the basin Ω and the escaping set $\mathcal{W}$:

$$\Gamma = \inf\{\, t \geq 0 \mid \boldsymbol{\theta}_t \notin \Omega^{-\eta^\gamma} \,\} \tag{4.10}$$

$$\mathcal{W} = \{\, \boldsymbol{y} \in \mathbb{R}^D \mid \boldsymbol{Q}_{\boldsymbol{\theta}^*}^{-1} \boldsymbol{\Sigma}_{\boldsymbol{\theta}^*}^{\frac{1}{2}} \boldsymbol{y} \notin \Omega^{-\eta^\gamma} \,\} \tag{4.11}$$

where $\boldsymbol{\Sigma}_{\boldsymbol{\theta}^*}^{\frac{1}{2}} = \lim\limits_{\boldsymbol{\theta}_t \to \boldsymbol{\theta}^*} \boldsymbol{\Sigma}_t^{\frac{1}{2}}$, $\boldsymbol{Q}_{\boldsymbol{\theta}^*} = \mathbb{1}$ for SGD and $\boldsymbol{Q}_{\boldsymbol{\theta}^*} = \lim\limits_{\boldsymbol{\theta}_t \to \boldsymbol{\theta}^*} \boldsymbol{Q}_t$ for Adam.
For the next theorem we need two main assumptions:

1. Suppose that $J(\boldsymbol{\theta})$ is a non-negative cost function with an upper bound that is also μ-strongly convex and l-smooth

2. For Adam we need additional assumptions. First of all we suppose that $\boldsymbol{\theta}_t$ satisfies $\int_0^\Gamma \left\langle \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_s)}{1+J(\boldsymbol{\theta}_s)}, \mu_s \boldsymbol{Q}_s^{-1} \boldsymbol{v}_s \right\rangle ds \geq 0$ almost surely with $\beta_1 \leq \beta_2 \leq 2\beta_1$. Furthermore we want $\|\boldsymbol{v}_t - \widehat{\boldsymbol{v}}_t\| \leq \tau_m \| \int_0^{t^-} (\boldsymbol{v}_s - \widehat{\boldsymbol{v}}_s) ds \|$ and $\|\widehat{\boldsymbol{v}}_t\| \geq \tau \|\nabla_{\boldsymbol{\theta}} J(\widehat{\boldsymbol{\theta}}_t)\|$ for some constant τ and τ_m where $\widehat{\boldsymbol{v}}_t$ and $\widehat{\boldsymbol{\theta}}_t$ follow the Adam update SDE with $\eta = 0$. The components of the first moment are uniformly bounded: $v_{min} \leq \sqrt{v}_{t,i} \leq v_{max}$.

Let $m(\cdot)$ be a non-zero Radon measure such that $m(\mathcal{U}) < m(\mathcal{V})$ for $\mathcal{U} \subset \mathcal{V}$: bigger volume means larger measure. Before stating the theorem we set some key constants that are different for SGD and Adam: $k_1 = l$ and $k_2 = 2\mu$ in the SGD case, while $k_1 = \frac{c_1 l}{(v_{min}+\delta)|\tau_m - 1|}$ and $k_2 = \frac{2\mu\tau}{\beta_1(v_{max}+\delta)+\mu\tau}\left(\beta_1 - \frac{\beta_2}{4}\right)$ for a constant c_1. The theorem then states:

Theorem 4.2. *Suppose that the previous two assumption hold. Let* $\Xi(\eta^{-1}) = \frac{2}{\alpha}\eta^\alpha$, $\rho_0 = \frac{1}{16(1+c_2 k_1)}$ *for a constant* c_2 *and* $\ln\left(\frac{2\Delta}{\mu\eta^{\frac{1}{3}}}\right) \leq k_2 \eta^{-\frac{1}{3}}$ *where* $\Delta = J(\boldsymbol{\theta}_0) - J(\boldsymbol{\theta}^*)$. *Then for any* $\boldsymbol{\theta}_0 \in \Omega^{-2\eta^\gamma}$, $u > -1$, $\eta \in (0, \eta_0]$, $\gamma \in (0, \gamma_0]$ *and* $\rho \in (0, \rho_0]$ *such that* $\eta^\gamma \leq \rho_0$ *and* $\lim\limits_{\eta \to 0} \rho = 0$, *then both SGD and Adam stochastic differential equation give us the following estimate*

$$\frac{1-\rho}{1+u+\rho} \leq \mathbb{E}\left[e^{-um(\mathcal{W})\Xi(\eta^{-1})\Gamma}\right] \leq \frac{1+\rho}{1+u-\rho}. \tag{4.12}$$

We notice that for η small enough $\rho \approx 0$ then for every $u > -1$ we have

$$\frac{1}{u} \lessapprox \mathbb{E}\left[e^{-um(\mathcal{W})\Xi(\eta^{-1})\Gamma}\right] \lessapprox \frac{1}{u}.$$

This means that for both Adam and SGD the bounds on the expected value of the escaping time Γ are of the order $O\left(\frac{1}{m(\mathcal{W})\Xi(\eta^{-1})}\right)$. This is the crucial result to understand why SGD converges to minima which often tend to generalize better than Adam ones. In fact, their two escaping sets are different because they have different $\boldsymbol{Q}_{\boldsymbol{\theta}^*}$. Therefore it can be proven that, if both algorithms start in the same basin Ω, SGD escapes in smaller expected time Γ since $m(\mathcal{W}_{SGD})$ is greater

than $m(\mathcal{W}_{Adam})$.
In order to see this property under a prospective of basin's characteristics, we observe that large $m(\mathcal{W})$ is equivalent to small $m(\mathcal{W}^c)$ which in turn is equivalent to large volume of the basin Ω, where $\mathcal{W}^c = \{ \boldsymbol{y} \in \mathbb{R}^D \mid \boldsymbol{Q}_{\boldsymbol{\theta}^*}^{-1} \boldsymbol{\Sigma}_{\boldsymbol{\theta}^*}^{\frac{1}{2}} \boldsymbol{y} \in \Omega^{-\eta^\gamma} \}$ is the complementary of $\mathcal{W}$. In conclusion, compared to Adam, SGD is more likely to stop in basin Ω with larger volume (the flat and asymmetric types that generalize well) because it escapes faster from the one with smaller volume while Adam could be stuck in them.

If we approximate the basin Ω with a quadratic one centered at $\boldsymbol{\theta}^*$ we can prove the difference between the two escaping sets. Let $h(\boldsymbol{\theta}^*)$ be the basin height and $\boldsymbol{H}(\boldsymbol{\theta}^*)$ be the Hessian at $\boldsymbol{\theta}^*$, then we can write $\Omega \approx \{ \boldsymbol{y} \in \mathbb{R}^D \mid J(\boldsymbol{\theta}^*) + \frac{1}{2}\boldsymbol{y}^\top \boldsymbol{H}(\boldsymbol{\theta}^*)\boldsymbol{y} \leq h(\boldsymbol{\theta}^*) \}$. Instead, the two escaping sets are

$$\mathcal{W}_{SGD} = \{ \boldsymbol{y} \in \mathbb{R}^D \mid \boldsymbol{y}^\top \boldsymbol{\Sigma}_{\boldsymbol{\theta}^*} \boldsymbol{H}(\boldsymbol{\theta}^*) \boldsymbol{\Sigma}_{\boldsymbol{\theta}^*} \boldsymbol{y} \geq h^* \}$$
$$\mathcal{W}_{Adam} = \{ \boldsymbol{y} \in \mathbb{R}^D \mid \boldsymbol{y}^\top \boldsymbol{\Sigma}_{\boldsymbol{\theta}^*} \boldsymbol{Q}_{\boldsymbol{\theta}^*}^{-1} \boldsymbol{H}(\boldsymbol{\theta}^*) \boldsymbol{Q}_{\boldsymbol{\theta}^*}^{-1} \boldsymbol{\Sigma}_{\boldsymbol{\theta}^*} \boldsymbol{y} \geq h^* \}$$

where $h^* = 2(h(\boldsymbol{\theta}^*) - J(\boldsymbol{\theta}^*))$. The geometry adaptation in the Adam algorithm due to $\boldsymbol{Q}_t^{-1}$ makes the escaping sets different. In particular, studying the behaviour of $\boldsymbol{\Sigma}_t \boldsymbol{H}(\boldsymbol{\theta}^*) \boldsymbol{\Sigma}_t$ and $\boldsymbol{\Sigma}_t \boldsymbol{Q}_t^{-1} \boldsymbol{H}(\boldsymbol{\theta}^*) \boldsymbol{Q}_t^{-1} \boldsymbol{\Sigma}_t$ when $\boldsymbol{\theta}_t$ tends to the minimum $\boldsymbol{\theta}^*$, we can see that the first one becomes more singular than the other. Computing the volumes exploiting the singular values of the Hessian and the covariance matrix, it can be proven that $\mathcal{W}_{SGD}^c$ is much smaller than $\mathcal{W}_{Adam}^c$ achieving the desired result.

4.2 Program Search to Find Variations

In this section we describe the method applied in the article "Symbolic Discovery of Optimization Algorithms" [5] to obtain the EvoLved Sign Momentum (Lion) algorithm. Since the algorithm isn't discovered but it's suggested through Machine Learning, someone could argue on the scientific importance of following this procedure. However, even if we could lose a part of the component on intuitions and idea built a priori, it is still an interesting challenge to formalize and analyse the features of Lion algorithm. Indeed, in Machine Learning it's not always easy to get theoretical results that reflect what it's empirically observed.
We can say that this modern approach remains in the fields of scientific method since we can consider the procedure as observing a real life phenomenon and explaining why it works. In fact, we observe that the suggested Lion algorithm performs better than other often employed method but the Machine Learning model doesn't tell us why. It's researchers duty to describe why and (sometimes more important) when it achieves better performance. Moreover, this assignment isn't easy as it could seem, indeed, in about a year since Lion came out few results have been obtained.

Program Search Model

The authors' aim was to build an efficient model to *learn* an optimization algorithm for deep networks. Since it's not easy to see this procedure as learning a parameters vectors $\boldsymbol{\theta}$, they decided to use a completely different strategy compared to the one we used so far: they implemented a symbolic representation. In symbolic representation the data are no more described as vectors of real numbers, instead they are represented by symbols usually easy to read and interpret by humans. With this strategy we can represent not only data but also concepts and relationships. In fact, symbols can be data structure as binary trees, operators like the logical ones, variables or even function. In our case the optimization algorithms are going to be represented by programs (code snippet) of a fictitious high-level programming language which can deal with n-dimensional arrays.

Before showing the model details, we make a parallelism to understand better what is the general idea of *program search* (searching a program through Machine Learning). In parametrized models we have a set of training examples and at each step we change $\boldsymbol{\theta}$ trying to decrease the error on those examples, lastly when we achieve convergence we check the actual performance computing the error on another examples set (test tet). In program search with symbolic representation, we have a set of proxy tasks and at each step we change the program trying to increase the performance on those tasks, lastly when we achieve convergence we check the actual efficiency computing the performance on another set of more complex tasks.

Every program in the search space has the following structure:

```
w = initialize_weights()
v1 = zero_initialization()
v2 = zero_initialization()
for i in range(max_iterations) do
    lr = learning_schedule(i)
    g = compute_gradient(w, get_batch(i))
    update, v1, v2 = train(w, g, v1, v2, lr)
    w = w - update
end for

def train(w, g, v1, v2, lr):
      ...
```

In other words, when we search for the optimal program, at each we are allowed to change only the definition of the function *train(...)*. Therefore, the optimization algorithms we can get are Gradient Descent variations that differ only on how they compute the update and how they track the two extra variables $v1$ and $v2$, which store historical information on previous steps. The authors chose to have only two extra "historical" variables in order to obtain an algorithm with amount of required memory similar to the Adam algorithm one, where in particular $v1$ and $v2$ are the first and second moment approximation but they can be used in different ways (as Lion does in fact).
Each line within the *train(...)* definition calls a function with constants and existing variables as argument and saves the output in a new variable or rewrite an existing one. The permitted changes in the code are of the three following type:

1. Insert a new line at a random location randomly calling a function and its arguments

2. Edit a function in a random line by changing one of its arguments: we can replace it with a new constant or a local variable or we can rescale it if it is a constant

3. Delete a random line

New constant are sampled from a standard normal distribution $\mathcal{N}(0, 1)$, while constants modified by rescaling are multiplied by a factor 2^{z} where $z \sim \mathcal{N}(0, 1)$. Functions can be selected from a set of 45 common mathematical functions, but there are no restrictions on the number of possible local variables or statements. Therefore the search space is really large and we need an efficient technique to find the right program. Moreover, it's a sparse search space since it can be shown that most of the program are inefficient even in simple tasks.

Remark 4.3. *Notice that, even if we fix a maximum number of local variable n_v and a maximum program length l, the amount of possible program is very large: it's more than $n_p = n_f^l n_a^{n_v l}$ where n_a is the average amount of arguments per statement.*

In order to limit the problem related to the large sparse search, the authors choose an evolutionary algorithm which is a type of population-based optimization algorithms inspired by biological evolution. Basically, the population is the set of candidates and at each iteration it evolves forming a new generation. Every generation is usually composed by some elements of the previous one (keeping at least the ones with best performances) and some modifications of previous elements.
In our case the population is composed by P programs in every step. Then we apply the so-called tournament selection: at each step $T < P$ programs are randomly selected and its taken as parent the one with the best performance on the proxy tasks. Following one of the previous three rules, we modify a copy of the parent element. The obtained element is called child and it's added to the population while its older element is deleted. The authors chose the default population size of $P = 1000$ and tournament size of $T = 2$.

The process can be improved applying two types of restart. Restarting the search from the initial population promotes exploration. It can be easily implemented running multiple process in parallel rather than actually restarting the process every time. Moreover it allows us to check real results, because comparing multiple initial restarts decreases the high variance of a process with such randomness. Instead, restarting the search from P copies of the best algorithm found so far promotes exploitation. The process can, in fact, reach a plateau and stop to improve. However, restarting from the best fully exploits the knowledge obtained so far and it can lead to additional improvements.

Moreover, in order to speed up the search, they used a warm-start, which in general means that we start from a set of elements or parameters $\boldsymbol{\theta}$ already known for performing well. In particular they start with a population of P copies of the AdamW algorithm. It simply is a slight variation of the Adam algorithm where we add a decoupled weight decay term for regularization. The regularization terms are functions of the weights added in the cost function. One of the most used is the L^2 weight decay: $\frac{\lambda}{2}\left\|\boldsymbol{\theta}\right\|_2^2$. When a regularization term is decoupled, its derivative (in the weight decay case $\lambda\boldsymbol{\theta}$) is not added when we compute the gradient or its estimate, but it's multiplied by the learning rate and added directly in the update.

Remark 4.4. $\lambda \geq 0$ *is an hyperparameter that determines the magnitude of the regularization: larger λ brings stronger a regularization effect, while small λ leads to weak regularization which in the extreme case of $\lambda = 0$ obviously becomes null-regularization (the standard algorithm). no-regularization when .*

We can clearly see the decoupled procedure in the AdamW algorithm comparing it with the standard one:

Algorithm 8 AdamW

```
Require: Starting earning rate $\varepsilon$, stability constant $\delta$, decay rates $\rho_1$ and $\rho_2$ in $[0, 1)$
Require: The size $m'$ of the minibatches
Require: Randomly initialized parameters vector $\boldsymbol{\theta}_0$
  Initial first and second moment estimate $\boldsymbol{v}_0 \leftarrow 0$ and $\boldsymbol{r}_0 \leftarrow 0$
  $k \leftarrow 1$
  while All stopping criterion aren't achieved do
      Randomly select a minibatch $B$
      Gradient estimator: $\widehat{\boldsymbol{g}}_k \leftarrow \frac{1}{m'} \sum_{h=1}^{m'} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i_h)}; \boldsymbol{\theta}_{k-1}), \boldsymbol{y}^{(i_h)}\big)$
      First moment estimate: $\boldsymbol{v}_k \leftarrow \rho_1 \boldsymbol{v}_{k-1} + (1-\rho_1)\widehat{\boldsymbol{g}}_k$
      Second moment estimate: $\boldsymbol{r}_k \leftarrow \rho_2 \boldsymbol{r}_{k-1} + (1-\rho_2)\widehat{\boldsymbol{g}}_k \odot \widehat{\boldsymbol{g}}_k$
      First moment correction: $\boldsymbol{v}_k^* \leftarrow \frac{\boldsymbol{v}_k}{1-\rho_1^k}$
      Second moment correction: $\boldsymbol{r}_k^* \leftarrow \frac{\boldsymbol{r}_k}{1-\rho_2^k}$
      Compute step: $\Delta\boldsymbol{\theta}_k \leftarrow -\varepsilon \left( \frac{\boldsymbol{v}_k^*}{\delta + \sqrt{\boldsymbol{r}_k^*}} + \lambda\boldsymbol{\theta} \right)$      ▷ Related line
      Update: $\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} + \Delta\boldsymbol{\theta}_k$
      $k \leftarrow k + 1$
  end while
  return $\boldsymbol{\theta}_{k-1}$
```

The amount of program computationally equivalent, invalid or with unnecessary lines is extremely high in this type of search space. That's why, when we generate the child for the new generation, the authors chose to run what they call an *abstract execution*. It's done when a program is created so previous than the actual execution. This abstract execution has a small cost, especially considering that it makes programs three times shorter and reduces the search cost of ten times (the factor are obtained studying the cache hit rate and unnecessary lines contained on average). The abstract execution monitor the presence of error due to wrong variables shapes or types. When an error occurs it keeps generating a modified version until a valid one is created. Moreover it produces an hash map to uniquely identified how the operations flow compute inputs and outputs. This hash is used to spot programs that perform equivalent computations. Lastly, it delete unnecessary lines, which are the ones that don't contribute to the output.

Since we are using an evolutionary algorithm, we don't get a unique best program but rather a set of promising programs that perform well on the proxy tasks. To really get the best one we have to compare them on the larger tasks chosen to check performances. In order to do it efficiently, they employ funnel selection. They first take subsets of the larger tasks and sort them with increasing complexity (more training steps, larger model size and other adjustments). Then they test all the promising programs on the subset with smallest complexity and keep only the ones with performance higher than a fixed threshold. They proceed in the same way iteratively increasing the complexity and keeping the best ones.

Lion Algorithm

After few simplifications like deleting redundant statements and removing statements that produce minimal differences in the output (can be done reusing the previous method disabling the possibility to insert lines of code), the authors obtained the following algorithm that seems to take features of both SGD and adaptive learning algorithms:

```
Algorithm 9 Lion Optimizer
Require: $\beta_1$, $\beta_2$, $\lambda$, $\eta$
Require: Initialize $v \leftarrow 0$
Require: The size $m'$ of the minibatches
Require: Randomly initialized parameters vector $\boldsymbol{\theta}$
  $k \leftarrow 1$
  while All stopping criterion aren't achieved do
      Randomly select a minibatch $B$
      Gradient estimator: $\widehat{\boldsymbol{g}} \leftarrow \frac{1}{m'} \sum_{k=1}^{m'} \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i_k)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i_k)}\big)$
      Pre-update: $\boldsymbol{c} \leftarrow \beta_1 v + (1 - \beta_1)\widehat{\boldsymbol{g}}$
      Update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\big(\mathrm{sgn}(c) + \lambda\theta\big)$
      Update EMA of $\widehat{\boldsymbol{g}}$: $v \leftarrow \beta_2 h + (1 - \beta_2)\widehat{\boldsymbol{g}}$
      $k \leftarrow k + 1$
  end while
  return $\boldsymbol{\theta}$
```

Appendix A

Point Estimator

In this chapter we present some concepts on point estimation which are applied but not formally introduced within the thesis. First of all, point estimation is a method that tries to find the individual "best" (in some sense) estimator of an object with the role of parameter in a statistical model. It differs from Bayesian statistic where we compute the probabilities of all possible values of the object and we make prediction weighting with respect to these probabilities. As we previously said, in point estimation, instead, we search for a single value since the true value of the object is unknown but fixed.
We talk about *object* since it can be a scalar, a parameters vector or even a function (intended as a point in a function space). However, in our case the object is of the first two types, for example during the training phase we try to estimate the parameters vector defining the Neural Network.

The main tools of this method are the previously mentioned estimators. An estimator is a random variable or vector that try to infer the unknown value of the statistical parameter by exploiting some observed data. In other words, it's a function of the random variables or vectors representing our sample. Since it doesn't require anything even on the actual codomain of the estimator, this definition is very general but it gives us more choices when we are building a point estimation method.

We are going to denote with $\boldsymbol{\theta}$ the unknown quantity we want to infer and with $\widehat{\boldsymbol{\theta}}$ its estimator. Since the data random vectors are $\mathbf{x}_1, \ldots, \mathbf{x}_m$, we write $\widehat{\boldsymbol{\theta}}_m = F_m(\mathbf{x}_1, \ldots, \mathbf{x}_m)$, then we are going to specify with the subscript how many samples are used.

Example A.1. *The expected value of an element in the data is* $\boldsymbol{\mu} = \mathbb{E}[\mathbf{x}]$. *One of its possible estimators is the sample or empirical mean: it's the average of the values of a data sample*

$$\widehat{\boldsymbol{\mu}}_m = \frac{1}{m}\sum_{k=1}^{m} \mathbf{x}_k. \tag{A.1}$$

Bias

The error performed by the estimator can be thought as a random variable or vector depending on the value of the statistical parameter $\boldsymbol{\theta}$: it's computed with $e_m(\boldsymbol{\theta}) = \widehat{\boldsymbol{\theta}}_m - \boldsymbol{\theta}$. We can take the expected value of this quantity with respect to the data distributions that are associated with

$\boldsymbol{\theta}$ in the statistical model $\mathcal{P}$. In this way we obtain a function of the statistical parameter we want to estimate. This function is the bias of the estimator $\widehat{\boldsymbol{\theta}}$ relative to $\boldsymbol{\theta}$ and it's defined in the following way:

$$\text{Bias}_{\boldsymbol{\theta}}\left(\widehat{\boldsymbol{\theta}}_m\right) = \text{Bias}\left(\widehat{\boldsymbol{\theta}}_m\right) = \mathbb{E}_{\boldsymbol{\theta}}\left[\widehat{\boldsymbol{\theta}}_m\right] - \boldsymbol{\theta}. \tag{A.2}$$

As it's shown in the formula, we simply denote it with $\text{Bias}\left(\widehat{\boldsymbol{\theta}}_m\right)$ to lighten the notation because $\boldsymbol{\theta}$ is always implied by the context and the estimator.
The action of the function can be describe with the following words: if the true value of the statistical parameter is $\boldsymbol{\theta}$, then the expected error of the estimator $\widehat{\boldsymbol{\theta}}$ is $\text{Bias}\left(\widehat{\boldsymbol{\theta}}_m\right)$ (since the data are drawn from the distribution $P_{\boldsymbol{\theta}}$).

An estimator is called unbiased if its bias has zero constant value: $\text{Bias}\left(\widehat{\boldsymbol{\theta}}_m\right) = 0$, which means that on average the estimator returns the correct value: explicitly, we always have $\mathbb{E}_{\boldsymbol{\theta}}\left[\widehat{\boldsymbol{\theta}}_m\right] = \boldsymbol{\theta}$ for every $\boldsymbol{\theta}$. Conversely, a biased estimator has bias different from zero.
We call an estimator asymptotically unbiased when $\lim\limits_{m\to\infty} \text{Bias}\left(\widehat{\boldsymbol{\theta}}_m\right) = 0$, or equivalently in terms of the expectation $\lim\limits_{m\to\infty} \mathbb{E}_{\boldsymbol{\theta}}\left[\widehat{\boldsymbol{\theta}}_m\right] = \boldsymbol{\theta}$.

Example A.2. *If all elements in the data are i.i.d, then the empirical mean defined in A.1 is an unbiased estimator:*

$$\mathbb{E}\left[\widehat{\boldsymbol{\mu}}_m\right] = \frac{1}{m}\sum_{k=1}^{m}\mathbb{E}[\mathbf{x}_k] = \mathbb{E}[\mathbf{x}] = \boldsymbol{\mu}$$

Variance

The estimator is a random variable or vector depending on the data, therefore it changes value when different samples are taken. In other words, different samples usually lead to different estimates even if they are all drawn by the same data generating distribution. This fact shows us that, even if the estimator is unbiased, we have another source of error when we compute an estimate with a specific sample.
If we want to quantify how much the estimator outcome can fluctuate using different data samples, then we can proceed similarly to how we did in the previous subsection. In fact, as we computed the expected value of the estimator, we can compute the variance to measure the variation in the estimate. The variance of an estimator is just the usual variance of a random variable, in this case, computed with respect to the distribution of the data.

$$\text{Var}\left(\widehat{\theta}_m\right) = \mathbb{E}\left[\left(\widehat{\theta}_m - \mathbb{E}\left[\widehat{\theta}_m\right]\right)^2\right]. \tag{A.3}$$

In the more general case where the estimator is a vector, we use the covariance matrix of the estimator:

$$\text{Cov}\left(\widehat{\boldsymbol{\theta}}_m\right) = \mathbb{E}\left[\left(\widehat{\boldsymbol{\theta}}_m - \mathbb{E}\left[\widehat{\boldsymbol{\theta}}_m\right]\right)\cdot\left(\widehat{\boldsymbol{\theta}}_m - \mathbb{E}\left[\widehat{\boldsymbol{\theta}}_m\right]\right)^{\top}\right]. \tag{A.4}$$

Remark A.1. *We recall that every entry of the covariance matrix is the covariance between two component of the random vector, therefore*

$$\text{Cov}\left(\widehat{\boldsymbol{\theta}}_m\right)_{ij} = \text{Cov}\left(\widehat{\theta}_{m,i}\,,\widehat{\theta}_{m,j}\right) = \mathbb{E}\left[\left(\widehat{\theta}_{m,i} - \mathbb{E}\left[\widehat{\theta}_{m,i}\right]\right)\left(\widehat{\theta}_{m,j} - \mathbb{E}\left[\widehat{\theta}_{m,j}\right]\right)\right]. \tag{A.5}$$

Example A.3. *Let* $\boldsymbol{\Sigma} = \mathbb{E}\big[(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^\top\big]$ *be the covariance matrix of an element in the data. We compute the estimator variance of the empirical mean to continue the previous examples: if all elements are i.i.d., we get*

$$\begin{aligned}
\mathrm{Cov}\big(\widehat{\boldsymbol{\mu}}_m\big)_{ij} &= \mathbb{E}\big[\big(\widehat{\mu}_{m,i} - \mathbb{E}[\widehat{\mu}_{m,i}]\big)\big(\widehat{\mu}_{m,j} - \mathbb{E}[\widehat{\mu}_{m,j}]\big)\big] = \\
&= \mathbb{E}\left[\left(\frac{1}{m}\sum_{k=1}^{m} \mathrm{x}_{k,i} - \mu_i\right)\left(\frac{1}{m}\sum_{h=1}^{m} \mathrm{x}_{h,j} - \mu_j\right)\right] = \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{k=1}^{m} (\mathrm{x}_{k,i} - \mu_i)\frac{1}{m}\sum_{h=1}^{m} (\mathrm{x}_{h,j} - \mu_j)\right] = \\
&= \frac{1}{m^2}\mathbb{E}\left[\left(\sum_{k=1}^{m} (\mathrm{x}_{k,i} - \mu_i)\right)\left(\sum_{h=1}^{m} (\mathrm{x}_{h,j} - \mu_j)\right)\right] = \\
&= \frac{1}{m^2}\mathbb{E}\left[\sum_{k=1}^{m} (\mathrm{x}_{k,i} - \mu_i)(\mathrm{x}_{k,j} - \mu_j) + \sum_{k\neq h} (\mathrm{x}_{k,j} - \mu_j)(\mathrm{x}_{h,j} - \mu_j)\right] = \\
&= \frac{1}{m^2} m\, \mathbb{E}\big[(\mathrm{x}_i - \mu_i)(\mathrm{x}_j - \mu_j)\big] = \frac{\Sigma_{ij}}{m}.
\end{aligned} \tag{A.6}$$

The first equality is given by the fact that $\widehat{\boldsymbol{\mu}}_m$ *is unbiased, then every component* $\widehat{\mu}_{m,i}$ *is unbiased with mean* μ_i*. The penultimate equality, instead, comes from the i.i.d. property, in particular: since all samples' components are identically distributed we get*

$$\mathbb{E}\left[\sum_{k=1}^{m} (\mathrm{x}_{k,i} - \mu_i)(\mathrm{x}_{k,j} - \mu_j)\right] = m\,\mathbb{E}\left[(\mathrm{x}_i - \mu_i)(\mathrm{x}_j - \mu_j)\right]$$

and since all samples are independent we have

$$\begin{aligned}
\mathbb{E}\left[\sum_{k\neq h} (\mathrm{x}_{k,j} - \mu_j)(\mathrm{x}_{h,j} - \mu_j)\right] &= \sum_{k\neq h} \mathbb{E}\left[(\mathrm{x}_{k,j} - \mu_j)(\mathrm{x}_{h,j} - \mu_j)\right] = \\
&= \sum_{k\neq h} \mathbb{E}\left[\mathrm{x}_{k,j} - \mu_j\right]\mathbb{E}\left[\mathrm{x}_{h,j} - \mu_j\right] = 0.
\end{aligned}$$

With the matrix notation we write:

$$\mathrm{Cov}\big(\widehat{\boldsymbol{\mu}}_m\big) = \frac{1}{m}\boldsymbol{\Sigma}. \tag{A.7}$$

If we are in the case where the observations are scalar with variance $\sigma^2 = \mathbb{E}\big[(\mathrm{x}-\mu)^2\big]$*, then we simply have* $\mathrm{Var}\big(\widehat{\mu}_m\big) = \frac{\sigma^2}{m}$.

Mean Squared Error of an Estimator

Now that we have defined the bias and the variance of an estimator, we can put together these two source of error in the Mean Squared Error (MSE). This is useful to decide which is the best estimator: suppose we have an unbiased estimator with high variance and a biased estimator with low variance, we want a quantitative method to select one of them. We can compare their

MSE and choose the one that returns the lowest. The MSE of an estimator is computed with the expected value of the squared norm of the error $e_m(\boldsymbol{\theta})$:

$$\mathrm{MSE}\left(\widehat{\boldsymbol{\theta}}_m\right) = \mathbb{E}\left[\left\|\widehat{\boldsymbol{\theta}}_m - \boldsymbol{\theta}\right\|^2\right]. \tag{A.8}$$

We can rewrite the formula to show the dependence on bias and variance of the estimator:

$$\begin{aligned}
&\mathrm{MSE}\left(\widehat{\boldsymbol{\theta}}_m\right) = \\
&\quad = \mathbb{E}\left[\left\|\widehat{\boldsymbol{\theta}}_m - \mathbb{E}\left[\widehat{\boldsymbol{\theta}}_m\right] + \mathbb{E}\left[\widehat{\boldsymbol{\theta}}_m\right] - \boldsymbol{\theta}\right\|^2\right] = \\
&\quad = \mathbb{E}\left[\left\|\widehat{\boldsymbol{\theta}}_m - \mathbb{E}\left[\widehat{\boldsymbol{\theta}}_m\right]\right\|^2 + \left\|\mathbb{E}\left[\widehat{\boldsymbol{\theta}}_m\right] - \boldsymbol{\theta}\right\|^2 + 2\left(\widehat{\boldsymbol{\theta}}_m - \mathbb{E}\left[\widehat{\boldsymbol{\theta}}_m\right]\right)^\top \left(\mathbb{E}\left[\widehat{\boldsymbol{\theta}}_m\right] - \boldsymbol{\theta}\right)\right] = \\
&\quad = \mathrm{tr}\left(\mathrm{Cov}\left(\widehat{\boldsymbol{\theta}}_m\right)\right) + \left\|\mathrm{Bias}\left(\widehat{\boldsymbol{\theta}}_m\right)\right\|^2 + 2\,\mathbb{E}\left[\left(\widehat{\boldsymbol{\theta}}_m - \mathbb{E}\left[\widehat{\boldsymbol{\theta}}_m\right]\right)\right]^\top \mathbb{E}\left[\left(\mathbb{E}\left[\widehat{\boldsymbol{\theta}}_m\right] - \boldsymbol{\theta}\right)\right] = \\
&\quad = \mathrm{tr}\left(\mathrm{Cov}\left(\widehat{\boldsymbol{\theta}}_m\right)\right) + \left\|\mathrm{Bias}\left(\widehat{\boldsymbol{\theta}}_m\right)\right\|^2
\end{aligned} \tag{A.9}$$

where the last equality is obtained by $\mathbb{E}\left[\left(\widehat{\boldsymbol{\theta}}_m - \mathbb{E}\left[\widehat{\boldsymbol{\theta}}_m\right]\right)\right] = \boldsymbol{0}$.

With this new formulation we clearly see that the best ideal estimator is unbiased and has low variance, however it's very difficult to find such an estimator. Usually, in fact, bias and variance are related and we aren't able to improve both. For instance, in the Machine Learning field they are both linked to the capacity and size of the model. Therefore we have to manage a trade-off between the two: for example, we often prefer a biased estimator to an unbiased one if it has very low variance and the bias is small enough.

Standard error

Another quantity with the same aim of the variance is the standard error. It's another way to measure the oscillation of an estimator, in the scalar case in fact, it's just the square root of the variance

$$\mathrm{SE}\left(\widehat{\theta}_m\right) = \sqrt{\mathrm{Var}\left(\widehat{\theta}_m\right)}. \tag{A.10}$$

It can be extended to the vectorial case defining the square root of a matrix:

Definition A.1. *Let $\boldsymbol{A}$ be a symmetric and positive semidefinite matrix. The square root of $\boldsymbol{A}$, denoted with $\boldsymbol{A}^{\frac{1}{2}}$, is the unique symmetric and positive semidefinite matrix $\boldsymbol{B}$ such that $\boldsymbol{A} = \boldsymbol{B}\boldsymbol{B}^\top = \boldsymbol{B}\boldsymbol{B}$.*

Since the covariance matrix is a positive semidefinite matrix then we can write the more general definition

$$\mathrm{SE}\left(\widehat{\boldsymbol{\theta}}_m\right) = \mathrm{Cov}\left(\widehat{\boldsymbol{\theta}}_m\right)^{\frac{1}{2}}. \tag{A.11}$$

Example A.4. *Proceeding with the previous examples, we can use the formula A.7 to compute the standard error of the sample mean:*

$$\mathrm{SE}(\widehat{\boldsymbol{\mu}}_m) = \mathrm{Cov}(\widehat{\boldsymbol{\mu}}_m)^{\frac{1}{2}} = \frac{1}{\sqrt{m}}\boldsymbol{\Sigma}^{\frac{1}{2}}. \tag{A.12}$$

Or just $\mathrm{SE}\big(\widehat{\mu}_m\big) = \frac{\sigma}{\sqrt{m}}$ *in the scalar case, where* σ *is the standard deviation of an element in the data.*

This quantity is important even if it's equivalent to the variance in its purpose. The variance, in fact, measures the oscillations exploiting the square of the estimator. The standard error, instead, brings the quantification of the oscillations back to their real order of magnitude since it undoes the power with the square root. Therefore the standard error gives, in some sense, a more correct quantitative measure of the expected error.
Moreover the standard error of the sample mean is directly used in confidence interval. With the estimator's outcome we obtain an estimate of the unknown parameter, but there exists a method to know if the true value is near the computed estimate. In order to do so, we fix a probability p and we search for an interval centered in the estimate value such that the true parameter is in that interval with probability $(1-p)$. In the case of a scalar sample mean, we can easily find this confidence interval using the Central Limit Theorem (CLT): $\frac{\sqrt{m}}{\sigma}\big(\widehat{\mu}_m - \mu\big) \to \mathcal{N}(0,1)$. Let $c = \Phi^{-1}\big(1-\frac{p}{2}\big)$ where $\Phi(x)$ is the cumulative distribution function of the standard Gaussian distribution $\mathcal{N}(0,1)$. Then, for m large enough we get a good approximation with the following probabilities:

$$\begin{aligned} p = P\big(-c \leq \mathcal{N}(0,1) \leq c\big) &\approx P\left(-c \leq \frac{\sqrt{m}}{\sigma}\big(\widehat{\mu}_m - \mu\big) \leq c\right) = \\ &= P\left(-c\frac{\sigma}{\sqrt{m}} \leq \widehat{\mu}_m - \mu \leq c\frac{\sigma}{\sqrt{m}}\right) = \\ &= P\left(\widehat{\mu}_m - c\frac{\sigma}{\sqrt{m}} \leq \mu \leq \widehat{\mu}_m + c\frac{\sigma}{\sqrt{m}}\right) = \\ &= P\Big(\widehat{\mu}_m - c\,\mathrm{SE}\big(\widehat{\mu}_m\big) \leq \mu \leq \widehat{\mu}_m + c\,\mathrm{SE}\big(\widehat{\mu}_m\big)\Big) \end{aligned}$$

which means that the confidence interval of probability $(1-p)$ is

$$\big(\widehat{\mu}_m - c\,\mathrm{SE}\big(\widehat{\mu}_m\big)\,,\, \widehat{\mu}_m + c\,\mathrm{SE}\big(\widehat{\mu}_m\big)\big).$$

Consistency

We have tried to quantify if an estimator is good or not with a fixed amount m of available observations. However, another very important property of an estimator is whether it tends or not to the true value if the number of available observations goes to infinity. It's a reasonable property to ask, since we would like that at least with an hypothetical infinite amount of samples we get the true value or that the estimator approaches the true value increasing m. This property is called consistency and we formally define it by making explicit the convergence we want:

Definition A.2. *An estimator* $\widehat{\boldsymbol{\theta}}_m$ *is consistence if it convergences in probability to the true value* $\boldsymbol{\theta}$ *as* $m \to \infty$.
Moreover, it is strongly consistence if it converges almost surely to $\boldsymbol{\theta}$.

Remark A.2. *We recall that a sequence of random variables or vectors* $(\mathbf{z}_m)_{m\in\mathbb{N}}$ *convergences in probability to* $\mathbf{z}$ *if for every* $\epsilon > 0$ *we have* $P(\|\mathbf{z}_m - \mathbf{z}\| > \epsilon) \to 0$ *as* $m \to \infty$.
While it converges almost surely to $\mathbf{z}$ *when* $P\left(\lim_{m\to\infty} \mathbf{z}_m = \mathbf{z}\right) = 1$*: we have pointwise convergence to* $\mathbf{z}$ *almost everywhere.*

The only property we analysed in this chapter that was connected to the estimators behaviour as $m \to \infty$ was the asymptotically unbiasedness. We show in two propositions how these two properties are related in the scalar case:

Proposition A.1. *If the sequence of estimators $\widehat{\theta}_m$ has a uniform bound on the second moments, then consistency implies asymptotically unbiasedness.*

Proof. Let $c \geq 0$ be the bound of the second moments: $\mathbb{E}\left[\widehat{\theta}_m{}^2\right] \leq c$ for every $m \in \mathbb{N}$. We can see the estimators $\widehat{\theta}_m$ as elements of a function space $L^2(\Omega, P_\theta)$, then we know by the Minkowski inequality that the second moment is a norm. In other words we can use the triangular inequality to get

$$\left\|\widehat{\theta}_m - \theta\right\|_2 \leq \left\|\widehat{\theta}_m\right\|_2 + \|\theta\|_2 \,,$$

or with the explicit expectation

$$\mathbb{E}\left[\left|\widehat{\theta}_m - \theta\right|^2\right] \leq \mathbb{E}\left[\widehat{\theta}_m{}^2\right] + \mathbb{E}[\theta^2] \leq c + \theta^2 =: C.$$

With the previous inequality, we can show the convergence in absolute value: for every $\epsilon > 0$ we have

$$\begin{aligned}\mathbb{E}\left[\left|\widehat{\theta}_m - \theta\right|\right] &= \mathbb{E}\left[\left|\widehat{\theta}_m - \theta\right|\mathbb{1}_{\left|\widehat{\theta}_m - \theta\right| \leq \epsilon} + \left|\widehat{\theta}_m - \theta\right|\mathbb{1}_{\left|\widehat{\theta}_m - \theta\right| > \epsilon}\right] \leq \\ &\leq \epsilon P\left(\left|\widehat{\theta}_m - \theta\right| \leq \epsilon\right) + \mathbb{E}\left[\left|\widehat{\theta}_m - \theta\right|\mathbb{1}_{\left|\widehat{\theta}_m - \theta\right| > \epsilon}\right] \leq \\ &\leq \epsilon + \sqrt{\mathbb{E}\left[\left|\widehat{\theta}_m - \theta\right|^2\right]\mathbb{E}\left[\mathbb{1}_{\left|\widehat{\theta}_m - \theta\right| > \epsilon}{}^2\right]} \leq \\ &\leq \epsilon + \sqrt{C}\sqrt{P\left(\left|\widehat{\theta}_m - \theta\right| > \epsilon\right)}\end{aligned}$$

where we use the Cauchy-Schwartz inequality in the third step.
Looking at the limit as $m \to \infty$ we see that

$$\lim_{m\to\infty} \mathbb{E}\left[\left|\widehat{\theta}_m - \theta\right|\right] \leq \epsilon + \sqrt{C}\cancel{\lim_{m\to\infty}\sqrt{P\left(\left|\widehat{\theta}_m - \theta\right| > \epsilon\right)}} = \epsilon.$$

which means that $\lim\limits_{m\to 0} \mathbb{E}\left[\left|\widehat{\theta}_m - \theta\right|\right] \leq \epsilon$ for every $\epsilon > 0$.
Since the $\epsilon > 0$ can be arbitrary small, we get the desired convergence with the limit as $\epsilon \to 0$:

$$0 \leq \lim_{m\to 0} \mathbb{E}\left[\left|\widehat{\theta}_m - \theta\right|\right] \leq \lim_{\epsilon\to 0} \epsilon = 0 \quad \Longrightarrow \quad \lim_{m\to 0} \mathbb{E}\left[\left|\widehat{\theta}_m - \theta\right|\right] = 0.$$

Knowing that $\left|\mathbb{E}\left[\widehat{\theta}_m - \theta\right]\right| \leq \mathbb{E}\left[\left|\widehat{\theta}_m - \theta\right|\right]$, we can conclude by the convergence in absolute value:

$$\lim_{m\to\infty} \mathbb{E}\left[\widehat{\theta}_m\right] - \theta = 0.$$

□

With a similar additional hypothesis we can obtain also the inverse:

Proposition A.2. *If the sequence of estimators $\widehat{\theta}_m$ has a vanishing variance, then asymptotically unbiasedness implies consistency.*

Proof. The result is directly obtained by the Markov's inequality: fix $\epsilon > 0$,

$$P\left(\left|\widehat{\theta}_m - \theta\right| > \epsilon\right) = P\left(\left|\widehat{\theta}_m - \theta\right|^2 > \epsilon^2\right) \leq$$

$$\leq \frac{\mathbb{E}\left[\left(\widehat{\theta}_m - \theta\right)^2\right]}{\epsilon^2} = \frac{\text{Bias}\left(\widehat{\theta}_m\right)^2 + \text{Var}\left(\widehat{\theta}_m\right)}{\epsilon^2}.$$

Since both $\lim\limits_{m\to\infty} \text{Bias}\left(\widehat{\theta}_m\right)$ and $\lim\limits_{m\to\infty} \text{Var}\left(\widehat{\theta}_m\right)$ are equal to 0 by hypothesis, then we easily obtain the thesis:

$$\lim_{m\to\infty} P\left(\left|\widehat{\theta}_m - \theta\right| > \epsilon\right) = 0.$$

□

Remark A.3. *Notice that in both proposition we need an additional hypothesis concerning the squared estimator. Neither consistency nor asymptotically unbiasedness alone can imply the other one, then neither of them is a stronger condition than the other. In fact, they deal with different aspects of the asymptotic behaviour as $m \to \infty$.*

Bibliography

[1] Christopher M. Bishop. *Pattern recognition and machine learning.* Information science and statistics. New York: Springer, 2006.

[2] Alan J. Bray and David S. Dean. "Statistics of critical points of Gaussian fields on large-dimensional spaces". In: *Physical Review Letters* 98 (2007), p. 150201.

[3] Eduardo R. Caianiello. "Outline of a theory of thought-processes and thinking machines". In: *Journal of Theoretical Biology* 1.2 (1961), pp. 204–235.

[4] Tom Carter. *An Introduction to Information Theory and Entropy.* California State University Stanislaus, 2014.

[5] Xiangning Chen et al. *Symbolic Discovery of Optimization Algorithms.* 2023.

[6] Yann N. Dauphin et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: NIPS'14 (2014), 2933–2941.

[7] Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning.* Cambridge University Press, 2020.

[8] Frederik Dekking et al. *A Modern Introduction to Probability and Statistics: Understanding Why and How.* Springer Texts In Statics. Springer, 2005.

[9] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159.

[10] Alexandre Défossez et al. *A Simple Convergence Proof of Adam and Adagrad.* 2022. arXiv: 2003.02395.

[11] Lawrence C. Evans. *Partial Differential Equations Second Edition.* Graduate Studies in Mathematics ; v.19. Providence: American Mathematical Society, 2010.

[12] Saeed Ghadimi and Guanghui Lan. *Stochastic First- and Zeroth-order Methods for Non-convex Stochastic Programming.* 2013. arXiv: 1309.5549.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016.

[14] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer series in statistics. New York: Springer, 2009.

[15] Haowei He, Gao Huang, and Yang Yuan. *Asymmetric Valleys: Beyond Sharp and Flat Local Minima.* 2019. arXiv: 1902.00744.

[16] Reinaldo Hernández and Sebastian Engell. “Stochastic Approximation in Online Steady State Optimization Under Noisy Measurements”. In: Computer Aided Chemical Engineering 40 (2017), pp. 1747–1752.

[17] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. “A Fast Learning Algorithm for Deep Belief Nets”. In: *Neural Computation* 18 (2006), pp. 1527–1554.

[18] Robert A. Jacobs. “Increased rates of convergence through learning rate adaptation”. In: *Neural Networks* 1.4 (1988), pp. 295–307.

[19] Tianchong Jiang. *Wigner Semicircle Law for Gaussian Random Matrices.* Chicago University, 2021.

[20] Nitish Shirish Keskar et al. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima.* 2017. arXiv: `1609.04836`.

[21] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2014. arXiv: `1412.6980`.

[22] Solomon Kullback and Richard A. Leibler. “On Information and Sufficiency”. In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86.

[23] Luc Laperriere and Gunther Reinhart. *CIRP Encyclopedia of Production Engineering.* Berlin: Springer, 2014.

[24] Erich L. Lehmann and George Casella. *Theory of Point Estimation.* New York, NY, USA: Springer, 1998.

[25] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *Bulletin of Mathematical Biophysics* 5 (1943), 115–133.

[26] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* Cambridge, MA, USA: MIT Press, 1969.

[27] Tom M. Mitchell. *Machine learning.* McGraw Hill series in computer science. New York: McGraw-Hill, 1997.

[28] Berndt Müller, Joachim Reinhardt, and Michael T. Strickland. *Neural Networks: An Introduction.* Springer, 1995.

[29] Yurii Nesterov. “A method for solving the convex programming problem with convergence rate $O(1/k^2)$”. In: *Proceedings of the USSR Academy of Sciences* 269 (1983), pp. 543–547.

[30] Whitney K. Newey and Daniel McFadden. “Chapter 36: Large sample estimation and hypothesis testing”. In: vol. 4. Handbook of Econometrics. Elsevier, 1994, pp. 2111–2245.

[31] Micheal A. Nielsen. *Neural Networks and Deep Learning.* 2015.

[32] Duc T. Pham and Xing Liu. *Neural Networks for Identification, Prediction and Control.* Springer, 1995.

[33] Allan Pinkus. “Approximation Theory of the MLP model in Neural Networks”. In: *Acta Numerica* 8 (1999), 143–195.

[34] Boris T. Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17.

[35] Frank Rosenblatt. “The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain”. In: *Psychological Review* 65.6 (1958), pp. 386–408.

[36] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning Internal Representations by Error Propagation”. In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* 1 (1986). Ed. by David E. Rumelhart and James L. Mcclelland, pp. 318–362.

[37] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[38] Claude E. Shannon. “A Mathematical Theory of Communication”. In: *The Bell System Technical Journal* 27 (1948), pp. 379–423.

[39] Umut Simsekli, Levent Sagun, and Mert Gurbuzbalaban. *A Tail-Index Analysis of Stochastic Gradient Noise in Deep Neural Networks*. 2019. arXiv: `1901.06053`.

[40] Ilya Sutskever et al. “On the importance of initialization and momentum in deep learning”. In: Proceedings of Machine Learning Research 28.3 (2013), pp. 1139–1147.

[41] Graham Upton and Ian Cook. *A Dictionary of Statistics*. Oxford Paperback Reference. Oxford University Press, 2008.

[42] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Statistics for engineering and information science. New York: Springer, 2000.

[43] Jeremy Watt, Reza Borhani, and Aggelos Katsaggelos. *Machine Learning Refined: Foundations, Algorithms, and Applications*. Cambridge University Press, 2016.

[44] D.Randall Wilson and Tony R. Martinez. “The general inefficiency of batch training for gradient descent learning”. In: *Neural Networks* 16.10 (2003), pp. 1429–1451.

[45] Pan Zhou et al. *Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning*. 2021. arXiv: `2010.05627`.