Deep Learning from a Mathematical Point of View

Carmen Mayora Cebollero

Trabajo de Fin de Máster
Universidad de Zaragoza

Directores del trabajo:
Roberto Barrio Gil
Rubén Vigara Benito

Junio 2021

Prologue

Artificial Neural Networks are a Machine Learning algorithm based on the structure of biological neurons (these neurons are organized in layers). Deep Learning is the branch of Machine Learning that includes all the techniques used to build Deep Neural Networks (Artificial Neural Networks with at least two hidden layers) that are able to learn from data with several levels of abstraction.

A Feed-forward Neural Network with fully-connected layers is a Deep Neural Network whose information flow goes forwards. The neurons that belong to consecutive layers are fully-connected. Its architecture is based on the weights of the connections between the neurons and on the bias that each neuron adds to its received information. The value of these parameters is fitted during training. This learning process is reduced to an optimization problem that can be solved using Gradient Descent or other recent algorithms as Scheduled Restart Stochastic Gradient Descent, being Back Propagation the algorithm used to compute the required derivatives. If the network is not able to learn correctly, overfitting or underfitting can arise. Other parameters of the neural network (hyperparameters) are not tuned during training. To perform, for example, image classification or prediction tasks we have to use other types of Deep Neural Networks as Convolutional Neural Networks or Recurrent Neural Networks.

This Master's Dissertation is organized into a summary (written in Spanish), three chapters, some conclusions and a bibliography.

In Chapter 1 we define the concepts of Artificial Intelligence and Machine Learning in order to introduce Artificial Neural Networks (we prove some results related to their capabilities).

In Chapter 2 we introduce the concept of Deep Learning. We explain the architecture of Feed-forward Neural Networks with fully-connected layers and we study some topics related to them (we prove some mathematical results).

In Chapter 3 we study Convolutional Neural Networks and Recurrent Neural Networks.

Resumen

La inteligencia artificial es el campo de la ciencia computacional que se centra en la construcción de máquinas que son capaces de aprender, razonar y actuar como seres humanos inteligentes. Una de sus principales ramas a día de hoy es el aprendizaje automático o *Machine Learning*, que se encarga de desarrollar algoritmos que sean capaces de aprender a partir de un conjunto de datos. Teniendo en cuenta si es necesaria la supervisión del programador durante el proceso de aprendizaje, podemos dividir los algoritmos de aprendizaje automático en dos grandes grupos: aprendizaje supervisado y aprendizaje no supervisado.

Las redes neuronales artificiales (*Artificial Neural Networks*) son un algoritmo de aprendizaje automático que está inspirado en las neuronas biológicas (dichas neuronas se organizan en capas). Vamos a centrarnos en las redes neuronales artificiales que pertenecen al grupo de aprendizaje supervisado. Las *Feed-forward Neural Networks* son aquellas en las que el flujo de información va hacia delante. El perceptrón es una red neuronal artificial de este tipo y está formado por una capa de entrada conectada con una capa de salida constituida por neuronas artificiales conocidas como TLUs (siglas en inglés de *Threshold Logic Unit*). Uno de los principales problemas de esta red neuronal artificial es que es incapaz de resolver, por ejemplo, el problema de clasificación XOR.

El perceptrón multicapa (*Multi-Layer Perceptron*) es una *Feed-forward Neural Network* formada por una capa de entrada, una o varias capas ocultas y una capa de salida, estando todas ellas totalmente conectadas. Esta red neuronal sí es capaz de resolver el problema de clasificación XOR. Además, si el problema que queremos solucionar puede describirse con una función continua, puede demostrarse que existe un perceptrón multicapa con unas ciertas características que se aproxima a dicha función. Es decir, existe una red neuronal con la que podemos resolver el problema.

El aprendizaje profundo o *Deep Learning* es la rama del aprendizaje automático que incluye todas las técnicas utilizadas para construir redes neuronales profundas (*Deep Neural Networks*) capaces de aprender de datos reales con diferentes niveles de abstracción. Por red neuronal profunda entendemos cualquier red neuronal que tenga al menos dos capas ocultas.

Dentro de las redes neuronales profundas se encuentran las *Feed-forward Neural Networks* que tienen sus capas totalmente conectadas (*Feed-forward Neural Networks with fully-connected layers*). Estas redes neuronales son perceptrones multicapa con al menos dos capas ocultas. Las conexiones entre neuronas tienen asociado un peso, añadiendo cada neurona un sesgo a la información recibida y aplicando una función de activación. Estos pesos y sesgos son parámetros ajustados durante el proceso de aprendizaje o entrenamiento (*training*). Dicho proceso se reduce a un problema de optimización que consiste en encontrar valores para los pesos y los sesgos que minimicen la función de pérdida (*loss function*), que nos permite saber cómo de bien se ajusta la red a los datos. Para resolver este problema podemos utilizar el método de *Gradient Descent* o el de *Stochastic Gradient*. Sin embargo, ambos tienen problemas de convergencia, por lo que se han desarrollado nuevos algoritmos de optimización, siendo uno de los más recientes el *Scheduled Restart Stochastic Gradient Descent*. Para calcular el gradiente que aparece en estos métodos de optimización se utiliza el algoritmo de retropropagación o *Back Propagation*, que explota la estructura hacia delante de la red para poder calcular las derivadas necesarias hacia atrás. A veces, durante el proceso de entrenamiento la red puede sufrir problemas de

aprendizaje: puede no ser capaz de generalizar (sobreajuste u *overfitting*) o tal vez no aprende de los datos (subajuste o *underfitting*). Aquellos parámetros que describen la red pero no se ajustan durante el entrenamiento, como el número de capas ocultas o el valor inicial de los pesos y sesgos, se conocen como hiperparámetros.

El uso de *Feed-forward Neural Networks* con todas sus capas totalmente conectadas no es factible cuando los datos de entrada son imágenes (esto es debido a que la capa de entrada tendría tantas neuronas como píxeles tiene la imagen a estudiar y considerando únicamente las conexiones entre esta capa y la primera capa oculta, ya tendríamos demasiados pesos para entrenar). Las redes neuronales convolucionales (*Convolutional Neural Networks*) son redes neuronales profundas utilizadas para tareas de clasificación y de reconocimiento de imágenes. Aunque son *Feed-forward Neural Networks*, sus capas no están totalmente conectadas (esto reducirá el número de pesos a entrenar). Su estructura se basa en capas convolucionales, mapas de características y *pooling layers*.

Si debemos resolver problemas de predicción en los que los datos de entrada son series temporales, necesitamos que la red neuronal artificial utilizada tenga memoria y por tanto, no podemos utilizar *Feed-forward Neural Networks* como las anteriores. Las redes neuronales recurrentes (*Recurrent Neural Networks*) son redes neuronales profundas con memoria. Como datos de entrada utilizan la información observable y los estados previos. Su estructura principal, donde se guardan dichos estados, es conocida como *memory cell*. Sin embargo, existe evidencia empírica de que estas redes neuronales recurrentes no tienen memoria a largo plazo. Para resolver este problema se utilizan *memory cells* con memoria a largo plazo como la *LSTM cell* o la *GRU cell*.

Contents

Chapter 1

Introduction to AI and ML

Before explaining Deep Learning, which is the main topic of this Master's Dissertation, we have to introduce some concepts as Artificial Intelligence and Machine Learning to contextualize it.

1.1 Artificial Intelligence

This short introduction to Artificial Intelligence is based on [1], [2] and [3].

Artificial Intelligence (AI) is the field of computer science focused on building machines that can learn, reason and act as intelligent humans.

The Turing Test was created to decide if a machine shows intelligent behaviour or not. This test was introduced by A. Turing in 1950 in his work *Computing Machinery and Intelligence* [4]. It is based on the simple idea that if a machine behaves intelligently, then it is. To carry out the test we need two human people, person A and person B, and the computer we want to test. Person A is in one room and person B and the computer are in a different room. Person A is the judge, who holds a conversation with the computer and with person B. The communication between them is only written (the capacity of the machine to convert words into audio is not tested) and person A does not know to whom he is talking to at each moment. If the judge is unable to distinguish correctly between the answers given by person B and the ones given by the computer, then it is concluded that the machine is intelligent.

The Turing Test received a lot of criticism. The main critique was The Chinese Room Argument by J. Searle. Let us explain the scenario of this argument. An English person A who does not understand Chinese is locked in a room and he receives a paper with a question written in Chinese by a person B who is outside the room. Person A has an auxiliary element that allows him to translate this question to English. Person A thinks the answer in English and with the auxiliary element he is able to write it in Chinese. Person B, that is outside the room, receives the answer. As it is written in Chinese, person B may think that the subject inside the room can speak Chinese, but this is not true. In some way, person A is misleading person B. This argument can be summarized in the following question: *Can the machine that passes the Turing Test trick us in that way?*

According to the *MIT Technology Review* [1] the main topics in AI right now are:

- Face recognition.
- Machine Learning.
- Robots.
- Voice assistants.

1.2 Machine Learning

Some information about Machine Learning can be consulted in [2], [5], [6], [7], [8], [9] and [10].

Machine Learning (ML) is the branch of Artificial Intelligence focused on programming systems that have to be able to learn from data. The term *Machine Learning* was coined by A. Samuel in 1959. He defined it as the *field of study that gives computers the ability to learn without being explicitly programmed.*

Machine Learning algorithms can help us to solve problems that would be unsolvable if they were solved in a traditional way (that is, writing all the code needed to find the solution). In addition, they can find patterns in data that in another way would be very complicated or even impossible to detect (this application of ML is known as data mining).

Example 1.1. An example of a ML application that we can find in everyday life is the spam filter. When an e-mail is received, the spam filter decides if it is spam or not. In this case, the data it has learned from to be able to do this classification (spam or nonspam) is a set of previous e-mails that were classified as spam or nonspam by the user.

Training set is the name given to the data that ML algorithms use to learn. Each of the elements of the set is called sample or training instance. The process of learning is called training. It can be easily understood with a quote from T. Mitchell (1997): *A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.*

Example 1.2. In the case of the spam filter given in Example 1.1, we can identify that the task T is to classify a new e-mail as spam or nonspam and the experience E is the training set we have defined. The performance measure P is the percentage of e-mails that are correctly classified.

1.2.1 Types of Machine Learning

Taking into account if human supervision is needed or not during training, ML algorithms can be classified into two big groups:

→ Supervised learning.

Human supervision is needed during training. We use labeled data: each instance of the training set is associated to a label that corresponds to the value we should obtain when the task is performed. The available data is divided into the training set and the test set. This last set is used to check how well the ML algorithm generalizes. One of the main problems that ML can present is overfitting or loss of generalization: the algorithm fits the training set but it does not work well for other data.

Supervised learning is used to perform classification tasks (as in Example 1.1) and prediction tasks (for example, to predict the price of a car given some features).

Some of the most famous supervised learning algorithms are Decision Trees (DTs) and Artificial Neural Networks (ANNs).

→ Unsupervised learning.

Human supervision is not needed during training. This means that the training instances do not have a label. The algorithm has to be able to extract features of the data without help.

Unsupervised learning algorithms as k-Means are used for clustering (for instance, it can be used to classify the clients of a company in k different groups to create specific ads for each group) and others as Principal Component Analysis (PCA) are utilized for dimensionality reduction. Some Artificial Neural Networks as autoencoders (they are used, for example, to remove the noise of an image) are unsupervised learning algorithms.

This classification may include another types of ML algorithms as

→ Semi-supervised learning.

Not all the training instances are labeled. This type of algorithms can be useful when we have unlabeled data and it is laborious to label all the samples.

→ Reinforcement Learning (RL).

The system is called agent and it knows which is the best to do in each situation (policy) by learning from the environment and past actions from which it has got rewards or penalties.

1.3 Artificial Neural Networks

In this section we introduce Artificial Neural Networks in order to contextualize Deep Neural Networks. For more infomation, [5], [11], [12] and [13] can be consulted.

Artificial Neural Networks (ANNs) are a Machine Learning algorithm inspired by biological neurons. In 1943 the neurophysiologist W. McCulloch and the mathematician W. Pitts presented the first ANN in their work *A Logical Calculus of the Ideas Immanent in Nervous Activity* [14].

In this work we focus on ANNs that need supervision during training (supervised learning).

1.3.1 The Structure of a Biological Neuron

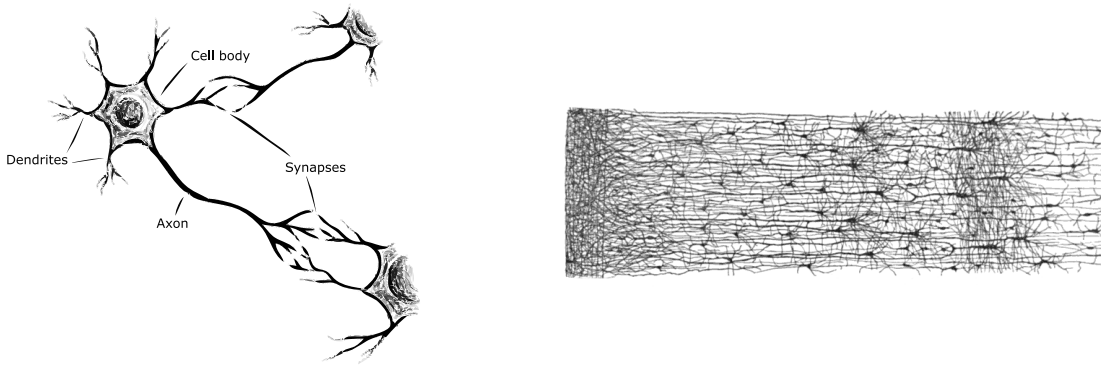

Figure 1.1: On the left, principal structure of a biological neuron. On the right, drawing of a cortical lamination by S. Ramón y Cajal (this image is reproduced from [15] and it has been adapted following its style in [5]).

A biological neuron is a cell that can be found in the brain. It has four principal parts:

- Cell body. It contains the nucleus of the neuron.
- Axon. It is a very long extension which transmits information from this neuron to the other neurons to which it is connected. Near the end, it divides into many branches called telodendria.
- Dendrites. They are branching extensions of the neuron that transmit the information received from the axon (telodendria) of the other neurons.
- Synapses. They are the regions that connect the axon (telodendria) with the dendrites. The electrical impulses (signals) pass from one neuron to another through them.

In the left image of Figure 1.1 we can see the representation of a neuron.

Biological Neural Networks (BNNs) are formed by billions of neurons and according to research, these neurons seem to be organized in consecutive layers. In the right image of Figure 1.1 we can see a drawing of a cortical lamination by S. Ramón y Cajal in which this structure of consecutive layers can be appreciated. Taking into account the structure of a BNN, an ANN can be designed. Let us start with one of the simplest ANNs, the Perceptron.

1.3.2 The Perceptron

The Perceptron is an ANN created by F. Rosenblatt in 1957. It is based on an artificial neuron known as Threshold Logic Unit (TLU).

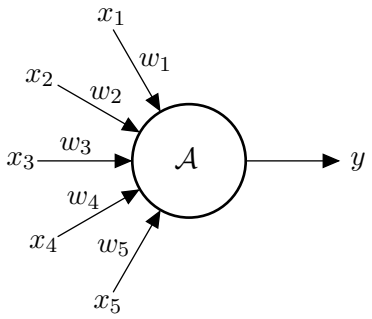

Figure 1.2: Graphic representation of a TLU with 5 inputs.

A TLU is a neuron that receives n different inputs (numerical values) that we can denote by x_i for $i = 1, ..., n$. A weight w_i $(i = 1, ..., n)$ is associated to the connection between the input x_i and the neuron. The output of the TLU is a numerical value y obtained from the sum of the weighted inputs and the application of an activation function $\mathcal{A}$. Therefore, the calculation of the output can be written mathematically as

$$y = \mathcal{A}\left(\sum_{i=1}^{n} w_i x_i\right).$$

In Figure 1.2 we have a diagram of the structure of a TLU with 5 inputs.

A Feed-forward Neural Network (FNN) is a neural network whose flow goes forwards, that is to say, the information is sent from the input layer to the output layer without forming cycles.

The Perceptron is an FNN that consists of an input layer with n input neurons (they represent the input data x_i, $i = 1, ..., n$) and an output layer with k TLUs, each of them receiving information from all the neurons at the input layer. The weight of the connection between input x_i and neuron j is denoted by w_{ji}. A bias term b_j $(j = 1, ..., k)$ is usually added to the linear combination of the inputs of each TLU. The output of the Perceptron is of the form

$$y_j = \mathcal{A}\left(\sum_{i=1}^{n} w_{ji} x_i + b_j\right); \quad j = 1, ..., k. \tag{1.1}$$

Notice that the Perceptron can be understood as a function that associates an output value to the input of the network. The activation function is a step function as the Heaviside function (its graph is on the right of Figure 1.3)

$$\mathcal{A}(x) = \mathrm{H}(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{if } x < 0. \end{cases}$$

Notice that it is quite natural to use this function because it can be interpreted as the behaviour of a neuron: value 1 when the neuron fires and value 0 if the neuron is inactive.

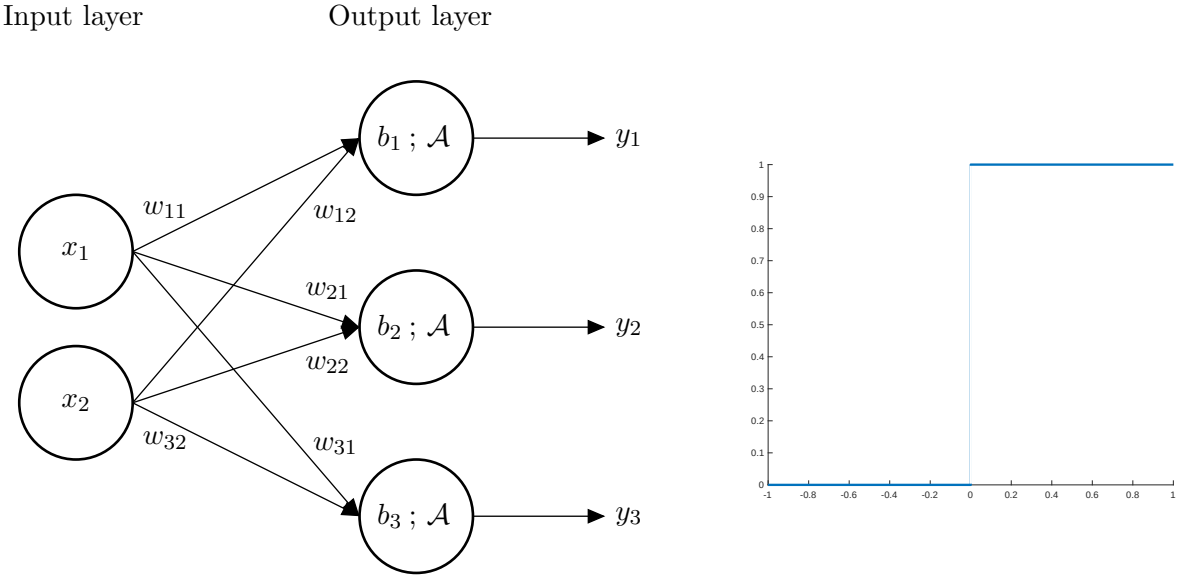

Figure 1.3: On the left, graphic representation of a Perceptron with $n = 2$ neurons in the input layer and $k = 3$ TLUs in the output layer. On the right, Heaviside function.

On the left of Figure 1.3 we can see a representation of a Perceptron with $n = 2$ neurons in the input layer and an output layer with $k = 3$ TLUs.

Remark 1.3. The term Perceptron is also used by some authors to refer to an ANN with a unique TLU.

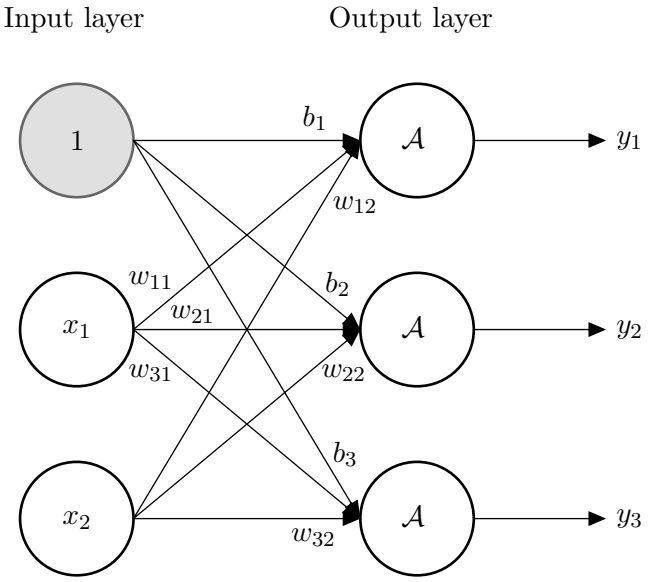

Figure 1.4: Graphic representation of the Perceptron in Figure 1.3 using a bias neuron.

Remark 1.4. Some authors use an extra neuron with value 1 in the input layer to represent the bias term. This neuron is called bias neuron. In Figure 1.4 we have the representation of the Perceptron in Figure 1.3 using a bias neuron (in grey).

To train the Perceptron, the training samples are used one by one. When the prediction $\hat{y}_j$ $(j = 1, ..., k)$ made by the Perceptron for a training sample does not coincide with the label y_j $(j = 1, ..., k)$, the connections that would have given the right solution are reinforced. This training process was proposed by F. Rosenblatt and it is based on Hebb's rule. Mathematically, the training algorithm of the Perceptron can be written as

$$w_{ji}(m+1) = w_{ji}(m) + \eta(y_j - \hat{y}_j)x_i,$$

where η is the learning rate and $w_{ji}(m)$ represents the weight of the connection between the input neuron i and the output neuron j when the m-th training sample is used.

The Perceptron is a relatively simple structure. However, it presents some problems. For example, it is incapable of solving the XOR classification problem. Let us prove it.

Lemma 1.5. [12] *The XOR classification problem cannot be solved using the Perceptron.*

Proof. In what follows, we identify 0 with false and 1 with true. The XOR function has two inputs. This function returns false when the inputs are both false or both true (that is to say, the inputs are $(0,0)$ or $(1,1)$), and true when the two inputs are different, that is to say, one is true and the other is false (the possibilities are $(0,1)$ and $(1,0)$).

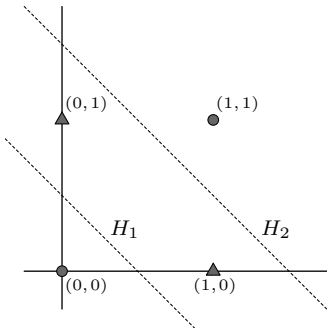

Figure 1.5: The XOR function is not linearly separable.

If we interpret geometrically how the Perceptron works (see (1.1)), we deduce that the input space has to be separated by a hyperplane. In our case we have four points in $\mathbb{R}^2$: $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$. Therefore, one of the half-spaces of the hyperplane has to contain all the points with which we obtain 0 with the XOR function (they are $(0,0)$ and $(1,1)$) and the other, all the points with which the solution is 1 (they are $(0,1)$ and $(1,0)$). As the XOR function is not linearly separable, it is impossible to find a unique hyperplane to separate the input space. In Figure 1.5, it can be seen that to separate $(0,1)$ and $(1,0)$ from $(0,0)$ and $(1,1)$ we need two hyperplanes, H_1 and H_2. □

If we stack some Perceptrons, some weakness of the Perceptron like the inability of solving the XOR problem can be solved.

1.3.3 The Multi-Layer Perceptron

The Multi-Layer Perceptron (MLP) is a Feed-forward Neural Network that consists of an input layer, one or more hidden layers (layers of neurons that are between the input layer and the output layer) and an output layer, being each layer (except the output layer) fully-connected to the following layer. The MLP can be understood as the ANN obtained stacking some Perceptrons. An example of an MLP with one hidden layer can be seen in Figure 1.6.

The way to calculate the output is a generalization of how the Perceptron computes its output (in Section 2.1 we will see it in detail). The training process is more complex than the one explained for the Perceptron (it will be studied in Section 2.2).

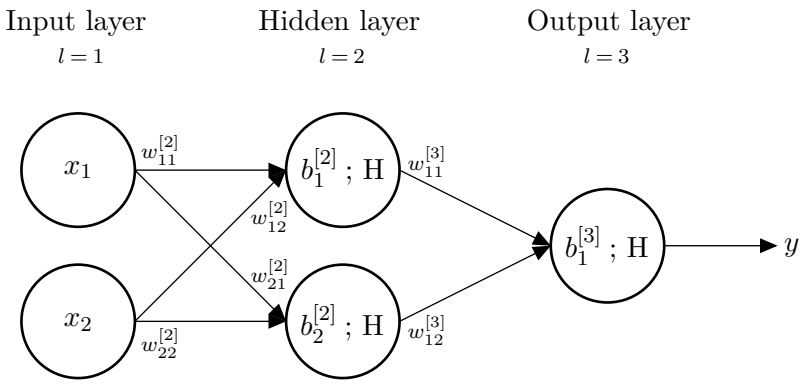

Figure 1.6: Graphic representation of an MLP with one hidden layer.

Remark 1.6. In the case of an MLP, if we use the graphic representation with bias neuron (see Remark 1.4), we have to add one neuron with value 1 in each layer except in the output one. The bias neurons are not connected to any neuron of the previous layer and they are fully-connected to all the neurons in the following one.

Notation 1.7. In what follows we use the following notation:

- x_i is the data of neuron i at the input layer.
- $w_{ji}^{[l]}$ denotes the weight of the connection between neuron i at layer $l-1$ and neuron j at layer l.
- $b_j^{[l]}$ is the bias associated to neuron j in layer l.
- $\mathcal{A}$ is the activation function.

Let us see that the XOR classification problem can be solved with an MLP.

Lemma 1.8. [5] *The XOR classification problem can be solved using an MLP.*

Proof. Let us consider a Multi-Layer Perceptron with an input layer (layer 1) formed by two neurons, one hidden layer (layer 2) constituted by two neurons and an output layer (layer 3) of a single neuron. In Figure 1.6 we can see a drawing of the situation.

We take $w_{11}^{[2]} = w_{21}^{[2]} = w_{12}^{[2]} = w_{22}^{[2]} = w_{12}^{[3]} = 1$, $w_{11}^{[3]} = -1$, $b_1^{[2]} = -1.5$, $b_2^{[2]} = b_1^{[3]} = -0.5$. We consider the Heaviside function H as the activation function $\mathcal{A}$. Then the output y of our MLP can be computed as

$$y = \mathrm{H}\left(-\mathrm{H}(x_1 + x_2 - 1.5) + \mathrm{H}(x_1 + x_2 - 0.5) - 0.5\right).$$

Notice that when $x_1 = x_2 = 0$ or $x_1 = x_2 = 1$, the output is $y = 0$ and when $x_1 = 0$, $x_2 = 1$ or $x_1 = 1$, $x_2 = 0$, the output is $y = 1$. So, the XOR problem is solved.

□

We have seen that using one hidden layer, we can solve problems that simple ANNs like the Perceptron cannot solve. In addition, it can be proved that an MLP with certain characteristics can approximate any continuous function. That is to say, if the process we want to represent with an MLP can be described with a continuous function, then such MLP exists. We state this in Theorem 1.13, but before we need to introduce some concepts and to explain the structure of the MLP of the theorem.

Definition 1.9. Sigmoidal function. [13]

A function σ is said to be sigmoidal if it satisfies that $\sigma(x) \underset{x\to+\infty}{\longrightarrow} 1$ and $\sigma(x) \underset{x\to-\infty}{\longrightarrow} 0$.

The MLP of the theorem has a single hidden layer and an output layer with one neuron. The activation function of the neurons of the hidden layer is a continuous sigmoidal function. We do not consider a bias term or an activation function for the output neuron.

Let us compute how the output of this ANN is. We have no restrictions on the input layer, so we suppose that it has n neurons and we denote the input data as $\boldsymbol{x} \in \mathbb{R}^n$. We do not have any restriction on the dimension of the unique hidden layer, we can consider that it has m neurons. The activation function of these neurons is a continuous sigmoidal function. The output layer has restrictions on the number of neurons, it has only one neuron. Moreover, we do not consider a bias term or an activation function on it. Taking into account that the output of the MLP is computed analogously to the output of the Perceptron, we have that the output y is given by

$$y = \sum_{j=1}^{m} w_{1j}^{[3]} \sigma\left(\langle \boldsymbol{w}_j^{[2]}, \boldsymbol{x}\rangle + b_j^{[2]}\right),$$

where $\boldsymbol{w}_j^{[2]} = (w_{j1}^{[2]}, \dots, w_{jn}^{[2]})^T$ and $\langle\cdot,\cdot\rangle$ denotes the scalar product of two vectors.

Notation 1.10. Let I^n be the n-dimensional cube, that is, $I^n \equiv [0,1]^n$. The set of continuous functions defined on I^n is denoted by $\mathcal{C}(I^n)$ and the supremum norm of a function $h \in \mathcal{C}(I^n)$ is $\|h\|_\infty = \sup_{\boldsymbol{x}\in I^n} |h(\boldsymbol{x})|$.

Definition 1.11. Accumulation point, derived set and closure. [16]

Let (X,τ) be a topological space. A point $x \in X$ is said to be an accumulation point of $A \subset X$ if it satisfies that for any open set $\mathcal{U}$ such that $x \in \mathcal{U}$, $(\mathcal{U}\setminus\{x\}) \cap A \neq \emptyset$. The set of all the accumulation points of A, which is denoted by A', is called derived set. The closure of the set A, that can be denoted by $\bar{A}$, is the closed set defined as $\bar{A} = A \cup A'$.

Definition 1.12. Dense. [16]

Let (X,τ) be a topological space. It is said that $A \subset X$ is dense in X if $\bar{A} = X$, where $\bar{A}$ is the closure of the set A.

Theorem 1.13. [13] *Let σ be any continuous sigmoidal function. Then, finite sums of the form*

$$G(\boldsymbol{x}) = \sum_{j=1}^{m} w_{1j}^{[3]} \sigma\left(\langle \boldsymbol{w}_j^{[2]}, \boldsymbol{x}\rangle + b_j^{[2]}\right) \tag{1.2}$$

(they are the output of an MLP with the properties described above) are dense in $\mathcal{C}(I^n)$. That is to say, given $g \in \mathcal{C}(I^n)$ and $\varepsilon > 0$ there is a function G of the form (1.2) such that

$$|G(\boldsymbol{x}) - g(\boldsymbol{x})| < \varepsilon, \quad \forall \boldsymbol{x} \in I^n.$$

To prove this theorem we have to prove two auxiliary results that make the proof of Theorem 1.13 trivial. These results are Lemma 1.21 and Lemma 1.28. To sketch out each result and to prove them we need to define and state some results of Functional Analysis and Measure Theory. For simplicity, we state some of these results for the space of continuous functions on the n-dimensional cube I^n, which is the space of functions we consider.

Definition 1.14. σ-algebra, measurable space and measure space. [17] [18]

A σ-algebra over a set X is a set $\mathscr{A} \subset \mathcal{P}(X)$ that verifies the following three properties:

- $\emptyset \in \mathscr{A}$.
- $A \in \mathscr{A} \Longrightarrow A^c \in \mathscr{A}$, *where A^c denotes the complement of the set A, that is, $A^c = X \setminus A$.*

- *If* $\{A_k\}_{k=1}^{\infty}$ *is a sequence of sets* $A_k \in \mathscr{A}$, *then* $\bigcup_{k=1}^{\infty} A_k \in \mathscr{A}$.

$(X, \mathscr{A})$ *is called measurable space.*
A real measure is a map

$$\mu : \mathscr{A} \longrightarrow \mathbb{R}$$

that is countably additive. To be countably additive means that

$$\mu\left(\bigcup_{k\in\mathbb{N}} A_k\right) = \sum_{k\in\mathbb{N}} \mu(A_k)$$

for any sequence $\{A_k\}_{k=1}^{\infty} \subset \mathscr{A}$ *whose elements are disjoint.*
$(X, \mathscr{A}, \mu)$ *is a measure space.*

Definition 1.15. Finite, signed regular Borel measure. [17] [18]
Let (X, τ) *be a topological space. The Borel* σ*-algebra of* X*, which is denoted by* $\mathcal{B}(X)$*, is the* σ*-algebra generated by the topology* τ *(that is, it is the smallest* σ*-algebra which contains all the open sets of the topology* τ*). Its elements are called Borel sets. A Borel measure* μ *is a measure defined on the Borel* σ*-algebra.*
A measure μ *is said to be finite if* $\mu(X)$ *is a finite number.*
A measure is said to be a signed measure if it is a real measure, that is to say, if it is defined as

$$\mu : \mathscr{A} \longrightarrow \mathbb{R}.$$

A Borel set A *is said to be outer regular if it satisfies that*

$$\mu(A) = \inf\{\mu(V) \mid A \subset V, V \text{ is open}\}.$$

A Borel set A *is said to be inner regular if it satisfies that*

$$\mu(A) = \sup\{\mu(K) \mid A \subset K, K \text{ is compact}\}.$$

A Borel measure μ *is called regular if any Borel set* $A \subset X$ *is outer and inner regular.*

Notation 1.16. We denote by $\mathcal{M}(I^n)$ the space of finite, signed regular Borel measures on I^n.

Definition 1.17. Discriminatory. [13]
It is said that a function σ *is discriminatory if for a measure* $\mu \in \mathcal{M}(I^n)$ *it satisfies that*

$$\int_{I^n} \sigma(\langle \boldsymbol{p}, \boldsymbol{x}\rangle + q)\, \mathrm{d}\mu = 0; \ \forall \boldsymbol{p} \in \mathbb{R}^n, q \in \mathbb{R} \implies \mu = 0.$$

Theorem 1.18. Hahn-Banach Theorem. [19]
Let M *be a subspace of a real vector space* X*. If*

- $u : X \to \mathbb{R}$ *is a functional such that*

$$u(\boldsymbol{x}+\boldsymbol{y}) \leq u(\boldsymbol{x}) + u(\boldsymbol{y}) \quad \text{and} \quad u(t\boldsymbol{x}) = tu(\boldsymbol{x}),$$

for all $\boldsymbol{x}, \boldsymbol{y} \in X$ *and* $t \geq 0$,

- $h : M \to \mathbb{R}$ *is a linear functional such that*

$$h(\boldsymbol{x}) \leq u(\boldsymbol{x}),$$

where $\boldsymbol{x} \in M$.

Then, there exists a linear functional $\Lambda : X \to \mathbb{R}$ *such that*

- $\Lambda \boldsymbol{x} = h(\boldsymbol{x})$ *for* $\boldsymbol{x} \in M$,
- $-u(-\boldsymbol{x}) \leq \Lambda \boldsymbol{x} \leq u(\boldsymbol{x})$ *for* $\boldsymbol{x} \in X$.

Corollary 1.19. Consequence of Hahn-Banach Theorem. [19]

Let $\mathcal{D}$ *be a subspace of* $\mathcal{C}(I^n)$. *If there exists some* $h \in \mathcal{C}(I^n)$ *such that* $h \notin \overline{\mathcal{D}}$, *then there exists a bounded continuous linear functional* Λ *on* $\mathcal{C}(I^n)$ *such that* $\Lambda \not\equiv 0$ *but* $\Lambda(\mathcal{D}) = 0$.

Theorem 1.20. Riesz Representation Theorem. [18]

Let us consider the n-dimensional unit cube I^n *and the space of continuous functions on* I^n *denoted by* $\mathcal{C}(I^n)$. *Then, any bounded linear functional* Λ *on* $\mathcal{C}(I^n)$ *is represented by some* $\mu \in \mathcal{M}(I^n)$ *as*

$$\Lambda h = \int_{I^n} h \, \mathrm{d}\mu,$$

where $h \in \mathcal{C}(I^n)$.

Lemma 1.21. [13] *Let* σ *be any continuous discriminatory function. Then, finite sums of the form* (1.2) *are dense in* $\mathcal{C}(I^n)$. *That is to say, given* $g \in \mathcal{C}(I^n)$ *and* $\varepsilon > 0$ *there is a finite sum* G *of the form* (1.2) *such that, for all* $\boldsymbol{x} \in I^n$,

$$|G(\boldsymbol{x}) - g(\boldsymbol{x})| < \varepsilon.$$

Proof. Let us denote by $\mathcal{S}$ the set of continuous functions of the form (1.2). It is obvious that $\mathcal{S}$ is a linear subspace of $\mathcal{C}(I^n)$. We want to prove that $\mathcal{S}$ is dense in $\mathcal{C}(I^n)$ so, by Definition 1.12, we have to prove that $\bar{\mathcal{S}} = \mathcal{C}(I^n)$, where $\bar{\mathcal{S}}$ denotes the closure of the corresponding set. Let us prove it by contradiction.

Let us assume that $\bar{\mathcal{S}} \neq \mathcal{C}(I^n)$. Therefore, $\bar{\mathcal{S}}$ is a proper subspace of $\mathcal{C}(I^n)$ (that is to say, $\bar{\mathcal{S}} \neq \emptyset, \mathcal{C}(I^n)$). Moreover, as $\bar{\mathcal{S}}$ is the closure of $\mathcal{S}$, $\bar{\mathcal{S}}$ is a closed subspace.

As $\bar{\mathcal{S}}$ is a closed proper subspace of $\mathcal{C}(I^n)$, applying the consequence of Hahn-Banach Theorem 1.18 sketched out in Corollary 1.19 we have that there exists a bounded linear functional L on $\mathcal{C}(I^n)$ such that $L \not\equiv 0$ but $L(\mathcal{S}) = 0$.

Applying Riesz Representation Theorem 1.20 to the linear bounded functional L, we conclude that it is of the form

$$L(h) = \int_{I^n} h \, \mathrm{d}\mu$$

for all $h \in \mathcal{C}(I^n)$ and for some $\mu \in \mathcal{M}(I^n)$.

Notice that $\sigma(\langle \boldsymbol{c}, \boldsymbol{x} \rangle + b) \in \mathcal{S}$ for all $\boldsymbol{c} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. Using the results we have just obtained from the consequence of Hahn-Banach Theorem and Riesz Representation Theorem we get that

$$0 = L(\sigma(\langle \boldsymbol{c}, \boldsymbol{x} \rangle + b)) = \int_{I^n} \sigma(\langle \boldsymbol{c}, \boldsymbol{x} \rangle + b) \, \mathrm{d}\mu. \tag{1.3}$$

As σ is supposed to be discriminatory, expression (1.3) implies that $\mu = 0$. This means that $L \equiv 0$, which is a contradiction since applying the consequence of Hahn-Banach Theorem we have obtained that $L \not\equiv 0$. Therefore, there is not any element of $\mathcal{C}(I^n)$ that does not belong to $\bar{\mathcal{S}}$ and we can conclude that $\bar{\mathcal{S}} = \mathcal{C}(I^n)$ and $\mathcal{S}$ is dense in $\mathcal{C}(I^n)$. □

Definition 1.22. Measurable function. [17]

Let $(X, \mathscr{A})$ *be a measurable space. A function* $h : X \to \mathbb{R}$ *is said to be a measurable function if*

$$h^{-1}(B) \in \mathscr{A}, \quad \forall B \in \mathcal{B}(\mathbb{R}).$$

Theorem 1.23. Lebesgue's Dominated Convergence Theorem. [18]

Let $\{h_k\}_{k\in\mathbb{N}}$ be a sequence of measurable functions on X such that the following limit exists for all $x \in X$:

$$h(x) = \lim_{k\to+\infty} h_k(x).$$

If h_k is pointwise bounded (for any k) by a Lebesgue integrable function, then

$$\lim_{k\to+\infty} \int_X h_k \, \mathrm{d}\mu = \int_X h \, \mathrm{d}\mu.$$

Definition 1.24. Simple function. [17] [18]

Let $(X, \mathscr{A})$ be a measurable space. A simple function s is a linear combination of indicator functions.

If $\alpha_1, \ldots, \alpha_K$ are the different values that s can take and we define $A_k = \{x \mid s(x) = \alpha_k\}$, we can write the simple function s in its canonical representation as

$$s = \sum_{k=1}^{K} \alpha_k \chi_{A_k},$$

where χ_{A_k} is the indicator function of the set A_k.

A simple function is measurable if and only if the sets A_k defined above are measurable for any $k = 1, \ldots, K$.

Theorem 1.25. [20] *Let $(X, \mathscr{A}, \mu)$ be a measure space. The set of all measurable simple functions is dense in $L^\infty(X)$ (this is the set of bounded functions defined on X).*

Definition 1.26. Fourier transform of a Borel measure. [19]

Let μ be a Borel measure on I^n. The Fourier transform of μ, denoted by $\widehat{\mu}$, is the function given by

$$\widehat{\mu}(\boldsymbol{\xi}) = \int_{I^n} e^{-i\langle \boldsymbol{\xi}, \boldsymbol{x}\rangle} \, \mathrm{d}\mu, \ \ \boldsymbol{\xi} \in \mathbb{R}^n.$$

Corollary 1.27. [21] *If $\widehat{\mu} = 0$, then $\mu = 0$ almost everywhere.*

Lemma 1.28. [13] *Any bounded measurable sigmoidal function σ is discriminatory.*

Proof. We want to prove that σ is discriminatory, so by Definition 1.17 we have to suppose that, for a measure $\mu \in \mathcal{M}(I^n)$,

$$\int_{I^n} \sigma(\langle \boldsymbol{p}, \boldsymbol{x}\rangle + q) \, \mathrm{d}\mu = 0 \tag{1.4}$$

for all $\boldsymbol{p} \in \mathbb{R}^n$ and $q \in \mathbb{R}$ and we have to prove that $\mu = 0$.

Let us define

$$\sigma_\lambda(\boldsymbol{x}) := \sigma(\lambda(\langle \boldsymbol{p}, \boldsymbol{x}\rangle + q) + r), \tag{1.5}$$

where $\boldsymbol{x}, \boldsymbol{p} \in \mathbb{R}^n$ and $q, r, \lambda \in \mathbb{R}$.

By Definition 1.9 of sigmoidal function, we have that for all $\boldsymbol{x}, \boldsymbol{p} \in \mathbb{R}^n$ and $q, r \in \mathbb{R}$,

$$\lim_{\lambda\to+\infty} \sigma_\lambda(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \langle \boldsymbol{p}, \boldsymbol{x}\rangle + q > 0, \\ 0 & \text{if } \langle \boldsymbol{p}, \boldsymbol{x}\rangle + q < 0, \\ \sigma(r) & \text{if } \langle \boldsymbol{p}, \boldsymbol{x}\rangle + q = 0. \end{cases}$$

So, if we define the function

$$\gamma(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \langle \boldsymbol{p}, \boldsymbol{x} \rangle + q > 0, \\ 0 & \text{if } \langle \boldsymbol{p}, \boldsymbol{x} \rangle + q < 0, \\ \sigma(r) & \text{if } \langle \boldsymbol{p}, \boldsymbol{x} \rangle + q = 0, \end{cases}$$

we have that $\lim_{\lambda \to +\infty} \sigma_\lambda(\boldsymbol{x}) = \gamma(\boldsymbol{x})$. That is to say, all the functions σ_λ converge pointwise to the function γ. As σ_λ is measurable (we assume that σ is measurable and σ_λ is given by (1.5)), by the Lebesgue's Dominated Convergence Theorem 1.23 we have that the following equality holds:

$$\lim_{\lambda \to +\infty} \int_{I^n} \sigma_\lambda \, \mathrm{d}\mu = \int_{I^n} \gamma \, \mathrm{d}\mu. \tag{1.6}$$

Let $H_{\boldsymbol{p},q}$ be the hyperplane

$$H_{\boldsymbol{p},q} := \{\boldsymbol{x} \mid \langle \boldsymbol{p}, \boldsymbol{x} \rangle + q = 0\}, \tag{1.7}$$

and let us consider the open half-space

$$H^+_{\boldsymbol{p},q} := \{\boldsymbol{x} \mid \langle \boldsymbol{p}, \boldsymbol{x} \rangle + q > 0\} \tag{1.8}$$

defined by the previous hyperplane. Then, function γ can be rewritten as

$$\gamma(\boldsymbol{x}) = \chi_{H^+_{\boldsymbol{p},q}}(\boldsymbol{x}) + \sigma(r)\chi_{H_{\boldsymbol{p},q}}(\boldsymbol{x}).$$

Integrating both sides of previous expression we obtain

$$\int_{I^n} \gamma \, \mathrm{d}\mu = \int_{I^n} \left[\chi_{H^+_{\boldsymbol{p},q}} + \sigma(r)\chi_{H_{\boldsymbol{p},q}}\right] \mathrm{d}\mu.$$

Moreover, as

$$\int_{I^n} \chi_A \, \mathrm{d}\mu = \int_A \mathrm{d}\mu = \mu(A) \quad \text{ with } \ A \subseteq I^n, \tag{1.9}$$

we conclude that

$$\int_{I^n} \gamma \, \mathrm{d}\mu = \mu(H^+_{\boldsymbol{p},q}) + \sigma(r)\mu(H_{\boldsymbol{p},q}). \tag{1.10}$$

Taking into account (1.4), (1.6) and (1.10), we have that

$$0 = \mu(H^+_{\boldsymbol{p},q}) + \sigma(r)\mu(H_{\boldsymbol{p},q}), \quad \forall \boldsymbol{p} \in \mathbb{R}^n;\ q, r \in \mathbb{R}.$$

Therefore,

$$\mu(H_{\boldsymbol{p},q}) = \mu(H^+_{\boldsymbol{p},q}) = 0, \quad \forall \boldsymbol{p} \in \mathbb{R}^n, q \in \mathbb{R}. \tag{1.11}$$

Fix $\boldsymbol{p}$. Let us consider a bounded measurable function f and let us define the linear functional F given by

$$F(f) := \int_{I^n} f(\langle \boldsymbol{p}, \boldsymbol{x} \rangle) \, \mathrm{d}\mu.$$

As f is bounded by definition and $\mu \in \mathcal{M}(I^n)$, we have that F is a bounded functional on $L^\infty(\mathbb{R})$.

Let us consider that f is the indicator function of the interval $[q, +\infty)$. Then, we have that

$$F(\chi_{[q,+\infty)}) = \int_{I^n} \chi_{[q,+\infty)}(\langle \boldsymbol{p}, \boldsymbol{x} \rangle) \, \mathrm{d}\mu.$$

Applying property (1.9) to the right-hand side of previous equality we have that

$$F(\chi_{[q,+\infty)}) = \int_{\{\boldsymbol{x} \in I^n \mid \langle \boldsymbol{p}, \boldsymbol{x} \rangle \geq q\}} \mathrm{d}\mu = \mu(\{\boldsymbol{x} \mid \langle \boldsymbol{p}, \boldsymbol{x} \rangle = q\}) + \mu(\{\boldsymbol{x} \mid \langle \boldsymbol{p}, \boldsymbol{x} \rangle > q\}).$$

Notice that the sets $\{\boldsymbol{x} \mid \langle \boldsymbol{p}, \boldsymbol{x} \rangle = q\}$ and $\{\boldsymbol{x} \mid \langle \boldsymbol{p}, \boldsymbol{x} \rangle > q\}$, according to the notation given in (1.7) and (1.8), are $H_{\boldsymbol{p},-q}$ and $H^+_{\boldsymbol{p},-q}$, whose measure is 0 by (1.11). Therefore, we conclude that

$$F(\chi_{[q,+\infty)}) = 0. \tag{1.12}$$

Let us consider now that f is the indicator function of the open interval $(q, +\infty)$. Arguing as in the case $\chi_{[q,+\infty)}$, we have that

$$F(\chi_{(q,+\infty)}) = 0. \tag{1.13}$$

Taking into account the linearity of the functional F and the results (1.12) and (1.13), we have that $F(f) = 0$ for f the indicator function of any interval. For example, let us consider the interval (q_1, q_2) for $q_1, q_2 \in \mathbb{R}$ with $q_1 < q_2$. The indicator function of this interval can be rewritten as $\chi_{(q_1,q_2)} = \chi_{(q_1,+\infty)} - \chi_{[q_2,+\infty)}$. Then, by the linearity of F we have that

$$F(\chi_{(q_1,q_2)}) = F(\chi_{(q_1,+\infty)} - \chi_{[q_2,+\infty)}) = F(\chi_{(q_1,+\infty)}) - F(\chi_{[q_2,+\infty)}),$$

and applying (1.12) and (1.13), we conclude that $F(\chi_{(q_1,q_2)}) = 0$. The argument for the remaining intervals is similar.

Simple functions are linear combinations of indicator functions (see Definition 1.24) so, by the linearity of F, we have that $F(f) = 0$ if f is a simple function. Moreover, as the set of measurable simple functions is dense in $L^\infty(\mathbb{R})$ (see Theorem 1.25), we can conclude that $F \equiv 0$. Then, in particular, if we take $f(s) = \cos(s) + i\sin(s)$ we have that

$$F(\cos(s) + i\sin(s)) = 0. \tag{1.14}$$

If we expand the expression $F(\cos(s) + i\sin(s))$ we get

$$F(\cos(s) + i\sin(s)) = \int_{I^n} \cos(\langle \boldsymbol{p}, \boldsymbol{x} \rangle) + i\sin(\langle \boldsymbol{p}, \boldsymbol{x} \rangle)\, \mathrm{d}\mu = \int_{I^n} e^{i\langle \boldsymbol{p}, \boldsymbol{x} \rangle}\, \mathrm{d}\mu. \tag{1.15}$$

Joining (1.14) and (1.15) we obtain that

$$\int_{I^n} e^{i\langle \boldsymbol{p}, \boldsymbol{x} \rangle}\, \mathrm{d}\mu = 0$$

for a fixed $\boldsymbol{p} \in \mathbb{R}^n$. As we have not considered any restriction over $\boldsymbol{p}$, the previous equality holds for any $\boldsymbol{p} \in \mathbb{R}^n$, and by Definition 1.26 we have that the Fourier Transform of the measure μ is equal to 0. Applying Corollary 1.27 we obtain that $\mu = 0$. We can conclude that σ is discriminatory. □

We have proved Lemma 1.21 and Lemma 1.28. Therefore, now we can present the proof of the main result, Theorem 1.13.

Proof of Theorem 1.13. In Lemma 1.28 we have proved that any bounded measurable sigmoidal function σ is discriminatory. So, in particular, as all the continuous functions are measurable (with Borel measure) we have that any continuous sigmoidal function is discriminatory. With Lemma 1.21 we can conclude that finite sums of the form (1.2) are dense in $\mathcal{C}(I^n)$. □

As we have seen, using one hidden layer we can solve problems that simple ANNs like the Perceptron cannot solve. Moreover, if the problem can be described with a continuous function, there exists an MLP (with certain characteristics) that can solve the problem quite well. Then, it is natural to think that if we add more hidden layers, ANNs will be able to solve more complex problems and to learn from data with several levels of abstraction. Deep Neural Networks (DNNs) are neural networks that have at least two hidden layers. The set of techniques used to design this type of ANNs is called Deep Learning (DL) and we will study it in Chapter 2.

Chapter 2

Deep Learning

Deep Learning (DL) is the branch of Machine Learning that includes all the techniques used to design Deep Neural Networks (ANNs with multiple hidden layers) that are able to learn from data with several levels of abstraction.

Real data is usually structured in a hierarchical way (from small details to large structures). To have several layers allows the network to *divide* the learning process into three different levels of abstraction:

- First hidden layers learn low-level structures like segments.
- Middle hidden layers work with the previous low-level structures to be able to model intermediate-level structures like squares.
- Last hidden layers and the output layer use the intermediate-level structures to model high-level structures like faces.

This hierarchical structure of DNNs helps them to improve their ability to generalize to other data sets.

DL has improved the state-of-the-art in some fields like speech recognition, object detection and drug discovery. To understand DNNs, in this chapter we introduce Feed-forward Neural Networks with fully-connected layers (in what follows we refer to them as FNNs with fully-connected layers). In the field of DL, an FNN with fully-connected layers is an MLP with at least two hidden layers (however, some topics we explain, as the architecture or the training process, can also be applied to MLPs with a unique hidden layer).

2.1 Architecture of Feed-forward Neural Networks with Fully-connected Layers

In this section we explain the architecture of Feed-forward Neural Networks with fully-connected layers. References that can be consulted are [5], [9] and [22].

FNNs with fully-connected layers have an input layer, at least two hidden layers (the word *hidden* is only used to indicate that these layers compute intermediate calculations) and an output layer (it provides the output of the ANN). Each neuron at a hidden layer receives an input from all the neurons of the previous layer and after some computations it sends information to all the neurons in the next layer.

Suppose that our FNN with fully-connected layers has L layers. We denote each layer by l, with $l = 1, \dots, L$. Layer 1 is the input layer, we have $L-2$ hidden layers ($l = 2, \dots, L-1$) and layer L is the output layer. If we denote by n_l the number of neurons at layer l ($l = 1, \dots, L$), the input data has dimension n_1, the output data has dimension n_L and the neural network is a map from $\mathbb{R}^{n_1}$ to $\mathbb{R}^{n_L}$. We have weights for the connections between two neurons of

consecutive layers, each neuron has a bias term and it applies an activation function. Let us use Notation 1.7. We denote by $a_i^{[l]}$ the information stored in the i-th neuron of layer l.

The computations performed by a neuron j at layer l can be described as follows:

- Neuron j receives information from all the neurons of the previous layer $l-1$. Taking into account the weights of the connections between these neurons and neuron j, we have that the information received by the neuron is

$$w_{ji}^{[l]} a_i^{[l-1]}$$

  for $i = 1, \ldots, n_{l-1}$.

- All these terms are added, obtaining

$$\sum_{i=1}^{n_{l-1}} w_{ji}^{[l]} a_i^{[l-1]}.$$

- To this sum, the bias term associated to the current neuron is added. We have

$$b_j^{[l]} + \sum_{i=1}^{n_{l-1}} w_{ji}^{[l]} a_i^{[l-1]}.$$

- Finally, the activation function $\mathcal{A}$ is applied to the previous expression,

$$\mathcal{A}\left(b_j^{[l]} + \sum_{i=1}^{n_{l-1}} w_{ji}^{[l]} a_i^{[l-1]} \right).$$

Thus, the value of neuron j at layer l is given by

$$a_j^{[l]} = \mathcal{A}\left(b_j^{[l]} + \sum_{i=1}^{n_{l-1}} w_{ji}^{[l]} a_i^{[l-1]} \right).$$

This number is known as the activation or output of neuron j at layer l (we can also refer to it as the value of the neuron).

Let $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$ be the matrix whose element in row j and column i is $w_{ji}^{[l]}$ and let $\boldsymbol{b}^{[l]} \in \mathbb{R}^{n_l}$ be the vector whose j-th element is $b_j^{[l]}$. We can write the vector $\boldsymbol{a}^{[l]} \in \mathbb{R}^{n_l}$, whose j-th element is the value of neuron j at layer l denoted by $a_j^{[l]}$, as

$$\boldsymbol{a}^{[l]} = \boldsymbol{\mathcal{A}}\left(W^{[l]} \boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]} \right) \in \mathbb{R}^{n_l},$$

where we understand that the activation function $\boldsymbol{\mathcal{A}}$ is applied to a vector in $\mathbb{R}^{n_l}$ as follows:

$$\begin{array}{rccc} \boldsymbol{\mathcal{A}}: & \mathbb{R}^{n_l} & \longrightarrow & \mathbb{R}^{n_l} \\ & \boldsymbol{z} \equiv (z_j)_{j=1}^{n_l} & \longmapsto & \boldsymbol{\mathcal{A}}(\boldsymbol{z}) = (\mathcal{A}(z_j))_{j=1}^{n_l}. \end{array} \tag{2.1}$$

The vector $\boldsymbol{a}^{[l]}$ can be interpreted as the activation of all the neurons at layer l.

So, if we suppose that the input data is a vector $\boldsymbol{x} \in \mathbb{R}^{n_1}$, the vector of activation of each layer (and in particular the output of the network) can be obtained as

$$\left\{ \begin{array}{rcl} \boldsymbol{a}^{[1]} & = & \boldsymbol{x} \in \mathbb{R}^{n_1}, \\ \boldsymbol{a}^{[l]} & = & \boldsymbol{\mathcal{A}}\left(W^{[l]} \boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]} \right) \in \mathbb{R}^{n_l}, \quad l = 2, \ldots, L. \end{array} \right. \tag{2.2}$$

Remark 2.1. It is necessary to use a non-linear activation function $\mathcal{A}$ in the computations performed by the DNN because, otherwise, it would be unnecessary to use hidden layers: the output would be a linear combination of the input.

The most common non-linear activation functions are:

- ReLU (Rectified Linear Unit) function (left graph of Figure 2.1):

$$\mathcal{A}(x) = \text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leq 0. \end{cases} \tag{2.3}$$

- Sigmoid (or logistic) function (middle graph of Figure 2.1):

$$\mathcal{A}(x) = \sigma(x) = \frac{1}{1+e^{-x}}. \tag{2.4}$$

- Hyperbolic tangent function (right graph of Figure 2.1):

$$\mathcal{A}(x) = \tanh(x) = 2\sigma(2x) - 1,$$

  where σ denotes the sigmoid function.

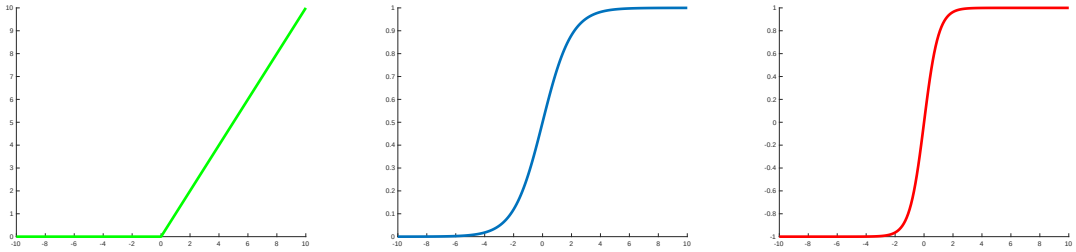

Figure 2.1: On the left, ReLU function. In the middle, sigmoid function. On the right, hyperbolic tangent function.

Example 2.2. [22] Let us suppose that we have a piece of land and we have to determine the points that are oil drilling sites and the ones that are not. To solve this problem we use an FNN with fully-connected layers.

The input of our ANN is a point in $\mathbb{R}^2$ (a point on the ground). We can take as output one vector with two coordinates interpreted as follows: if the first coordinate is bigger than the second one, the input point corresponds to an oil drilling site; otherwise, it is not. Our network is a function $\boldsymbol{g} \equiv (g_1, g_2) : \mathbb{R}^2 \to \mathbb{R}^2$.

We take an FNN with fully-connected layers that has four layers ($L = 4$). The first layer ($l = 1$) is the input one. It has two neurons ($n_1 = 2$) because, as we have said, the input data is a point in $\mathbb{R}^2$. The fourth layer ($l = 4$) is the output layer, it has two neurons ($n_4 = 2$) because of the structure we have defined for the output of the network. The second and third layer ($l = 2$ and $l = 3$, respectively) are the hidden layers. We build the second one with two neurons ($n_2 = 2$) and the third one with three ($n_3 = 3$). We use the same matrix notation for the weights and biases that we have used in the general case. As the activation function, we take the sigmoid function σ given by (2.4). In Figure 2.2 we can see a representation of the situation.

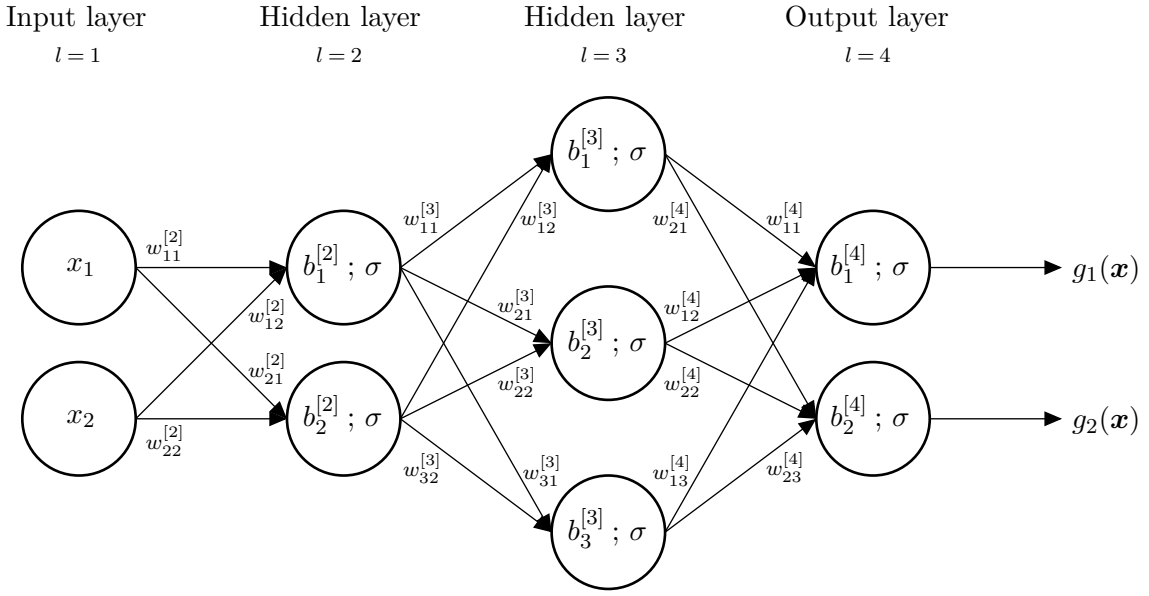

Figure 2.2: Graphic representation of the FNN with fully-connected layers used to solve a problem about the detection of oil drilling sites.

The input point is $\boldsymbol{a}^{[1]} = \boldsymbol{x} \in \mathbb{R}^2$. As the input is a point in $\mathbb{R}^2$ represented by two neurons and the second layer has two neurons too, we have that $W^{[2]} \in \mathbb{R}^{2\times 2}$ and $\boldsymbol{b}^{[2]} \in \mathbb{R}^2$. The activation of the neurons of the second layer, $\boldsymbol{a}^{[2]}$, is

$$\boldsymbol{a}^{[2]} = \sigma\left(W^{[2]}\boldsymbol{x} + \boldsymbol{b}^{[2]}\right) \in \mathbb{R}^2. \tag{2.5}$$

The next layer that receives information is the third one. It has three neurons, so taking into account that the previous one (the second layer) has two, we have that $W^{[3]} \in \mathbb{R}^{3\times 2}$ and $\boldsymbol{b}^{[3]} \in \mathbb{R}^3$. The value of the neurons at the third layer is given by

$$\boldsymbol{a}^{[3]} = \sigma\left(W^{[3]}\boldsymbol{a}^{[2]} + \boldsymbol{b}^{[3]}\right) \overset{(2.5)}{=} \sigma\left(W^{[3]}\left(\sigma\left(W^{[2]}\boldsymbol{x} + \boldsymbol{b}^{[2]}\right)\right) + \boldsymbol{b}^{[3]}\right) \in \mathbb{R}^3. \tag{2.6}$$

The following layer is the output layer. This implies that the output of its neurons is the output of the network. As this last layer has two neurons and the third layer (the previous one) has three, $W^{[4]} \in \mathbb{R}^{2\times 3}$ and $\boldsymbol{b}^{[4]} \in \mathbb{R}^2$. The output is

$$\boldsymbol{a}^{[4]} = \sigma\left(W^{[4]}\boldsymbol{a}^{[3]} + \boldsymbol{b}^{[4]}\right) \overset{(2.6)}{=} \sigma\left(W^{[4]}\left(\sigma\left(W^{[3]}\left(\sigma\left(W^{[2]}\boldsymbol{x} + \boldsymbol{b}^{[2]}\right)\right) + \boldsymbol{b}^{[3]}\right)\right) + \boldsymbol{b}^{[4]}\right) \in \mathbb{R}^2.$$

The function $\boldsymbol{g}$ that defines our network is

$$\begin{aligned} \boldsymbol{g}: \quad \mathbb{R}^2 &\longrightarrow \mathbb{R}^2 \\ \boldsymbol{x} &\longmapsto \sigma\left(W^{[4]}\left(\sigma\left(W^{[3]}\left(\sigma\left(W^{[2]}\boldsymbol{x} + \boldsymbol{b}^{[2]}\right)\right) + \boldsymbol{b}^{[3]}\right)\right) + \boldsymbol{b}^{[4]}\right). \end{aligned}$$

In this example we have obtained an expression to deduce if a point on the land is an oil drilling site or not. But, how have we chosen the number of layers or the number of neurons per layer? Our DNN depends on 23 parameters (the elements of the weight matrices and the bias vectors), how do we give values to them? We answer these questions in the following sections.

2.2 Training Deep Neural Networks

In Example 2.2 we wanted to know if a point on the land was an oil drilling site or not. To do this we have built an FNN with fully-connected layers. However, we have observed that we needed to determine the value of 23 parameters to be able to do the classification of the input data (possible oil drilling points). To find these values we have to train the neural network. In this section, which is based on [5], [22], [23] and [24], we explain how to do it.

As we have introduced at the beginning of the chapter, Deep Learning techniques are carried out to allow ANNs to learn from data. In this context, *to learn from data* means to use data to determine the weights and biases that give us the correct result when new data is introduced to the network. To do this, it is clear that the DNN needs labeled data (input data associated to a target output) to learn from. This data is the training data.

Remark 2.3. [10] Before using the labeled data, it must be preprocessed. For example, outliers have to be deleted and if the input variables are category variables, the data has to be transformed into numeric values.

The process of learning from data, known as training, consists in introducing a training point (input point) in the network to obtain an output. This output is compared to the target output (the label of the training point) using the loss function (also called cost function). The loss function $f(W,\boldsymbol{b})$ is a function of weights (we represent them with W) and biases (we denote them by $\boldsymbol{b}$) that allows us to know how well the DNN with given weights and biases fits the data. Our objective is to minimize it for the training data.

Training the network is an optimization problem which consists in computing values for the weights and biases that minimize the loss function (objective function). One of the most used loss functions is the quadratic cost function

$$f(W,\boldsymbol{b}) = \frac{1}{M}\sum_{m=1}^{M} \|\boldsymbol{y}^{\{m\}} - \widehat{\boldsymbol{y}}^{\{m\}}\|_2^2, \tag{2.7}$$

where M is the size of the training set (number of training points), $\boldsymbol{y}^{\{m\}}$ denotes the target output for the m-th training data point $\boldsymbol{x}^{\{m\}}$, $\widehat{\boldsymbol{y}}^{\{m\}}$ is the output of the network when the input is the m-th training data point (that is, $\widehat{\boldsymbol{y}}^{\{m\}} = \boldsymbol{a}^{[L]}$) and $\|\cdot\|_2$ is the Euclidean norm.

Remark 2.4. If we multiply the loss function by a scalar, the optimization problem does not change, but later computations may be easier. For example, we can take

$$f(W,\boldsymbol{b}) = \frac{1}{M}\sum_{m=1}^{M} \frac{1}{2}\|\boldsymbol{y}^{\{m\}} - \widehat{\boldsymbol{y}}^{\{m\}}\|_2^2, \tag{2.8}$$

instead of (2.7). Moreover, if we define

$$f_{\{\boldsymbol{x}^{\{m\}}\}}(W,\boldsymbol{b}) = \frac{1}{2}\|\boldsymbol{y}^{\{m\}} - \widehat{\boldsymbol{y}}^{\{m\}}\|_2^2$$

for $m = 1,\ldots,M$, we can rewrite it as

$$f(W,\boldsymbol{b}) = \frac{1}{M}\sum_{m=1}^{M} f_{\{\boldsymbol{x}^{\{m\}}\}}(W,\boldsymbol{b}).$$

Let us return to the example of the oil drilling sites given in Example 2.2. Notice that it is relatively simple since it has only two hidden layers and nine neurons. However, to train the network is a difficult optimization problem that consists in minimizing a non-convex function over 23 parameters. Due to the difficulty of training DNNs, the beginnings of DL were hard (this epoch is known as AI winter). Now, we are going to study how to solve this optimization problem.

2.2.1 Gradient Descent and Stochastic Gradient

As we have seen, to train a network consists in solving an optimization problem: we have to minimize the loss function $f(W, \boldsymbol{b})$ over the space of all the possible values for the weights and biases. Our problem can be formulated as

$$\min_{W,\boldsymbol{b}} f(W, \boldsymbol{b}). \tag{2.9}$$

To solve it we may use a classical optimization method known as Gradient Descent (or Steepest Descent).

Notation 2.5. To make it easier, instead of assuming that we have $L-1$ matrices for the weights ($W^{[2]} \in \mathbb{R}^{n_2 \times n_1}, \ldots, W^{[L]} \in \mathbb{R}^{n_L \times n_{L-1}}$) and $L-1$ vectors for the biases ($\boldsymbol{b}^{[2]} \in \mathbb{R}^{n_2}, \ldots,$ $\boldsymbol{b}^{[L]} \in \mathbb{R}^{n_L}$), we assume that the weights and biases are stored in a unique vector $\boldsymbol{p} \in \mathbb{R}^s$ with

$$s = \sum_{l=2}^{L} n_l (n_{l-1} + 1).$$

Now, the loss function is denoted by $f(\boldsymbol{p}) := f(W, \boldsymbol{b})$ and the minimization problem (2.9) is rewritten as

$$\min_{\boldsymbol{p}} f(\boldsymbol{p}). \tag{2.10}$$

Gradient Descent (GD) is a classical optimization method that consists in computing iteratively vectors $\boldsymbol{p}^1$, $\boldsymbol{p}^2, \ldots \in \mathbb{R}^s$ to converge to a vector $\boldsymbol{p}^{min} \in \mathbb{R}^s$ that minimizes $f(\boldsymbol{p})$. At each iteration k, we have to compute the gradient of the loss function at $\boldsymbol{p}^k$ and we have to move $\boldsymbol{p}^k$ on the direction of descending gradient to obtain $\boldsymbol{p}^{k+1}$. Let us see that this direction is the right one.

At each iteration k, we have to choose a perturbation $\Delta\, \boldsymbol{p}^k$ such that

$$f(\boldsymbol{p}^k) \geq f(\boldsymbol{p}^k + \Delta\, \boldsymbol{p}^k).$$

Then, $\boldsymbol{p}^{k+1} = \boldsymbol{p}^k + \Delta\, \boldsymbol{p}^k$.

Assuming that $\Delta \boldsymbol{p}^k$ is small, we can use Taylor series expansion to approximate $f(\boldsymbol{p}^k + \Delta\, \boldsymbol{p}^k)$ as

$$f(\boldsymbol{p}^k + \Delta\, \boldsymbol{p}^k) \approx f(\boldsymbol{p}^k) + \sum_{r=1}^{s} \frac{\partial f}{\partial p_r}(\boldsymbol{p}^k)\, \Delta\, p_r^k, \tag{2.11}$$

where $\dfrac{\partial f}{\partial p_r}$ denotes the partial derivative of f with respect to the r-th component of the vector $\boldsymbol{p}$ and $\Delta\, p_r^k$ is the r-th component of $\Delta\, \boldsymbol{p}^k$.

The previous formula (2.11) can be rewritten as

$$f(\boldsymbol{p}^k + \Delta\, \boldsymbol{p}^k) \approx f(\boldsymbol{p}^k) + \langle \nabla\, f(\boldsymbol{p}^k), \Delta\, \boldsymbol{p}^k \rangle, \tag{2.12}$$

with $\nabla\, f(\boldsymbol{p}) = \left(\dfrac{\partial f}{\partial p_1}(\boldsymbol{p}), \dfrac{\partial f}{\partial p_2}(\boldsymbol{p}), \ldots, \dfrac{\partial f}{\partial p_s}(\boldsymbol{p}) \right)^T$ the gradient of $f(\boldsymbol{p})$.

As we need $\Delta\, \boldsymbol{p}^k$ such that $f(\boldsymbol{p}^k + \Delta\, \boldsymbol{p}^k) \leq f(\boldsymbol{p}^k)$, seeing (2.12) it is natural to think that we have to select $\Delta\, \boldsymbol{p}^k$ to make $\langle \nabla\, f(\boldsymbol{p}^k), \Delta\, \boldsymbol{p}^k \rangle$ as negative as possible. Cauchy-Schwarz-Bunyakovski Inequality allows us to find an expression for $\Delta\, \boldsymbol{p}^k$.

Proposition 2.6. Cauchy-Schwarz-Bunyakovski Inequality. [17]

For any $\boldsymbol{u}$, $\boldsymbol{v} \in \mathbb{R}^s$,

$$|\langle \boldsymbol{u}, \boldsymbol{v} \rangle| \leq \|\boldsymbol{u}\| \|\boldsymbol{v}\|,$$

and the equality holds if and only if $\boldsymbol{u}$ and $\boldsymbol{v}$ are linearly dependent.

Taking $\boldsymbol{u} = \nabla f(\boldsymbol{p}^k)$ and $\boldsymbol{v} = \Delta \boldsymbol{p}^k$ in Proposition 2.6 we have that

$$|\langle \nabla f(\boldsymbol{p}^k), \Delta \boldsymbol{p}^k \rangle| \leq \|\nabla f(\boldsymbol{p}^k)\| \|\Delta \boldsymbol{p}^k\|. \tag{2.13}$$

Equivalently,

$$-\|\nabla f(\boldsymbol{p}^k)\| \|\Delta \boldsymbol{p}^k\| \leq \langle \nabla f(\boldsymbol{p}^k), \Delta \boldsymbol{p}^k \rangle \leq \|\nabla f(\boldsymbol{p}^k)\| \|\Delta \boldsymbol{p}^k\|.$$

It is obvious that the 'most negative' that $\langle \nabla f(\boldsymbol{p}^k), \Delta \boldsymbol{p}^k \rangle$ can be is $-\|\nabla f(\boldsymbol{p}^k)\| \|\Delta \boldsymbol{p}^k\|$. This happens when the equality holds in (2.13), that is, when $\nabla f(\boldsymbol{p}^k)$ and $\Delta \boldsymbol{p}^k$ are linearly dependent. We conclude that $\Delta \boldsymbol{p}^k$ has to lie in the direction of $-\nabla f(\boldsymbol{p}^k)$. As (2.12) is an approximation for a small $\Delta \boldsymbol{p}^k$, the step in that direction has to be small. So,

$$\Delta \boldsymbol{p}^k = -\eta \nabla f(\boldsymbol{p}^k), \tag{2.14}$$

where η is the small constant step size known as learning rate. Since $\Delta \boldsymbol{p}^k = \boldsymbol{p}^{k+1} - \boldsymbol{p}^k$, we can rewrite the last equality (2.14) as

$$\boldsymbol{p}^{k+1} = \boldsymbol{p}^k - \eta \nabla f(\boldsymbol{p}^k). \tag{2.15}$$

This is the Gradient Descent update rule. We have proved that the direction of descending gradient is the correct one to try to converge to the vector $\boldsymbol{p}^{min}$ that minimizes the loss function.

In practice, we choose an initial value $\boldsymbol{p}^0$ for the weights and biases (notice that $\boldsymbol{p}^0$ is a parameter not fitted during training) and we apply iteratively (2.15) until a stopping criterion is reached. Common stopping criterions are to iterate a certain number of times $maxiter$ or to iterate until the distance between $\boldsymbol{p}^k$ and $\boldsymbol{p}^{k+1}$ is smaller than a given value tol.

Notice that if we use GD to train a neural network, we have to compute $\nabla f(\boldsymbol{p}^k)$ at each iteration. It is computationally expensive to calculate it on the entire training set, so it is convenient to find less computationally expensive alternatives to Gradient Descent.

Stochastic Gradient (SG), also called Stochastic Gradient Descent (SGD), is a computationally cheaper alternative to Gradient Descent. However, unlike GD, in SG the reduction of $f(\boldsymbol{p}^k)$ is not guaranteed at each iteration. The idea of this optimization algorithm is to choose an approximation of $\nabla f(\boldsymbol{p}^k)$ to substitute it in the update rule (2.15). We have several possibilities for this replacement.

The simplest way is to choose randomly (with a uniform distribution) an index $m \in \{1, \ldots, M\}$ at each iteration and to move in the direction of $-\nabla f_{\{\boldsymbol{x}^{\{m\}}\}}(\boldsymbol{p}^k)$. Formula (2.15) is rewritten as

$$\boldsymbol{p}^{k+1} = \boldsymbol{p}^k - \eta \nabla f_{\{\boldsymbol{x}^{\{m\}}\}}(\boldsymbol{p}^k). \tag{2.16}$$

The index at each iteration is chosen by sampling with replacement.

Another option is similar to the one described above, but instead of choosing the index by sampling with replacement, it is chosen by sampling without replacement. When all the indexes have been used, it is said that an epoch has been completed (notice that an epoch consist in computing M iterations). In this case, the number of maximum epochs can be used as a stopping criterion.

Other alternative closer to Gradient Descent is to take $N \ll M$, to choose randomly (with a uniform distribution) N indexes $\{\alpha_1, \ldots, \alpha_N\}$ from $\{1, \ldots, M\}$. We move in the descending direction of the mean of the gradient of $f_{\{\boldsymbol{x}^{\{\alpha_m\}}\}}$ for $m \in \{1, \ldots, N\}$. That is to say, (2.15) is replaced by

$$\boldsymbol{p}^{k+1} = \boldsymbol{p}^k - \eta \frac{1}{N} \sum_{m=1}^{N} \nabla f_{\{\boldsymbol{x}^{\{\alpha_m\}}\}}(\boldsymbol{p}^k). \tag{2.17}$$

The set $\{\boldsymbol{x}^{\{\alpha_m\}}\}_{m=1}^N$ chosen at each iteration is known as mini-batch.

The choice of the indexes $\{\alpha_1, \ldots, \alpha_N\} \subset \{1, \ldots, M\}$ can be done sampling with replacement or sampling without replacement. In the last case, N is usually chosen in a way that $\frac{M}{N}$ is a natural number. The training set is divided in $\frac{M}{N}$ pairwise disjoint mini-batches.

Notice that if we take $N = 1$ in (2.17) we obtain (2.16). So, in what follows, we are going to use Stochastic Gradient to refer to formula (2.17), which is more general.

We have defined the learning rate η as a constant. However, it can depend on the iteration. If we denote by η_k the step size of iteration k, we can rewrite the update of the weights and biases for GD (2.15) and SG (2.17) as

$$\boldsymbol{p}^{k+1} = \boldsymbol{p}^k - \eta_k \nabla f(\boldsymbol{p}^k), \tag{2.18}$$

$$\boldsymbol{p}^{k+1} = \boldsymbol{p}^k - \eta_k \frac{1}{N} \sum_{m=1}^{N} \nabla f_{\{\boldsymbol{x}^{\{\alpha_m\}}\}}(\boldsymbol{p}^k),$$

respectively.

Observe that if the convergence of the optimization methods is not fast, the process of training the network can be incredibly computationally expensive. The optimization methods we have presented so far (Gradient Descent and Stochastic Gradient) have convergence issues. In Subsection 2.2.2 we focus on convex smooth optimization (that is, we consider that the loss function f is convex and $\mathcal{L}$-smooth) to study variants of GD and in Subsection 2.2.3 we introduce other algorithms based on SG.

2.2.2 Accelerating Gradient Descent in Convex Smooth Optimization

In this subsection we suppose that f is a convex $\mathcal{L}$-smooth function.

Definition 2.7. Convex.

The function f is convex if

$$f(\lambda \boldsymbol{p}_1 + (1-\lambda)\boldsymbol{p}_2) \leq \lambda f(\boldsymbol{p}_1) + (1-\lambda) f(\boldsymbol{p}_2); \ \forall \boldsymbol{p}_1, \boldsymbol{p}_2 \in \mathbb{R}^s, \forall \lambda \in [0,1].$$

Definition 2.8. $\mathcal{L}$-smooth. [25]

The function f is $\mathcal{L}$-smooth if $f \in \mathcal{C}^1$ (that is, it is differentiable and its first-order derivatives are continuous) and its gradient is Lipschitz continuous with constant $\mathcal{L}$, which means that

$$\|\nabla f(\boldsymbol{p}_1) - \nabla f(\boldsymbol{p}_2)\|_2 \leq \mathcal{L} \|\boldsymbol{p}_1 - \boldsymbol{p}_2\|_2; \quad \forall \boldsymbol{p}_1, \boldsymbol{p}_2 \in \mathbb{R}^s.$$

If $f \in \mathcal{C}^2$ (that is to say, f is twice continuously differentiable), then the previous condition is equivalent to

$$\|\nabla^2 f\|_2 \leq \mathcal{L},$$

where $\nabla^2 f$ is the Hessian of f.

Notation 2.9. [23] Let $\{a_n\}$ and $\{b_n\}$ be two sequences. We use $a_n = \mathcal{O}(b_n)$ (big $\mathcal{O}$ notation) to indicate that there exists a positive constant $\mathcal{K}$ such that $a_n \leq \mathcal{K} b_n$.

As we have seen, Gradient Descent is a classical optimization method that we can use to solve the minimization problem (2.10) (related to the training process of a DNN), where f is the loss function. Remember that if we use GD, the update rule for $\boldsymbol{p}^k$ (vector with the weights and biases of the ANN) is given by (2.18). If the loss function f is convex and $\mathcal{L}$-smooth and we take $\eta_k \equiv 1/\mathcal{L}$ (in what follows we consider this value for the learning rate), the convergence rate of Gradient Descent is $\mathcal{O}\left(\frac{1}{k}\right)$, where k is the iteration number.

If we add the momentum $\boldsymbol{p}^k - \boldsymbol{p}^{k-1}$ to the Gradient Descent update rule (2.18), we may accelerate it. The scheme that we obtain is known as Heavy Ball (HB) and it is given by

$$\boldsymbol{p}^{k+1} = \boldsymbol{p}^k - \eta_k \nabla f(\boldsymbol{p}^k) + \mu(\boldsymbol{p}^k - \boldsymbol{p}^{k-1}), \tag{2.19}$$

where $\mu > 0$ is a constant called momentum coefficient. It has a convergence rate of $\mathcal{O}\left(\frac{1}{k}\right)$.

We can also accelerate GD using Nesterov momentum. In this case, the update rule has two steps:

$$\begin{aligned} \boldsymbol{q}^{k+1} &= \boldsymbol{p}^k - \eta_k \nabla f(\boldsymbol{p}^k), \\ \boldsymbol{p}^{k+1} &= \boldsymbol{q}^{k+1} + \mu(\boldsymbol{q}^{k+1} - \boldsymbol{q}^k). \end{aligned} \tag{2.20}$$

As in the case of GD and HB, it has a convergence rate of $\mathcal{O}\left(\frac{1}{k}\right)$.

We have seen that GD and the algorithms obtained adding momentum to GD have the same convergence rate. However, there is empirical evidence that the convergence of GD is slower when momentum is not used. In Figure 2.3 we have the evolution of $\{f(\boldsymbol{p}^k) - f(\boldsymbol{p}^{min})\}_k$ when f is a convex quadratic function with Lipschitz constant 4 defined as

$$\frac{1}{2}\boldsymbol{p}^T A \boldsymbol{p} - \boldsymbol{p}^T \boldsymbol{d}, \tag{2.21}$$

where $A \in \mathbb{R}^{1000\times 1000}$ is the Laplacian of a cycle graph and $\boldsymbol{d} \in \mathbb{R}^{1000}$ is a vector whose first component has value 1 and the remaining have value 0. The optimization algorithms are applied during 50000 iterations with $\eta_k = \frac{1}{4}$. In dark green we have the evolution with GD and in red with GD+Momentum (Nesterov momentum is used). It is clearly seen that GD is accelerated when momentum is added.

If in (2.20) we take

$$\mu = \frac{t_k - 1}{t_{k+1}} \tag{2.22}$$

with t_k given by the recursive formula

$$t_0 = 1, \;\; t_{k+1} = \frac{1 + \sqrt{1 + 4t_k^2}}{2} \text{ for } k \geq 0, \tag{2.23}$$

we obtain

$$\begin{aligned} \boldsymbol{q}^{k+1} &= \boldsymbol{p}^k - \eta_k \nabla f(\boldsymbol{p}^k), \\ \boldsymbol{p}^{k+1} &= \boldsymbol{q}^{k+1} + \frac{t_k - 1}{t_{k+1}}(\boldsymbol{q}^{k+1} - \boldsymbol{q}^k). \end{aligned} \tag{2.24}$$

This update rule gives us the optimization method known as Nesterov Accelerated Gradient (NAG). It has a convergence rate of $\mathcal{O}\left(\frac{1}{k^2}\right)$ (this rate is the optimal one for general convex smooth optimization problems).

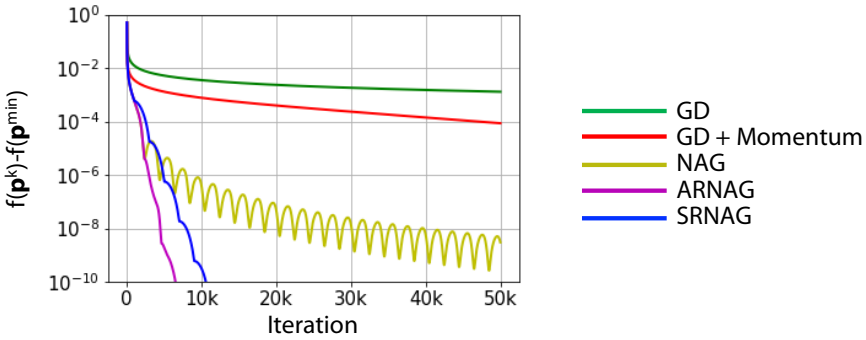

Figure 2.3: Evolution of $\{f(\boldsymbol{p}^k) - f(\boldsymbol{p}^{min})\}_k$ when f is the quadratic function (2.21). This image has been modified from [23].

Remark 2.10. In [26] it is said that the asymptotic limit of the momentum coefficient given by (2.22) is

$$\frac{k-1}{k+2}, \tag{2.25}$$

so we can consider the expression of the limit instead of the original one.

The sequence $\{f(\boldsymbol{p}^k) - f(\boldsymbol{p}^{min})\}_k$ converges monotonically to zero when GD (2.18), HB (2.19) or GD+Nesterov momentum (2.20) are used. However, in the case of NAG (2.24), it oscillates. In Figure 2.3 we can see the evolution of this sequence for GD (dark green), GD+Momentum (red) and NAG (olive-green) when we take the quadratic function (2.21) defined before. We can clearly see that NAG converges faster than the other two, but the sequence $\{f(\boldsymbol{p}^k) - f(\boldsymbol{p}^{min})\}_k$ oscillates.

To alleviate this phenomenon, the Adaptive Restart NAG (ARNAG) method can be used. It is obtained changing properly the momentum coefficient. Its update rule is given by

$$\begin{aligned} \boldsymbol{q}^{k+1} &= \boldsymbol{p}^k - \eta_k \nabla f(\boldsymbol{p}^k), \\ \boldsymbol{p}^{k+1} &= \boldsymbol{q}^{k+1} + \frac{\mathrm{m}(k)-1}{\mathrm{m}(k)+2}(\boldsymbol{q}^{k+1} - \boldsymbol{q}^k), \end{aligned}$$

where $\mathrm{m}(1) = 1$ and $\mathrm{m}(k+1) = \begin{cases} \mathrm{m}(k)+1 & \text{if } f(\boldsymbol{p}^{k+1}) \leq f(\boldsymbol{p}^k), \\ 1 & \text{otherwise.} \end{cases}$

Notice that if the condition $f(\boldsymbol{p}^{k+1}) \leq f(\boldsymbol{p}^k)$ holds, we have the NAG scheme (using the momentum coefficient (2.25)). Otherwise, $\boldsymbol{q}^{k+1} - \boldsymbol{q}^k$ is multiplied by 0 and the algorithm is restarted in the sense that the last point obtained with it is used as the starting point to apply the algorithm again. That is to say, NAG (with momentum coefficient (2.25)) is applied until $f(\boldsymbol{p}^{k+1}) > f(\boldsymbol{p}^k)$, then NAG starts again with the last point as the initial point.

Scheduled Restart NAG (SRNAG) is other scheme that uses NAG and a restart strategy (different from the one used in ARNAG). In this case, the interval of iterations $(0, maxiter]$ is divided into few intervals $\{I_\beta\}_\beta$ such that $(0, maxiter] = \bigsqcup_\beta I_\beta$. To each interval I_β we associate a number F_β known as restart frequency. Inside each interval, NAG (using an appropriate momentum coefficient) is applied until the iteration number k satisfies $k \equiv 0 \mod F_\beta$, then the momentum is restarted and NAG is applied again with the last point as the initial point. The scheme for each I_β can be written as

$$\begin{aligned} \boldsymbol{q}^{k+1} &= \boldsymbol{p}^k - \eta_k \nabla f(\boldsymbol{p}^k), \\ \boldsymbol{p}^{k+1} &= \boldsymbol{q}^{k+1} + \frac{k \mod F_\beta}{(k \mod F_\beta)+3}(\boldsymbol{q}^{k+1} - \boldsymbol{q}^k). \end{aligned} \tag{2.26}$$

Definition 2.11. Polyak-Łojasiewicz Condition. [27]

A function $f : \mathbb{R}^s \to \mathbb{R}$ satisfies Polyak-Łojasiewicz (PL) Condition if there exists $\rho > 0$ such that

$$\frac{1}{2}\|\nabla f(\boldsymbol{p})\|_2^2 \geq \rho(f(\boldsymbol{p}) - f(\boldsymbol{p}^{min})); \quad \forall \boldsymbol{p} \in \mathbb{R}^s,$$

where $\boldsymbol{p}^{min}$ denotes the minimizer of f.

If the loss function f satisfies Polyak-Łojasiewicz Condition, ARNAG and SRNAG have linear convergence.

In Figure 2.3 we can see the evolution of $\{f(\boldsymbol{p}^k) - f(\boldsymbol{p}^{min})\}_k$ for the optimization methods studied so far, when f is the quadratic function (2.21). It can be seen that ARNAG (in purple) and SRNAG (in dark blue) converge faster than the others.

To sum up, in convex smooth optimization, if we add constant momentum to GD (obtaining HB if we consider $\boldsymbol{p}_k - \boldsymbol{p}_{k-1}$ momentum, or the update rule in (2.20) if Nesterov momentum is used), we can accelerate it. Moreover, changing properly the constant μ of (2.20), we obtain NAG, whose convergence rate is the optimal one for general convex smooth optimization problems. We can also restart NAG to obtain ARNAG and SRNAG algorithms that have linear convergence.

2.2.3 Accelerating Stochastic Gradient

In Subsection 2.2.2 we have seen that we have good convergence results for NAG in convex smooth optimization when exact gradient is used. Remember that it is computationally expensive to compute the expression of the gradient on the entire training set, so let us approximate the gradient $\nabla f(\boldsymbol{p}^k)$ by $\frac{1}{N}\sum_{m=1}^{N} \nabla f_{\{\boldsymbol{x}^{\{\alpha_m\}}\}}(\boldsymbol{p}^k)$ in the update rule (2.24). With this approximation, we obtain the following scheme:

$$\begin{aligned} \boldsymbol{q}^{k+1} &= \boldsymbol{p}^k - \eta_k \frac{1}{N} \sum_{m=1}^{N} \nabla f_{\{\boldsymbol{x}^{\{\alpha_m\}}\}}(\boldsymbol{p}^k), \\ \boldsymbol{p}^{k+1} &= \boldsymbol{q}^{k+1} + \frac{t_k - 1}{t_{k+1}}(\boldsymbol{q}^{k+1} - \boldsymbol{q}^k). \end{aligned} \tag{2.27}$$

This algorithm is called Nesterov Accelerated Stochastic Gradient Descent (NASGD).

Theorem 2.12. *Let f be a convex $\mathcal{L}$-smooth function. The sequence $\{\boldsymbol{p}^k\}_{k\geq 0}$ obtained with NASGD algorithm* (2.27) *using any constant learning rate $\eta_k \equiv \eta \leq \frac{1}{\mathcal{L}}$ satisfies*

$$\mathbb{E}\left[f(\boldsymbol{p}^k) - f(\boldsymbol{p}^{min})\right] = \mathcal{O}(k),$$

where $\boldsymbol{p}^{min}$ denotes the minimizer of function f and the expectation is taken over the random mini-batch samples.

According to Theorem 2.12, NASGD accumulates error even when the function f is convex and $\mathcal{L}$-smooth. To prove it we need to introduce some definitions and results (we prove some of them).

We consider that the loss function f is convex and $\mathcal{L}$-smooth. Let us suppose that we have computed $\boldsymbol{p}^k$, then the update rule of Gradient Descent with learning rate $\eta = \frac{1}{r}$ can be obtained with the minimization of

$$\mathcal{Q}_r(\boldsymbol{q}, \boldsymbol{p}^k) := \langle \boldsymbol{q} - \boldsymbol{p}^k, \nabla f(\boldsymbol{p}^k)\rangle + \frac{r}{2}\|\boldsymbol{q} - \boldsymbol{p}^k\|_2^2. \tag{2.28}$$

It can be proved that

$$\mathcal{Q}_r(\boldsymbol{q}^{k+1}, \boldsymbol{p}^k) - \min_{\boldsymbol{q}} \mathcal{Q}_r(\boldsymbol{q}, \boldsymbol{p}^k) = \frac{\|\boldsymbol{\Lambda}^k - \nabla f(\boldsymbol{p}^k)\|_2}{2r}, \tag{2.29}$$

where $\boldsymbol{\Lambda}^k$ is used to denote the expression of the gradient with mini-batch, that is,

$$\boldsymbol{\Lambda}^k := \frac{1}{N} \sum_{m=1}^{N} \nabla f_{\{\boldsymbol{x}^{\{\alpha_m\}}\}}(\boldsymbol{p}^k).$$

If we assume that Stochastic Gradient (with mini-batch) has bounded variance, we obtain the Stochastic Gradient Rule $\mathcal{R}_\delta$:

$$\mathbb{E}\left[\mathcal{Q}_r(\boldsymbol{q}^{k+1}, \boldsymbol{p}^k) - \min_{\boldsymbol{q}} \mathcal{Q}_r(\boldsymbol{q}, \boldsymbol{p}^k) \mid \chi^k\right] \leq \delta, \tag{2.30}$$

where δ is a constant and $\chi^k := \sigma(\boldsymbol{p}^1, \boldsymbol{p}^2, \ldots, \boldsymbol{p}^k)$, that is, χ^k is the σ-algebra generated by $\boldsymbol{p}^1, \boldsymbol{p}^2, \ldots, \boldsymbol{p}^k$.

Remark 2.13. In formula (2.30) we take the expectation conditioned to the σ-algebra of the weights and biases that have already been computed (we have used χ^k to denote this σ-algebra). In what follows, although it is not written explicitly, we consider this conditional expectation.

If we consider (2.28) and (2.29), we have that the update rule of NASGD given by (2.27) can be formulated as

$$\begin{aligned} \boldsymbol{q}^{k+1} &\approx \underset{\boldsymbol{q}}{\operatorname{argmin}}\, \mathcal{Q}_r(\boldsymbol{q}, \boldsymbol{p}^k) \quad \text{with rule } \mathcal{R}_\delta, \\ \boldsymbol{p}^{k+1} &= \boldsymbol{q}^{k+1} + \frac{t_k - 1}{t_{k+1}}(\boldsymbol{q}^{k+1} - \boldsymbol{q}^k). \end{aligned} \tag{2.31}$$

Notation 2.14. We define $\widetilde{\boldsymbol{q}}^{k+1} := \underset{\boldsymbol{q}}{\operatorname{argmin}}\, \mathcal{Q}_r(\boldsymbol{q}, \boldsymbol{p}^k)$.

With the notation we have just introduced we can rewrite (2.31) as

$$\begin{aligned} \boldsymbol{q}^{k+1} &\approx \widetilde{\boldsymbol{q}}^{k+1} \quad \text{with rule } \mathcal{R}_\delta, \\ \boldsymbol{p}^{k+1} &= \boldsymbol{q}^{k+1} + \frac{t_k - 1}{t_{k+1}}(\boldsymbol{q}^{k+1} - \boldsymbol{q}^k). \end{aligned} \tag{2.32}$$

Lemma 2.15. Schwarz Inequality. [23]
Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be two vectors and let $\lambda > 0$ be a real constant. Then,

$$\langle \boldsymbol{a}, \boldsymbol{b} \rangle \leq \frac{\|\boldsymbol{a}\|_2^2}{2\lambda} + \frac{\lambda}{2}\|\boldsymbol{b}\|_2^2.$$

Lemma 2.16. [23] *Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be two vectors. Then, we have the following equality:*

$$\|\boldsymbol{a} - \boldsymbol{b}\|_2^2 + 2\langle \boldsymbol{a} - \boldsymbol{b}, \boldsymbol{b} \rangle = \|\boldsymbol{a}\|_2^2 - \|\boldsymbol{b}\|_2^2.$$

Lemma 2.17. [28] *Let h be a differentiable function such that $\operatorname{dom}(h)$ is a convex set. Then h is convex if and only if*

$$h(\boldsymbol{y}) \geq h(\boldsymbol{x}) + \langle \nabla\, h(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle; \quad \forall \boldsymbol{x}, \boldsymbol{y} \in \operatorname{dom}(h),$$

where $\operatorname{dom}(h)$ denotes the domain of h.

Definition 2.18. ν-strongly convex. [23]
A function h is ν-strongly convex if

$$h(\boldsymbol{y}) \geq h(\boldsymbol{x}) + \langle \nabla\, h(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle + \frac{\nu}{2}\|\boldsymbol{y} - \boldsymbol{x}\|_2^2; \quad \forall \boldsymbol{x}, \boldsymbol{y} \in \operatorname{dom}(h).$$

Lemma 2.19. [23] *Let h be ν-strongly convex and let us denote the minimizer of h by $\boldsymbol{x}^{min}$. Then,*

$$h(\boldsymbol{x}) - h(\boldsymbol{x}^{min}) \geq \frac{\nu}{2}\|\boldsymbol{x} - \boldsymbol{x}^{min}\|_2^2; \quad \forall \boldsymbol{x} \in \operatorname{dom}(h).$$

Lemma 2.20. [23] *Let h be an $\mathcal{L}$-smooth function. Then,*

$$h(\boldsymbol{y}) \leq h(\boldsymbol{x}) + \langle \nabla h(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle + \frac{\mathcal{L}}{2}\|\boldsymbol{y} - \boldsymbol{x}\|_2^2; \quad \forall \boldsymbol{x}, \boldsymbol{y} \in \operatorname{dom}(h).$$

Lemma 2.21. [23] *If $r > 0$, then*

$$\mathbb{E}\left[\|\boldsymbol{q}^{k+1} - \widetilde{\boldsymbol{q}}^{k+1}\|_2^2\right] \leq \frac{2\delta}{r}.$$

Proof. The expression of $\mathcal{Q}_r$, given in (2.28), is

$$\mathcal{Q}_r(\boldsymbol{q},\boldsymbol{p}^k) := \langle \boldsymbol{q}-\boldsymbol{p}^k, \nabla f(\boldsymbol{p}^k)\rangle + \frac{r}{2}\|\boldsymbol{q}-\boldsymbol{p}^k\|_2^2.$$

With Definition 2.18 we can deduce that $\mathcal{Q}_r$ is r-strongly convex.

We have that $\mathcal{Q}_r$ is r-strongly convex, $\boldsymbol{q}^{k+1} \in \text{dom}(\mathcal{Q}_r)$ and $\widetilde{\boldsymbol{q}}^{k+1}$ is the minimizer of $\mathcal{Q}_r$. Applying Lemma 2.19 we have the following inequality:

$$\mathcal{Q}_r(\boldsymbol{q}^{k+1},\boldsymbol{p}^k) - \mathcal{Q}_r(\widetilde{\boldsymbol{q}}^{k+1},\boldsymbol{p}^k) \geq \frac{r}{2}\|\boldsymbol{q}^{k+1}-\widetilde{\boldsymbol{q}}^{k+1}\|_2^2.$$

Previous expression implies that

$$\mathbb{E}\left[\mathcal{Q}_r(\boldsymbol{q}^{k+1},\boldsymbol{p}^k) - \mathcal{Q}_r(\widetilde{\boldsymbol{q}}^{k+1},\boldsymbol{p}^k)\right] \geq \frac{r}{2}\mathbb{E}\left[\|\boldsymbol{q}^{k+1}-\widetilde{\boldsymbol{q}}^{k+1}\|_2^2\right]. \tag{2.33}$$

Notice that $\widetilde{\boldsymbol{q}}^{k+1} = \underset{\boldsymbol{q}}{\text{argmin}}\ \mathcal{Q}_r(\boldsymbol{q},\boldsymbol{p}^k)$ means that

$$\mathcal{Q}_r(\widetilde{\boldsymbol{q}}^{k+1},\boldsymbol{p}^k) = \min_{\boldsymbol{q}} \mathcal{Q}_r(\boldsymbol{q},\boldsymbol{p}^k).$$

Therefore, we have the following equality:

$$\mathbb{E}\left[\mathcal{Q}_r(\boldsymbol{q}^{k+1},\boldsymbol{p}^k) - \mathcal{Q}_r(\widetilde{\boldsymbol{q}}^{k+1},\boldsymbol{p}^k)\right] = \mathbb{E}\left[\mathcal{Q}_r(\boldsymbol{q}^{k+1},\boldsymbol{p}^k) - \min_{\boldsymbol{q}} \mathcal{Q}_r(\boldsymbol{q},\boldsymbol{p}^k)\right].$$

Applying Stochastic Gradient Rule $\mathcal{R}_\delta$ (2.30) in previous identity we conclude that

$$\mathbb{E}\left[\mathcal{Q}_r(\boldsymbol{q}^{k+1},\boldsymbol{p}^k) - \mathcal{Q}_r(\widetilde{\boldsymbol{q}}^{k+1},\boldsymbol{p}^k)\right] \leq \delta. \tag{2.34}$$

Taking into account (2.33), (2.34) and that $r>0$ by hypothesis, we obtain that

$$\mathbb{E}\left[\|\boldsymbol{q}^{k+1}-\widetilde{\boldsymbol{q}}^{k+1}\|_2^2\right] \leq \frac{2\delta}{r}.$$

□

Lemma 2.22. [23] *If constants r and $\mathcal{L}$ satisfy the relation $r>\mathcal{L}$, then*

$$\mathbb{E}\left[f(\widetilde{\boldsymbol{q}}^{k+1}) + \frac{r}{2}\|\widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{p}^k\|_2^2 - \left(f(\boldsymbol{q}^{k+1}) + \frac{r}{2}\|\boldsymbol{q}^{k+1}-\boldsymbol{p}^k\|_2^2\right)\right] \geq -\tau\delta - \frac{r-\mathcal{L}}{2}\mathbb{E}\left[\|\boldsymbol{p}^k-\widetilde{\boldsymbol{q}}^{k+1}\|_2^2\right],$$

where $\tau := 1 + \dfrac{\mathcal{L}^2}{r(r-\mathcal{L})}$.

Proof. The proof of this lemma, that we can find in [23], follows from Lemma 2.17, the Stochastic Gradient Rule given by (2.30), Lemma 2.15 (Schwarz Inequality) and Lemma 2.21. □

Lemma 2.23. [23] *If constants r and $\mathcal{L}$ satisfy $r>\mathcal{L}$, then we have the following bounds:*

$$\mathbb{E}\left[f(\boldsymbol{q}^k) - f(\boldsymbol{q}^{k+1})\right] \geq \frac{r}{2}\mathbb{E}\left[\|\boldsymbol{p}^k-\boldsymbol{q}^{k+1}\|_2^2\right] + r\mathbb{E}\left[\langle \boldsymbol{p}^k-\boldsymbol{q}^k, \widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{p}^k\rangle\right] - \tau\delta, \tag{2.35a}$$

$$\mathbb{E}\left[f(\boldsymbol{q}^{min}) - f(\boldsymbol{q}^{k+1})\right] \geq \frac{r}{2}\mathbb{E}\left[\|\boldsymbol{p}^k-\boldsymbol{q}^{k+1}\|_2^2\right] + r\mathbb{E}\left[\langle \boldsymbol{p}^k-\boldsymbol{q}^{min}, \widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{p}^k\rangle\right] - \tau\delta. \tag{2.35b}$$

In both formulas we define $\tau := 1 + \frac{\mathcal{L}^2}{r(r-\mathcal{L})}$. In the second formula we have used $\boldsymbol{q}^{min}$ to denote the minimizer of f.

Proof. Take into account that

$$\widetilde{\boldsymbol{q}}^{k+1} := \underset{\boldsymbol{q}}{\operatorname{argmin}}\, \mathcal{Q}_r(\boldsymbol{q}, \boldsymbol{p}^k) \quad \text{and} \quad \mathcal{Q}_r := \langle \boldsymbol{q} - \boldsymbol{p}^k, \nabla f(\boldsymbol{p}^k)\rangle + \frac{r}{2}\|\boldsymbol{q} - \boldsymbol{p}^k\|_2^2.$$

Notice that the second addend of $\mathcal{Q}_r$ is always positive ($r > 0$), so to minimize $\mathcal{Q}_r$, the first addend has to be as negative as possible. In the justification of the update rule of GD we have seen that, using Cauchy-Schwarz-Bunyakovski Inequality (Proposition 2.6), the first addend $\langle \boldsymbol{q} - \boldsymbol{p}^k, \nabla f(\boldsymbol{p}^k)\rangle$ is as negative as possible when $\boldsymbol{q} - \boldsymbol{p}^k$ is on the direction of $-\nabla f(\boldsymbol{p}^k)$. Taking $r > 0$ properly, we have that

$$\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k = -\frac{1}{r}\nabla f(\boldsymbol{p}^k),$$

where we have taken into account the definition of $\widetilde{\boldsymbol{q}}^{k+1}$. Isolating the gradient in the previous identity we obtain

$$\nabla f(\boldsymbol{p}^k) = r(\boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}). \tag{2.36}$$

The function f is $\mathcal{L}$-smooth and $\widetilde{\boldsymbol{q}}^{k+1}, \boldsymbol{p}^k \in \operatorname{dom}(f)$. Applying Lemma 2.20 we have that

$$f(\widetilde{\boldsymbol{q}}^{k+1}) \leq f(\boldsymbol{p}^k) + \langle \nabla f(\boldsymbol{p}^k), \widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\rangle + \frac{\mathcal{L}}{2}\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2.$$

With (2.36) the previous inequality can be written as

$$-f(\widetilde{\boldsymbol{q}}^{k+1}) \geq -f(\boldsymbol{p}^k) - \langle r(\boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}), \widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\rangle - \frac{\mathcal{L}}{2}\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2. \tag{2.37}$$

As f is $\mathcal{L}$-smooth, it is differentiable. Moreover, it is convex, $\operatorname{dom}(f) \equiv \mathbb{R}^s$ is a convex set and $\boldsymbol{q}^k, \boldsymbol{p}^k \in \operatorname{dom}(f)$. Applying Lemma 2.17 we have the following inequality:

$$f(\boldsymbol{q}^k) \geq f(\boldsymbol{p}^k) + \langle \nabla f(\boldsymbol{p}^k), \boldsymbol{q}^k - \boldsymbol{p}^k\rangle.$$

Using (2.36) in this last expression we obtain

$$f(\boldsymbol{q}^k) \geq f(\boldsymbol{p}^k) + \langle r(\boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}), \boldsymbol{q}^k - \boldsymbol{p}^k\rangle. \tag{2.38}$$

Adding (2.37) and (2.38) we get

$$f(\boldsymbol{q}^k) - f(\widetilde{\boldsymbol{q}}^{k+1}) \geq -\langle r(\boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}), \widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\rangle - \frac{\mathcal{L}}{2}\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2 + \langle r(\boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}), \boldsymbol{q}^k - \boldsymbol{p}^k\rangle,$$

that can be rewritten as

$$f(\boldsymbol{q}^k) - f(\widetilde{\boldsymbol{q}}^{k+1}) \geq r\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2 - \frac{\mathcal{L}}{2}\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2 + r\langle \boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}, \boldsymbol{q}^k - \boldsymbol{p}^k\rangle.$$

Joining terms we have

$$f(\boldsymbol{q}^k) - f(\widetilde{\boldsymbol{q}}^{k+1}) \geq \left(r - \frac{\mathcal{L}}{2}\right)\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2 + r\langle \boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}, \boldsymbol{q}^k - \boldsymbol{p}^k\rangle.$$

Using this last inequality and basic properties of the expectation we obtain

$$\mathbb{E}\left[f(\boldsymbol{q}^k)\right] - \mathbb{E}\left[f(\widetilde{\boldsymbol{q}}^{k+1})\right] \geq \left(r - \frac{\mathcal{L}}{2}\right)\mathbb{E}\left[\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2\right] + r\mathbb{E}\left[\langle \boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}, \boldsymbol{q}^k - \boldsymbol{p}^k\rangle\right]. \tag{2.39}$$

As we assume that $r > \mathcal{L}$, we can use Lemma 2.22. Using basic properties of the expectation we rewrite the formula of this lemma as

$$\mathbb{E}\left[f(\widetilde{\boldsymbol{q}}^{k+1})\right] + \frac{r}{2}\mathbb{E}\left[\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2\right] - \mathbb{E}\left[f(\boldsymbol{q}^{k+1})\right] - \frac{r}{2}\mathbb{E}\left[\|\boldsymbol{q}^{k+1} - \boldsymbol{p}^k\|_2^2\right] \geq$$

$$\geq -\tau\delta - \frac{r-\mathcal{L}}{2}\mathbb{E}\left[\|\boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}\|_2^2\right], \tag{2.40}$$

where $\tau := 1 + \dfrac{\mathcal{L}^2}{r(r-\mathcal{L})}$.

Adding (2.39) and (2.40), and using basic properties of the expectation and the scalar product we obtain

$$\mathbb{E}\left[f(\boldsymbol{q}^k) - f(\boldsymbol{q}^{k+1})\right] \geq \frac{r}{2}\mathbb{E}\left[\|\boldsymbol{p}^k - \boldsymbol{q}^{k+1}\|_2^2\right] + r\mathbb{E}\left[\langle \boldsymbol{p}^k - \boldsymbol{q}^k, \widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\rangle\right] - \tau\delta,$$

which is the inequality (2.35a) of the statement.

As we have seen before, f is convex and differentiable and its domain is convex. Applying Lemma 2.17 with $\boldsymbol{q}^{min}, \boldsymbol{p}^k \in \mathrm{dom}(f)$ we have that

$$f(\boldsymbol{q}^{min}) \geq f(\boldsymbol{p}^k) + \langle \nabla f(\boldsymbol{p}^k), \boldsymbol{q}^{min} - \boldsymbol{p}^k\rangle.$$

Using (2.36) in this inequality we obtain

$$f(\boldsymbol{q}^{min}) \geq f(\boldsymbol{p}^k) + \langle r(\boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}), \boldsymbol{q}^{min} - \boldsymbol{p}^k\rangle. \tag{2.41}$$

Adding (2.37) and (2.41) we have that

$$f(\boldsymbol{q}^{min}) - f(\widetilde{\boldsymbol{q}}^{k+1}) \geq -\langle r(\boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}), \widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\rangle - \frac{\mathcal{L}}{2}\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2 + \langle r(\boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}), \boldsymbol{q}^{min} - \boldsymbol{p}^k\rangle.$$

Previous inequality can be rewritten as

$$f(\boldsymbol{q}^{min}) - f(\widetilde{\boldsymbol{q}}^{k+1}) \geq r\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2 - \frac{\mathcal{L}}{2}\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2 + r\langle \boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}, \boldsymbol{q}^{min} - \boldsymbol{p}^k\rangle,$$

and joining terms we get

$$f(\boldsymbol{q}^{min}) - f(\widetilde{\boldsymbol{q}}^{k+1}) \geq \left(r - \frac{\mathcal{L}}{2}\right)\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2 + r\langle \boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}, \boldsymbol{q}^{min} - \boldsymbol{p}^k\rangle.$$

With this inequality and basic properties of the expectation we obtain

$$\mathbb{E}\left[f(\boldsymbol{q}^{min})\right] - \mathbb{E}\left[f(\widetilde{\boldsymbol{q}}^{k+1})\right] \geq \left(r - \frac{\mathcal{L}}{2}\right)\mathbb{E}\left[\|\widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\|_2^2\right] + r\mathbb{E}\left[\langle \boldsymbol{p}^k - \widetilde{\boldsymbol{q}}^{k+1}, \boldsymbol{q}^{min} - \boldsymbol{p}^k\rangle\right]. \tag{2.42}$$

Adding (2.42) and the expression in Lemma 2.22 (we can apply it because we assume $r > \mathcal{L}$, in particular, we use the expression (2.40) that is equivalent to the one in the statement of the lemma), and using basic properties of the expectation and the scalar product we have that

$$\mathbb{E}\left[f(\boldsymbol{q}^{min}) - f(\boldsymbol{q}^{k+1})\right] \geq \frac{r}{2}\mathbb{E}\left[\|\boldsymbol{p}^k - \boldsymbol{q}^{k+1}\|_2^2\right] + r\mathbb{E}\left[\langle \boldsymbol{p}^k - \boldsymbol{q}^{min}, \widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\rangle\right] - \tau\delta,$$

where $\tau := 1 + \dfrac{\mathcal{L}^2}{r(r-\mathcal{L})}$. This is the inequality (2.35b) of the statement. □

Before proving Theorem 2.12, we restate it. Notice that in the update rule of NASGD (2.27), the approximation of the gradient is in the first step. Therefore, the following are equivalent:

- Study the sequence $\{\boldsymbol{p}^k\}_{k\geq 0}$ and prove
$$\mathbb{E}\left[f(\boldsymbol{p}^k) - f(\boldsymbol{p}^{min})\right] = \mathcal{O}(k).$$
- Study the sequence $\{\boldsymbol{q}^k\}_{k\geq 0}$ and prove
$$\mathbb{E}\left[f(\boldsymbol{q}^k) - f(\boldsymbol{q}^{min})\right] = \mathcal{O}(k).$$

Moreover, if we define $\eta := \frac{1}{r}$, where η is the step size or learning rate, we can substitute $\eta_k \equiv \eta \leq \frac{1}{\mathcal{L}}$ by $r > \mathcal{L}$ (observe that the second condition is more restrictive).

Taking into account the previous considerations, we can restate Theorem 2.12 as follows:

Theorem 2.24. Restated version of Theorem 2.12. [23]

Let f be a convex $\mathcal{L}$-smooth function. Let $\{\boldsymbol{q}^k\}_{k\geq 0}$ be the sequence obtained with NASGD algorithm (2.27) and let r and $\mathcal{L}$ satisfy $r > \mathcal{L}$. Then,

$$\mathbb{E}\left[f(\boldsymbol{q}^k) - f(\boldsymbol{q}^{min})\right] = \mathcal{O}(k),$$

where $\boldsymbol{q}^{min}$ denotes the minimizer of function f and the expectation is taken over the random mini-batch samples.

Proof of Theorem 2.24 or, equivalently, of Theorem 2.12. First of all, let us define

$$\mathcal{F}^k := \mathbb{E}\left[f(\boldsymbol{q}^k) - f(\boldsymbol{q}^{min})\right]. \tag{2.43}$$

Notice that if we use the linearity of the expectation and the previous notation, we have that

$$\mathbb{E}\left[f(\boldsymbol{q}^k) - f(\boldsymbol{q}^{k+1})\right] = \mathbb{E}\left[f(\boldsymbol{q}^k) - f(\boldsymbol{q}^{min}) + f(\boldsymbol{q}^{min}) - f(\boldsymbol{q}^{k+1})\right] =$$

$$= \mathbb{E}\left[f(\boldsymbol{q}^k) - f(\boldsymbol{q}^{min})\right] - \mathbb{E}\left[f(\boldsymbol{q}^{k+1}) - f(\boldsymbol{q}^{min})\right] = \mathcal{F}^k - \mathcal{F}^{k+1}.$$

Using this fact, we can rewrite inequality (2.35a) of Lemma 2.23 (notice that we can use Lemma 2.23 because we suppose that $r > \mathcal{L}$) as

$$\mathcal{F}^k - \mathcal{F}^{k+1} \geq \frac{r}{2}\mathbb{E}\left[\|\boldsymbol{p}^k - \boldsymbol{q}^{k+1}\|_2^2\right] + r\mathbb{E}\left[\langle \boldsymbol{p}^k - \boldsymbol{q}^k, \widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\rangle\right] - \tau\delta. \tag{2.44}$$

Using the notation introduced in (2.43) we can also rewrite inequality (2.35b) of Lemma 2.23 as

$$-\mathcal{F}^{k+1} \geq \frac{r}{2}\mathbb{E}\left[\|\boldsymbol{p}^k - \boldsymbol{q}^{k+1}\|_2^2\right] + r\mathbb{E}\left[\langle \boldsymbol{p}^k - \boldsymbol{q}^{min}, \widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\rangle\right] - \tau\delta. \tag{2.45}$$

Remember that t_k (with it we define the non-constant factor that multiplies the momentum) is given by formula (2.23). Notice that $t_k \geq 1$ for any $k \geq 0$. In effect, $t_0 = 1 \geq 1$ and for $k > 0$,

$$t_k = \frac{1 + \sqrt{1 + 4t_{k-1}^2}}{2} > \frac{1+1}{2} = 1.$$

Therefore $t_k - 1 \geq 0$ and multiplying both sides of (2.44) by $t_k - 1$, we have the following inequality:

$$(t_k - 1)\left[\mathcal{F}^k - \mathcal{F}^{k+1}\right] \geq (t_k - 1)\left[\frac{r}{2}\mathbb{E}\left[\|\boldsymbol{p}^k - \boldsymbol{q}^{k+1}\|_2^2\right] + r\mathbb{E}\left[\langle \boldsymbol{p}^k - \boldsymbol{q}^k, \widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\rangle\right] - \tau\delta\right]. \tag{2.46}$$

Adding (2.45) and (2.46) we have that

$$(t_k - 1)\mathcal{F}^k - t_k\mathcal{F}^{k+1} \geq$$

$$\geq \frac{r}{2}t_k\mathbb{E}\left[\|\boldsymbol{p}^k - \boldsymbol{q}^{k+1}\|_2^2\right] + r\mathbb{E}\left[\langle (\boldsymbol{p}^k - \boldsymbol{q}^k)(t_k - 1) + (\boldsymbol{p}^k - \boldsymbol{q}^{min}), \widetilde{\boldsymbol{q}}^{k+1} - \boldsymbol{p}^k\rangle\right] - t_k\tau\delta.$$

As $r > 0$ (observe that $r > \mathcal{L}$ and as $\mathcal{L}$ is a Lipschitz constant, it is positive, so $r > 0$), we can

rewrite previous expression as

$$\frac{2}{r}\left[(t_k-1)\mathcal{F}^k - t_k\mathcal{F}^{k+1}\right] \geq$$
$$\geq t_k\mathbb{E}\left[\|\boldsymbol{p}^k-\boldsymbol{q}^{k+1}\|_2^2\right] + 2\mathbb{E}\left[\langle\widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{p}^k, t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\rangle\right] - \frac{2t_k\tau\delta}{r}.$$

Multiplying both sides of this inequality by t_k (notice that $t_k>0$, $\forall k\geq 0$) we obtain

$$\frac{2}{r}\left[t_k(t_k-1)\mathcal{F}^k - t_k^2\mathcal{F}^{k+1}\right] \geq$$
$$\geq t_k^2\mathbb{E}\left[\|\boldsymbol{q}^{k+1}-\boldsymbol{p}^k\|_2^2\right] + 2t_k\mathbb{E}\left[\langle\widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{p}^k, t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\rangle\right] - \frac{2t_k^2\tau\delta}{r}. \quad (2.47)$$

Taking into account the recursive formula of t_k (2.23) we have that for any $k>0$

$$t_k(t_k-1) = \frac{1+\sqrt{1+4t_{k-1}^2}}{2}\left(\frac{1+\sqrt{1+4t_{k-1}^2}}{2}-1\right) = \frac{\sqrt{1+4t_{k-1}^2}+1}{2}\frac{\sqrt{1+4t_{k-1}^2}-1}{2} = t_{k-1}^2.$$

So, $t_k(t_k-1)=t_{k-1}^2$ for $k>0$. Using this expression we can rewrite (2.47) as

$$\frac{2}{r}\left[t_{k-1}^2\mathcal{F}^k - t_k^2\mathcal{F}^{k+1}\right] \geq$$
$$\geq \mathbb{E}\left[\|t_k\boldsymbol{q}^{k+1}-t_k\boldsymbol{p}^k\|_2^2\right] + 2t_k\mathbb{E}\left[\langle\widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{p}^k, t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\rangle\right] - \frac{2t_k^2\tau\delta}{r}. \quad (2.48)$$

Let us study right-hand side of previous inequality. It is obvious that

$$\mathbb{E}\left[\|t_k\boldsymbol{q}^{k+1}-t_k\boldsymbol{p}^k\|_2^2\right] + 2t_k\mathbb{E}\left[\langle\widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{p}^k, t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\rangle\right] - \frac{2t_k^2\tau\delta}{r} =$$
$$= \mathbb{E}\left[\|t_k\boldsymbol{q}^{k+1}-t_k\boldsymbol{p}^k\|_2^2\right] + 2t_k\mathbb{E}\left[\langle\widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{p}^k+\boldsymbol{q}^{k+1}-\boldsymbol{q}^{k+1}, t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\rangle\right] - \frac{2t_k^2\tau\delta}{r} =$$
$$= \mathbb{E}\left[\|t_k\boldsymbol{q}^{k+1}-t_k\boldsymbol{p}^k\|_2^2\right] + 2t_k\mathbb{E}\left[\langle\boldsymbol{q}^{k+1}-\boldsymbol{p}^k, t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\rangle\right] +$$
$$+2t_k\mathbb{E}\left[\langle\widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{q}^{k+1}, t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\rangle\right] - \frac{2t_k^2\tau\delta}{r}. \quad (2.49)$$

If we take

$$\boldsymbol{a} = t_k\boldsymbol{q}^{k+1}-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min} \quad \text{and} \quad \boldsymbol{b} = t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}$$

in Lemma 2.16, we obtain

$$\|t_k\boldsymbol{q}^{k+1}-t_k\boldsymbol{p}^k\|_2^2 + 2t_k\langle\boldsymbol{q}^{k+1}-\boldsymbol{p}^k, t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\rangle =$$
$$= \|t_k\boldsymbol{q}^{k+1}-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\|_2^2 - \|t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\|_2^2.$$

Taking into account this expression we have the following equality for the expectation:

$$\mathbb{E}\left[\|t_k\boldsymbol{q}^{k+1}-t_k\boldsymbol{p}^k\|_2^2\right] + \mathbb{E}\left[2t_k\langle\boldsymbol{q}^{k+1}-\boldsymbol{p}^k, t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\rangle\right] =$$
$$= \mathbb{E}\left[\|t_k\boldsymbol{q}^{k+1}-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\|_2^2\right] - \mathbb{E}\left[\|t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\|_2^2\right]. \quad (2.50)$$

Using (2.50), we have that expression (2.49) (remember that it is the right-hand side of

inequality (2.48)) is equal to

$$\mathbb{E}\left[\|t_k\boldsymbol{q}^{k+1}-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\|_2^2\right]-\mathbb{E}\left[\|t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\|_2^2\right]+$$
$$+2t_k\mathbb{E}\left[\langle\widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{q}^{k+1},t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\rangle\right]-\frac{2t_k^2\tau\delta}{r}. \quad (2.51)$$

The sequence $\{\boldsymbol{p}^k\}_{k\geq 0}$ is obtained applying NASGD given in (2.27), or equivalently, in (2.32). In particular, $\boldsymbol{p}^k$ is calculated in the second step of the update rule:

$$\boldsymbol{p}^k=\boldsymbol{q}^k+\frac{t_{k-1}-1}{t_k}(\boldsymbol{q}^k-\boldsymbol{q}^{k-1}).$$

Previous equality can be easily rewritten as

$$t_k\boldsymbol{p}^k-(t_k-1)\boldsymbol{q}^k=t_{k-1}\boldsymbol{q}^k-(t_{k-1}-1)\boldsymbol{q}^{k-1}. \quad (2.52)$$

With expression (2.52) we have that (2.51) is equal to

$$\mathbb{E}\left[\|t_k\boldsymbol{q}^{k+1}-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\|_2^2\right]-\mathbb{E}\left[\|t_{k-1}\boldsymbol{q}^k-(t_{k-1}-1)\boldsymbol{q}^{k-1}-\boldsymbol{q}^{min}\|_2^2\right]+$$
$$+2t_k\mathbb{E}\left[\langle\widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{q}^{k+1},t_{k-1}\boldsymbol{q}^k-(t_{k-1}-1)\boldsymbol{q}^{k-1}-\boldsymbol{q}^{min}\rangle\right]-\frac{2t_k^2\tau\delta}{r}.$$

Last expression is equal to the right-hand side of (2.48). Therefore, inequality (2.48) can be written as

$$\frac{2}{r}\left[t_{k-1}^2\mathcal{F}^k-t_k^2\mathcal{F}^{k+1}\right]\geq$$
$$\geq\mathbb{E}\left[\|t_k\boldsymbol{q}^{k+1}-(t_k-1)\boldsymbol{q}^k-\boldsymbol{q}^{min}\|_2^2\right]-\mathbb{E}\left[\|t_{k-1}\boldsymbol{q}^k-(t_{k-1}-1)\boldsymbol{q}^{k-1}-\boldsymbol{q}^{min}\|_2^2\right]+$$
$$+2t_k\mathbb{E}\left[\langle\widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{q}^{k+1},t_{k-1}\boldsymbol{q}^k-(t_{k-1}-1)\boldsymbol{q}^{k-1}-\boldsymbol{q}^{min}\rangle\right]-\frac{2t_k^2\tau\delta}{r}.$$

If we define

$$u^k:=\mathbb{E}\left[\|t_{k-1}\boldsymbol{q}^k-(t_{k-1}-1)\boldsymbol{q}^{k-1}-\boldsymbol{q}^{min}\|_2^2\right],$$

we can express last inequality as

$$\frac{2}{r}\left[t_{k-1}^2\mathcal{F}^k-t_k^2\mathcal{F}^{k+1}\right]\geq$$
$$\geq u^{k+1}-u^k+2t_k\mathbb{E}\left[\langle\widetilde{\boldsymbol{q}}^{k+1}-\boldsymbol{q}^{k+1},t_{k-1}\boldsymbol{q}^k-(t_{k-1}-1)\boldsymbol{q}^{k-1}-\boldsymbol{q}^{min}\rangle\right]-\frac{2t_k^2\tau\delta}{r}.$$

Previous formula is equivalent to

$$u^{k+1}+\frac{2t_k^2}{r}\mathcal{F}^{k+1}\leq$$
$$\leq u^k+\frac{2t_{k-1}^2}{r}\mathcal{F}^k+2t_k\mathbb{E}\left[\langle\boldsymbol{q}^{k+1}-\widetilde{\boldsymbol{q}}^{k+1},t_{k-1}\boldsymbol{q}^k-(t_{k-1}-1)\boldsymbol{q}^{k-1}-\boldsymbol{q}^{min}\rangle\right]+\frac{2t_k^2\tau\delta}{r}. \quad (2.53)$$

Let us take

$$\boldsymbol{a}=\boldsymbol{q}^{k+1}-\widetilde{\boldsymbol{q}}^{k+1},\quad \boldsymbol{b}=t_{k-1}\boldsymbol{q}^k-(t_{k-1}-1)\boldsymbol{q}^{k-1}-\boldsymbol{q}^{min}\quad\text{and}\quad\lambda=\frac{1}{t_k}>0$$

in Schwarz Inequality (Lemma 2.15), then we have that

$$2t_k\langle\boldsymbol{q}^{k+1}-\widetilde{\boldsymbol{q}}^{k+1},t_{k-1}\boldsymbol{q}^k-(t_{k-1}-1)\boldsymbol{q}^{k-1}-\boldsymbol{q}^{min}\rangle\leq$$
$$\leq t_k^2\|\boldsymbol{q}^{k+1}-\widetilde{\boldsymbol{q}}^{k+1}\|_2^2+\|t_{k-1}\boldsymbol{q}^k-(t_{k-1}-1)\boldsymbol{q}^{k-1}-\boldsymbol{q}^{min}\|_2^2.$$

Notice that we have multiplied Schwarz Inequality by $2t_k$, but as $2t_k > 0$, the inequality holds. Using the expression we have just obtained, we have the following relation for the expectation:

$$\mathbb{E}\left[2t_k\langle \boldsymbol{q}^{k+1} - \widetilde{\boldsymbol{q}}^{k+1}, t_{k-1}\boldsymbol{q}^k - (t_{k-1}-1)\boldsymbol{q}^{k-1} - \boldsymbol{q}^{min}\rangle\right] \leq$$

$$\leq t_k^2 \mathbb{E}\left[\|\boldsymbol{q}^{k+1} - \widetilde{\boldsymbol{q}}^{k+1}\|_2^2\right] + \mathbb{E}\left[\|t_{k-1}\boldsymbol{q}^k - (t_{k-1}-1)\boldsymbol{q}^{k-1} - \boldsymbol{q}^{min}\|_2^2\right].$$

Applying Lemma 2.21 (remember that $r > 0$) in the first addend of the right-hand side of previous expression and defining

$$R = \sup\left\{\left\|\boldsymbol{q}^k - \frac{t_{k-1}-1}{t_{k-1}}\boldsymbol{q}^{k-1} - \frac{1}{t_{k-1}}\boldsymbol{q}^{min}\right\|_2\right\}$$

for the second addend, we have that last inequality can be reformulated as

$$\mathbb{E}\left[2t_k\langle \boldsymbol{q}^{k+1} - \widetilde{\boldsymbol{q}}^{k+1}, t_{k-1}\boldsymbol{q}^k - (t_{k-1}-1)\boldsymbol{q}^{k-1} - \boldsymbol{q}^{min}\rangle\right] \leq t_k^2\frac{2\delta}{r} + t_{k-1}^2 R^2. \tag{2.54}$$

Applying the bound given in (2.54) on the right-hand side of inequality (2.53) we obtain that

$$u^{k+1} + \frac{2t_k^2}{r}\mathcal{F}^{k+1} \leq u^k + \frac{2t_{k-1}^2}{r}\mathcal{F}^k + \frac{2t_k^2\delta}{r} + t_{k-1}^2 R^2 + \frac{2t_k^2\tau\delta}{r}.$$

Defining

$$\xi_k := u^k + \frac{2t_{k-1}^2}{r}\mathcal{F}^k \quad \text{and} \quad \tau^* = 1 + \tau,$$

we can rewrite the previous inequality as

$$\xi_{k+1} \leq \xi_k + t_k^2\frac{2\tau^*\delta}{r} + t_{k-1}^2 R^2.$$

According to [23], this formula can be rewritten as

$$\xi_{k+1} \leq \left(\frac{2\tau^*\delta}{r} + R^2\right)\frac{(k+1)^3}{3}$$

and, in addition, we have that

$$\xi_{k+1} \geq \frac{(k+1)^2\mathcal{F}^{k+1}}{4}.$$

With these bounds we can deduce, taking into account the definition of $\mathcal{F}^k$ (2.43), that

$$\mathbb{E}\left[f(\boldsymbol{q}^k) - f(\boldsymbol{q}^{min})\right] = \mathcal{O}(k),$$

as we wanted to prove. □

We have proved that NASGD accumulates error even when we take a function f that is convex and $\mathcal{L}$-smooth. So, if we try to accelerate SG with non-constant Nesterov momentum (this is what we were trying to do with NASGD), we do not obtain good results. Recently, a new algorithm to speed up SG has been introduced in [23]. It is obtained using SRNAG (we have seen its expression for GD in (2.26)) with SG. This new scheme is called Scheduled Restart Stochastic Gradient Descent (SRSGD). According to [23] (article where the algorithm is explained), the advantages of SRSGD can be summarized in four points:

- SRSGD can significantly accelerate the training process of DNNs.
- A DNN generalizes better if it has been trained using SRSGD.

- SRSGD helps to reduce overfitting (this concept will be defined in Section 2.3) when the DNN has many hidden layers.
- It is easy to implement SRSGD, only few lines of the implementation of SG have to be modified.

The update rule of SRSGD, that is obtained replacing $\nabla f(\boldsymbol{p}^k)$ by $\frac{1}{N}\sum_{m=1}^{N} \nabla f_{\{\boldsymbol{x}^{\{\alpha_m\}}\}}(\boldsymbol{p}^k)$ in (2.26), is given by

$$\boldsymbol{q}^{k+1} = \boldsymbol{p}^k - \eta_k \frac{1}{N}\sum_{m=1}^{N} \nabla f_{\{\boldsymbol{x}^{\{\alpha_m\}}\}}(\boldsymbol{p}^k),$$

$$\boldsymbol{p}^{k+1} = \boldsymbol{q}^{k+1} + \frac{k \mod F_\beta}{(k \mod F_\beta)+3}(\boldsymbol{q}^{k+1} - \boldsymbol{q}^k).$$

2.2.4 Back Propagation

It is clear that if we train a neural network using one of the previous optimization methods, we need to compute

$$\nabla f_{\{\boldsymbol{x}^{\{\alpha_m\}}\}}(\boldsymbol{p}^k),$$

which is the gradient of the loss function at $\boldsymbol{p}^k$ for the training point $\boldsymbol{x}^{\{\alpha_m\}}$. Let us see how to calculate it properly.

Notation 2.25. We consider again that the loss function depends on matrices for the weights (we have represented them by W) and on vectors for the biases (we have denoted them by $\boldsymbol{b}$). So, now, instead of using the notation $f(\boldsymbol{p})$ (where $\boldsymbol{p}$ is a vector with all the weights and biases) we return to the first notation we have used in this chapter, that is, $f(W, \boldsymbol{b})$.

Back Propagation (shortened as backprop) is a mathematical technique that helps us to compute these partial derivatives easily (in fact, the development of Back Propagation helped to arrive to the end of the AI Winter). It exploits the forward structure of the neural network to be able to compute the partial derivatives in the backward direction.

To understand how backprop works we consider that our loss function is the quadratic cost function given by (2.7) multiplied by $\frac{1}{2}$ (that is, we consider the formula (2.8)).

Notation 2.26. For the sake of notation, in what follows we use $\widetilde{f}$ to denote $f_{\{\boldsymbol{x}^{\{\alpha_m\}}\}}(W, \boldsymbol{b})$.

Following previous notation and taking into account the expression we consider for the loss function, we have that

$$\widetilde{f} = \frac{1}{2}\|\boldsymbol{y}^{\{m\}} - \widehat{\boldsymbol{y}}^{\{m\}}\|_2^2. \tag{2.55}$$

Let us remember that $\boldsymbol{y}^{\{m\}}$ is the target output (when the input is the data point $\boldsymbol{x}^{\{m\}}$) and $\widehat{\boldsymbol{y}}^{\{m\}} = \boldsymbol{a}^{[L]}$ is the output of our FNN with fully-connected layers. Thus, the only dependence on the weights and biases of f is via the term $\widehat{\boldsymbol{y}}^{\{m\}} = \boldsymbol{a}^{[L]}$.

To formulate the main result that helps us to understand how Back Propagation works, we need to introduce some notation.

Notation 2.27.

$$\boldsymbol{z}^{[l]} = W^{[l]}\boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]} \in \mathbb{R}^{n_l} \ ; \ l = 2, \ldots, L, \tag{2.56}$$

where $z_j^{[l]}$ can be understood as the weighted input for neuron j at layer l. With this notation, the general architecture of FNNs with fully-connected layers given in (2.2) can be rewritten as

$$\begin{cases} \boldsymbol{a}^{[1]} &= \boldsymbol{x} \in \mathbb{R}^{n_1}, \\ \boldsymbol{a}^{[l]} &= \boldsymbol{\mathcal{A}}\left(\boldsymbol{z}^{[l]}\right) \in \mathbb{R}^{n_l}; \quad l = 2, \ldots, L. \end{cases} \tag{2.57}$$

We use $\xi_j^{[l]}$ to denote the partial derivative of $\widetilde{f}$ respect to the weighted input for neuron j at layer l, that is,

$$\xi_j^{[l]} = \frac{\partial \widetilde{f}}{\partial z_j^{[l]}} \text{ for } 1 \leq j \leq n_l,\ 2 \leq l \leq L. \tag{2.58}$$

It is usually called the error in the j neuron at layer l. This name is due to the fact that the loss function can only reach a minimum if all its derivatives are zero. In [22], it is recommended to think about it as a *measure of the sensitivity of the loss function to the weighted input for neuron j at layer l.*

The main result needed to understand Back Propagation is the next lemma.

Lemma 2.28. [22] *Let us consider that we have a general Feed-forward Neural Network with fully-connected layers and that the loss function is the quadratic cost function given by* (2.55)*. Then,*

$$\boldsymbol{\xi}^{[l]} = \begin{cases} \boldsymbol{\mathcal{A}}'(\boldsymbol{z}^{[l]}) \otimes (W^{[l+1]})^T \boldsymbol{\xi}^{[l+1]} & \text{for } l = 2, \dots, L-1, \\ \boldsymbol{\mathcal{A}}'(\boldsymbol{z}^{[L]}) \otimes (\boldsymbol{a}^{[L]} - \boldsymbol{y}^{\{m\}}) & \text{for } l = L. \end{cases} \tag{2.59a}$$

$$\frac{\partial \widetilde{f}}{\partial b_j^{[l]}} = \xi_j^{[l]} \quad \text{for } l = 2, \dots, L. \tag{2.59b}$$

$$\frac{\partial \widetilde{f}}{\partial w_{ji}^{[l]}} = \xi_j^{[l]} a_i^{[l-1]} \quad \text{for } l = 2, \dots, L. \tag{2.59c}$$

We have used $\otimes$ to denote the componentwise multiplication.

Remark 2.29. We use $(\,\cdot\,)'$ to denote the derivative respect to $\boldsymbol{z}^{[l]}$ $(l = 2, \dots, L)$. In the previous lemma we have $\boldsymbol{\mathcal{A}}'(\boldsymbol{z}^{[l]})$ where $\boldsymbol{z}^{[l]} \in \mathbb{R}^{n_l}$. Taking into account the way in which $\boldsymbol{\mathcal{A}}$ is applied to a vector (see (2.1)), we have that

$$\boldsymbol{\mathcal{A}}'(\boldsymbol{z}^{[l]}) = \frac{\partial \boldsymbol{\mathcal{A}}(\boldsymbol{z}^{[l]})}{\partial \boldsymbol{z}^{[l]}} = \left(\frac{\partial \mathcal{A}(z_1^{[l]})}{\partial z_1^{[l]}}, \dots, \frac{\partial \mathcal{A}(z_{n_l}^{[l]})}{\partial z_{n_l}^{[l]}} \right)^T.$$

Proof of Lemma 2.28. The proof of this lemma is based on the chain rule for the derivation.

Let us start proving (2.59a). For $l = L$ and for each $j = 1, \dots, n_L$, using (2.58) and the chain rule, we have that

$$\xi_j^{[L]} = \frac{\partial \widetilde{f}}{\partial z_j^{[L]}} = \frac{\partial \widetilde{f}}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}}. \tag{2.60}$$

The expression of $\widetilde{f}$ is given in (2.55), so

$$\frac{\partial \widetilde{f}}{\partial a_j^{[L]}} = \frac{\partial \left(\frac{1}{2} \| \boldsymbol{y}^{\{m\}} - \boldsymbol{a}^{[L]} \|_2^2 \right)}{\partial a_j^{[L]}} = \frac{\partial \left(\frac{1}{2} \sum_{n=1}^{n_L} \left(y_n^{\{m\}} - a_n^{[L]} \right)^2 \right)}{\partial a_j^{[L]}},$$

where we have applied the definition of the Euclidean norm for a vector. Notice that if we compute the partial derivative, all the addends vanish except the j-th one. Observe that it is here (in the process of computing the derivative of the j-th term) where the constant $\frac{1}{2}$ is useful, thanks to it we avoid having the constant 2 multiplying the derivative. We conclude that

$$\frac{\partial \widetilde{f}}{\partial a_j^{[L]}} = a_j^{[L]} - y_j^{\{m\}}. \tag{2.61}$$

Taking into account the relation between $\boldsymbol{a}^{[L]}$ and $\boldsymbol{z}^{[L]}$ given in (2.57), we have that

$$\frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \mathcal{A}'(z_j^{[L]}). \tag{2.62}$$

Replacing in (2.60) the corresponding expressions obtained in (2.61) and (2.62), we get

$$\xi_j^{[L]} = \left(a_j^{[L]} - y_j^{\{m\}}\right) \mathcal{A}'(z_j^{[L]}).$$

The last equality holds for any $j = 1, \ldots, n_L$, so using componentwise product we obtain (2.59a) for $l = L$.

Let us consider now $l = 2, \ldots, L-1$. Applying expression (2.58) and using the chain rule taking into account that the output of the neural network is computed forwards (that is to say, $\boldsymbol{z}^{[l+1]}$ is obtained from $\boldsymbol{z}^{[l]}$) we have that

$$\xi_j^{[l]} = \frac{\partial \widetilde{f}}{\partial z_j^{[l]}} = \sum_{n=1}^{n_{l+1}} \frac{\partial \widetilde{f}}{\partial z_n^{[l+1]}} \frac{\partial z_n^{[l+1]}}{\partial z_j^{[l]}} = \sum_{n=1}^{n_{l+1}} \xi_n^{[l+1]} \frac{\partial z_n^{[l+1]}}{\partial z_j^{[l]}}. \tag{2.63}$$

By (2.56) and (2.57) we have that

$$\boldsymbol{z}^{[l+1]} = W^{[l+1]} \boldsymbol{\mathcal{A}}(\boldsymbol{z}^{[l]}) + \boldsymbol{b}^{[l+1]},$$

whose componentwise version is

$$z_n^{[l+1]} = \sum_{r=1}^{n_l} w_{nr}^{[l+1]} \mathcal{A}(z_r^{[l]}) + b_n^{[l+1]} \quad \text{for } n = 1, \ldots, n_{l+1}.$$

If we derive respect to $z_j^{[l]}$ we have that

$$\frac{\partial z_n^{[l+1]}}{\partial z_j^{[l]}} = \frac{\partial}{\partial z_j^{[l]}} \left[\sum_{r=1}^{n_l} w_{nr}^{[l+1]} \mathcal{A}(z_r^{[l]}) + b_n^{[l+1]}\right] = w_{nj}^{[l+1]} \mathcal{A}'(z_j^{[l]}) \quad \text{for } n = 1, \ldots, n_{l+1}. \tag{2.64}$$

Substituting (2.64) in (2.63) we obtain

$$\xi_j^{[l]} = \sum_{n=1}^{n_{l+1}} \xi_n^{[l+1]} w_{nj}^{[l+1]} \mathcal{A}'(z_j^{[l]}) = \mathcal{A}'(z_j^{[l]}) \left((W^{[l+1]})^T \boldsymbol{\xi}^{[l+1]}\right)_j.$$

This last expression holds for $j = 1, \ldots, n_l$ and for $l = 2, \ldots, L-1$, so, taking into account how componentwise product works, we have proved (2.59a) for $l = 2, \ldots, L-1$.

Let us prove (2.59b). Let $l = 2, \ldots, L$. Using the chain rule we have that

$$\frac{\partial \widetilde{f}}{\partial b_j^{[l]}} = \frac{\partial \widetilde{f}}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}}. \tag{2.65}$$

By (2.56) and (2.57) we obtain

$$\boldsymbol{z}^{[l]} = W^{[l]} \mathcal{A}(\boldsymbol{z}^{[l-1]}) + \boldsymbol{b}^{[l]},$$

whose componentwise version is

$$z_j^{[l]} = \left(W^{[l]} \mathcal{A}(\boldsymbol{z}^{[l-1]})\right)_j + b_j^{[l]}.$$

Notice that $\boldsymbol{z}^{[l-1]}$ does not depend on $b_j^{[l]}$ (remember that the expressions of $\boldsymbol{z}^{[l]}$ for $l=2,\dots,L$ are computed forwards). Therefore, we conclude that the derivative respect to $b_j^{[l]}$ of the last expression is

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1. \tag{2.66}$$

From (2.66) and the expression of $\xi_j^{[l]}$ given in (2.58) we have that (2.65) can be rewritten as

$$\frac{\partial \widetilde{f}}{\partial b_j^{[l]}} = \xi_j^{[l]},$$

which gives us (2.59b).

Finally, let us prove (2.59c). Let $l=2,\dots,L$. Applying the chain rule we have that

$$\frac{\partial \widetilde{f}}{\partial w_{ji}^{[l]}} = \sum_{n=1}^{n_l} \frac{\partial \widetilde{f}}{\partial z_n^{[l]}} \frac{\partial z_n^{[l]}}{\partial w_{ji}^{[l]}}. \tag{2.67}$$

By (2.56) we have that

$$\boldsymbol{z}^{[l]} = W^{[l]}\boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]},$$

whose componentwise version is

$$z_n^{[l]} = \sum_{r=1}^{n_{l-1}} w_{nr}^{[l]} a_r^{[l-1]} + b_n^{[l]} \quad \text{for } n=1,\dots,n_l.$$

Computing the derivative respect to $w_{ji}^{[l]}$ of the previous expression, we get

$$\frac{\partial z_n^{[l]}}{\partial w_{ji}^{[l]}} = \begin{cases} a_i^{[l-1]} & \text{if } n=j, \\ 0 & \text{if } n\neq j. \end{cases}$$

Taking into account this last equality and the definition of $\xi_j^{[l]}$ given in (2.58) we have that (2.67) is rewritten as

$$\frac{\partial \widetilde{f}}{\partial w_{ji}^{[l]}} = \frac{\partial \widetilde{f}}{\partial z_j^{[l]}} a_i^{[l-1]} = \xi_j^{[l]} a_i^{[l-1]},$$

which is (2.59c). □

With Lemma 2.28 it is easy to see that Back Propagation exploits the forward structure of the neural network to be able to compute the partial derivatives in the backward direction. With (2.56) and (2.57), which go forwards on the network, we can obtain $\boldsymbol{a}^{[L]}$, that is, the output of the FNN with fully-connected layers. As we know this value, we can obtain $\boldsymbol{\xi}^{[L]}$ applying (2.59a) of Lemma 2.28 with $l=L$. As we have $\boldsymbol{\xi}^{[L]}$, we can compute $\boldsymbol{\xi}^{[l]}$ for $l=2,\dots,L-1$ backwards with expression (2.59a) (that is to say, we calculate $\boldsymbol{\xi}^{[L-1]}$ using $\boldsymbol{\xi}^{[L]}$, then we obtain $\boldsymbol{\xi}^{[L-2]}$ from $\boldsymbol{\xi}^{[L-1]}$,...). The partial derivatives respect to the weights and biases can be computed with (2.59b) and (2.59c).

Remark 2.30. Observe that in expression (2.59a) of Lemma 2.28 we have to compute $\mathcal{A}'$, where $\mathcal{A}$ is the activation function. If we take, for example, the sigmoid function given in (2.4), it is easy to compute $\mathcal{A}'$:

$$\mathcal{A}'(x) = \sigma'(x) = \left(\frac{1}{1+e^{-x}}\right)' = \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(x)(1-\sigma(x)).$$

Notice that if we call $D^{[l]}$ the diagonal matrix with dimension $n_l \times n_l$ whose (i,i) entry is $\mathcal{A}'(z_i^{[l]})$, we can replace componentwise product in (2.59a) of Lemma 2.28 by a matrix product. The expression is rewritten as

$$\boldsymbol{\xi}^{[l]} = \begin{cases} D^{[l]}(W^{[l+1]})^T \boldsymbol{\xi}^{[l+1]} & \text{for } l = 2, \ldots, L-1, \\ D^{[L]}(\boldsymbol{a}^{[L]} - \boldsymbol{y}^{\{m\}}) & \text{for } l = L. \end{cases}$$

Using this matrix expression we can write the following algorithm [22] for the training process of an FNN with fully-connected layers when we use Stochastic Gradient with replacement and mini-batch size equal to 1 (see (2.16)) and Back Propagation:

```
Initialize W^[l] and b^[l] for l = 2,...,L
for k = 1,...,maxiter
    m is chosen randomly from {1,...,M}
    x^{m} is the training point
    a^[1] = x^{m}
    for l = 2,...,L
        z^[l] = W^[l] a^[l-1] + b^[l]
        a^[l] = A(z^[l])
        D^[l] = diag(A'(z^[l]))
    end for
    ξ^[L] = D^[L] (a^[L] - y^{m})
    for l = L-1,...,2
        ξ^[l] = D^[l] (W^[l+1])^T ξ^[l+1]
    end for
    for l = L,...,2
        W^[l] = W^[l] - η ξ^[l] a^[l-1]^T
        b^[l] = b^[l] - η ξ^[l]
    end for
end for
```

2.3 Problems with Training

Sometimes during training, the DNN does not learn correctly: it has focused on some details that are not representative of the general input data and it is not able to generalize (overfitting) or it cannot learn from the given training data (underfitting). In this section we explain these problems and how to detect and avoid them. Information about overfitting and underfitting can be found in [5], [9], [22] and [29].

2.3.1 Overfitting

When a network that has already been trained is not able to generalize but it fits perfectly the training data, we say that overfitting occurs. Intuitively, overfitting is due to the fact that the network has learnt in a wrong way: it has focused on details of the training set that are not representative of all possible inputs of the network. In Figure 2.4 we have a visual representation of overfitting.

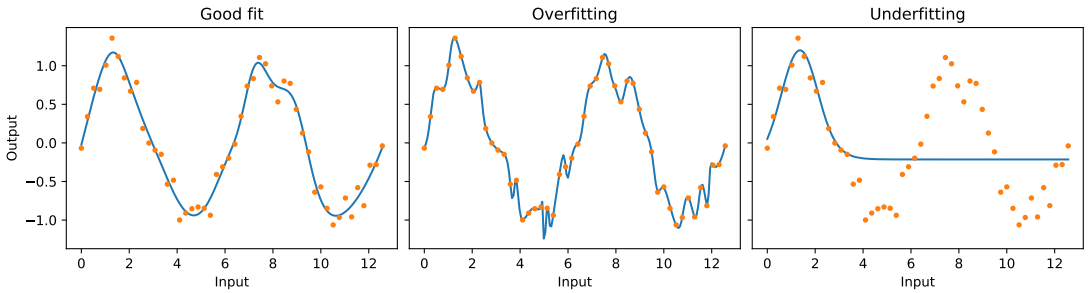

Figure 2.4: In all the pictures we have in orange the training points and in blue the result given by a neural network. From left to right, graph obtained when the neural network has learnt correctly, when overfitting occurs and when underfitting arises.

If we want to detect if overfitting occurs, we have to split the available labeled data into two different groups: training data and test data. Test data, that has to be preprocessed as we have done with training data, is the main tool that helps us to notice if overfitting arises. If the value of the loss function for test data is relatively high and this value for training data is small, the network is not able to generalize and we can conclude that overfitting is happening.

The term regularization describes the different techniques used to avoid overfitting by rewarding smoothness. Some of the most popular regularization techniques are early stopping, dropout, ℓ_1 and ℓ_2 regularization, max-norm regularization and data augmentation.

→ Early stopping.

If we want to use early stopping, we need to split the labeled data into three different sets (they do not have data in common): training set, test set and validation set. As we have seen, the training set is used to update the weights and biases during training. However, validation set is not used to change the values of weights and biases, it is used to check if the DNN is learning correctly.

During the process of training (for example, after every update of the weights and biases), using the loss function, the target outputs of the data of the validation set are compared with the outputs obtained applying the current neural network to measure how well the trained weights and biases fit unseen data (that is, data not used for training). If the network is learning correctly, the value of the loss function decreases at each application of the validation data set. Nevertheless, if overfitting occurs, at some point of the training process, that value does not decrease (even it can increase) and it is better to stop training.

Early stopping is usually used together with other techniques that also help us to avoid overfitting.

→ Dropout.

Dropout is a regularization technique that at first can seem to be extreme, but it works quite well. It was proposed in *Improving Neural Networks by Preventing Co-adaptation of Feature Detectors* [30] by G.E. Hinton et al. in 2012. In 2014, in the article *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* [31], N. Srivastava et al. showed that this technique can improve the performance of neural networks created to carry out vision and speech recognition tasks. Dropout consists in deleting the neurons (and its corresponding connections) of the input and hidden layers (neurons of the output layer cannot be removed) with probability P at every training step and training the resultant network. After training, the obtained weights and biases have to be multiplied by $1-P$. The probability P is a parameter whose value is not computed during training, it is known as dropout rate. After training, we consider the entire DNN.

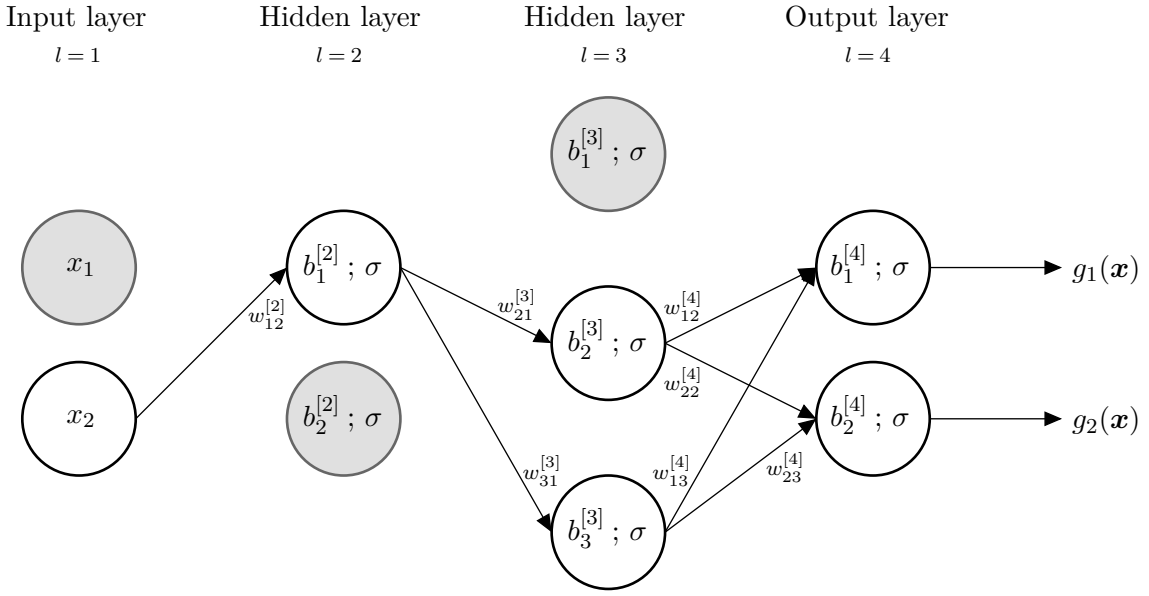

Figure 2.5: Illustrative example of dropout in the FNN with fully-connected layers of Figure 2.2.

To remove neurons and connections and to train the resulting network make neurons more robust and less dependent on their neighbours. This helps the network to learn to generalize and overfitting is avoided. If we use dropout and we detect that overfitting is still happening, we could increase P and try again.

Example 2.31. If we have the FNN with fully-connected layers of Figure 2.2 and at training step k the removed neurons are the first one of the input layer, the second one of the first hidden layer and the first one of the second hidden layer (the neurons of each layer are numbered from top to bottom and the layers from left to right), the neural network that we have to train is in Figure 2.5.

$\rightarrow$ ℓ_1 and ℓ_2 regularization.

To avoid overfitting we can use ℓ_1 and ℓ_2 regularization. Their goal is to force weights to be small (conditions are not usually imposed on the biases). To do this, some special terms are added to the loss function $f(W, \boldsymbol{b})$.

- ℓ_1 regularization. The term we add to the loss function is of the form

$$\beta \sum_{l=2}^{L} \sum_{j=1}^{n_l} \sum_{i=1}^{n_{l-1}} |w_{ji}^{[l]}|.$$

  That is to say, the extra addend is proportional to the sum of the absolute value of all the weights of the DNN.

- ℓ_2 regularization. The special addend inserted in the loss function is of the form

$$\beta \sum_{l=2}^{L} \sum_{j=1}^{n_l} \sum_{i=1}^{n_{l-1}} \left(w_{ji}^{[l]}\right)^2.$$

  It is obvious that it is proportional to the sum of the square of all the weights of the neural network.

In both cases, we have a parameter $\beta > 0$ known as regularization hyperparameter. It is a measure of the strength of the regularization: if β is small, we give more importance to minimizing the original expression of the loss function, but if β is big, it is more important to minimize the extra addend, which is equivalent to say that small weights are required.

Remark 2.32. This regularization terms are only added to the loss function during training. In other cases, as in test process, the loss function is used with its original expression.

→ Max-norm regularization.

Max-norm regularization consists in imposing that the vector of weights of each neuron has to verify that its Euclidean norm is not greater than a certain value r known as max-norm hyperparameter. That is to say, if we denote by $W^{[l,j]}$ the vector of weights of the input connections of the j-th neuron at layer l, this vector has to satisfy

$$\|W^{[l,j]}\|_2 = \left[\sum_{i=1}^{n_{l-1}} \left(w_{ji}^{[l]}\right)^2\right]^{1/2} \leq r. \tag{2.68}$$

To impose this condition, after each update of weights and biases, we have to check if (2.68) is verified for each neuron at each layer. If it is not, we replace $W^{[l,j]}$ by

$$\frac{W^{[l,j]}}{\|W^{[l,j]}\|_2}\, r.$$

When r decreases, the amount of regularization increases. Therefore, if overfitting is still happening when we use this technique, we could reduce the value of r.

→ Data augmentation.

Data augmentation consists in increasing the number of elements of the training set using the available instances of this data set. For example, if the elements of a training data set are images, we can resize, rotate, shift or change the saturation of them to obtain new images that can be used to train the network (in Figure 2.6 we can see from left to right, an original image of a training data set, its rotation version, its shifted instance and the version obtained with a gamma correction). Training the network with the original pictures and with these new images created artificially, we can make the network less sensitive to size, orientation, position and brightness changes. If the DNN is more tolerant, overfitting can be avoided.

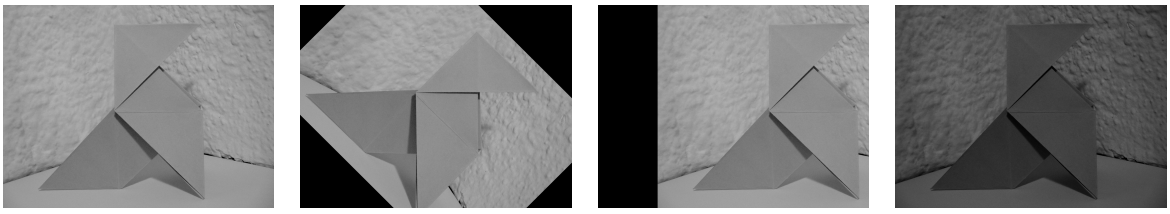

Figure 2.6: Data augmentation. From left to right: original image, rotation version, shifted instance and version of the image when a gamma correction is used. To obtain these images we have used the library *imgaug* for Python [32].

2.3.2 Underfitting

Other problem that we can find after training is underfitting. It is the opposite of overfitting. It arises when we have built a so simple neural network that it is not able to learn from data. In this case, the network does not fit neither the training set nor the test data. To avoid underfitting we can build a more complex network, for example, adding layers or neurons. If we detect underfitting while we are using dropout, we should decrease the dropout rate to avoid the problem. In Figure 2.4 we have a visual representation of underfitting.

2.4 Hyperparameters

We have seen that some parameters of neural networks like weights and biases are tuned during training. However, there are others as the initial value of the weights and biases, the number of layers or the number of neurons per layer whose values are not fitted during training. These parameters are known as hyperparameters. To determine their optimal values is one of the main drawbacks of DNNs and it makes difficult to find an optimal architecture for the neural network. Some information about hyperparameters can be consulted in [5] and [33].

The most intuitive way to tune hyperparameters is to try all the possible values, that is to say, we train the network with all the possible combinations of all the possible values and then we decide the best option. This technique is known as grid search. Let us imagine that we have a network with 6 hyperparameters, each of them can take 4 different values and the training time is around 10 minutes. To train the network with all the possible combinations to decide the best values for the hyperparameters would take us $4^6 \times 10 = 40960$ minutes, that is, almost a month. Therefore, even when we have few hyperparameters and few values for them, to obtain the optimal values is computationally expensive. We conclude that this is not a good way to proceed.

These values can be optimized using metaheuristics algorithms (they are based on nature guiding principles and they help us to search more intelligently in the search space, which is the space of all the possible values for the hyperparameters) instead of using exhaustive methods like grid search. Examples of these metaheuristics algorithms are:

- Particle Swarm Optimization (PSO): it is based on the flight of birds during migration or during the search for food.
- Genetic Algorithm (GA): it is based on the selection and crossover of species taking into account how the evolution and the improvement of the species are affected.

To look for the best values for the hyperparameters we can also use tools like `Oscar` [34]. It is described as *Your new (free) Data Scientist* and it helps us to optimize the hyperparameters of our DNNs. Given the results obtained with our current network, `Oscar` chooses the best values for them and it shows us the influence of each of them in our network.

To fit some hyperparameters like the number of layers and neurons per layer and the initial value of weights and biases of an FNN with fully-connected layers, there are some particular strategies:

→ Number of layers.

To determine the number of layers we usually start with a network with one or two hidden layers (in the case of one hidden layer, it is an MLP but not a DNN) and then, we gradually increase the number of hidden layers until we detect overfitting.

→ Number of neurons per layer.

The number of neurons at the first layer (input layer) and the last layer (output layer) depends on the structure of the input and the output data, respectively. However, there is not a general rule for choosing the optimal number of neurons at each hidden layer. We can take some simplifications as to consider that all the hidden layers have the same number of neurons (this reduces the problem to just one hyperparameter) and to increase this number until overfitting occurs. We can also consider that the number of neurons decreases at each layer (it is an idea based on the hierarchical structure of DNNs). But, in general, it has been observed that it is better to increase the number of layers than the number of neurons per layer.

→ Initial value of weights and biases.

When we work with FNNs with fully-connected layers, biases can be initialized to zero and weights can be initialized using different strategies (they tend to be heuristic) as normalized initialization proposed by X. Glorot and Y. Bengio in their article *Understanding the Difficulty of Training Deep Feedforward Neural Networks* [35]. Normalized initialization consists in obtaining the initial values of the weights with a uniform distribution

$$W^{[l]} \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{l-1}+n_l}}, \frac{\sqrt{6}}{\sqrt{n_{l-1}+n_l}}\right],$$

where $l = 2, \ldots, L$ and n_l is the number of neurons at layer l.

We have seen that it is complicated and usually expensive to obtain the optimal values of the hyperparameters. For this and as a consequence of the hierarchical structure of DNNs, it is common to *recycle* networks that have already been trained. For example, if we have trained a DNN that is able to recognize people, we can use the neurons of the first hidden layers to construct a DNN to recognize different dress styles. In this way, it is only necessary to determine the architecture of the middle and last hidden layers, and the number of hyperparameters is reduced.

In Chapter 3 we study other types of Deep Neural Networks that allow us to perform some tasks that are not feasible to carry out with Feed-forward Neural Networks with fully-connected layers.

Chapter 3

CNNs and RNNs

To perform, for example, image classification or prediction tasks, it is not feasible to use Feed-forward Neural Networks with fully-connected layers. In these cases, we have to use other types of Deep Neural Networks as Convolutional Neural Networks or Recurrent Neural Networks.

3.1 Convolutional Neural Networks

Let us imagine that we want to build a deep Feed-forward Neural Network with fully-connected layers (this type of networks has been described in Chapter 2) to solve an image classification problem. In this case, the input of the neural network is an image. Let us suppose that this image is black and white and its dimension in pixels is 150×150. Therefore, the input layer has $150 \cdot 150 = 22500$ neurons. If we have 100 neurons in the first hidden layer, as this FNN is fully-connected, there are $22500 \cdot 100 = 2250000$ connections between the input layer and the first hidden layer. This means that only in the first two layers we have 2250000 weights that we need to fit. It is obvious that it is not practical to build an FNN with fully-connected layers to solve this kind of problems.

Convolutional Neural Networks (CNNs or ConvNets) are Deep Neural Networks used for image recognition, voice recognition and Natural Language Processing [36] (its objective is to programme computers capable of understanding human language).

CNNs are based on human visual cortex. In the late 50s, D.H. Hubel and T. N. Wiesel carried out some studies of the visual cortex of cats [37] [38]. They concluded that some neurons of this brain region are only able to react to the stimuli of small parts of the visual field and to specific simple patterns (for example, vertical lines). From their experiments it can be deduced that neurons which react to complex patterns (for example, faces) receive the information from the neurons that detect simpler structures. As a consequence of these discoveries, in 1980, K. Fukushima presented the neocognitron [39], which is constructed with an input layer and a cascade connection of modular structures with a layer of S-cells (which corresponds with the neurons that recognize simple patterns) and another one with C-cells (which represents the neurons that detect complex structures). The neocognitron is the predecessor of CNNs.

In this work we focus on the application of ConvNets to image recognition. Thus, we consider that the input of the CNN is an image.

More information about CNNs can be consulted in [5], [22], [33] and [40].

3.1.1 Architecture of Convolutional Neural Networks

The input of a Convolutional Neural Network is a pixelated image understood as a three-dimensional tensor, which is a two-dimensional matrix whose elements are vectors with dimension equal to the number of channels (in general, one channel if we have a black and white

image and three channels if it is a color image). This determines the structure of the input layer.

The main structures of CNNs are convolutional layers. Convolutional layers are 2D layers (matrices) inspired by the fact that some neurons only react to small regions of the visual field. Therefore, each neuron of each non-input layer is not connected to all the neurons of the previous layer (unlike what happens with fully-connected DNNs), it is only connected to some neurons that constitute its local receptive field.

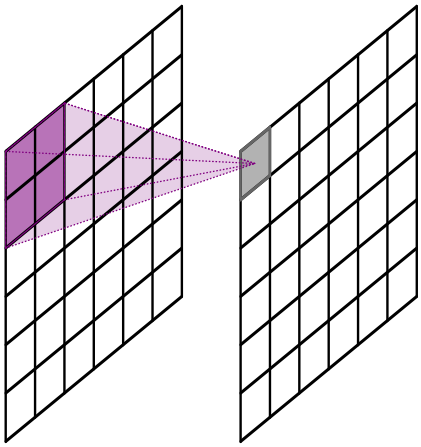

Figure 3.1: Representation of two consecutive convolutional layers of a CNN whose flow goes from left to right. We can see the local receptive field (in purple) of a neuron (in grey).

Figure 3.1 shows two consecutive convolutional layers whose flow goes from left to right. In purple we have the receptive field of the grey neuron. We only need the values of these four neurons at the previous layer to obtain the value of the neuron in grey, we do not need the values of the 36 neurons.

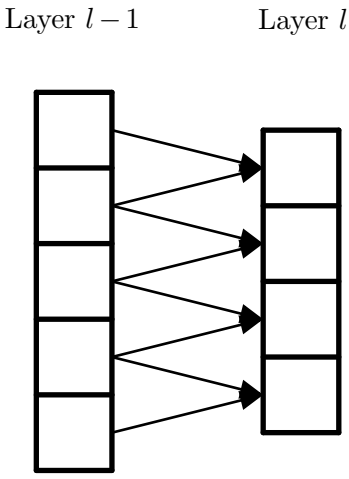

Figure 3.2: Graphic representation of the connections between two consecutive layers of a CNN. The receptive field of layer l has dimension 2×1.

The height and width that define the local receptive field of a neuron are common to all the neurons in the same layer (all the neurons at a layer are connected to the same number of neurons of the previous layer). For layer l, we use $\beta_h^{[l]}$ and $\beta_w^{[l]}$ to denote these dimensions.

The neuron at position (i,j) (row and column) in convolutional layer l is connected to all the neurons in the previous layer that belong to the region between rows i and $i+\beta_h^{[l]}-1$ and columns j and $j+\beta_w^{[l]}-1$.

Example 3.1. For simplicity, we use a column vector (matrix with dimension $n\times 1$) to understand how convolutional layers are connected. Let us consider two consecutive convolutional layers, layer $l-1$ and layer l, that have dimensions 5×1 and 4×1, respectively. The height and width of the local receptive field of the neurons of layer l are $\beta_h^{[l]}=2$ and $\beta_w^{[l]}=1$ (notice that no other value makes sense). The connections between the two layers can be seen in Figure 3.2.

Notice that if in Example 3.1 we consider that the convolutional layer l has dimension 5×1 (the value of $\beta_h^{[l]}$ and $\beta_w^{[l]}$ and the dimension of the previous layer do not change), then the last neuron of layer l does not have enough neurons to connect to. We can see this in the left picture of Figure 3.3, where the missing connection is represented in red. The dimensions of both layers are not compatible. To solve this problem we perform zero padding. This consists in adding zeros around the layer $l-1$ in order to make dimensions compatible. In right picture of Figure 3.3 we have the connections between the two layers when zero padding is used (in grey we have the neuron with value 0 that we have added).

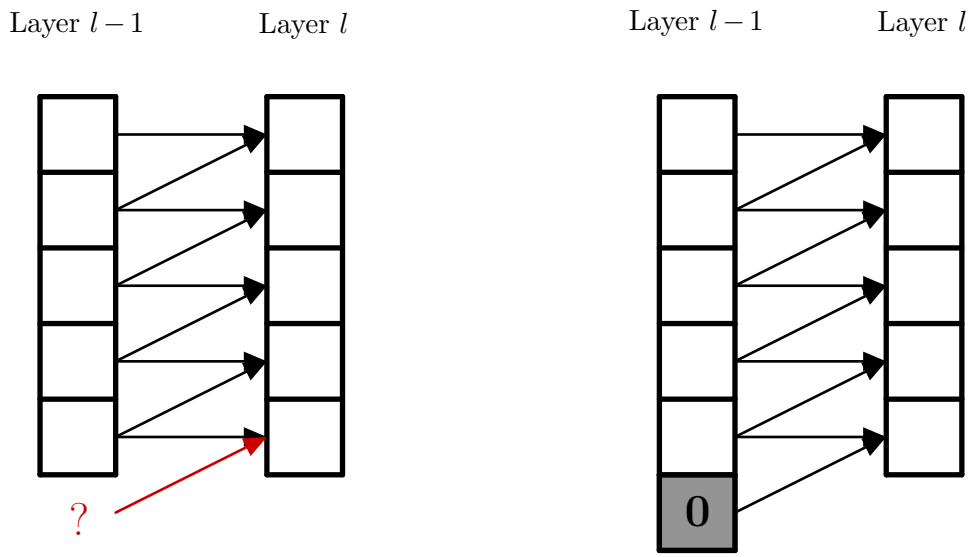

Figure 3.3: On the left, the dimensions of both layers are not compatible. On the right, we perform zero padding to have compatibility.

Observe that in Example 3.1 the distance between the beginnings of the receptive fields of two consecutive neurons is 1, that is to say, if we move the local receptive field of neuron $(i,1)$ one position down, we obtain the local receptive field of neuron $(i+1,1)$. Such distance is called stride. In the case of Figure 3.4 it is 2. We use $\alpha_h^{[l]}$ and $\alpha_w^{[l]}$ to denote the vertical and horizontal stride, respectively. If only one number is given for the stride, we understand that vertical and horizontal stride are equal. If $\alpha_h^{[l]},\alpha_w^{[l]}\neq 1$, neuron (i,j) at layer l is connected to the neurons of previous layer that belong to the rectangle with rows $(i-1)\alpha_h^{[l]}+1$ to $(i-1)\alpha_h^{[l]}+\beta_h^{[l]}$ and columns $(j-1)\alpha_w^{[l]}+1$ to $(j-1)\alpha_w^{[l]}+\beta_w^{[l]}$.

So far, we have seen how connections between neurons of consecutive layers work. Let us study the weights and biases of such connections.

A filter or convolutional kernel is a matrix in which the weights of the connections between two consecutive layers are stored. This matrix (which is sparse and well-structured) and the

local receptive field of the layer receiving these connections have the same dimension. The same filter is applied to the receptive field of all the neurons at the same layer. The bias term is also the same for all the neurons in one layer.

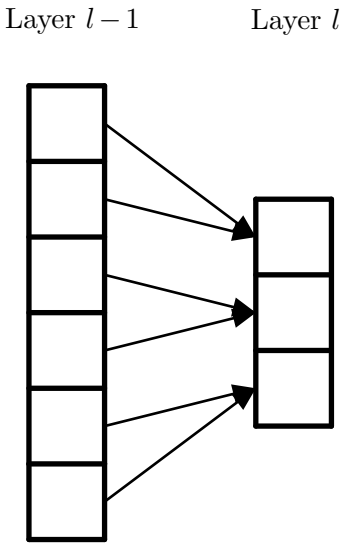

Figure 3.4: Graphic representation of the connections between two consecutive layers of a CNN when the vertical stride and the dimension of the local receptive field of layer l are 2 and 2×1, respectively.

We denote by $x_{i,j}^{[l]}$ the value of the neuron in the i-th row and j-th column of the convolutional layer l. The filter is represented by the matrix $W^{[l]} = (w_{i,j}^{[l]})_{i,j} \in \mathbb{R}^{\beta_h^{[l]} \times \beta_w^{[l]}}$ and $b^{[l]}$ is the bias term. It is common to use the ReLU function (2.3) as the activation function $\mathcal{A}$. The value of neuron (i,j) at layer l is computed as follows:

$$x_{i,j}^{[l]} = \mathcal{A}\left(b^{[l]} + \sum_{u=1}^{\beta_h^{[l]}} \sum_{v=1}^{\beta_w^{[l]}} x_{i',j'}^{[l-1]} \, w_{u,v}^{[l]} \right), \tag{3.1}$$

where

$$\begin{cases} i' = u + (i-1)\alpha_h^{[l]}, \\ j' = v + (j-1)\alpha_w^{[l]}. \end{cases}$$

CNNs and convolutional layers are called this way because the computation process (3.1) is similar to a convolution operation.

As we apply the same filter and bias to all the neurons at the same layer, all of them are extracting the same structures (determined by the filter) from their corresponding local receptive field. Let us see this with an example. At the bottom of Figure 3.5 we have an image of a building. We consider a vertical filter that is represented by a sparse matrix that only has 1's in the middle column. If we apply it, we obtain the top-left image in which all the vertical lines of the original image are highlighted, while the other structures seem to be 'discarded'. We use a horizontal filter now. It corresponds to a sparse matrix with only 1's in the middle row. Applying it to the original image we obtain the top-right image in which the horizontal features of the image have been captured. So, using different filters we extract the different structures of the image. A layer whose neurons use the same filter and bias is known as feature map.

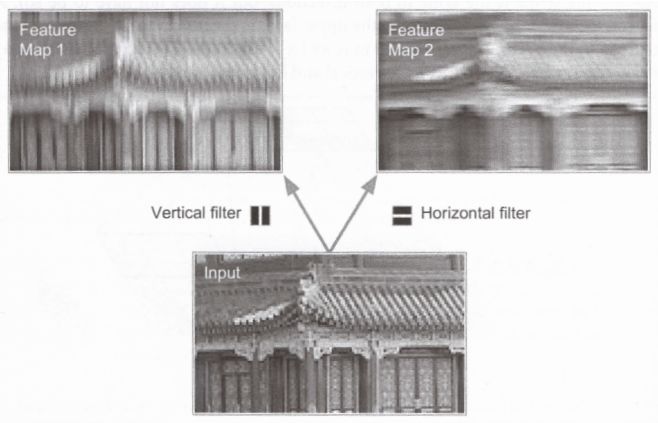

Figure 3.5: Visual explanation of how filters highlight some features of an image. This picture has been obtained from [5].

At this moment, a convolutional layer and a feature map seem to be the same, but they are not. To explain how a CNN works we have defined a convolutional layer as a unique layer of neurons with the same weights and bias, that is to say, we have considered that a convolutional layer has only one feature map. However, convolutional layers are usually built with several feature maps with the same size. Each feature map is connected to all the feature maps of the previous convolutional layer using a different filter for each of them (each one highlights a different structure of the data). The set of all these filters is called filter bank and it is characteristic of this feature map.

In Figure 3.6 we have two consecutive convolutional layers whose flow goes from left to right. Each layer has three feature maps. In grey we have a neuron of a feature map. In purple we can see all the information it receives from all the feature maps of the previous layer.

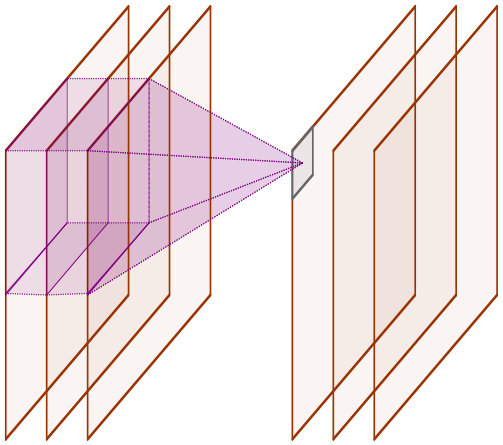

Figure 3.6: Representation of two consecutive convolutional layers of a CNN whose flow goes from left to right. We can see the information (in purple) that receives a neuron (in grey) from the feature maps of the previous convolutional layer.

As all the neurons of a feature map use the same bank of filters, CNNs have the following two characteristics:

- The number of training parameters (weights and biases) is lower, so the network needs less time to learn.

- CNNs have the property of invariance under location: any pattern or object can be recognized in any part of the image, unlike Feed-forward Neural Networks with fully-connected layers that only are able to recognize objects in one part of the image if they have seen one similar there during training.

We denote by $x_{i,j,k}^{[l]}$ the value of the neuron (i,j) in the feature map k of the convolutional layer l. The filter applied to the feature map k' at convolutional layer $l-1$ to obtain the feature map k of layer l is the matrix $W_{k,k'}^{[l]} = (w_{i,j,k,k'}^{[l]})_{i,j}$. The bias term of feature map k at convolutional layer l is $b_k^{[l]}$. The number of feature maps of the convolutional layer l is $f^{[l]}$. We can write mathematically the value of neuron (i,j) in the feature map k of the convolutional layer l as

$$x_{i,j,k}^{[l]} = \mathcal{A}\left(b_k^{[l]} + \sum_{u=1}^{\beta_h^{[l]}} \sum_{v=1}^{\beta_w^{[l]}} \sum_{m=1}^{f^{[l-1]}} x_{i',j',m}^{[l-1]}\, w_{u,v,k,m}^{[l]}\right), \tag{3.2}$$

where

$$\begin{cases} i' = u + (i-1)\alpha_h^{[l]}, \\ j' = v + (j-1)\alpha_w^{[l]}. \end{cases}$$

If $l = 2$, we are computing the values of the neurons of the first hidden layer. In this case, the information is obtained from the input layer ($l = 1$). Instead of feature maps, the input layer has channels that behave similarly to feature maps.

Taking into account formula (3.2), it is easy to deduce that the patterns highlighted in layer $l-1$ are combined in convolutional layer l to create more complex structures. ConvNets are able to take advantage of the hierarchical structure of real data and for that, they are good for solving, for example, classification problems.

After each convolutional layer, we often have a pooling layer (some authors refer to it as sub-sampling layer) which allows us to reduce the dimension of the feature maps. In the pooling layer we have a 'copy' of each feature map that has less neurons: each neuron of the 'copy' is replacing all the neurons of the feature map which are in a rectangular receptive field. The value of such replacement is obtained using a function that returns the maximum of the receptive field (in this case the pooling layer is called max pooling layer) or the average (it receives the name of average pooling layer). As in the case of convolutional layers, we have to fix a stride value $\alpha^{[l],p}$ (two values if we consider that vertical and horizontal strides are different) and the dimension $\beta_h^{[l],p} \times \beta_w^{[l],p}$ of the rectangular receptive field (we have used the superscript $[l],p$ to refer to the pooling layer that follows convolutional layer l).

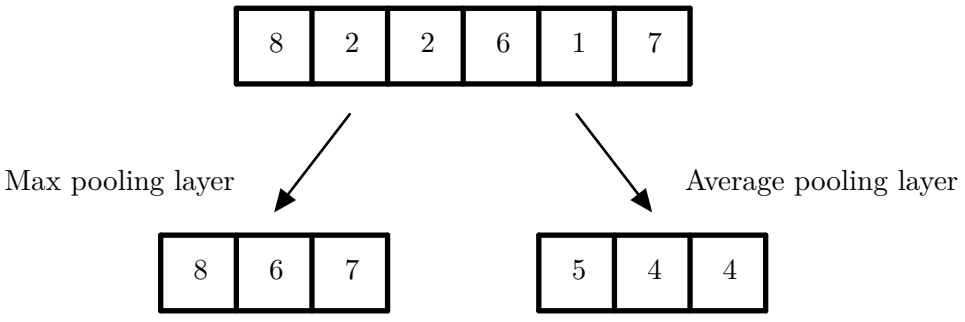

Figure 3.7: 'Copy' of the max pooling layer (bottom-left) and 'copy' of the average pooling layer (bottom-right) of a feature map (top).

Example 3.2. As in Example 3.1, for simplicity, we work with vectors. Let us consider that we have a feature map of dimension 1×6 in the convolutional layer l that we can see at the top of Figure 3.7. The pooling layer that follows it is built with 'copies' of dimension 1×3, its horizontal stride is $\alpha_w^{[l],p} = 2$ and the dimension of the rectangular receptive field is $\beta_h^{[l],p} \times \beta_w^{[l],p} = 1 \times 2$. At the bottom-left of the figure we have the corresponding 'copy' of our feature map when the pooling layer is a max pooling layer and at the bottom-right of such figure we have it when the pooling layer is an average pooling layer.

To sum up, in the architecture of a CNN we have convolutional layers (they are built stacking features maps) and pooling layers (they can be understood as 'copies' of the feature maps of the previous convolutional layer that have less neurons). If we stack two convolutional layers, the connections between them, and the corresponding computations, are based on the local receptive field and the filters. The convolutional layer that follows a pooling layer is connected to it and computed in the same way as when we have two consecutive convolutional layers.

As we have introduced, ConvNets are used for image recognition. Therefore, we can use them to carry out detection and classification tasks. In these cases, after the CNN, we stack an FNN with fully-connected layers in order to obtain the wanted output.

→ Detection task.

We want to know the position of an object in the image. The output of the network is a sequence of bounding boxes. A bounding box is a rectangle, usually represented by its corners, that bounds an object. So, the output is a set of points and to obtain it we need to stack behind the ConvNet a Feed-forward Neural Network with fully-connected layers.

→ Classification task.

The neural network has to be able to classify some objects. The category of each of them is represented by a numerical value. The output of the network is a vector of dimension $\mathcal{N}$ where $\mathcal{N}$ is the number of categories in which the object can be classified. Its i-th coordinate is the probability of the object to belong to the i-th category. To get this type of output we need to stack behind the CNN a Feed-Forward Neural Network with fully-connected layers and to apply the softmax operation to the last layer. The softmax operation converts the values of a vector into positive values whose sum is 1, so we can understand them as probabilities. Its expression is given by

$$\widetilde{v}_i = \frac{e^{v_i}}{\sum_{j=1}^{\mathcal{N}} e^{v_j}},$$

where $\boldsymbol{v} = (v_i)_{i=1}^{\mathcal{N}}$ is the original output of the network and $\widetilde{\boldsymbol{v}} = (\widetilde{v}_i)_{i=1}^{\mathcal{N}}$ is the vector of probabilities.

3.1.2 Other Considerations about Convolutional Neural Networks

The training process for CNNs is equal to the one explained in Section 2.2. In the case of classification tasks, if we use softmax operation in the output layer, in order to obtain a probability close to 1 for the correct category, the following logarithmic cost function is used:

$$-\sum_{i=1}^{M} \log \frac{e^{\widetilde{v}_{c_i}^{\{i\}}}}{\sum_{j=1}^{\mathcal{N}} e^{\widetilde{v}_j^{\{i\}}}}, \tag{3.3}$$

where M is the number of points of the training data, $\mathcal{N}$ is the number of categories, $\widetilde{v}_j^{\{i\}}$ denotes the j-th component of the vector of probabilities that corresponds to the i-th element of the training data and $\widetilde{v}_{c_i}^{\{i\}}$ is the component of the previous vector that corresponds to the correct category of the input (this index is obtained from the label of the training point).

Back Propagation (see Subsection 2.2.4) can also be used to compute the gradient of the loss function during training. The techniques introduced in Section 2.3 to avoid overfitting can be applied to CNNs.

Some hyperparameters of the ConvNets are the number of convolutional and pooling layers, the number of feature maps in each convolutional layer and the stride and dimension of the receptive fields.

3.1.3 Examples of Convolutional Neural Networks

In this subsection we explain two tasks performed by CNNs in order to illustrate them.

We want to build a CNN in order to solve a classification problem [22]. We have a set of black and white images (one channel) of dimension 150×200 of cars, motorbikes, trucks, buses and vans that we want to classify. We identify previous vehicles with numbers 1, 2, 3, 4 and 5, respectively. The label of the training instances is a vector of dimension 5 whose unique non-zero coordinate is the one that corresponds to the correct category (it has value 1). Let us define a CNN to try to solve the problem.

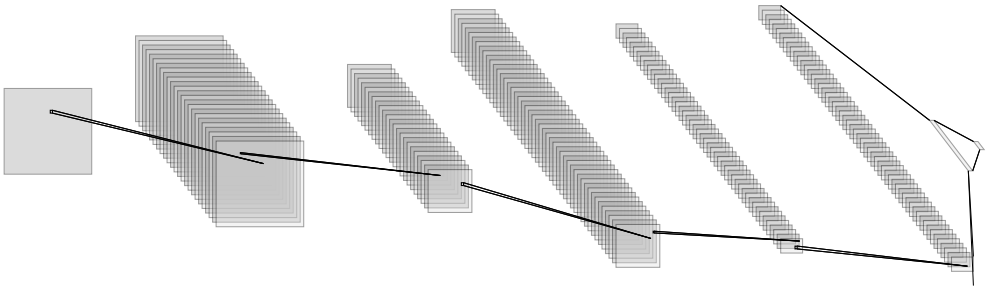

Figure 3.8: CNN that can be used to solve a classification task. This picture has been made with [41].

The input layer is of dimension $150 \times 200 \times 1$ since we work with black and white images of dimension 150×200. This layer is followed by a convolutional layer with 24 features maps with dimension 150×200. We use an stride of 1 and a local receptive field of dimension 5×5 (if it is needed, zero padding is applied). We consider a bias term and we take the ReLU function as the activation function. Following this convolutional layer we place an average pooling layer with stride 2 and dimension 2×2 for the receptive field in order to reduce the dimension into 75×100. Behind it we put a convolutional layer with 48 feature maps with dimension 75×100. The filters have dimension 5×5 and we consider an stride of 2 (as we have pointed out before, zero padding is applied if it is necessary). A bias term is applied and we use the ReLU function as the activation function. This convolutional layer is followed by a max pooling layer with stride 3 for the vertical dimension and 2 for the horizontal one, being the dimension of the receptive field equal to 3×2. The dimension is reduced to 25×50. We place the last convolutional layer which has 56 feature maps with dimension 25×50. The stride is 1 and the local receptive field has dimension 5×5 (if it is necessary, zero padding is applied). We consider a bias term and the ReLU activation function. This network ends with a Feed-forward Neural Network with fully-connected layers whose output is a vector of dimension 5 (each element of the vector represents one of the five categories), to which we apply the softmax operation in order to obtain a probability for each category. To train the network we use the logarithmic cost function defined in (3.3). In Figure 3.8 we have a representation of a network of this type.

Let us think now in a more complex task. We want to detect and classify the vehicles that cross the street. In this case we have a video, so we have to study it frame by frame. To detect and classify the vehicles of each still, we use a CNN followed by an FNN with fully-connected layers. The output is a sequence of vectors, each of them containing the bounding box (position) and the probabilities for the categories of an object. This output may be used, for example, to identify the vehicle over the video to compute its velocity. This has been develop in the project *Smart Traffic manager suite* (reference: 2020/0075) carried out in the University of Zaragoza and financed by Lector Vision S.L. In Figure 3.9 we have a sample of the graphic interpretation of the output of the network, the colours represent the category of the vehicles (green for cars and orange for motorbikes).

Figure 3.9: Graphic interpretation of the output of a DNN used for detection and classification tasks. The image has been provided by Lector Vision S.L.

3.2 Recurrent Neural Networks

So far, we have only studied neural networks whose flow goes forwards, that is to say, the information goes into the input layer and it crosses all the hidden layers until it arrives to the output layer where the output of the network is obtained. However, if, for example, the input data is a sequence and we need to predict the next element of it, we need a structure 'with memory'. For this reason, the flow cannot only go forwards.

Recurrent Neural Networks (RNNs) are Deep Neural Networks which receive data sequentially, using previous and current information at each time step. We can say that they have memory. This makes them very useful to study time series to predict next events. For example, they can be used to predict the next word of a sentence or the following musical note of a melody. Moreover, some Natural Language Processing applications such as machine translation use this sort of networks. RNNs can also be used to try to do weather predictions. Dynamical systems that model the weather are complex systems whose dynamics are not easy to predict, however RNNs could do it [42].

More information about RNNs can be consulted in [5], [33], [40] and [43].

3.2.1 Architecture of Recurrent Neural Networks

Remark 3.3. In what follows, when we talk about observable input we refer to the information introduced in the network that is new (it has not been used or obtained from it).

The simplest RNN is built with a unique neuron that is called recurrent neuron. At each time step t (also called frame), the input of the neuron consists of an observable data point $\boldsymbol{x}(t)$ and its previous output $y(t-1)$. The output is a scalar $y(t)$ that is used as input in the next

time step. The row vector $\boldsymbol{w}^{\boldsymbol{x}}$ of the weights of the connections between the observable data and the neuron and the weight w^y of the link between the previous output and the neuron do not depend on time. Adding a bias term b that we can also consider independent of time and choosing an activation function $\mathcal{A}$, we can write the output at time t of this simple RNN as

$$y(t) = \mathcal{A}(\boldsymbol{w}^{\boldsymbol{x}}\boldsymbol{x}(t) + w^y y(t-1) + b). \tag{3.4}$$

With this expression it is easy to see that the output $y(t)$ depends on the observable data $\boldsymbol{x}(t)$ and on the previous output $y(t-1)$, which, in turn, depends on $\boldsymbol{x}(t-1)$ and $y(t-2)$ and so on. Therefore, $y(t)$ is a function that depends on all the introduced data until time t. Somehow, the neural network has memory.

Remark 3.4. For time step $t = 0$, according to formula (3.4), the previous output is needed. As it does not exist, we consider that it is equal to 0.

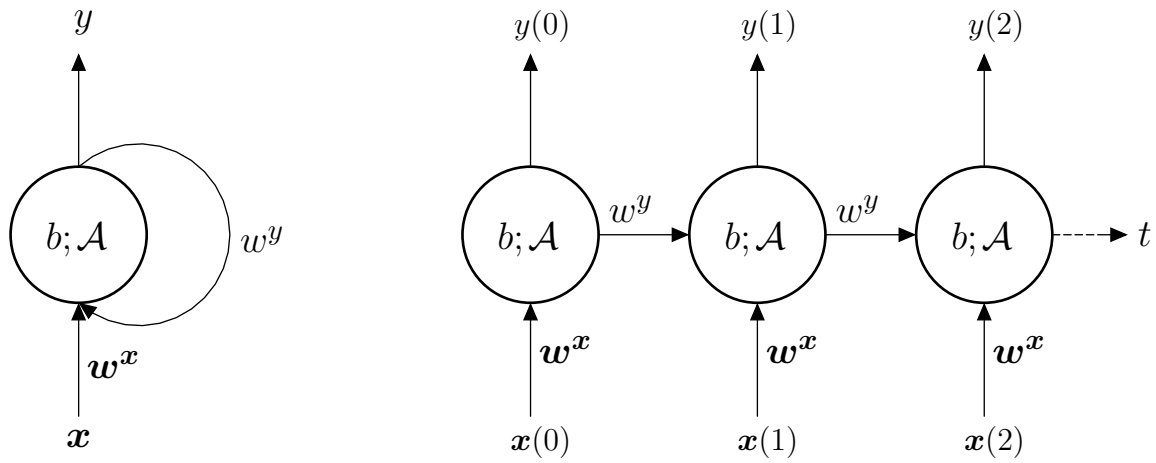

Figure 3.10: On the left, scheme of the simplest RNN. On the right, drawing obtained unrolling the network through time.

On the left of Figure 3.10 we can see a graphic representation of this RNN. On the right of the same figure we have represented the network with respect to time. Now, the RNN looks like a Feed-forward Neural Network. The process to change from the representation on the left to the representation on the right is known as unrolling the network through time.

A more complicated RNN can be obtained if we build a layer with some recurrent neurons. In this case, we have an observable input connected to all the neurons in the layer. All of them also receive the output of this neural network, which is a vector (the output of each recurrent neuron is a scalar, so the output of all the network has to be a vector). As in the case of a unique neuron, the weights and biases are constant for all time steps. If $W^{\boldsymbol{x}}$ is the matrix whose rows are the weights of the connections between the observable input and each recurrent neuron, $W^{\boldsymbol{y}}$ is the matrix for the links between previous output and each recurrent neuron, and $\boldsymbol{b}$ is the bias vector, we have that the output of the network at time step t is given by

$$\boldsymbol{y}(t) = \mathcal{A}(W^{\boldsymbol{x}}\boldsymbol{x}(t) + W^{\boldsymbol{y}}\boldsymbol{y}(t-1) + \boldsymbol{b}).$$

On the left of Figure 3.11 we have a drawing of this RNN and on the right we have the picture obtained unrolling the network through time. Notice that in the unrolled representation we have used a unique structure represented by a square to symbolize the layer. This layer is the part of the RNN that is preserving the previous states and it is considered as a unique structure called memory cell. The recurrent neuron of our first RNN (the simplest one) is a memory cell too.

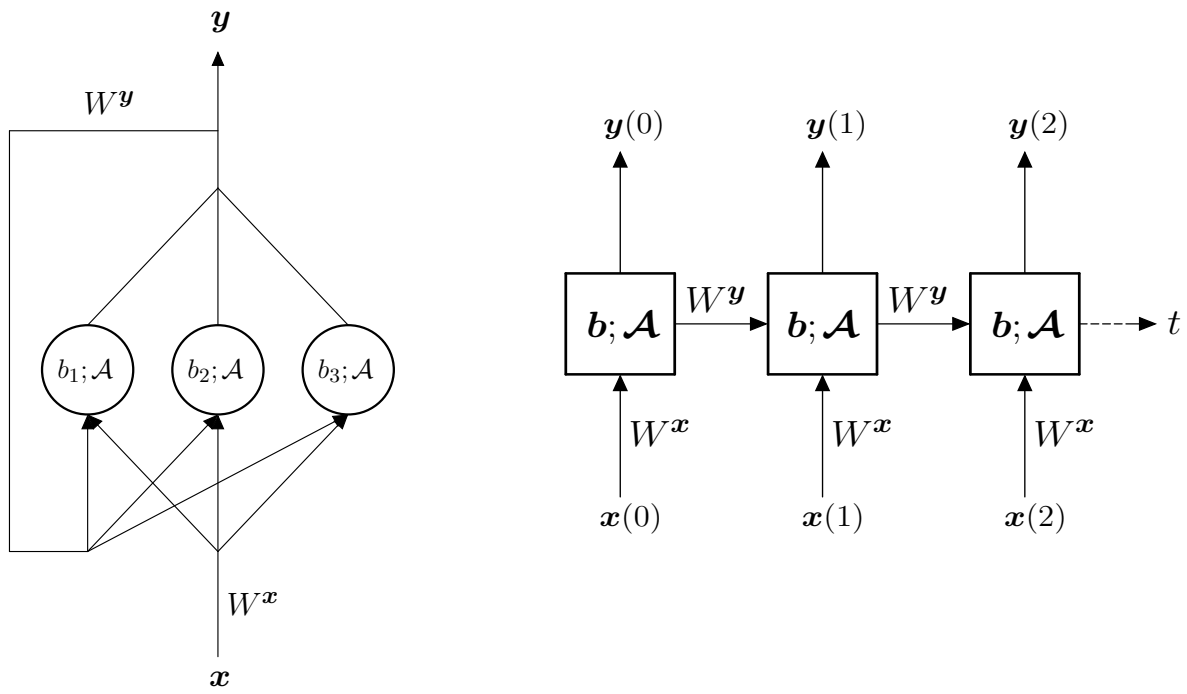

Figure 3.11: On the left, scheme of an RNN with a layer of recurrent neurons. On the right, drawing obtained unrolling the network through time.

Remark 3.5. Up to now, we have considered that the previous information of the network used as input at time step t is $\boldsymbol{y}(t-1)$. However, in a more general case, the input of the network that contains previous information can be a vector $\boldsymbol{s}(t-1)$ called state that is computed by the network but it could not coincide with $\boldsymbol{y}(t-1)$.

Remark 3.6. We can stack some layers of recurrent neurons to obtain a deep RNN. However, it is common to have only one layer.

It is not necessary to have an input $\boldsymbol{x}(t)$ and an output $\boldsymbol{y}(t)$ for each time step t. Depending on the task we need to perform, we can choose different sequences for the observable inputs and for the outputs. A famous example is the Encoder-Decoder. In this case, for the first time steps, the elements of the observable input sequence are non zero and we do not obtain the outputs (however, at each time step the network is still receiving information from the previous time steps). This part of the RNN is the Encoder, that converts the input sequence to a vector. For the next time steps, we consider that the observable input sequence is zero and we obtain all the outputs. This part of the network is the Decoder, which transforms the vector obtained in the Encoder into a sequence. In Figure 3.12 we have an example of this RNN. It can be used to translate sentences from one language to another (in [5] it is explained in detail how it works).

3.2.2 Back Propagation Through Time

Once we have defined the structure of our RNN, we need to train it. To do this we have to unroll the network through time. As we have pointed out before, when it is represented in this way, it seems to be a Feed-forward Neural Network. Remember that the weights and biases remain the same for all time steps. When we use one of the optimization algorithms studied in Section 2.2, we need to compute the gradient of the loss function. The derivatives of the

loss function respect to the weights and biases are calculated using Back Propagation Through Time (BPTT).

BPTT is an algorithm similar to Back Propagation. It is applied to the unrolled RNN. The loss function depends on all the outputs along time (if, as in the case of an Encoder-Decoder, there are some time steps for which we have not an output, the loss function does not depend on the outputs of these time steps). The derivatives are computed backwards from the output of each time step as we can see in Figure 3.12, where the red arrows represent the calculation flow. The obtained derivatives are added over all time steps.

Notice that unrolling through time, we transform the RNN into a network that has a layer for each time step. If the task needs a lot of time steps, this network has many layers and BPTT needs much time to compute the derivatives. In this case, we can use Truncated BPTT. Such algorithm consists in separating the usual computation flow of BPTT into some subsequences. Then, the backwards computations are done separately for each group. Finally, the results are added in order to obtain the gradient.

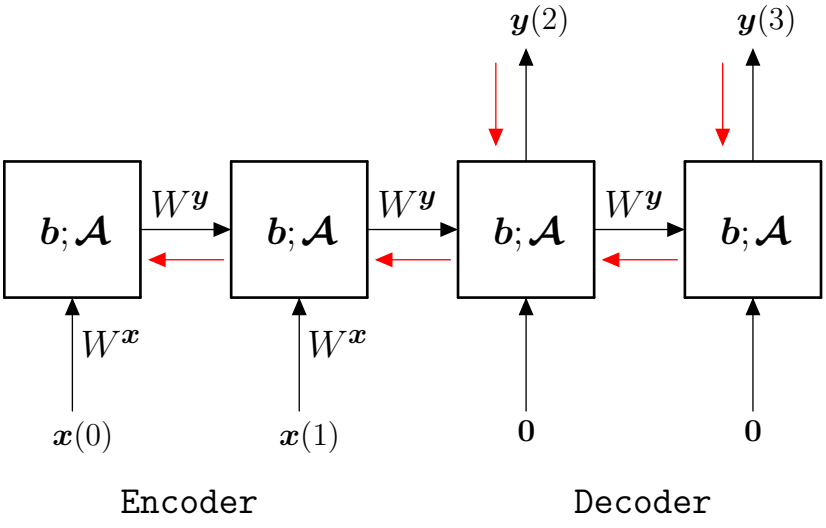

Figure 3.12: Graphic representation of an Encoder-Decoder. Flow of the calculations of BPTT algorithm in red. This picture is inspired by the image used to explain BPTT in [5].

3.2.3 LSTM and GRU

One of the main characteristics of RNNs is that they have 'memory', that is to say, we store information from previous time steps in order to use it in the current time step. However, there is empirical evidence that RNNs cannot store such information for a long time and this can be a problem. To try to solve this issue, some long-term memory cells have been defined. Two of the most used are LSTM cell and GRU cell, being the second one a simpler version of the first one.

LSTM

Long Short-Term Memory (LSTM) is a Recurrent Neural Network with long-term memory. Unlike RNNs with memory cells formed by only one recurrent neuron or a layer of recurrent networks, LSTM is able to keep information of the states of earlier time steps. It is based on a memory cell called LSTM cell. It was presented by S. Hochreiter and J. Schmidhuber in 1997 in their paper *Long Short-Term Memory* [44].

Let us study how an LSTM cell works at each time step t. First of all, we have to remark that the state is divided into a short-term state $\boldsymbol{h}(t)$ and a long-term state $\boldsymbol{c}(t)$. The inputs of

the cell are the observable input $\boldsymbol{x}(t)$ and the short-term and long-term state of the previous time step, they are denoted by $\boldsymbol{h}(t-1)$ and $\boldsymbol{c}(t-1)$, respectively. The output of the network is $\boldsymbol{y}(t)$.

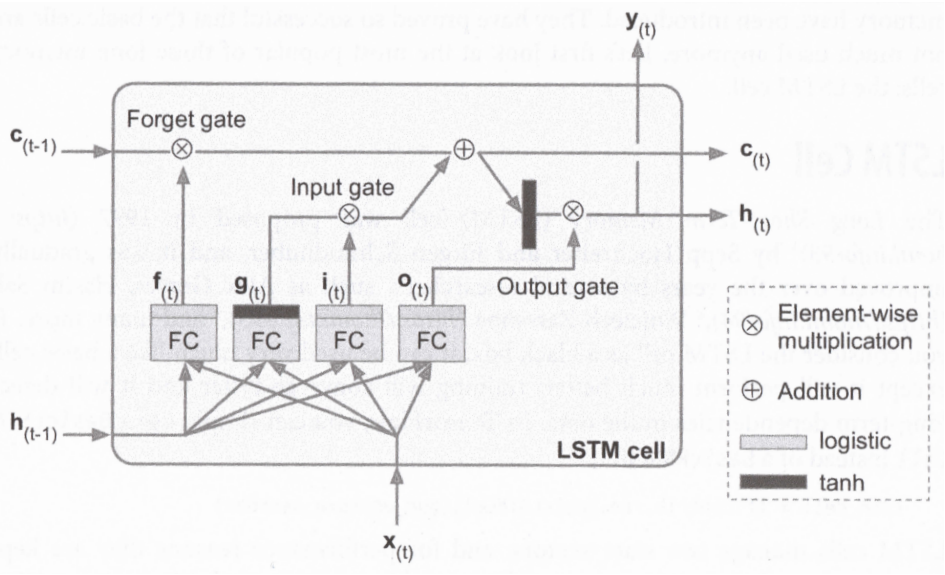

Figure 3.13: LSTM cell. This picture has been obtained from [5].

Inside the LSTM cell there are four fully-connected layers whose inputs are $\boldsymbol{x}(t)$ and $\boldsymbol{h}(t-1)$. We refer to these layers with letters f, g, i and o. They can be understood as the structures of the cell that decide the information that is in the state vectors and in the output vector.

Let $\alpha = f,g,i,o$. We denote by $W^{\alpha,\boldsymbol{x}}$ and $W^{\alpha,\boldsymbol{h}}$ the weights of the connections between $\boldsymbol{x}(t)$ and layer α, and between $\boldsymbol{h}(t-1)$ and layer α, respectively. The bias term of layer α is $\boldsymbol{b}^{\alpha}$ and $\boldsymbol{\alpha}(t)$ represents its output at time step t. Using this notation, we study each of the fully-connected layers introduced before.

- Layer f. It is one of the gate controllers. Its output $\boldsymbol{f}(t)$ is

$$\boldsymbol{f}(t) = \boldsymbol{\sigma}\left(W^{f,\boldsymbol{x}}\boldsymbol{x}(t) + W^{f,\boldsymbol{h}}\boldsymbol{h}(t-1) + \boldsymbol{b}^{f}\right). \tag{3.5}$$

  Notice that we have used the sigmoid (or logistic) function as the activation function, so the components of vector $\boldsymbol{f}(t)$ take values between 0 and 1. Because of this, we can use this output as a control term that is said to be responsible of a gate: if we multiply $\boldsymbol{f}(t)$ by vector $\boldsymbol{c}(t-1)$ element by element (componentwise multiplication of two vectors), the components of the long-term state vector that are multiplied by zero are erased and the others are stored (they are multiplied by a number in the interval $(0,1]$). Therefore, if we call this vector $\widehat{\boldsymbol{c}}(t)$, we have that

$$\widehat{\boldsymbol{c}}(t) = \boldsymbol{f}(t) \otimes \boldsymbol{c}(t-1).$$

  In this case the gate is known as forget gate.

- Layer g. It is responsible for studying $\boldsymbol{x}(t)$ and $\boldsymbol{h}(t-1)$. The output $\boldsymbol{g}(t)$ of this layer is given by

$$\boldsymbol{g}(t) = \mathbf{tanh}\left(W^{g,\boldsymbol{x}}\boldsymbol{x}(t) + W^{g,\boldsymbol{h}}\boldsymbol{h}(t-1) + \boldsymbol{b}^{g}\right), \tag{3.6}$$

  where we have used the hyperbolic tangent function as the activation function.

- Layer i. It is one of the three gate controllers. Its output $\boldsymbol{i}(t)$ is given by

$$\boldsymbol{i}(t) = \boldsymbol{\sigma}\left(W^{i,x}\boldsymbol{x}(t) + W^{i,h}\boldsymbol{h}(t-1) + \boldsymbol{b}^i\right). \tag{3.7}$$

  As in the case of layer f, the output is a vector whose elements belong to the closed interval $[0,1]$. This output and $\boldsymbol{g}(t)$ are multiplied component by component in the input gate (it is controlled by $\boldsymbol{i}(t)$ that decides the elements of $\boldsymbol{g}(t)$ that are stored). The result is added to $\widehat{\boldsymbol{c}}(t)$, obtaining the long-term state of the current time step. We can write that

$$\boldsymbol{c}(t) = \boldsymbol{i}(t) \otimes \boldsymbol{g}(t) + \widehat{\boldsymbol{c}}(t), \tag{3.8}$$

  where $\widehat{\boldsymbol{c}}(t) = \boldsymbol{f}(t) \otimes \boldsymbol{c}(t-1)$.

- Layer o. It is one of the three gate controllers. The output of this layer is

$$\boldsymbol{o}(t) = \boldsymbol{\sigma}\left(W^{o,x}\boldsymbol{x}(t) + W^{o,h}\boldsymbol{h}(t-1) + \boldsymbol{b}^o\right). \tag{3.9}$$

  This gate controller is responsible for the output gate in which $\mathbf{tanh}(\boldsymbol{c}(t-1))$ and $\boldsymbol{o}(t)$ are multiplied point by point in order to obtain the short-term state $\boldsymbol{h}(t)$ and the output $\boldsymbol{y}(t)$. Therefore we can write

$$\boldsymbol{h}(t) = \boldsymbol{y}(t) = \boldsymbol{o}(t) \otimes \mathbf{tanh}(\boldsymbol{c}(t)). \tag{3.10}$$

Formulas (3.5), (3.6), (3.7), (3.8), (3.9) and (3.10) give us all the computations performed inside the memory cell. In Figure 3.13 we can see the structure of the LSTM cell.

GRU

A Gated Recurrent Unit (GRU) is a simpler version of an LSTM, so it is an RNN with long-term memory. Its main structure is a memory cell called GRU cell. GRUs were introduced by K. Cho et al. in 2014 in the article *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation* [45].

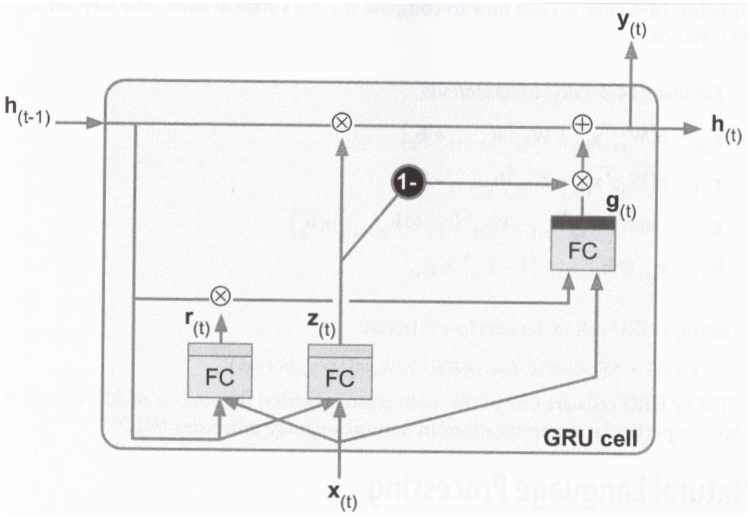

Figure 3.14: GRU cell. This picture has been obtained from [5].

Let us see how a GRU cell works. It has only one vector state denoted by $\boldsymbol{h}(t)$ (in the case of the LSTM cell we have two). The inputs are the observable input $\boldsymbol{x}(t)$ and the previous state $\boldsymbol{h}(t-1)$. The output is $\boldsymbol{y}(t)$. Unlike LSTM cell, GRU cell has only three fully-connected layers to which we refer with letters r, g and z, and it does not have an output gate. Following a notation for the weights and biases equivalent to the one used in the case of LSTM cells, we can study each layer.

- Layer r. It is a gate controller. Its output is given by

$$\boldsymbol{r}(t) = \boldsymbol{\sigma}\left(W^{r,\boldsymbol{x}}\boldsymbol{x}(t) + W^{r,\boldsymbol{h}}\boldsymbol{h}(t-1) + \boldsymbol{b}^r\right). \tag{3.11}$$

  This output is responsible for a gate that, with a componentwise multiplication, controls the part of the previous state that is received by layer g as input.

- Layer g. It is equivalent to the layer g in the LSTM cell. However, a gate controlled by $\boldsymbol{r}(t)$ has power over the previous state $\boldsymbol{h}(t-1)$ that layer g receives as input. Its output is

$$\boldsymbol{g}(t) = \mathbf{tanh}\left(W^{g,\boldsymbol{x}}\boldsymbol{x}(t) + W^{g,\boldsymbol{h}}\left(\boldsymbol{r} \otimes \boldsymbol{h}(t-1)\right) + \boldsymbol{b}^g\right). \tag{3.12}$$

- Layer z. It is a gate controller. Its output is

$$\boldsymbol{z}(t) = \boldsymbol{\sigma}\left(W^{z,\boldsymbol{x}}\boldsymbol{x}(t) + W^{z,\boldsymbol{h}}\boldsymbol{h}(t-1) + \boldsymbol{b}^z\right), \tag{3.13}$$

  that controls the input and the forget door: when a component of $\boldsymbol{z}(t)$ is equal to 1, the forget gate does not erase this element from $\boldsymbol{h}(t-1)$ and the input gate removes it from $\boldsymbol{g}(t)$; when it has value 0, the gates work in the other way around. Therefore, we have that

$$\boldsymbol{h}(t) = \boldsymbol{y}(t) = \boldsymbol{z}(t) \otimes \boldsymbol{h}(t-1) + (\mathbf{1} - \boldsymbol{z}(t)) \otimes \boldsymbol{g}(t). \tag{3.14}$$

All the computations carried out by the GRU cell are given by equations (3.11), (3.12), (3.13) and (3.14). This memory cell is represented in Figure 3.14, where we can see clearly that its structure is simpler than the structure of the LSTM cell.

Conclusions

- Artificial Intelligence (AI) is the field of computer science focused on building machines that can learn, reason and act as intelligent humans. One of the main topics in AI is Machine Learning.
- Machine Learning (ML) is the branch of AI concentrated on programming systems that have to be able to learn from data. The learning process is known as training. Taking into account if human supervision is needed or not, we can classify ML algorithms in two groups: supervised learning and unsupervised learning.
- Artificial Neural Networks (ANNs) are a ML algorithm based on the structure of biological neurons (these neurons are organized in layers). We focus on ANNs that are supervised learning algorithms. The Perceptron is a simple type of ANN which cannot solve some problems like the XOR classification problem that other ANNs like Multi-Layer Perceptron can.
- A Multi-Layer Perceptron (MLP) is a Feed-forward Neural Network (FNN) with fully-connected layers that has an input layer, some hidden layers and an output layer. Any continuous function can be approximated by an MLP with certain characteristics.
- Deep Learning (DL) is the branch of ML that includes all the techniques used to build Deep Neural Networks (ANNs with multiple hidden layers) that are able to learn from real data.
- In the field of DL, an FNN with fully-connected layers is an MLP with at least two hidden layers. Its architecture is based on the weights assigned to the connections between the neurons of consecutive layers, the biases added at each neuron and the activation function. To determine the values of the weights and biases we need to train the network. This process is reduced to an optimization problem that consists in minimizing a function called loss function (it measures how well the neural network fits the data).
- Gradient Descent (GD) and Stochastic Gradient (SG) are optimization algorithms that can be used to solve the minimization problem. However, they have convergence issues. In the case of convex $\mathcal{L}$-smooth functions, we can fix them adding momentum or restart techniques to GD. In the case of SG, if we only add momentum (NASGD), even for convex smooth optimization problems, we accumulate error at each iteration. We need to use a restart technique to speed up SG (SRSGD algorithm).
- In the optimization algorithms used during training, we have to compute the gradient of the loss function. Back Propagation exploits the forward structure of FNNs with fully-connected layers to compute the derivatives backwards.
- During training, the neural network cannot be able to learn to generalize or it cannot learn from the training data. These problems are known as overfitting and underfitting, respectively. The first one can be solved with regularization techniques as dropout and the second one, building a more complex neural network.

- Hyperparameters are parameters that are not fitted during training. In the case of FNNs with fully-connected layers they are, for example, the number of layers or the initial value of the weights and biases.
- Convolutional Neural Networks (CNNs) are Deep Neural Networks used when the input of the network is an image. They can be used to perform classification and detection tasks. They are FNNs, but they do not have fully-connected layers. Their architecture is based on convolutional and pooling layers. The process of training is similar to the one of FNNs with fully-connected layers and Back Propagation can also be used.
- Recurrent Neural Networks (RNNs) are Deep Neural Networks with memory (they use an observable input and the previous state of the network that is stored in the memory cell). They are used when the input data is a sequence. If we unroll them through time, they seem to be FNNs, so the process of training is equivalent and the algorithm to compute the derivatives is a version of Back Propagation known as Back Propagation Through Time. There is empirical evidence that they do not have long-term memory, we can get it using an LSTM cell or a GRU cell, which are long-term memory cells.

Bibliography

[1] MIT Technology Review, *Artificial Intelligence*, https://www.technologyreview.com/topic/artificial-intelligence

[2] I. Belda, *Mentes, Máquinas y Matemáticas. La Inteligencia Artificial y sus Retos*, National Geographic. El Mundo es Matemático, RBA Coleccionables (2011).

[3] K. Kumar, G.S.M. Thakur, *Advanced Application of Neural Networks and Artificial Intelligence: A Review*, International Journal of Information Technology and Computer Science **4** (6) (2012), 57 – 68.

[4] A. Turing, *Computing Machinery and Intelligence*, Mind **59** (236) (1950), 433 – 460.

[5] A. Géron, *Hands-On Machine Learning with Scikit-Learn & TensorFlow. Concepts, Tools, and Techniques to Build Intelligent Systems*, O'Reilly (2019) (Twelfth Release).

[6] R. Follmann, E. Rosa Jr., *Predicting Slow and Fast Neuronal Dynamics with Machine Learning*, Chaos: An Interdisciplinary Journal of Nonlinear Science **29** (11) (2019), 113 – 119.

[7] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press (2016), http://www.deeplearningbook.org

[8] A. Dey, *Machine Learning Algorithms: A Review*, International Journal of Computer Science and Information Technologies **7** (3) (2016), 1174 – 1179.

[9] Neuromatch Academy, *Computational Neuroscience. Lecture Notes for Neuromatch Academy 2020*, https://github.com/NeuromatchAcademy/course-content/tree/master/tutorials

[10] AFI Escuela de Finanzas Madrid, *Machine Learning con Python. Lecture Notes for MOOC: Machine Learning con Python 2020.*

[11] R. Morris, M. Fillenz, *Neuroscience. Science of the Brain. An Introduction for Young Students*, British Neuroscience Association - European Dana Alliance for the Brain (2003).

[12] S. Russell, P. Norvig, *Artificial Intelligence. A Modern Approach*, Prentice-Hall Inc. (1995).

[13] G. Cybenko, *Approximation by Superpositions of a Sigmoidal Function*, Mathematics of Control, Signals and Systems **2** (4) (1989), 303 – 314.

[14] W. McCulloch, W. Pitts, *A Logical Calculus of the Ideas Immanent in Nervous Activity*, Bulletin of Mathematical Biophysics **5** (4) (1943), 115 – 133.

[15] Wikipedia the free encyclopedia, *Cerebral Cortex*, https://en.wikipedia.org/wiki/Cerebral_cortex

[16] J.L. Navarro, *Topología General. Lecture Notes for Academic Year 2017/2018*. Universidad de Zaragoza.

[17] P.J. Miana, *Integral de Lebesgue. Lecture Notes for Academic Year 2019/2020*. Universidad de Zaragoza.

[18] W. Rudin, *Real and Complex Analysis*, McGraw-Hill International Editions (1987).

[19] W. Rudin, *Functional Analysis*, McGraw-Hill International Editions (1991).

[20] P.J. Miana, *Curso de Análisis Funcional*, Colección Textos Docentes (2006).

[21] O. Ciaurri, *Análisis Funcional y de Fourier. Introducción al Análisis de Fourier. Lecture Notes for Academic Year 2020/2021*. Universidad de La Rioja.

[22] C.F. Higham, D.J. Higham, *Deep Learning: An Introduction for Applied Mathematicians*, SIAM Review **61** (4) (2019), 860 – 891.

[23] B. Wang, T.M. Nguyen, T. Sun, A.L. Bertozzi, R.G. Baraniuk, S.J. Osher, *Scheduled Restart Momentum for Accelerated Stochastic Gradient Descent*, arXiv preprint arXiv:2002.10583 (2020).

It is under review in
`https://openreview.net/pdf/24b9ad22e22211b234923e134f9a732cb25edcc7.pdf`

[24] P. Mehta, M. Bukov, C.H. Wang, A.G.R. Day, C. Richardson, C.K. Fisher, D.J. Schwab, *A High-bias, Low-variance Introduction to Machine Learning for Physicists*, Physics Reports **810** (2019) 1 – 124.

[25] D. Kressner, *Advanced Numerical Analysis. Lecture Notes Part 3a. Version 24.04.2015 (Optimization, Part 2)*,
`https://www.epfl.ch/labs/anchp/wp-content/uploads/2018/05/part3a.pdf`

[26] W. Su, S. Boyd, E.J. Candes, *A Differential Equation for Modeling Nesterov's Accelerated Gradient Method: Theory and Insights*, Journal of Machine Learning Research **17** (2016) 1 – 43.

[27] H. Karimi, J. Nutini, M. Schmidt, *Linear Convergence of Gradient and Proximal-Gradient Methods Under the Polyak-Łojasiewicz Condition*, Joint European Conference on Machine Learning and Knowledge Discovery in Databases (2016) 795 – 811.

[28] S. Boyd, L. Vandenberghe, *Convex Functions*, Lecture Notes from
`https://web.stanford.edu/class/ee364a/lectures/functions.pdf`

[29] M. Nielsen, *Neural Networks and Deep Learning* (2019)
`http://neuralnetworksanddeeplearning.com/chap3.html#overfitting_and_regularization`

[30] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, *Improving Neural Networks by Preventing Co-adaptation of Feature Detectors*, arXiv preprint arXiv:1207.0580 (2012).

[31] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research **15** (2014) 1929 – 1958.

[32] *imgaug*, `https://imgaug.readthedocs.io/en/latest/index.html`

[33] A. Shrestha, A. Mahmood, *Review of Deep Learning Algorithms and Architectures*, IEEE Access **7** (2019) 53040 – 53065.

[34] *Oscar, your New Data Scientist*, `http://oscar.calldesk.ai/`

[35] X. Glorot, Y. Bengio, *Understanding the Difficulty of Training Deep Feedforward Neural Networks*, Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (2010) 249 – 256.

[36] IBM Cloud Education, *Natural Language Processing*, IBM Cloud (2020) `https://www.ibm.com/cloud/learn/natural-language-processing`

[37] D.H. Hubel, *Single Unit Activity in Striate Cortex of Unrestrained Cats*, Journal of Physiology **147** (1959) 226 – 238.

[38] D.H. Hubel, T.N. Wiesel, *Receptive Fields of Single Neurones in the Cat's Striate Cortex*, Journal of Physiology **148** (1959) 574 – 591.

[39] K. Fukushima, *Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position*, Biological Cybernetics **36** (1980) 193 – 202.

[40] Y. LeCun, Y. Bengio, G. Hinton, *Deep Learning*, Nature **521** (2015) 436 – 444.

[41] A. LeNail, *NN-SVG: Publication-Ready Neural Network Architecture Schematics*, Journal of Open Source Software **4** (2019) 747 `http://alexlenail.me/NN-SVG/LeNet.html`

[42] P.R. Vlachas, J. Pathak, B.R. Hunt, T.P. Sapsis, M. Girvan, E. Ott, P. Koumoutsakos, *Backpropagation Algorithms and Reservoir Computing in Recurrent Neural Networks for the Forecasting of Complex Spatiotemporal Dynamics*, Neural Networks **126** (2020) 191 – 217.

[43] C. Tallec, Y. Ollivier, *Unbiasing Truncated Backpropagation Through Time*, arXiv preprint arXiv:1705.08209 (2017).

[44] S. Hochreiter, J. Schmidhuber, *Long Short-Term Memory*, Neural Computation **9** (8) (1997) 1735 – 1780.

[45] K. Cho, B. van Merriënboer, C. Gulcehre, F. Bougares, H. Schwenk, Y. Bengio, *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, arXiv preprint arXiv:1406.1078 (2014).