CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduate in Computer Engineering

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Bachelor's thesis

Bike Sharing Demand Forecasting using Recurrent Neural Networks

Author: Máximo García Martínez

Director: Antonio García Dopico

MADRID, JANUARY, 2021

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Abstract

Las redes neuronales recurrentes (RNN en inglés) son un tipo de redes neuronales que utilizan conexiones de retroalimentación. Existen diferentes aplicaciones para este tipo de redes neuronales como por ejemplo el estudio de la demanda de un activo durante un tiempo. Este proyecto se llevó a cabo para estudiarlas teóricamente y desarrollar modelos para predecir la demanda de bicicletas en la ciudad de Chicago basados en el enfoque de las RNN y comparar distintas arquitecturas. Los modelos empleados han sido redes neuronales Feedforward, Simple Recurrent Neural Network, LSTM y arquitecturas auto-regresivas. Los modelos predicen en base a una ventana deslizante de intervalos de 1 hora. La salida de los modelos es un vector con valores con la predicción de cada estación de la ciudad para un determinado intervalo. La fecha y la hora es la única entrada que la red neuronal y con ella se han obtenido resultados sobresalientes. Los resultados obtenidos con las redes LSTM y autoregresivas han obtenido resultados que compiten con proyectos de última generación, a pesar de que se han encontrado varios puntos de fallo y posibles mejoras.

Palabras clave: Redes neuronales, redes neuronales recurrentes, predicción, series de tiempo, alquiler de bicicletas.

Abstract

Recurrent neural networks (RNN) are a type of neural network that uses feedback connections. There are different applications for this type of neural network, such as studying the demand for an asset over time. This project was carried out to study them theoretically and develop models to predict the demand for bicycles in the city of Chicago based on the RNN approach and to compare different architectures. The models used were Feedforward Neural Networks, Simple Recurrent Neural Network, LSTM and auto-regressive architectures. The models predict on the basis of a sliding window of 1 hour intervals. The output of the models is a vector with values with the prediction of each station in the city for a given interval. The date and time is the only input that the neural network and with it outstanding results have been calculated. The results obtained with the LSTM and autoregressive networks have obtained results that compete with state-of-the-art projects, despite the fact that several points of failure and possible improvements have been found.

Keywords: Neural network, recurrent neural network, forecasting, time series, bike sharing.

1 INTRODUCTION TO THE PROJECT

1.1 Introduction

1.1.1 Bikesharing

Since the middle of the 20^{th} century, the private car has been the prevailing means of transport in most European cities, probably constituting the main factor in urban design today. It is a fast and efficient transport that has brought negative impacts to both the environment and its inhabitants: congestion, pollution, noise, high energy consumption per person transported, accidents that add up to many years of life lost and disabilities, contribution to the increase of CO_2.... All this is making the use of more sustainable transport increasingly attractive. These negative effects have raised concerns and promoted public transport, cycling or simply walking. It is also known that the economic growth of a territory is closely related to its mobility capabilities [1], so it is essential that there is a shift towards more sustainable and efficient modes of transport with its use optimally integrated into the evolution of a city. The bicycle is currently in the focus of interest by public agencies for being the most sustainable, cheapest and easiest vehicle to implement as a form of transport in cities.

A paradigm shift is emerging in Western world society towards a collaborative economy where access to a good is allowed rather than ownership. It is an economy that is thriving considerably this past decade due to the rise of social networks, real-time technologies, new consumption patterns and environmental concerns. The Product Service System (PSS) is a result of this model of economy and is based on the fact that you do not have to own a product to enjoy the need it satisfies. Mobility systems have also evolved and taken their place as a PSS and as a result of this, new car sharing, bike sharing or ridesharing companies have appeared.

In most cases, the bike sharing business consists of setting up numerous stations throughout large cities. The use of these bikes is simple: A user wants to go from one point to another point of the city, with an app installed on his cell phone rents a bike at one of these stations. Once verified by the bicycle company that the user has sufficient funds, one of the bicycles is unlocked. The user will start paying for each minute of use, a fee that depends on the company. Alternately, the company offers a payment method in the form of a monthly or annual fee. Once the user arrives at the destination, he or she will park the bicycle in some available bicycle parking space and lock it. The companies that manage and maintain these services usually have tools to facilitate the use by citizens, in the form of customer services, websites or mobile applications that report availability in real time. Although, they do not have information on predicting availability in the near future, at least as a service to citizens. In this aspect, is where this thesis takes its sense and its being.

1.1.2 Big Data

The advance of information technologies has generated new requirements that traditional tools are unable to provide. Finding answers with "traditional" processes with the huge amount of data nowadays would be very slow and expensive. These problems can be solved using new algorithms based on finding patterns in the data using data mining and machine learning techniques. The owner of the data can analyse and understand more beyond what conventional tools create new business opportunities that he can take advantage of.

Big data is defined as systems for storing large volumes of data. One of the first examples of the use of this technology can be attributed to Alan Turing, when he worked on the Enigma machine during World War II [2]. This machine managed to decipher the code used by the German army to encrypt its messages. It is considered as such since the amount of data to be able to fulfil its objective was gigantic. Since then, this field has not stopped growing and its evolution has always been linked to Artificial Intelligence. For the detection of the aforementioned patterns, it is necessary to use methods that Artificial Intelligence proposes.

But it has not been until the birth of the Internet and specifically of social networks that there has been a data revolution. This has led to the generation of massive data that, together with the increase in computing power, new industries have emerged around this technology. This amount of data has also facilitated that new methods of Machine Learning have been invented making the field of Artificial Intelligence a booming field that is proposing a large number of solutions.

1.1.3 Project summary

Having commented on the concepts of bikesharing and Big data, the present work combines both notions. This practical work aims to demonstrate the capabilities of neural networks to perform prediction tasks and the study of their operation, behaviour and optimisation.

The problem to be solved is the prediction of bicycle use in the different stations in a city. The aim is to optimise service, customer satisfaction and profitability. The data analysis will predict where bicycles will be needed, in a suitable state for use, in the immediate future. In addition, many of the bicycles that are rented are electric and if they are not charged at the station, the company has to take care of them by taking them to a charging point and relocating them in the city. With the help of a model that predicts where there will be more demand in the near future, bicycles could be located to optimise their supply.

For the development of this project a simple problem has been chosen which will be solved with different neural network architectures and will be progressively improved by adding more concepts and complexity in each iteration. Different neural network ar-

chitectures and their different patterns will be studied and compared. It will use architectures such as feed-forward, Recurrent Neural Network (RNN), Long-Short Term Memory (LSTM) and Autoregressive (AR). The reason for the use of each of these networks and the results obtained will be explained.

A dataset provided by the City of Chicago and the Wibee Company will be used. This dataset contains crucial information regarding all trips that were rented during the years 2014 and 2019. The city of Chicago has more than 600 stations of this company and more than 26 million trips were made during this period.

1.2 Methodology and development

The present work consists of a theoretical part, on the study of neural networks, focusing on the recurrent ones, and on the other hand a practical part. Both parts will converge, complementing each other and with the help of the tutor, doubts or suggestions for improvement will be resolved.

1.3 Goals

The goals to be achieved are the following:

1.3.1 General goals

- Study on neural networks: Understanding of neural networks, their functioning and algorithms used in them.
- Study on recurrent neural networks as an extension of the neural networks and challenges that comes with it.
- Enable decision makers in the bike sharing system to proactively rebalance the performance and assistance provided at service stations based on accurate predictions of future bike flow.

1.3.2 Specific goals

- Writing of this document referencing the whole process carried out as well as an explanation of the concepts learned.
- Use of Python 3 data mining libraries such as: NumPy or Pandas.
- Use of machine learning Python 3 libraries such as: Tensorflow or Keras.
- Creation of a feed-forward neuronal network and study of its behaviour with different values for its hyperparameters.
- Creation of a recurrent neural network and study of its behaviour with different values for its hyperparameters.

- Creation of an LSTM neural network and study of its behaviour with different values for its hyperparameters.
- Creation of an AR neural network and study of its behaviour with different values for its hyperparameters.
- Compare the results of the different models and justify their metrics.

2 STATE OF THE ART

2.1 Artificial Intelligence

Artificial Intelligence (AI) belongs to the field of Computer Science that has achieved great recognition and popularity during the last decade. Despite being a discipline that has been developed over the last five years, it has already managed to provide solutions to a large number of problems that in the past were considered unsolvable or highly complex. Achievements such as transcription from voice to text [3] or vice versa [4], automation of processes with robots [5], automatic driving [6], style transfer [7] are some of the most famous examples. The problems that AI attempts to solve cover problems that affect any area of life.

Providing an exact definition of AI is difficult, since it is a concept that depends on the very definition of intelligence that today has multiple interpretations. Many authors propose their own definition of AI [8]–[14] and if we take all of them, we could extract a common idea: AI is a discipline of computer science that aims to create entities that can **imitate** intelligent behaviour, from analysing patterns, classifying elements in an image, predicting values to recognising voices, driving, trading on the stock exchange and many other entities in which a system can simulate intelligent behaviour.

The definition of AI can mainly be grouped into one of the following approaches [15]:

1. Systems that think like humans [8], [9]: The capacities of the system must be specific to human beings. Alan Turing (1950) proposed the Turing Test [14], which aims to test these capacities and if it is passed, it will be demonstrated that the system will behave like a human being. The reliability of this test is questionable since it is easy to create a system that imitates human behaviour. Imitating is not having intelligence of your own [15].

2. Systems that act like humans [12], [13]: Turing's test did not consider the physical person but only their intelligence. Hanard (1991) proposed Turing's total test [16] which considers all kinds of external stimuli, perception skills and object manipulation. In other words, it adds new requirements to the system: computer vision and robotics that require new capabilities: robotics, natural language processing, knowledge representation, automated reasoning, and automatic learning. These six disciplines are six of the AI disciplines currently being studied as will be explained later [15].

3. Systems that think rationally [10], [11]: The system would try to imitate the structure and processing of information as a human would. This would require research and further development of precise theory of mind, in order to translate this theory into a system. This could be done either by an introspective study on an individual basis or through psychological experiments that serve to make different definitions and requirements so that the system can be programmed. To carry out a test, the

same data would be provided with a structure and reaction times would be measured. If the structure and output data, together with the reaction time are similar to those of a human, there is evidence that some of the programmed mechanisms can be compared with those used by a human and therefore be considered intelligent [15].

4. Systems that act rationally [14], [17]: It would no longer be systems that are built but agents. An agent is something that reasons (agent comes from the Latin *agere*, to do). This agent is expected to act with the intention of achieving the best result, and can even take the decision to inaction, if it considers it appropriate in situations of uncertainty.

   It is considered that this agent must be able to make inferences when establishing which actions will help to achieve the desired objective. To obtain a correct inference not only depends on rationality, since sometimes the best decision will not be the consequence of a slow process of deliberation but of a reflex action or even situations where there is nothing right to do and a decision has to be made. This approach is the most general and complex at a technical level because it proposes the development of different parts: a rational part that is common to the other approaches and a part of logical inference. On the other hand, it is simpler to carry out tests with these agents since rationality is well defined mathematically, and therefore it can be broken down to design agents that can achieve this in a verifiable way [15].

Today, created artificial intelligences are what is known as the weak type. These are systems that can exceed the capabilities of a human in some task. However, if you select one of these systems that excels in a very specific domain and try to get it to perform another task, the result of that task will be much worse than that of a human being. This ability to be able to multi-task is a much sought-after feature that is still being researched in all AI departments today. At the moment, there are only weak AI's capable of doing one or more groups of tasks extraordinarily well. In contrast, systems with strong AI are systems that can carry out tasks in different domains, but there is no example of such a system yet [15].

It is important to mention that some of the four approaches to defining AI explained above are aimed at imitating intelligent behaviour. Trying to imitate a behaviour can be relatively easy. For example, a system can be programmed to play chess according to a set of rules and predicates. Being a machine, it may have memorised similar games in the past and know what strategies its opponent took and thus have access to much more information than its opponent. In this way, the system will be able to choose the best decision in each move and will be unbeatable, but if we try to make it play with a strategy that it has not seen before, it will not be able to use a strategy that it has tactically valid, that is to say, it will not work for other cases because the system is only imitating and not learning.

Imitating does not mean that such behaviour is in essence cognitive behaviour, however, according to the definition given of AI, the system playing chess would be considered a system with AI, even if the mechanics of the move have been memorised. That is why within this field of computer science we can find different subcategories that respond to different intelligent behaviours. These subcategories are:

- Robotics.
- Computer vision.
- Natural Language Processing.
- Knowledge Representation.
- Automatic Reasoning.
- Automatic Learning.

Above all, if there is one category that defines us as intelligent agents, it is the ability to learn. This capacity is the focus of this work and is the object of study of Machine Learning which we will see in more detail below.

2.2 Machine learning

Machine Learning (ML) is the branch of AI that studies how to give machines the ability to learn, understanding it as the generalisation of knowledge from a set of experiences. This learning can be divided into [15]:

- Supervised: The algorithm learns on a labeled dataset, providing an answer key that the algorithm can use to evaluate its accuracy on training data [18].
- Unsupervised: Given the unlabeled data, the algorithm tries to capture meaning by extracting characteristics and patterns by itself [18].
- Reinforced: An algorithm is trained with a system of rewards, providing feedback when an artificial intelligence agent performs the best action in a particular situation [18].

Machine Learning is the main branch of AI, since the other branches either imitate behaviour or learn from experience, i.e., they use ML to perform the task at hand. It is one thing to program a machine so that it can move, but it is another thing for the machine to learn to move. Likewise, it is not the same to program which components form a face of a person as it is to automatically learn that it is a face. This paradigm shift is what differentiates the ML from the AI, and that is why it should not be thought that they are the same concepts.

Machine Learning is the systematic study of algorithms and systems that improve your knowledge or experienced performance. In this subcategory there are different types of applications, some of them [15]:

- Regression models: These are models that predict the value of one or more variables given an input vector. A linear function is the simplest regression model, but the more complex regression models are formed by a fixed set of non-linear functions, known as base functions [19]. Algorithms such as the acrfullsvm or the Neural Network (NN) are based on this type of model. The latter will be used in this paper.
- Decision trees
- Classification models
- Grouping techniques

2.2.1 Models

The universe we know is in constant evolution and is complex, chaotic and enigmatic, however, the intelligence of the human being manages to give meaning to all that chaos in a search for the elegance and symmetry that is hidden among the patterns it identifies in our reality. The ability to detect patterns and use them for our own good has been one of the main reasons for the development of the human species. Science has allowed us to understand, observe and simplify the world by turning all this enigma into knowledge, that is, by reconstructing reality through models [20].

A model is a conceptual and simplified construction of a complex reality allowing a better understanding of that reality. There are many models that we use every day, for example, a map. A map allows us to reflect a three-dimensional world on a two-dimensional surface, eliminating information that we do not need to process, such as environmental artifacts or types of vegetation. Another example is a physical equation, where different constants and values are related and in this way, we can approximate the physical behaviour of reality. A score is another example of a model. It reflects information on how different instruments must be coordinated in order to always produce the same song. The frequency spectrum of the song could be used to better represent reality, but this is obviously much more complex to interpret as a human being. In short, a model seeks a balance between correctly representing reality and being simple so that it can be used [21].

Let's imagine that you want to model the weather. For this purpose, different pieces of evidence are collected, and, after observation, a first model can be made:

> "The summer will be sunny, hot and clear. The winter will be cold, cloudy, and rainy."

If you keep collecting evidence, you will soon realise that this model is very simple as there will be days that are cold in summer and hot in winter. There will be storms in summer and clear days in winter. This will lead to improvements in the model in each iteration.

But if you keep studying other evidence in some tropics or in some poles, for example, you will conclude that the model was very simple and therefore more rules will have to be added to make this model more similar to reality anywhere in the world.

In the end, you will get a very complex model with all the exceptions and conditions. An alternative to this is to make use of probability to be able to say mathematically that most of the time a summer day will be sunny and not such a complex model to depend on.

Probability is the perfect tool to limit the uncertainty about a subject due to lack of knowledge or data or to avoid the work of making a complex model that would lead to less understanding for the human mind and dispersion in its approach. Being able to use a probability is much easier than having to study all the physical conditions of an environment and the behaviour of its entities in order to know for sure what is going to happen. Using probability to build models results in probabilistic models. These models compress many of the variability of our reality based on probabilities, making it easier to manage the information we receive from the environment [21].

Our brain applies similar schemes to these probabilistic models and it is thanks to them that we have the ability to conceptualise, predict, generalise, reason or learn. For this reason, discovering what these models are is one of the basic objectives of the field of Machine Learning and one of the fundamental tools of AI.

2.3 Human brain neural networks

The "artificial" neural networks are inspired by the organic brain of human beings brought to the computers. This is not a perfect comparison, but where there are most similarities between a neural network in the brain and an artificial one is in the neurons that constitute it [22]. Neurons are the simplest parts of the neural network, and by bringing several of them together, networks can be formed. In the following paragraphs we will look at the basic functioning of a neuron and some of the processes of the human brain so that it can be compared with the functioning of an artificial neuron network.

Neurons are a specific type of cell responsible for sending information using electrical signals called nerve impulses to other neurons or to effector organs. Their study dates back to 1888, when *Ramón y Cajal* began to postulate a theory known as the "neuron doctrine"[23]. In it, the concept of the neuron as a discrete unit and the law of dynamic polarisation, a model capable of explaining the unidirectional transmission of the nerve impulse, are highlighted. He later published many papers, among other contributions he described the synaptic cleft, suggesting that communication between neurons was made by chemically well-defined molecules, called neurotransmitters, using as an example one of the best-known molecules, acetylcholine [24]. Living beings can react to changes in the environment. In multicellular beings the capture of these changes and the response

or lack of it is the function of the nervous tissue, formed by neurons. The most easily "understood" example is the response to a stimulus, be it a movement produced by a muscular contraction [25].

The tasks of a neuron are:

- Conduction and transmission of nervous impulses. The simplest mechanism of nerve action is represented by the monosynaptic reflex arc, which consists of a neuronal circuit formed by two neurons: A sensory neuron has a receptor at one end to receive the stimulus (in the more "classical" neurons it is in the dendrites). The information is propagated along the axon by transient variations of the resting potential. This is a change in the electrical transmembrane potential of the cell, which produces an excitation wave that starts at the sensing end and is exhausted at the end of the axon with the release of the neurotransmitter into the synaptic cleft. This neurotransmitter will be the one that causes another electrical potential in the next neuron (which will capture it in its dendrites) or an effect if it is an effector organ that is on the other side of the cleft. The electrical potential that is triggered is of all or nothing type in a neuron. The gradation of intensity of a response will depend on the recruitment of neurons achieved by the intensity and type of stimulus and the transmission of information to other neurons will depend on the amount of neurotransmitter released that will stimulate one or many neurons (or none, one or many effector cells) [26].
- Store instinctive and acquired information. Adaptation to specialised functions is achieved by means of different types of extensions [26]. Unlike the memory that we have today on hard disks, the memory in the brain cannot be stored as such in tangible elements, i.e., memories are not stored in the neurons. Memories are signals. Short-term memory is a pattern of signals in the brain that occur in specific neurons in the prefrontal cortex. That signal starts as an electrical potential created by ions entering and leaving the cells at one end of the neuron creating an electrical signal which triggers a release of chemicals that will produce in the next neuron a change of its electrical potential in its cell membrane, the transfer of information is a release of a molecule that produces an electrical change in the next neuron. Basically, electricity and chemistry are responsible for a signal being transported through many neurons [27].

  In short, a memory is a chain reaction and as mentioned above, one neuron is connected to at least thousands of other neurons. So, the number of unique combinations and patterns that can be generated from connections between neurons is almost infinite [27].

  For a new memory to be created, there must first be a stimulus. These stimuli are captured by one of the neurons and send that signal to other neurons with a specific pattern. This newly created memory is totally unique and, therefore, this memory is the set of neurons activated with a specific activity between them. In

the future, if that pattern is activated again it will cause the memory to return in the form of memories to the person. If it is reactivated again and again (or there are accompanying stimuli, important for the receiving subject, among other details) the pattern will be stored in the long-term memory area in the hippocampus. In this area, neurons are closer together and the signal between them is stronger so that chemicals and electrical signals do not have to travel as far and over time those memories become more entrenched as memories are reactivated more frequently. On the other hand, if a memory is not reactivated given a period of time it may fade or become nuanced or changed into new memories [27].

2.4 Artificial neural networks. Forward-pass algorithm.

Neural networks are algorithms used in the field of Artificial Intelligence and try to solve problems in which one works with a large amount of data trying to find patterns in them. Furthermore, it is one of the few alternatives to other algorithms that can treat large volumes of data.

The beginnings of neural networks date back to the 1950s, when McCulloch and WalterPitts [28] worked on a mathematical model that resembled the behaviour of a neuron that they knew. The scientist Frank Rosenblatt, inspired by this work, developed what are known as networks of perceptions. This was the first approach to what is now known as neural networks [29].

2.4.1 Perceptron

A perceptron, or neuron is a basic unit of inference in the form of a linear discriminator, it is the basis of what is known today as an artificial neuron [29]. Basically, a perceptron takes several binary values as input $x_1, x_2, \ldots, x_n$ and produces a single output value y.

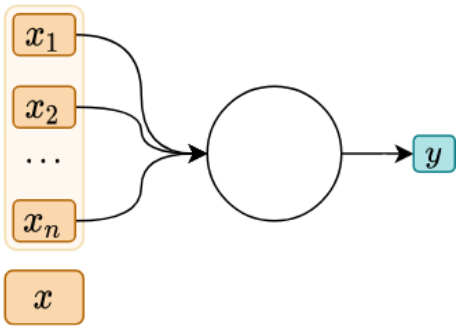


Figure 1: Representation of how a perception works

In addition to the x-vector and the y-value, the perceptron has two other components:

- The weight vector w which expresses the importance of the respective inputs for the

output and will have the same number of elements as x. This vector will be used to calculate the weighted sum $\sum_i w_i x_i$.

- The threshold or bias (b): The value of the summation will be checked to see if it is lower or higher than this b value. Depending on this, the y-value will be 0 or 1. The threshold is a value that represents how easy it is to produce a 1 in y-value. For a perception with a very large bias, it is extremely easy for the perception to produce a 1. But if the bias is very negative, then it is difficult for the perception to produce a 1.

Both the vector w and the threshold value b are parameters that must be known in advance. The value y produced by the perception is given by the following equation:

$$y = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq b \\ 1 & \text{if } \sum_i w_i x_i > b \end{cases} \tag{1}$$

Viewed graphically:

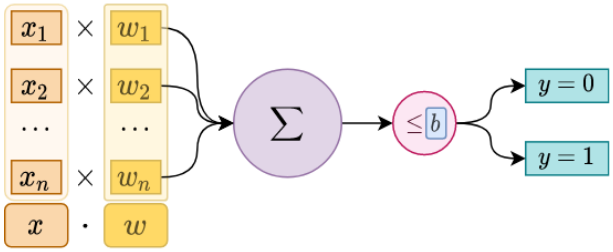


Figure 2: Representation of a perception

Simplifying equation 1 by using the vector product and changing side b:

$$y = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \tag{2}$$

Viewed graphically:

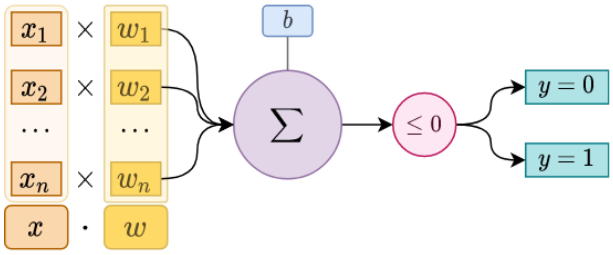


Figure 3: Representation of a perception

Perceptrons can be grouped in layers and these layers can be grouped in a network. The operation of a neural network can be seen by programming a logical XOR gateway that receives two binary input arguments and emits an output as shown below:

A	B	Ouput
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: XOR gateway

A neural network that acts as an XOR gateway is the following:

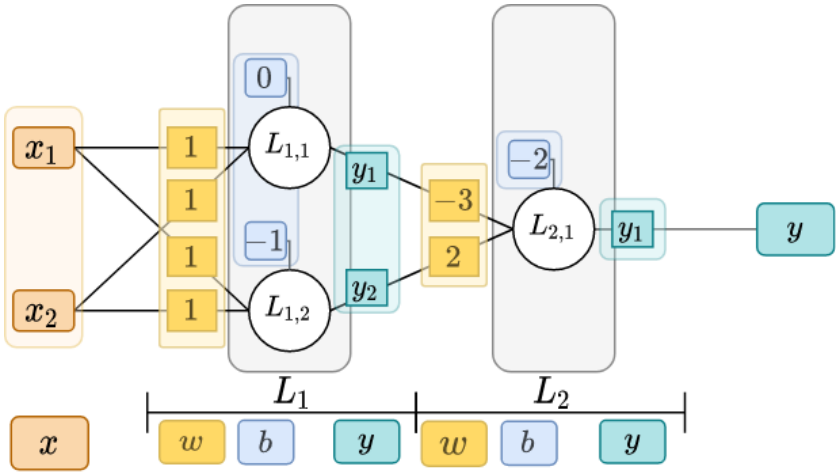


Figure 4: Perception network programmed for a logic gate XOR

Networks are usually represented with a unidirectional weighted graph. The network is divided into layers, in this case into two layers (L_1 and L_2). Each layer has at least one neuron, which will be represented by $L_{l,i}$ being l the number of the layer and i the index of the neuron within the layer. A network can have as many layers as desired.

The first layer is called the input layer. The last layer is called the output layer. These two layers are the ones that indicate the size of the input vector and the size of the output vector they produce. If a network consists of only one layer, that same layer will be called the input and output layer. On the other hand, if a neural network has more than two layers, all the layers between the input layer and the output layer are called hidden layers and the auto-adjustment that is done by the backpropagation algorithm (see section 2.5.4) is difficult to understand and explain and that is why neural networks are sometimes called black boxes.

The values of the weights are represented as if they were the weights of the edges. The amount of weights that a layer will have associated with it is given by multiplying the number of neurons in the L_l layer by the number of neurons in the previous L_{l-1} layer. Finally, each neuron will have an associated b-value which will be represented as an independent term connected to the neuron by a line. For example, the neuron $L_{2,1}$ has the vector $w = [3, -2]$ associated with it and the value $b = -2$.

With the network already explained, the problem that had been proposed it can be solved: Programming a logical XOR gateway with a neural network. This network receives two binary input values and will return a single binary value. As an example, the calculations are made for different examples:

$$
\begin{aligned}
For \quad & \boxed{x = (0,0)} \\
y^{L_{1,1}} \quad & \Longrightarrow \quad \begin{pmatrix} 1 & 1 \end{pmatrix} \cdot x^T + 0 = 0 \implies y^{L_{1,1}} = 0 \\
y^{L_{1,2}} \quad & \Longrightarrow \quad \begin{pmatrix} 1 & 1 \end{pmatrix} \cdot x^T - 1 = -1 \implies y^{L_{1,2}} = 0 \\
y = y^{L_{2,1}} \quad & \Longrightarrow \quad \begin{pmatrix} 3 & -2 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix} - 2 = -2 \implies \boxed{y = 0}
\end{aligned}
$$

$$
\begin{aligned}
For \quad & \boxed{x = (0,1)} \\
y^{L_{1,1}} \quad & \Longrightarrow \quad \begin{pmatrix} 1 & 1 \end{pmatrix} \cdot x^T + 0 = 1 \implies y^{L_{1,1}} = 1 \\
y^{L_{1,2}} \quad & \Longrightarrow \quad \begin{pmatrix} 1 & 1 \end{pmatrix} \cdot x^T - 1 = 0 \implies y^{L_{1,2}} = 0 \\
y = y^{L_{2,1}} \quad & \Longrightarrow \quad \begin{pmatrix} 3 & -2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} - 2 = 1 \implies \boxed{y = 1}
\end{aligned}
$$

$$
\begin{aligned}
For \quad & \boxed{x = (1,1)} \\
y^{L_{1,1}} \quad & \Longrightarrow \quad \begin{pmatrix} 1 & 1 \end{pmatrix} \cdot x^T + 0 = 2 \implies y^{L_{1,1}} = 1 \\
y^{L_{1,2}} \quad & \Longrightarrow \quad \begin{pmatrix} 1 & 1 \end{pmatrix} \cdot x^T - 1 = 1 \implies y^{L_{1,2}} = 1 \\
y = y^{L_{2,1}} \quad & \Longrightarrow \quad \begin{pmatrix} 3 & -2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} - 2 = -1 \implies \boxed{y = 0}
\end{aligned}
$$

2.4.2 Linear Regression

In fact, the network architecture in figure 4 uses one neuron to simulate an OR logic gate ($L_{1,1}$) and another neuron to simulate an AND logic gate ($L_{1,2}$). Seeing the logic of an XOR gate:

$$A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B) = (A \vee B) \wedge (\neg A \vee \neg B) \tag{3}$$

It makes sense to use the OR and AND gates in the network. You can see graphically how each neuron works in the following image:

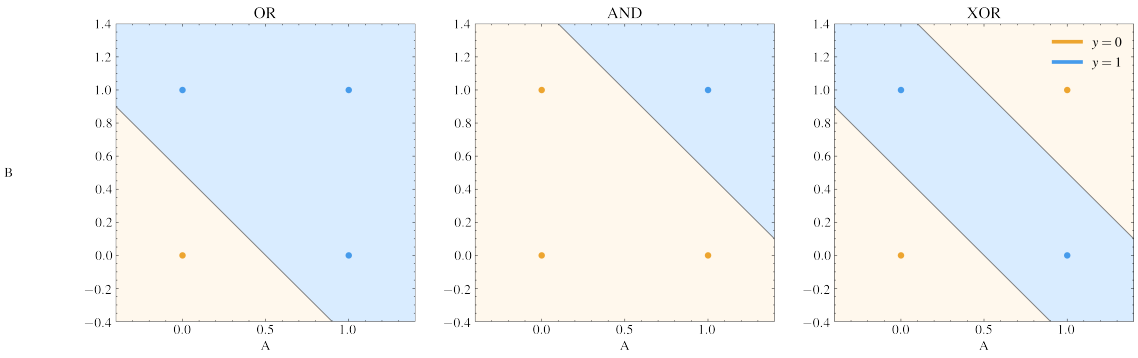


Figure 5: OR($L_{1,1}$), AND($L_{1,2}$) y XOR($L_{2,1}$) logic gates

Each of these neurons is performing a sorting task, simply checking whether the input data is on one side of the line or the other. This is because the perception has been defined in such a way that it can only return 0 or 1. But eliminating that classification step that has been defined in equation 2, would result in a linear regression such that:

$$y = w \cdot x + b \tag{4}$$

A linear regression is the other type of task that a neuron can solve. This is one of the main differences between modern neurons and perceptrons. Perceptrons are only programmed to perform one sorting task, but neurons can be programmed for other types of cases. Although as will be explained later, a perceptron is a type of neuron with a step activation function (see section 2.4.3).

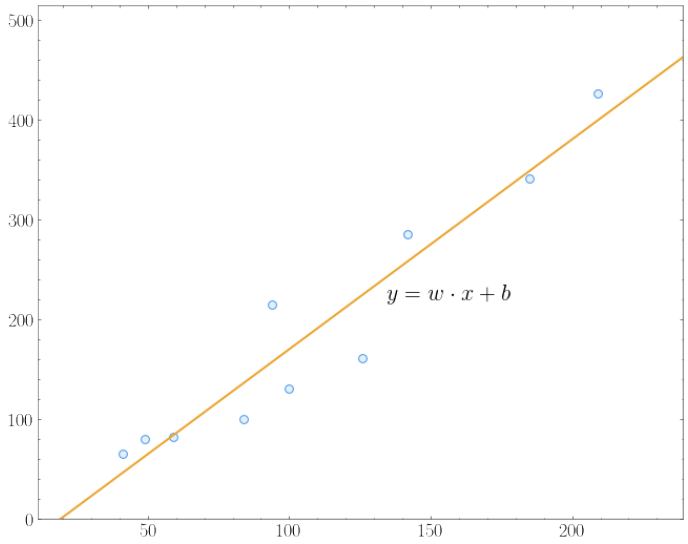


Figure 6: Example of linear regression in a two-dimensional space

A linear regression is a method that studies the relationship between several variables and thus generates a model that can be used to estimate other values. Mathematically, in a multidimensional space n, a regression is defined as follows:

$$y = w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n \tag{5}$$

In this equation there is an independent term w_0, and for each dimension, there will be a value w and an x-value associated with it. Geometrically, in a space of two dimensions, z will be a line, a plane if they are three dimensions and a hyperplane for greater than three dimensions, which cannot be represented graphically. In fact, it can be observed that equation 4 is the function of a line, being b the independent term.

From this point onwards, z will be used to refer to the linear regression that is calculated in a neuron:

$$z \equiv y = w \cdot x + b \tag{6}$$

2.4.3 Activation function

The neural network explained until this point can be proved mathematically that the effect of adding up many linear regression operations is equivalent to a single linear regression. This is demonstrated in section 2.4.5. The perception that has been created can collapse to the equivalent of having a single neuron. To avoid this, we need to calculate values that do not result in a linear function (a straight line in two dimensions), we need each of these neurons to apply some kind of non-linear manipulation that distorts their output values and for this we use the activation functions [29].

As an example of the importance of activation functions, it will be shown with an example with a model with only linear functions compared to a model using non-linear functions. To do so, the following example is presented: we want to create a regression model that tries to predict the value of a sine function. Since the sine is a non-linear function, non-linear functions will be needed to solve the problem.

$$y = \sin(x) \tag{7}$$

Code will also be shown to introduce the operation of the libraries that are popularly used in the Machine Learning world. As it is explained in section 3.2, in this work `keras`(neural networks), `numpy`(vectors and matrices) and `matplotlib`(represent functions) among others have been used.

As explained in the training section (section 2.5.1), a dataset is needed for the network to be trained. The dataset will be created as follows:

```
# real sin data
x = np.linspace(0, 2*np.pi, 1000)
y = np.sin(x)

# some noise is added to the real dataset to make it
# look like a real world data set
noise = 0.1
```

```
noisex = x + np.random.normal(-noise, noise, 1000)
noisey = y + np.random.normal(-noise, noise, 1000)
```

Graphically a series of points would be scattered around the sine function as expected:

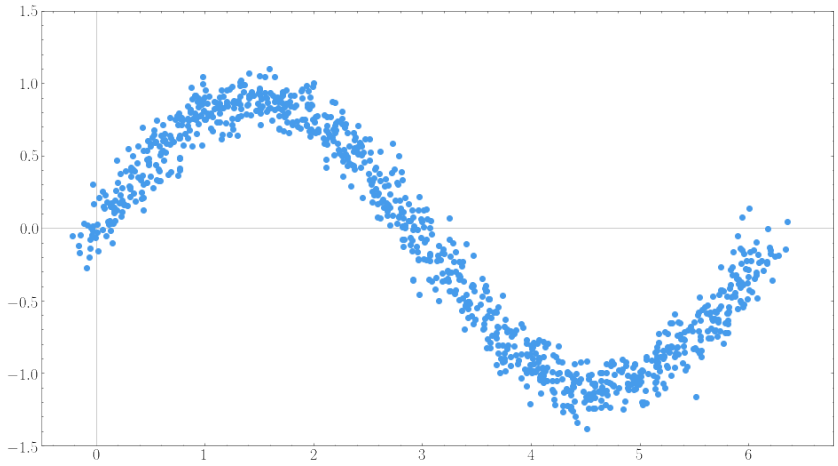


Figure 7: Dataset emulating a sine function.

Two models will be created, i.e. two networks with different architectures: one model with linear activation functions (`linear_model`) and one with non-linear activation functions (`relu_model`).

The first model is defined as follows:

```
linear_model = Sequential()
linear_model.add(Dense(32, activation='linear'))
linear_model.add(Dense(32, activation='linear'))
linear_model.add(Dense(1))
```

What is done in that code is to create the architecture of the network. It will be a network with three dense layers. The first two layers are the hidden layers of the model and it is indicated that 32 neurons are wanted in each layer with the `linear` activation function. The last layer is simply so that the model only returns a single value.

The other network will use a non-linear ReLU activation function, explained below. The code for the architecture of this network is as follows:

```
relu_model = Sequential()
relu_model.add(Dense(32, activation='relu'))
relu_model.add(Dense(32, activation='relu'))
relu_model.add(Dense(1))
```

The model is then compiled. In this step, Keras will be told that both mandatory and optional parameters are wanted for the network. These parameters will be explained in the next sections. Right after that the models are trained.

```
# Compile linear model
linear_model.compile(loss='mean_squared_error',
                     optimizer='adam')
# Train linear model
linear_model.fit(noisex, noisey, epochs=100, verbose=0)

# Compile ReLU model
relu_model.compile(loss='mean_squared_error',
                   optimizer='adam')
# Train ReLU model
relu_model.fit(noisex, noisey, epochs=100, verbose=0)
```

To study the models you can simply use the `predict()` function of the model:

```
predicted_by_linear = linear_model.predict(x)
predicted_by_relu = relu_model.predict(x)

# Plot predictions
plt.plot(x, predicted_by_linear)
plt.plot(x, predicted_by_relu)
```

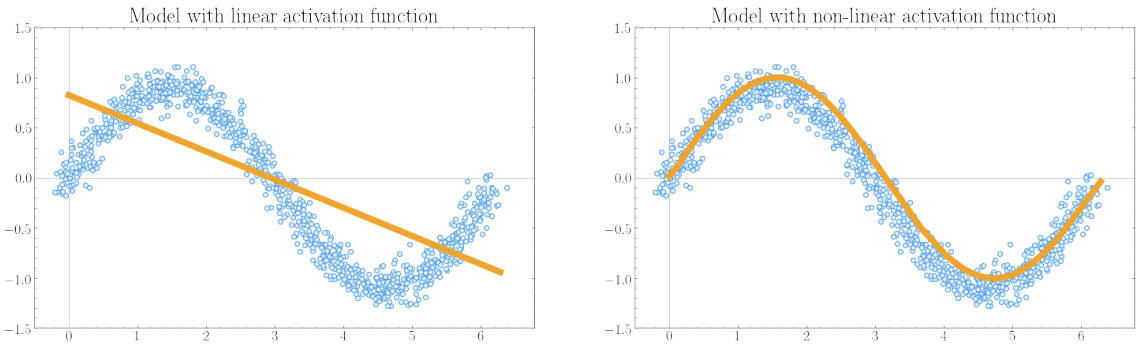


Figure 8: Comparison of linear and non-linear activation functions

As can be seen, the model using a linear trigger function is reduced to a single line while the model using a non-linear trigger function is adapted correctly.

In short, the trigger function is a function that applies to the result of the weighted sum of the input values, i.e., a z. The aim is that it distorts the result of the neuron by adding non-linear deformations to it so that the computation of several neurons can be effectively linked. This activation function will be represented as follows:

$$a(z) = a(w \cdot x + b) \tag{8}$$

By updating the image that has been used before to define a perception, it would look like this:

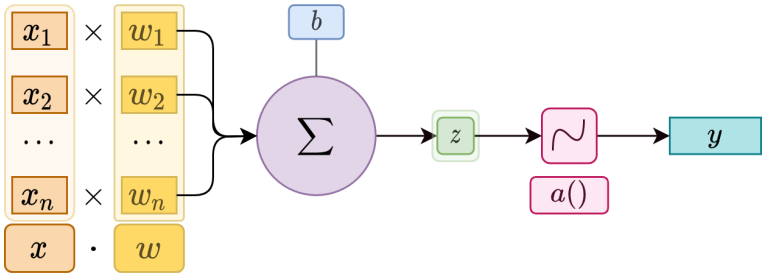


Figure 9: Representation of how a neuron works.

There are several types of activation function, these are the most used:

- Linear: It is simply the equation of a line and therefore a linear function. It is usually used in the last layer in a regression model, that is, a model that performs a regression task and not a classification task.

$$a(z) = z \tag{9}$$
$$a'(z) = 1 \tag{10}$$

- Step: The objective of the function is to convert the values to 0 or 1 as a function of b. The purpose of this activation function is to imitate a neuron that "activates" or "does not activate" based on the input information. Traditionally, this function was used in networks of perceptrons and in fact, it is the one used in equation 1:

$$a(z) = \begin{cases} 0 & \text{si } z \leq 0 \\ 1 & \text{si } z > 0 \end{cases} \tag{11}$$
$$a'(z) = 0 \tag{12}$$

- Rectified Linear Unit: It is a linear function when it is positive and constant at 0 when the value is negative. It is widely used because it is a non-linear function and similar to the linear function, so it is fast to compute.

$$a(z) = max(0, z) \tag{13}$$
$$a'(z) = \begin{cases} 0 & \text{si } z \leq 0 \\ 1 & \text{si } z > 0 \end{cases} \tag{14}$$

  There is a variant of this activation function (known as leaky). In this variant, the slope is changed to a value s of the negative part with respect to the abscissa axis.

$$a(z) = \begin{cases} s & \text{si } sz \leq 0 \\ z & \text{si } z > 0 \end{cases} \tag{15}$$
$$a'(z) = \begin{cases} s & \text{si } sz \leq 0 \\ 1 & \text{si } z > 0 \end{cases} \tag{16}$$

- Sigmoid (σ): This is the most commonly used function in neural networks. The distortion it produces to very large values make them saturated at 1 and very small values makes them saturated at 0. This function is very useful to represent probabilities since they always come in the range of $[0, 1]$ and as explained in section 2.2.1, probability is the perfect tool for Machine Learning. Moreover, this function, together with $tanh$, achieves a fluidity that is not achieved with the functions explained above. A small change in w or b will produce a small change in the output value.

$$a(z) = \frac{1}{1 + e^{-z}} \tag{17}$$
$$a'(z) = a(z)(1 - a(z)) \tag{18}$$

- Hyperbolic tangent ($tanh$): This is similar to the sigmoid, but the range of output values is $[-1, 1]$.

$$a(z) = tanh(z) \tag{19}$$
$$a'(z) = 1 - tanh(z)^2 \tag{20}$$

Each of the functions explained in graphic form is shown below:

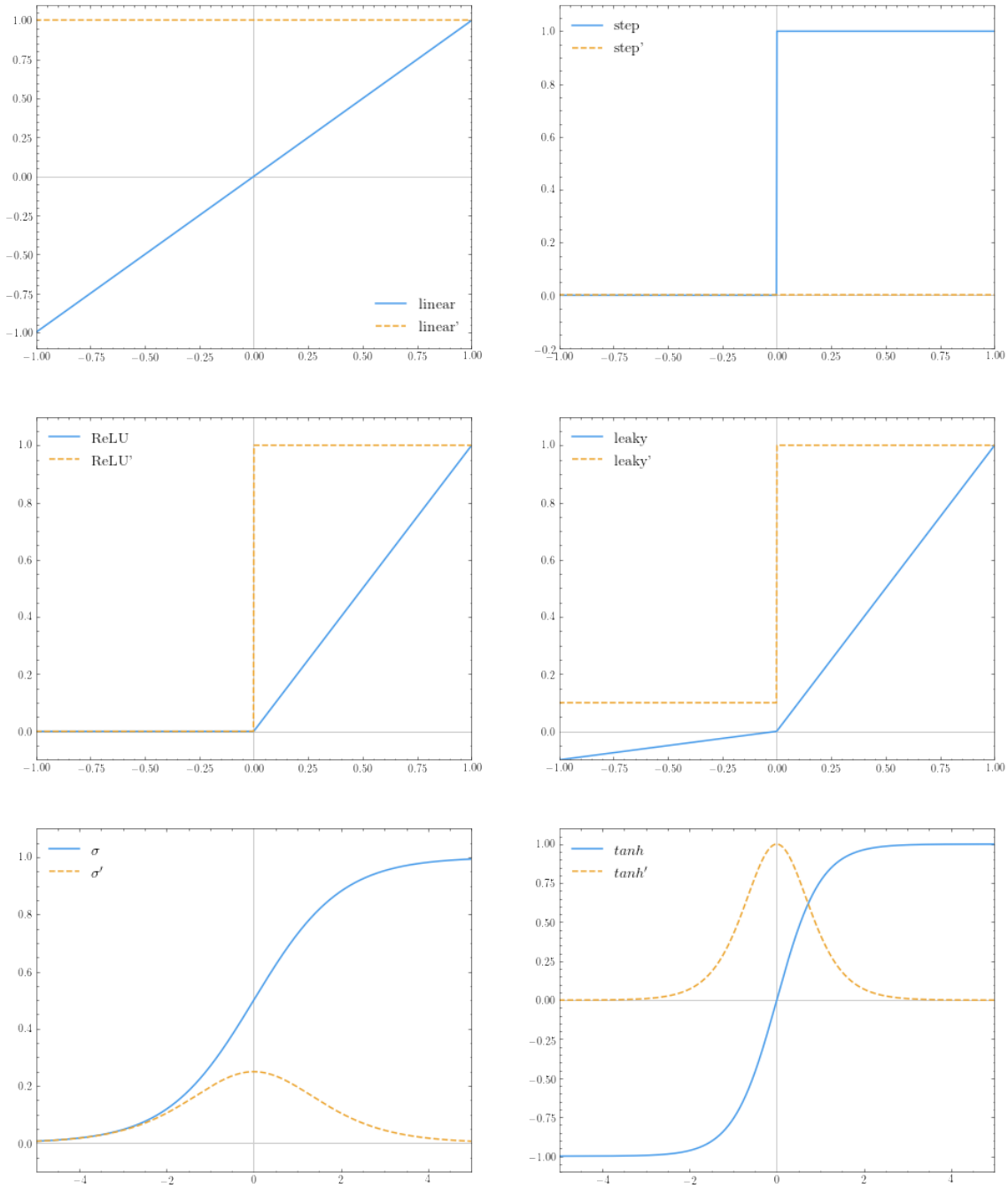


Figure 10: Graphs of the activation functions.

Each neuron may have an associated activation function, but by convention only one type of activation function is used for each layer. Both solutions give similar results and neither brings in principle significant improvements to the outcome of the model. The main difference is the added complexity of developing a network if a different activation function is to be used per neuron. In conclusion, the use of a single activation function on all the neurons in a layer is always chosen for simplicity of development.

2.4.4 Working with matrixes

To facilitate the explanation from this point, we proceed to explain different parts of the network and their mathematical notation with some simplification.

The following definition of a neural network is used as a starting point:

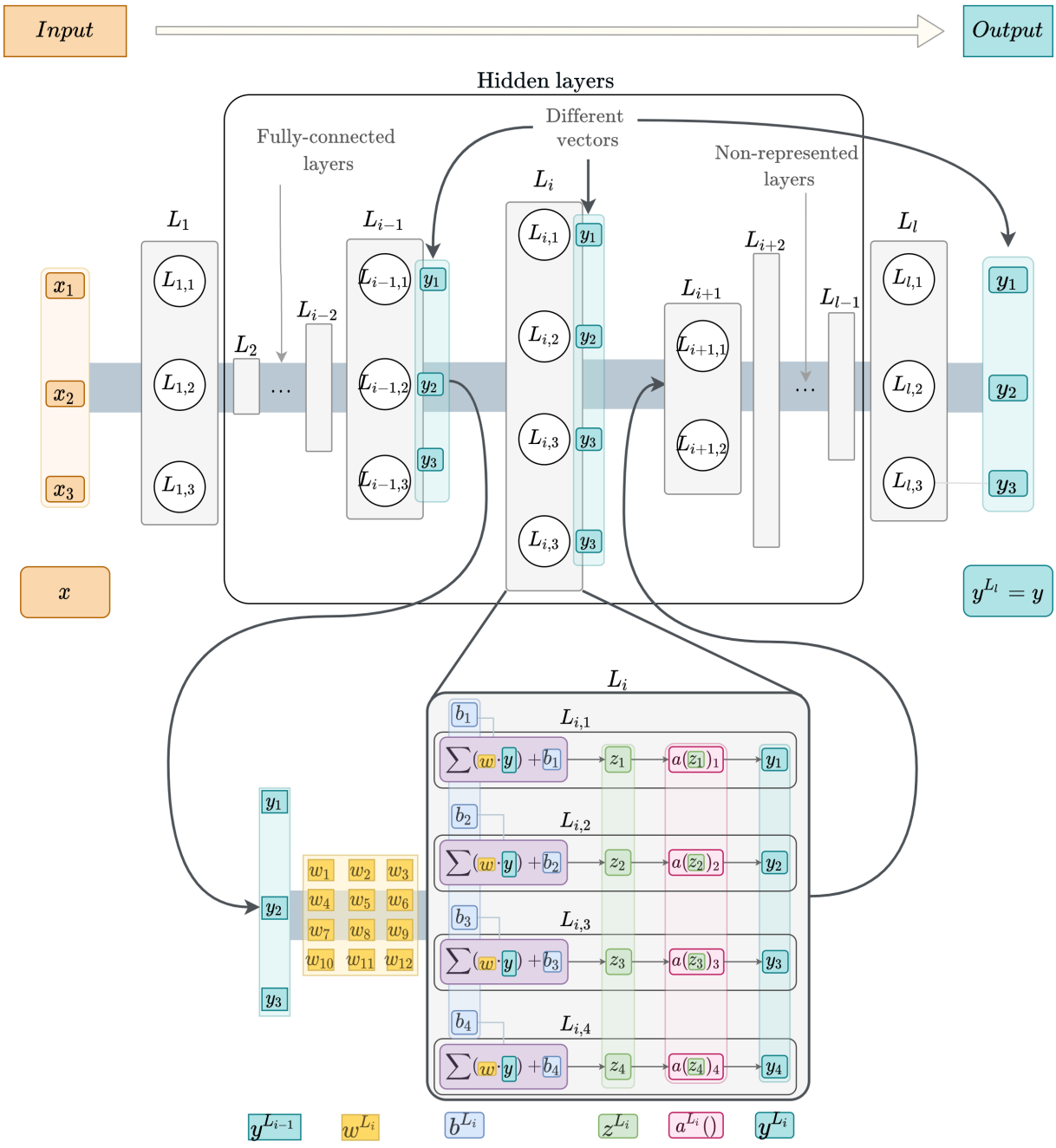


Figure 11: Forward pass in NN.

The network represented needs three input values that return a single value, so the

model will be trained so that, given a vector of three elements, the model will calculate three other elements. The value returned by the model will be equal to the value y, calculated in the last layer:

$$\begin{aligned} y = a^{L_l}(z_l) &= w_l \cdot a^{L_{l-1}}(z_{l-1}) + b_l \\ \text{where } L_l &= \text{Last layer} \\ w_{L_l} &= \text{Weights in last layer} \\ b_l &= \text{Biases in last layer} \\ a^{L_u}(z_u) &= \text{Result of the activation function of the} L_u \text{layer on the vector } z_u \\ a^{L_{l-1}}(z_{l-1}) &= \text{Vector with the values of the penultimate layer output} \end{aligned} \tag{21}$$

The number of z_{l-1} elements will be the same number of neurons as there are in the $L_{_{l-1}}$ layer. This occurs $\forall z_i$. The z_{l-1} vector is calculated in the L_{l-1} layer as follows:

$$z_{L_{l-1}} = w_{L_{l-1}} \cdot a^{L_{l-2}}(z_{L_{l-2}}) + b_{L_{l-1}} \tag{22}$$

This process will be repeated until the z_1 is calculated:

$$\begin{aligned} z_1 &= w_1 \cdot x + b_1 \\ \text{where } x &= \text{Input vector} \end{aligned} \tag{23}$$

To refer to a layer of the network, L_i will be used, where i is the index of the layer ($i = 1$ for the first layer, $i = 2$ for the second layer...). To refer to the last layer, L_l will be used, the penultimate one will be L_{l-1} and so on. A second index can be added to refer to a neuron such that $L_{i,j}$ and like the layer index, if $j = 1$, will refer to the first neuron in the layer, if $j = 2$, will refer to the second neuron in the layer and so on respectively. As an example, the third neuron in the second layer will be referred to as $L_{2,3}$.

The vector w and the value b of $L_{2,3}$ will be referenced such that $w^{L_{2,3}}$ and $b^{L_{2,3}}$ respectively. If we want to refer to a specific value of vector w we will use a third index such as $w_k^{L_{i,j}}$. For example, it will refer to the first weight of the third neuron in the second layer.

Knowing this new notation, the equations will be simplified from this point on. Therefore, the parameters of a neuron will come in a single vector by adding the value b to the already existing vector w. All the vectors will be grouped forming a matrix W, where each row will be the vector associated with a neuron. That is, the parameters of a layer L_i

will be given as a matrix which will be referred to as W.

$$W^{L_i} = \begin{pmatrix} b^{L_{i,1}} & w_1^{L_{i,1}} & w_2^{L_{i,1}} & \cdots & w_n^{L_{i,1}} \\ b^{L_{i,2}} & w_1^{L_{i,2}} & w_2^{L_{i,2}} & \cdots & w_n^{L_{i,2}} \\ b^{L_{i,3}} & w_1^{L_{i,3}} & w_2^{L_{i,3}} & \cdots & w_n^{L_{i,3}} \\ \vdots & \vdots & \vdots & & \vdots \\ b^{L_{i,m}} & w_1^{L_{i,m}} & w_2^{L_{i,m}} & \cdots & w_n^{L_{i,m}} \end{pmatrix} \tag{24}$$

where $i =$ Index of the layer in the network

$n =$ Number of neurons inL_{i-1} or input vector length if $i = 1$

$m =$ Number of neurons in the L_i layer

As an example, the W^{L-1} matrix of a neural network with 4 neurons in the second last layer and 3 neurons in the third last layer is:

$$W^{L_{l-1}} = W^{L_2} = \begin{pmatrix} b_1 & w_1 & w_2 & w_3 \\ b_2 & w_4 & w_5 & w_6 \\ b_3 & w_7 & w_8 & w_9 \\ b_4 & w_{10} & w_{11} & w_{12} \end{pmatrix} \tag{25}$$

Using the example in figure 4, where specific values are used for w and b, the matrices of each layer with the parameters would be such that:

$$W^{L_1} = \begin{pmatrix} 0 & 1 & 1 \\ -1 & 1 & 1 \end{pmatrix} \tag{26}$$

$$W^{L_2} = \begin{pmatrix} -2 & 3 & -2 \end{pmatrix} \tag{27}$$

With this new notation in a single matrix, the parameters that need to be calculated in the regression are collected. In fact, this matrix is what the model will have to "learn" in order to get better results. This will be discussed in more detail in section 2.5.1.

In equation 8 the value was calculated with the function $a()$ of the neuron. From this point, an activation function will receive a vector with the z values of each neuron and therefore this function $a()$ will also return a vector of the same size. In the first layer of a network, this vector will be calculated in such a way that:

$$y_1 = a^{L_1}(z) = a(W^{L_1}x) \tag{28}$$

But by defining the W-matrix in this way, the $W^{L_1}x$ product is impossible to calculate. This can be demonstrated using the example in the equation 24. The dimension of W^{L_1} is (3×6). However, the x-vector with which it is going to be multiplied has a dimension (1×5). To solve this problem, a 1 will be added to vector x, a vector known as x', and the

transposition of this vector will be performed. In this way it will be possible to multiply the matrix W with dimension 3×6 with a vector of dimension 6×1.

$$\begin{aligned} a_1^L(z) &= a(W^{L_2}x') \\ \text{where } x' &= [[1] \oplus x]^T = [1, x_1, x_2, ...x_n]^T \\ n &= \text{Vector size } x \\ A \oplus B &: \text{Concatenation of vectors } A \text{ and } B \end{aligned} \tag{29}$$

The rest of the layers do not have this problem. The vector obtained in the previous layer already fulfils the necessary conditions to be able to produce a matrix by a vector. The vector $a^{L_2}(z)$ will use the same equation as equation 28, but the variable x will be the vector y calculated in the previous layer:

$$a^{L_2}(z) = a(W^{L_2} \cdot a(W^{L_1}x')) \tag{30}$$

This process will be carried out until the last layer of the network is reached. More information can be found in section 2.4.5.

Also updating equations 21, 22 and 23:

$$y = a^{L_3}(z_3) = W^{L_3} \cdot a^{L_2}(z_2) \tag{31}$$

$$z_2 = W^{L_2} \cdot a^{L_1}(z_1) \tag{32}$$

$$z_1 = W^{L_1} \cdot x' \tag{33}$$

Grouping all the equations into one, the value of y in a neural network of three layers is given by:

$$y = a(W^{L_3} \cdot (a(W^{L_2} \cdot (a(W^{L_1} \cdot x')^{L_1}))^{L_2}))^{L_3} \tag{34}$$

This equation is the one that solves the model in order to obtain the y-value and is what is known as the forward-pass algorithm.

2.4.5 Forward-pass algorithm

The forward-pass algorithm is the basic algorithm with which a neural network predicts y. As seen in equation 34, the algorithm is ultimately a combination of matrix products and activation functions. A general equation can be defined as follows:

$$y = a(W^{L_l} \cdot (a(W^{L_{l-1}} \cdot ... \cdot (a(W^{L_1} \cdot x')^{L_1})) \; ... \; ^{L_{l-1}}))^{L_l} \tag{35}$$

This equation is computing the forward-pass algorithm, which is summarised in the following steps:

1. There will be a vector x:

(a) If it is the first layer of the model, x is the die to the model as an input argument (vector x) but transposed.

(b) IOC x will be calculated by the previous layer (vector y).

2. x' is computed by entering a 1 at the beginning of the vector.

$$x' = [[1] \oplus x]^T \tag{36}$$

3. y is computed:

$$y = a(Wx') \tag{37}$$

4. If it is the last layer, the model prediction will be y, IOC go back to step 1.b.

The same algorithm in pseudocode:
Result: Vector computation y
i = 0;
n = Number of network layers;
x = Input vector;
y = Output vector;
while $i \neq n$ **do**
  if *i == 0* **then**
    $y' = x^T$;
  else
    $y' = y$;
  end
  $x' = [[1] \oplus y']$;
  $z = W^{L_i} x'$;
  $y = a^{L_i}(z)$;
  $i++$;
end

Algorithm 1: *forward-pass* algorithm

With this equation it is easy to see that the use of one activation function per layer is necessary. As explained in section 2.4.3, without the use of the activation functions, a set of layers can be compacted into a single layer. The mathematical demonstration is shown below, the forward-pass algorithm without activation functions:

$$\begin{gathered} y = W^{L_l} \cdot W^{L_{l-1}} \cdot ... \cdot W^1 \cdot x' = (\prod_{l;i--=1}^{i=1} W^{L_i}) x' = W' x' \\ \text{where } W' = (\prod_{l;i--=1}^{i=1} W^{L_i}) \end{gathered} \tag{38}$$

This network collapses to a single layer whose parameters are given by the matrix of W'. Using the example in figure 11, if it were decided not to use activation functions, the

dimension of the matrix W' would be $(3, 1)$ which coincides with the size of the input vector(x) and the size of the output vector(y).

$$\begin{aligned} dim(W^1) &= (3, 6) \\ dim(W^2) &= (6, 5) \\ dim(W^3) &= (5, 1) \end{aligned} \tag{39}$$

$$\begin{aligned} dim(W^1 \cdot W^2) &= (3, 5) \\ dim(W') = dim(W^1 \cdot W^2 \cdot W^3) &= (3, 1) \end{aligned}$$

Reminding the following rule for the matrix product for a matrix product $A \times B = C$:

$$\begin{aligned} dim(A) &= (m, n) \\ dim(B) &= (n, k) \\ dim(C) &= (m, k) \end{aligned} \tag{40}$$

- The dimension of matrix C is given by the number of rows of A (m), and the number of columns of B (n).
- The number of columns of A (m) must coincide with the number of columns of B (n), a condition that is met since we are working with fully-connected neuronal networks, all the neurons in one layer are connected to the neurons in the next layer.

Returning to equation 38 and performing the same calculation as in equation 40 on the desired network, the size of y can be obtained and will be calculated by the model. The number of rows will indicate the number of data that have been predicted and the number of columns is the number of labels for each data.

In the network shown in figure 11, it has five variables as input. As an example, this network could be used to model the following problem: Predicting the cost of a house in a given city from a set of housing data in that same city. We want to predict the price value from the following variables: number of rooms, type of house, percentage of crime, geographical coordinates and square metres. This can be represented as a table where each row is a house.

	type	coords	lc[1]	m2[2]	r[3]	price
1	Den	$(40.35717, -3.81053)$	5	41	2	65.378
2	Flat	$(40.53125, -3.72054)$	2	49	3	79.519
3	Duplex	$(40.45990, -3.60053)$	5	59	3	82.578
4	Flat	$(40.43437, -3.74870)$	1	84	3	99.701
5	Flat	$(40.43646, -3.84071)$	3	100	5	130.978
6	Chalet	$(40.41922, -3.57909)$	1	126	6	160.890
7	Flat	$(40.47372, -3.58115)$	4	94	4	215.000
8	Penthouse	$(40.52186, -3.57051)$	4	142	6	285.398
9	Chalet	$(40.44979, -3.68003)$	3	185	7	340.589
10	Duplex	$(40.49347, -3.67969)$	2	209	8	426.000

[1] lc: Level of crime in the neighbourhood.
[2] m2: square meters.
[3] r: Number of rooms.

Table 2: Example of a dataset with dwellings.

2.5 Artificial neural networks. Training.

Neural Network are algorithms used in the field of AI and try to solve problems in which one works with a large amount of data trying to find patterns in them. Furthermore, it is one of the few alternatives to other algorithms that are capable of treating large volumes of data.

The beginnings of neural networks date back to the 1950s, when McCulloch and WalterPitts [28] worked on a mathematical model that resembled the behaviour of a neuron that they knew. The scientist Frank Rosenblatt, inspired by this work, developed what are known as networks of perceptions. This was the first approach to what is now known as neural networks [29].

2.5.1 Training of a network

As a summary of seen so far, four different elements are necessary to create a network:

- Input values: Information with which a data or a data set is to be predicted.
- Structure of the network: This is information that must be known a priori before creating the model. It has different properties:
  - Number of layers: The higher the number of layers, the longer it will take the model to compute the y output vector because the more calculations it will have to perform. The number of layers together with the number of neurons per layer are important parameters, since a very low number in the model and this will not have a good accuracy. On the contrary a very high number can produce what is known as overfitting (see section 2.5.7).

  - Number of neurons in each layer: It should be noted that the number of neurons in the last layer will be the size of the y vector, i.e. the number of labels that the model will predict.
  - Activation function for each layer.
  - Network architecture

- The matrices W: These are matrices that collect the information associated with each layer on the weights w and b of each neuron in the network.

W matrices are matrices with randomly created values. The network training process will try to optimise these matrices so that the resulting y vector is as accurate as possible. To be able to train a model it is necessary to have previously a dataset with a set of vectors x and their real value. The amount of data we provide to the model to learn is related to the accuracy of the model. With the example described in the table 2, we could use the columns: text type, text coordinates, text *lc*, text *m2* and text *r* to predict the value of the price.

Basically, the network initialises the W-matrix randomly. The model given a vector x makes all the calculations for each neuron and returns a vector y. y is what the model has predicted. This process is known as forward-pass.

For the network to learn, it first needs to know if it has made a mistake and the magnitude of the error. If the magnitude of this error is very large, the W values should be adjusted to minimise the error. On the other hand, if the error is small, the W values will not be adjusted much because making an adjustment in W can cause a slight improvement in that prediction, but it can disregard other predictions that the model has made previously and were also quite accurate. The elegance of this algorithm is that it finds a balance between adjusting values so that the network learns and at the same time not misadjusting too much so that the network decompensates in its global learning. It is the application of the imitation of human learning, an error with a great impact will have to modify future behaviours, an error without impact can be ignored or incorporate changes proportionate to the impact.

The error is quantified with a function called cost function or loss function explained in section 2.5.2. From the value of the error, an algorithm will be used to try to calculate the responsibility of each neuron in this error and thus be able to adjust the weights and biases associated with this neuron. This algorithm is called backpropagation and is explained in section 2.5.4.

It can be thought that this process can be repeated infinitely until a perfect model is obtained, but as seen in section 2.5.7 this can cause the network to memorise the dataset that is used for training and not have the capacity to generalise. Therefore, it is not only a matter of executing the algorithm, but also of performing certain optimisations and considering several concepts such as: the selection of the activation function, design of the

input and output vector, metrics to be used, among others.

As an analogy to understand the backpropragation algorithm, the hierarchy of a company can be used. This company, after a disastrous quarter, produces a summary of the results. These results are equivalent to the error that the model produces. The CEO (the last layer of the network) will try to be accountable to the managers. These managers will in turn deal with other managers with less responsibility and these in turn will deal with managers below them and so on until they reach the last level in the company (spreading the error to the most basic layers). Subsequently, the company will produce a report studying the responsibility of each person in the result of the quarter (which in the backpropagation algorithm is known as the gradient vector). This report will reach the human resources department, which will try to modify the behaviour of each worker in the company according to their responsibility in the error.

In the following diagram you can visualise the process that takes place to train a neural network:

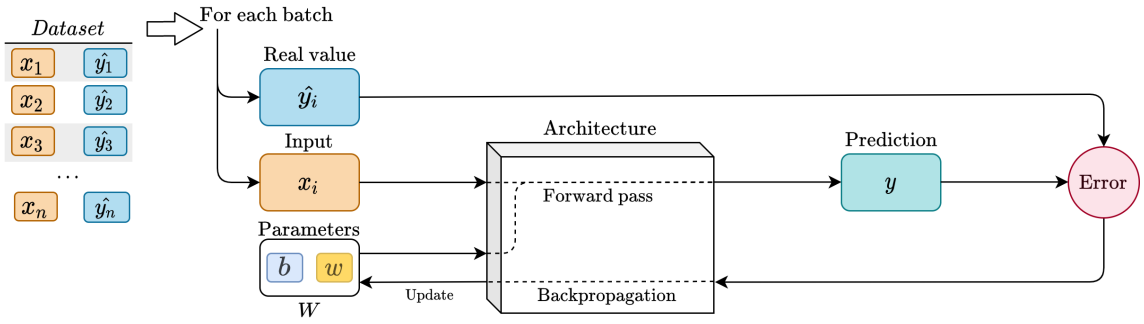


Figure 12: Training process of a neural network

Training a network is an iterative process. In each iteration, or also known as epoch, a set of data will be taken from the training dataset at random. Each of these datasets taken for each epoch is called a batch and the size of the batch is usually a parameter set by the user.

The size of a batch is usually 32, 64 or 128 values. In each epoch, the model will only work with these data. It will predict a value for each of the inputs and together with the real value, the cost function, the backpropagation algorithm and the gradient decrease, the W matrices will be adjusted iteratively.

2.5.2 Cost function

For the model to learn, it is first necessary to be able to identify errors in the process. The cost function is a function that will calculate the error that is occurring in a model. This function will have two parameters: The expected value and the value calculated by the model. The difference between both values is what is known as error or loss, so this

function is also known as loss function or objective function.

The simplest error is the error given by the difference between the expected value and the actual value:

$$\begin{aligned} c_i &= Real_{value} - Expected_{valuue} = \hat{y}_i - y_i, \\ \text{where } c_i &= \text{The value of the sample loss} \\ i &= \text{i}^{\text{th}} \text{ sample of the dataset} \\ \hat{y}_i &= \text{Result of the model} \\ y_i &= \text{Real value} \end{aligned} \tag{41}$$

Using the example seen before in the regression of figure 6 the error for the data i seen graphically would be as follows:

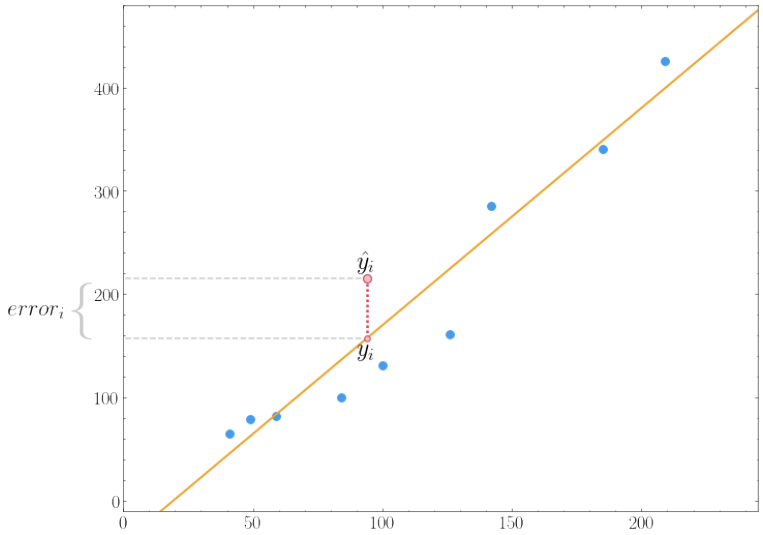


Figure 13: Regression error for data i

In a network you need to know the error of the layer and not the error of each neuron. To do this some function will be applied that receives a vector as input (the errors of each neuron in a layer) and a single output value (layer error). Some of the most commonly used functions are listed below [30]:

- Mean Absolute Error (MAE) [31] : This is the simplest regression error metric to understand. The absolute error of each data is taken, so that negative and positive errors are not cancelled out. Then the arithmetic mean is calculated. In fact, the MAE describes the typical magnitude of the residuals. The equation is as follows:

$$\text{MAE} = c_i(y_1, \hat{y_1}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y_i}| \tag{42}$$

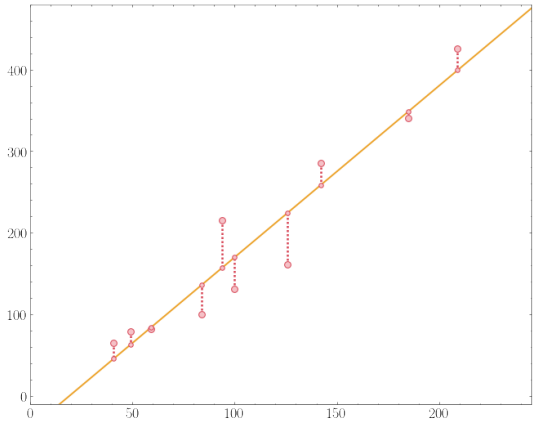


Figure 14: Visualisation of the MAE. The error is the arithmetic mean of all errors.

- Mean Squared Error (MSE) [31]: This equation calculates the average of all squared errors. By squaring it is possible to penalise with greater intensity those points that are further away from the estimation of the linear regression and with less intensity those that are closer. This equation is widely used when the task to be solved is of a regression type. The equation is as follows:

$$\begin{aligned} \text{MSE} = c_i(y_1, \hat{y_1}) &= \frac{1}{n}\sum_{i=1}^{n}(\sqrt{(\hat{y_i} - y_i)^2}^2) \\ &= \frac{1}{n}\sum_{i=1}^{n}(\hat{y_i} - y_i)^2 \end{aligned} \tag{43}$$

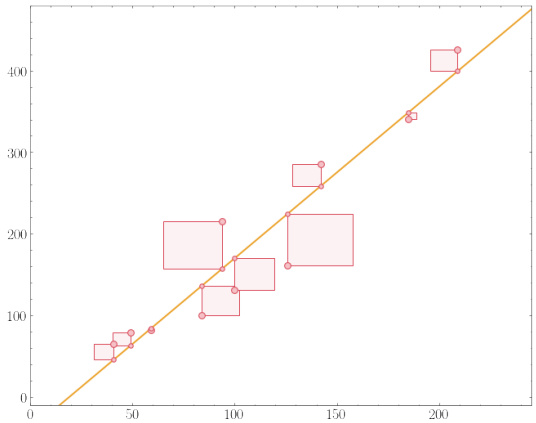


Figure 15: Visualisation of the MSE. The error is the average of the areas of the squares.

- Root Mean Squared Error (RMSE) [31] : This equation is the square root of MSE. If compared at the value level, they are interchangeable although they use different

scales. The equation is as follows:

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2} \tag{44}$$

- Mean Squared Logarithmic Error (MSLE) [31] : This equation is similar to MSE. It is often used when it is known in advance that the results are normally distributed and large errors are not intended to be significantly more penalised than small ones. The equation is as follows:

$$\begin{aligned} \text{MSLE} = c_i(y_1, \hat{y_1}) &= \frac{1}{n}\sum_{i=1}^{n}(\log y_i + 1 - \log \hat{y}_i + 1) \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(\log\left(\frac{y_i + 1}{\hat{y}_i + 1}\right)\right)^2 \end{aligned} \tag{45}$$

- Huber loss [32]: It is already known that MSE is better for learning outliers in the data set, on the other hand, MAE is good for ignoring outliers. But in some cases, data that appear to be outliers do not bother and should not be given high priority. The loss of Huber is a combination of MSE and MAE. δ will be used to define a bias for using MSE or MAE. The equation is as follows:

$$\begin{aligned} \text{Huber\_loss} = c_i(y_1, \hat{y_1}) &= \begin{cases} \text{MSE} & \text{if } |\hat{y} - y| \leq \delta, \\ \text{MAE} & \text{e.o.c.} \end{cases} \\ &= \begin{cases} \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 & \text{if } |\hat{y} - y| \leq \delta, \\ \delta\left(\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| - \frac{\delta}{n}\right) & \text{e.o.c.} \end{cases} \end{aligned} \tag{46}$$

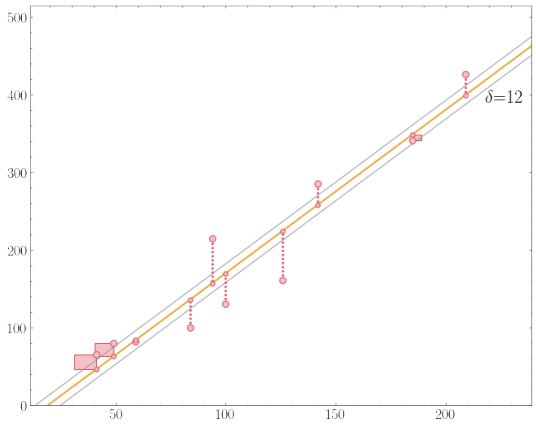


Figure 16: Visualisation of Huber loss MSE is calculated if $|\hat{y} - y| \leq \delta$, in other case is calculated MSE.

- Mean Absolute Percentage Error (MAPE) [31] : This is the equivalent percentage of MAE. The equation is equal to the MAE, but with adjustments to convert the values into percentages. They allow to see the distance between the model results and the real result showing the data in an easier way to interpret for human beings. The equation is as follows:

$$\text{MAPE} = c_i(y_1, \hat{y_1}) = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y} \right| \tag{47}$$

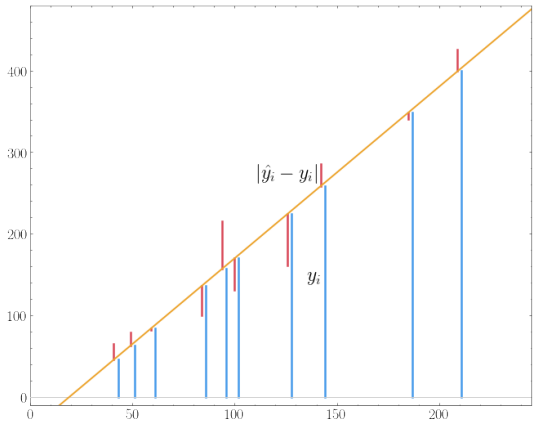


Figure 17: MAPE visualisation. The error is the mean of the error proportions to with respect to the value y.

- Mean Percentage Error (MPE) [31] : This is exactly what MAPE is, but without the absolute value. Since positive and negative errors are cancelled out, no statement can be made about the overall performance of the model's predictions. However, if there are more negative or positive errors, this bias will be shown in the MPE. Unlike MAPE and MAE, the MPE is useful because it allows one to see whether the model systematically underestimates (more negative errors) or overestimates (positive errors). The equation is as follows:

$$\text{MPE} = c_i(y_1, \hat{y_1}) = \frac{100\%}{n} \sum_{i=1}^{n} \frac{y_i - \hat{y}_i}{y} \tag{48}$$

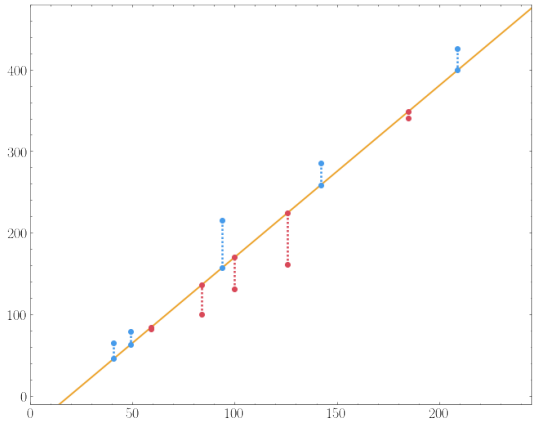


Figure 18: Visualisation of the MPE. It shows the number of errors that are positive and the number that are negative.

- Loss of cross entropy: Used explicitly to compare a probability of "fundamental truth" (y or "targets") and some predicted distribution ($\hat{y}$ or "predictions"). It aims to calculate the average of the probabilities that a value belongs to one class or another, so it is very useful for classification problems. It is widely used when the softmax activation function is used, since it also works with probabilities.

$$c_i(y_1, \hat{y_1}) = -\sum_i y_{i,j} log(\hat{y_{i,j}}) \\ \text{where } j = \text{"True" probability index} \tag{49}$$

  The "true" probability is a vector with all values at 0 except one that has the value equal to 1. This type of vector is known as an one-hot vector, where a value is hot if it is equal to 1 or cold if it is equal to 0. When the results of the model are compared with a one-hot vector using the cross entropy, the values equal to 0 are not used, and the log loss of the target probability is multiplied by 1, making the calculation of cross entropy relatively simple. This is also a special case of the calculation of cross entropy, called category cross entropy.

  An example of a one-hot vector would be the following: $[0, 1, 0, 0]$ where for example the following classes would be represented: Dog, cat, horse and elephant. The 1 represents that the data given to the model represents a cat. These types of vectors are widely used in classification tasks and in Natural Language Processing (NLP).

  There is a subtype called binary cross entropy loss. This type of error can be calculated when trying to classify only with two types: 0 or 1. It is usually used as if it were a boolean value in a programming language. A `true` if it is type A or a `false` if it is not type A. For example: cat or no cat or indoor or outdoor. The

mathematical equation is as follows:

$$\begin{aligned} c_{i,j}(y_1, \hat{y_1}) &= (y_{i,j})(-log(\hat{y}_{i,j})) + (1 - y_{i,j})(-log(1 - \hat{y}_{i,j})) \\ &= -y_{i,j} \cdot log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot log(1 - \hat{y}_{i,j}) \end{aligned} \tag{50}$$

2.5.3 Minimising error

Once one of the cost functions has been chosen, the aim is to minimise the error and thus improve the model:

$$\begin{aligned} W^* &= \underset{W}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} c(a(x^{(i)}; W), y^{(i)}) \\ &= \underset{W}{\operatorname{argmin}}\, c_i(W) \\ \text{where } x^{(i)} &= \text{Input vector for data } i \\ y^{(i)} &= \text{Actual vector for data } i \\ W &= \text{Matrices with the weights and biases of each} \end{aligned} \tag{51}$$

To calculate the minimum of a function, it is necessary to calculate the derivative of that function and equal it to 0. Minimising the cost function, that is, deriving the cost function, also minimises the error. To do this there are different ways of minimising the error:

- Ordinary Least Square (OLS).

  This was the algorithm used in the perceptron networks to calculate the W matrix. Starting from the definition of any cost function that allows the quantification of the model error, an attempt will be made to minimise this value. One of the most popular equations is the one known as the Ordinary Least Square. This function is based on the MSE derivative:

$$\begin{aligned} L_i &= (\hat{y_i} - y_i)^2 \\ &= (y - x \cdot W)' \cdot (y - x \cdot W) \\ &= y'y - W'x'y - y'xW + W'x'xW \end{aligned} \tag{52}$$

  Deriving and clearing:

$$\begin{aligned} L_i'() &= -2x'y + 2x'xW \\ W^* &= (x'x)^{-1}x'y \end{aligned} \tag{53}$$

  This equation allows the calculation of the optimal W-matrix possible only by using the model input and output values as arguments. Although this training algorithm has several limitations:

- It cannot be extended to more complex networks.
- The calculation of the inverse matrix is very costly in terms of computation.
- The minimum square error has been used, one of the simplest, since it is a function with a convex shape and therefore an easy derivative to calculate, but there are other cost functions that do not allow minimising the error with this technique.

These limitations, demonstrated mathematically in the book "*Perceptron*" [33] by Minsky and Papert (1969), led to a sudden cut in funding for artificial intelligence projects and more specifically those related to neural network systems over a period of more than 15 years known as the winter of artificial intelligence.

- Descending the gradient.

  This algorithm is not a formula like Ordinary Least Square, but an iterative method that gradually minimizes error. Somehow it can be assimilated to how we human beings learn: Not with a single formula, but through experience reducing our errors over time.

  The calculation of the gradient is a process that can be represented as follows:

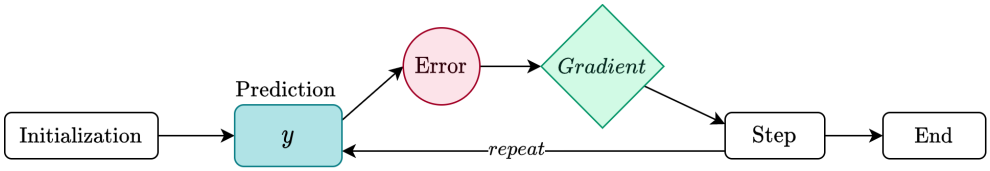


Figure 19: PIterative process to minimize error.

The OLS method minimizes the error by equalizing the derivative to 0 in order to find the minimum MSE. With this, the local and global minima can be calculated, but it is also possible to calculate local maxima, turning points or chair points, causing a large and rather inefficient system of equations to be solved [33].

One way of understanding this method is as follows: A person is in a mountainous terrain and the objective of this person is to reach the lowest point. To do this, he or she will analyse the terrain where he or she is located and evaluate the slope and move towards where the slope descends most intensely. He or she will descend a number of steps and repeat the process: analyse the slope and descend. This will be repeated until it gets as far down as possible and there is no way to go any further down. This is the logic of the descending gradient algorithm.

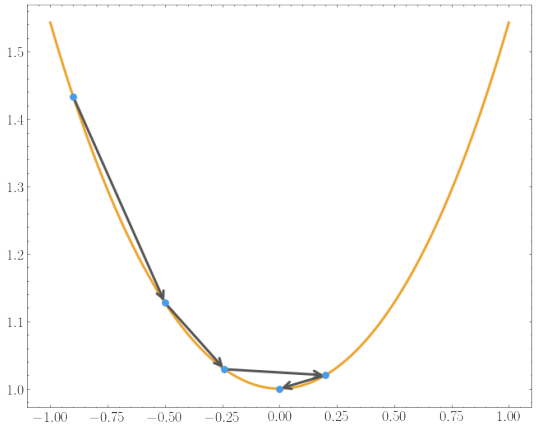


Figure 20: Graphical example of the iteration of the descent of the gradient.

The terrain in this metaphor would be the function of cost. The person is the value calculated by the cost function, which we want to minimise and therefore we evaluate the slope or gradient only at that point and in that way we do not depend on the derivative of the cost function but on the partial derivative with respect to each of the input values in the neuron. The number of steps that the person will descend is a value known as the Learning Rate and is one more parameter of the neuronal network.

In figure 22, there are two axes, representing a neuron that has only one input argument. The ordinate axis represents the error of the neuron. A two-dimensional space has been represented, but any dimension can be used since the number of inputs of the neuron is not limited. Using a three-dimensional space, the cost function would be an irregular surface. In fact, the vector ∇f will always have at least two values, a weight w and the bias b.

This paper does not explain how to calculate the derivative of a function or partial derivative (σ). That said, mathematically, the process of this algorithm is as follows:

1. 1. The w and bias b weights of the neuron want to be adjusted randomly:

$$N(0, \sigma_2) \tag{54}$$

2. A loop is made until convergence::
   (a) Partial derivatives are calculated for each of the parameters. A partial derivative measure how much impact a single input variable has on the output of the function and is calculated in the same way as a derivative, the only thing that the derivative repeats for each input variable. Each of these values will indicate what is the slope on the axis of that parameter

related to a single input value.

$$\begin{aligned} \nabla f &= \nabla f_b \oplus \nabla f_w \\ \nabla f_b &= \left(\frac{\partial c}{\partial b}\right) \\ \nabla f_w &= \left(\frac{\partial c}{\partial w_1}, \quad \frac{\partial c}{\partial w_2}, \quad \cdots, \quad \frac{\partial c}{\partial w_n}\right) \end{aligned} \tag{55}$$

Together all the directions, that is, all the partial derivatives form a vector that indicates the direction in which the slope is rising, this vector is also known as a gradient(∇f) and it will have the same number of elements as the input vector and each value will contain the solution to the partial derivative with respect to each of the input values.

The objective of this function is to minimize and not maximize, so the gradient will be denied indicating the direction in which the slope is descending.

$$\nabla f = -\nabla f_b \oplus \nabla f_w \tag{56}$$

(b) The weights are updated with the new values of the negative gradient multiplied by the learning rate. The learning rate is simply a value that determines how large the step will be in each iteration which will be discussed in more detail in section 2.5.5:

$$\begin{aligned} b^* &= b - \eta * \nabla f_b \\ w^* &= w - \eta * \nabla f_w \end{aligned} \tag{57}$$

As with the feed-forward algorithm, the descent of the gradient will work in layers, so this last step should be represented as follows::

$$W^* = W - \eta * \nabla f \tag{58}$$

This equation represents that the weights of all the neurons in a layer will be updated.

This method will be applied to all the neurons in the network. Given an error, the descent of the gradient will obtain a vector containing the derivative of the parameters with respect to the cost. That is to say, the gradient will be a vector with different values, the larger this value is, the more correction must be applied to the corresponding parameter. Seen graphically it can be seen in the following way:

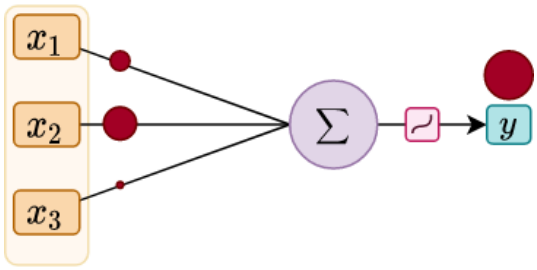


Figure 21: Representation of the descent of the gradient in a neuron The red is the representation of the error.

The partial derivatives that must be solved to obtain the gradient vector ∇f, indicates how the cost varies when the parameters w and b are changed. An image of the different parts of the partial derivatives is shown below:

¿How the cost varies when the parameters w and b are changed?

$$\frac{\partial c_i}{\partial w} \qquad \frac{\partial c_i}{\partial b}$$

Figure 22: Partial derivatives used in the descent of the gradient.

2.5.4 Backpropagation

In 1986, Rumelhart and other researchers published "Learning representation by back-propagating errors" [34], a work that brought back the popularity of neural networks. The paper, based on other works based on automatic differentiation, showed experimentally how using a new learning algorithm could make a neural network self-adjust its parameters in order to learn an internal representation of the information it was processing, an algorithm known as backpropragation.

This algorithm is adaptable to any model and with it, ended what is historically known as the AI Winter by obtaining new funding and new projects in the field of Deep Learning (DL).

The prediction of a model is obtained using an algorithm called forward-pass as explained in section 2.4.5. This value should be adjusted as closely as possible to the real value. To do this, the W matrices must be adjusted in an iterative method in a process known as training (see section 2.5.1). In each iteration of the training, the W matrices will be adjusted, first using the cost function, which will show the magnitude of the error (see section 2.5.2), then this error will be propagated backwards in the model to know the responsibility of each neuron using an algorithm known as backpropagation and finally,

for each neuron the gradient vector ∇f will be calculated, which represents how the neuron has affected the final result and with this its different weights w and bias b will be adjusted (see section 2.5.3).

The backpropagation algorithm can be represented graphically:

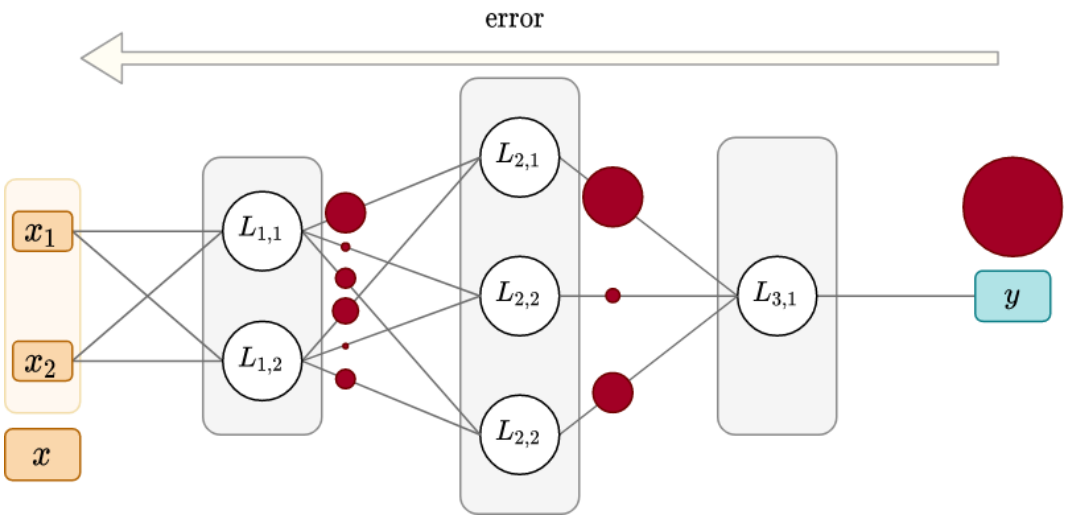


Figure 23: Representation of the backpropagation algorithm.

In this network, the error of each neuron is represented by red circles. The error calculated by the cost function is the largest error which indeed is the error of the output of the model. From that point, it will be backward propagated and computed which part of the error belongs to each neuron up to the first layer. For example, in the L_2 layer, the main neuron that has mismatched parameters is $L_{2,1}$. Likewise, this error is mainly due to the $L_{1,1}$ neuron. Therefore, it is logical to think that we should adjust neurons such as $L_{2,1}$ or $L_{1,1}$, however, leave neurons such as $L_{1,2}$ or $L_{2,2}$ as they are.

The simplest neural network that can be constructed with a single hidden neuron and an output neuron is defined as follows:

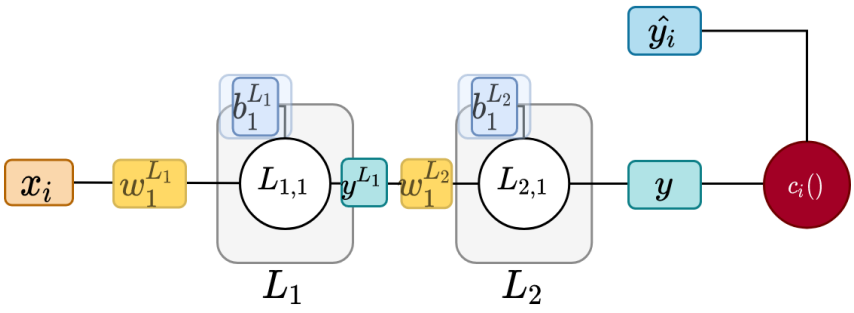


Figure 24: Basic neural network.

To perform the backpropagation algorithm, first the loss must be computed with the c_i function. Then, the backpropagation algorithm will "distribute" the loss of the model to the previous layers (always starting with the last layer). This computation expresses how important the W matrices are by indicating how much impact they have on the model. For example, if you change any value of W_2 in any way, you will be able to know how it affects the outcome of the model's prediction. Mathematically:

$$\begin{aligned} \frac{\partial c}{\partial w^{L_2}} &= \frac{\partial c(a(z^{L_2}))}{\partial w^{L_2}} \\ \frac{\partial c}{\partial b^{L_2}} &= \frac{\partial c(a(z^{L_2}))}{\partial b^{L_2}} \end{aligned} \tag{59}$$

The numerator is a composition of functions and is basically the calculation of the forward-pass algorithm (see section 2.4.5). To calculate the partial derivative of a composition of functions, it is necessary to use a calculation tool known as the chain rule.

An example to understand the chain rule would be the following. The Sun is known to be 100 times larger than the Earth. The Earth is 4 times larger than the Moon. Knowing these two relationships, we can tell how much the Sun is bigger than the Moon. To do this we simply multiply $100 \times 4 = 400$. The Sun is therefore 400 times bigger than the Moon. Defining this problem with equations:

$$\begin{aligned} \frac{\partial T_{\text{Sun}}}{\partial T_{\text{Earth}}} &= 100 \qquad \frac{\partial T_{\text{Earth}}}{\partial T_{\text{Moon}}} = 4 \\ \frac{\partial T_{\text{Sun}}}{\partial T_{\text{Moon}}} &= \frac{\partial T_{\text{Sun}}}{\partial T_{\text{Earth}}} \cdot \frac{\partial T_{\text{Earth}}}{\partial T_{\text{Moon}}} = 400 \end{aligned}$$

It is called the chain rule because it is chained in relation to the common data, in this case T_{Earth} is the common part that on one side is in the denominator and on the other in the numerator. Therefore, to derive the composition of functions seen in equation 68, we simply have to multiply each of the intermediate derivatives. Recalling the equations of the forward-pass algorithm and the composition of functions to calculate the model error for the network shown in figure 28:

$$z^{L_2} = W^{L_2} \cdot y^{L_1} + b^{L_2} \qquad c(a^{L_2}(z^{L_2})) \tag{60}$$

It is calculated how the error is propagated from the L_2 layer to the L_1 layer:

$$\begin{aligned} \frac{\partial c}{\partial w^{L_2}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial w^{L_2}} \\ \frac{\partial c}{\partial b^{L_2}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial b^{L_2}} \end{aligned} \tag{61}$$

These derivatives are easy to calculate and the explanation of each derivative would be as follows:

- First derivative. Derived from the activation function with respect to cost:

$$\frac{\partial c}{\partial a^{L_2}} \tag{62}$$

How does the cost of the network (it is the last layer) vary when the output of the network activation function is varied? In other words, the derivative of the cost function is calculated with respect to the output of the neural network. It is therefore necessary to know the derivative of the cost function used.

- Second derivative. Derived from activation with respect to z:

$$\frac{\partial a^{L_2}}{\partial z^{L_2}} \tag{63}$$

It tries to reflect how the output of the neuron varies when the weighted sum of the neuron is varied. The derivative to be used is the derivative of the activation function. Derived from the sum of the weighted values of the weights and biases.

- Third derivative: Derived from z with respect to the parameters:

$$\frac{\partial z^{L_2}}{\partial w^{L_2}} \qquad\qquad \frac{\partial z^{L_2}}{\partial b^{L_2}}$$

It indicates how the weighted sum z varies with respect to a variation in the parameters. Parameter b is an independent value so its derivative is a constant equal to 1. On the other hand, the derivative $\frac{\partial z^{L_2}}{\partial w^{L_2}}$ depends on the previous layer.

$$\frac{\partial z^{L_2}}{\partial w^{L_2}} = a^{L_1} \qquad\qquad \frac{\partial z^{L_2}}{\partial b^{L_2}} = 1$$

Based on this definition, the first and second derivatives can be combined into one:

$$\frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} = \frac{\partial c}{\partial z^{L_2}} \tag{64}$$

This derivative indicates the degree to which the error is modified when there is a change in the error when there is a small change in the sum of the neurons in the layer. If the value is large it means that a small change in any of the neurons in the layers in the z value will be reflected in the result of the model. On the contrary, if the value is small, a large change in any of the neurons in the z value of will not affect the final result. In summary, this derivative shows how the layer (or some neuron in particular since it is a vector) affects the final result and therefore the error of the network and in that way the algorithm of the descent of the gradient will modify the parameters to be able to obtain the best possible result. This derivative is also known as the layer error and is represented by δ:

$$\frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} = \frac{\partial c}{\partial z^{L_2}} = \delta^{L_2} \tag{65}$$

With this new definition, equation 61 can be rewritten as follows:

$$\begin{aligned}\frac{\partial c}{\partial w^{L_2}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial w^{L_2}} \\ &= \delta^{L_2} \cdot \frac{\partial z^{L_2}}{\partial w^{L_2}} = \delta^{L_2} \cdot a^{L_1}\end{aligned} \tag{66}$$

$$\begin{aligned}\frac{\partial c}{\partial b^{L_2}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial b^{L_2}} \\ &= \delta^{L_2} \cdot \frac{\partial z^{L_2}}{\partial b^{L_2}} = \delta^{L_2} \cdot 1\end{aligned} \tag{67}$$

But the model does not only depend on the second layer and W_2, it also depends on the first layer and W_1. This is where the beauty of the backpropagation algorithm comes in. The error can be backpropagated in a similar way using the same reasoning. First the composition of functions is shown:

$$\begin{aligned}\frac{\partial c}{\partial w^{L_1}} &= \frac{\partial c(a(z^{L_2}(a(z^{L_1}))))}{\partial w^{L_2}} \\ \frac{\partial c}{\partial b^{L_1}} &= \frac{\partial c(a(z^{L_2}(a(z^{L_1}))))}{\partial b^{L_2}}\end{aligned} \tag{68}$$

Using the chain rule it is divided into different parts:

$$\begin{aligned}\frac{\partial c}{\partial w^{L_1}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial a^{L_1}} \cdot \frac{\partial a^{L_1}}{\partial z^{L_1}} \cdot \frac{\partial z^{L_1}}{\partial x} \\ \frac{\partial c}{\partial b^{L_1}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial a^{L_1}} \cdot \frac{\partial a^{L_1}}{\partial z^{L_1}} \cdot \frac{\partial z^{L_1}}{\partial 1}\end{aligned} \tag{69}$$

In fact, of the 6 derivatives shown, only one would need to be calculated. The derivatives $\frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}}$ have already been calculated and are equal to δ^{L_2}. As for the derivative $\frac{\partial z^{L_1}}{\partial x}$ If you want to use the first layer, simply get the result of the previous layer or use x if it is the first layer as it is in this case. The derivative $\frac{\partial z^{L_1}}{\partial 1}$ is a constant value so nothing happens. Finally, the derivative $\frac{\partial a^{L_1}}{\partial z^{L_1}}$ is simply the derivative of the activation function of that layer. This is why in section 2.4.3 it was explained that it is better to use a single activation function for the whole layer and thus facilitate the calculations.

The only derivative that remains to be explained is $\frac{\partial z^{L_2}}{\partial a^{L_1}}$ the one that expresses how the weighted sum of a layer varies when the input vector of that layer is varied. The calculation of this derivative is simply the matrix of parameters W^{L_1} that connects both layers. Basically, this derivative moves the error from layer L_2 to layer L_1 distributing the error according to the weightings of the connections. As it was done with the previous layer, with this layer it is also possible to define the imputed error:

$$\frac{\partial c}{\partial a^{L_1}} = \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial a^{L_1}} \cdot \frac{\partial a^{L_1}}{\partial z^{L_1}} = \delta^{L_1} \tag{70}$$

Therefore, equation 69 would look like this:

$$\begin{aligned}\frac{\partial c}{\partial w^{L_1}} &= \delta^{L_1} \cdot \frac{\partial z^{L_1}}{\partial x} \\ \frac{\partial c}{\partial b^{L_1}} &= \delta^{L_1} \frac{\partial z^{L_1}}{\partial 1}\end{aligned} \tag{71}$$

In summary, it has been possible to explain how to propagate the error to any layer of the network and calculate the derivative with respect to the parameters with the following equations:

$$\begin{aligned}\frac{\partial c}{\partial w^{L_1}} &= \delta^{L_1} \cdot \frac{\partial z^{L_1}}{\partial x} \\ \frac{\partial c}{\partial b^{L_1}} &= \delta^{L_1} \\ \frac{\partial c}{\partial w^{L_2}} &= \delta^{L_2} \cdot \frac{\partial z^{L_2}}{\partial a^{L_1}} \\ \frac{\partial c}{\partial b^{L_2}} &= \delta^{L_2}\end{aligned} \tag{72}$$

In view of the backpropagation algorithm for the network defined in figure 28, the algorithm is divided into different steps:

1. Initialise index to know in the layer to be computed

$$i = l; l > 0 \tag{73}$$

2. Calculation of the imputed error of the layer:

   (a) If $i = l$:

$$\delta^{L_i} = \frac{\partial c}{\partial a^{L_i}} \cdot \frac{\partial a^{L_i}}{\partial z^{L_i}} \tag{74}$$

   (b) IOC:

$$\delta^{L_i} = \delta^{L_{i+1}} \cdot \frac{z^{L_{i+1}}}{\partial a^{L_i}} \cdot \frac{\partial a^{L_i}}{\partial z^{L_i}} \tag{75}$$

3. 3. Calculation of derivatives on parameters b and W:

   (a) Calculation on b:

$$\frac{\partial c}{\partial b^{L_i}} = \delta^{L_i} \tag{76}$$

   (b) Calculation on w:

      i. If $i = 1$:

$$\frac{\partial c}{\partial w^{L_i}} = \delta^{L_i} \cdot x \tag{77}$$

ii. e.o.c:

$$\frac{\partial c}{\partial w^{L_i}} = \delta^{L_i} \cdot a^{L_{i-1}} \tag{78}$$

4. Update the index:

$$i = i - 1 \tag{79}$$

5. If $i > 0$ go back to step 2

With this, it has been possible to obtain the partial derivatives of each layer of the network so that it will be possible to calculate the new W matrices using the method of the descent of the gradient explained in section 2.5.3.

2.5.5 Learning rate

Selecting an appropriate learning rate for the model is a fundamental step in designing a neural network and a minimum modification of this value can have a great impact on the final model.

If a very small learning rate is selected, it means that it does not trust the result of the gradient and therefore in each iteration the change that the W matrix will suffer will be small and the algorithm will be able to get stuck in some of the local minima because the change between the old W and the new W is not being so drastic as it should not end in these local minima. On the contrary, if a very large learning rate is selected, the algorithm will be completely exceeded and will diverge.

Therefore, the value for the learning rate should be neither too small so that the algorithm does not get stuck nor too large so that the model can converge. One way to choose a good learning rate is to test several values and study which value works best. This algorithm is known as Stochastic gradient descent (SGD) [35], but another option is to use a optimiser.

2.5.6 Optimisers

Optimizers are algorithms used to calculate a learning rate dynamically. It is not a static value but varies according to the state of the training. As you run the training, the learning rate may increase, or it may decrease.

If we use the metaphor of the person on a mountain, the number of steps that person will descend will be conditioned by different criteria. Some of these criteria may be how far or how fast the person has descended recently.

In other words, the learning rate used will be adapted according to different criteria. One of these criteria is how fast the learning is going with respect to the loss of our model among others.

Depending on the optimizer chosen, they will use one or other criteria to modify this value. The most used optimises are: Adam[36], Adadelta[37], Adagra[38] o RMSProp[38].

2.5.7 Overfitting

Knowing the basic functioning of a neural network using the forward-pass and backpropagation algorithm, it can be thought that if in each iteration of the model, the network trains and learns new patterns, the conclusion can be drawn that by carrying out infinite training, a perfect model can be obtained whose error has been minimised as much as possible and therefore the computed results are the expected ones. But this is not the case, having a wrong configuration of the network or training the network longer than necessary causes what is known as overfitting.

Overfitting is a well-known problem when training neural networks. It is usually caused by using a training that has been performed for a long time, causing the model to fit perfectly to the training data and somehow "memorize" the data and therefore, not know how to extrapolate and adapt to other cases. In other words, the W-matrix of all the layers is over-adjusted and thus, the model adapts perfectly to the input data producing that the model is not able to estimate a correct value when using an input vector that it had not seen before.

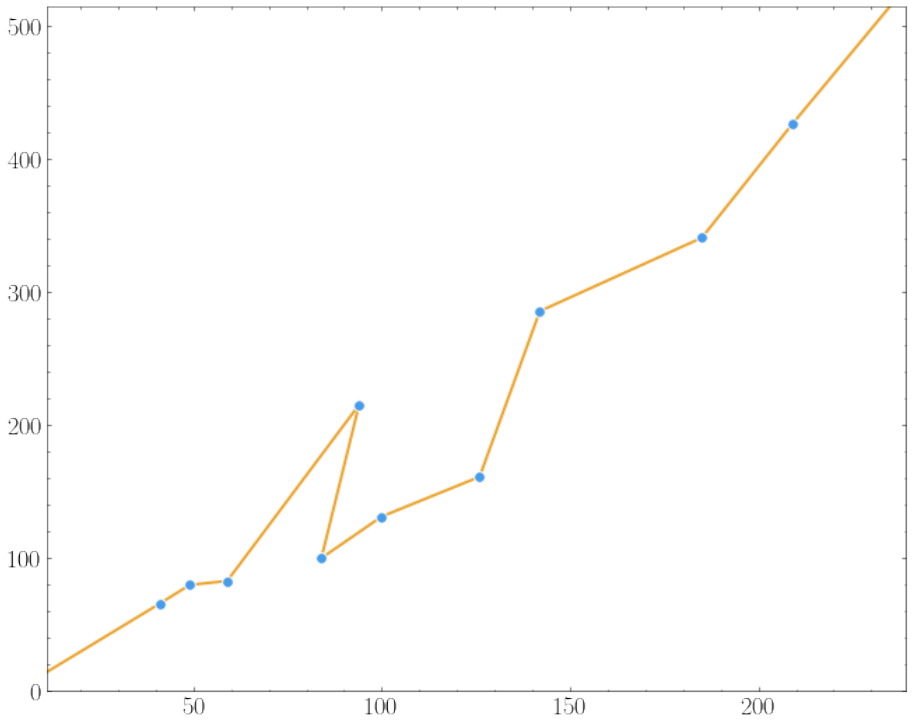


Figure 25: Example of overfitting in a linear regression.

One of the fastest ways to know if a model is overfitting is by using a second dataset in

addition to the training one. Normally, before creating a neural network, a pre-processing must be done. The original dataset is usually divided into three different datasets: training, validation and test. Obviously, the training dataset is used to train and adjust the parameters of the W matrices in an iterative way. On the other hand, there are the validation and test datasets that are used to evaluate different metrics. If the problem to be solved with the network is of regression type, some kind of error explained in section 2.5.2 is usually used as a metric, such as: MAE, MSE or Huber error. On the other hand, if the problem to be solved is a classification problem, precision is usually used as a metric.

If we see that the accuracy of the test data no longer improves, then we should stop training. Of course, strictly speaking, this is not necessarily a sign of overfitting. It could be that the accuracy of the test data and the training data stop improving at the same time. Even so, adopting this strategy will avoid overfitting.

At the end of each epoch, a subset of data from the training dataset and another subset from the validation dataset will be used. With these, the value of the selected metric will be calculated and two metrics will be obtained. These metrics can be compared between them. Two metrics that are similar indicate that the model is capable of extrapolating to other cases that it has not used for training. On the contrary, if the value associated to the training dataset is much better than the value associated to the validation dataset this may mean that the model may be suffering from overfitting. A training carried out without overfitting can be visualised in figure 26.

At the beginning of the training, as the network has not been trained and the parameters initialised randomly, the model will have quite bad metrics, i.e. if any error is being used, this value will be very high. On the other hand, if you are measuring the accuracy of the model, this value will be far from 100%. These metrics will be equally bad for both datasets. As epochs occur during the training, the model will not be able to improve anymore and will start to memorise the data with which it is being trained from the training dataset causing the metric associated with the training to be almost perfect and the validation metric to get progressively worse. An example of an overfitting training can be seen as follows:

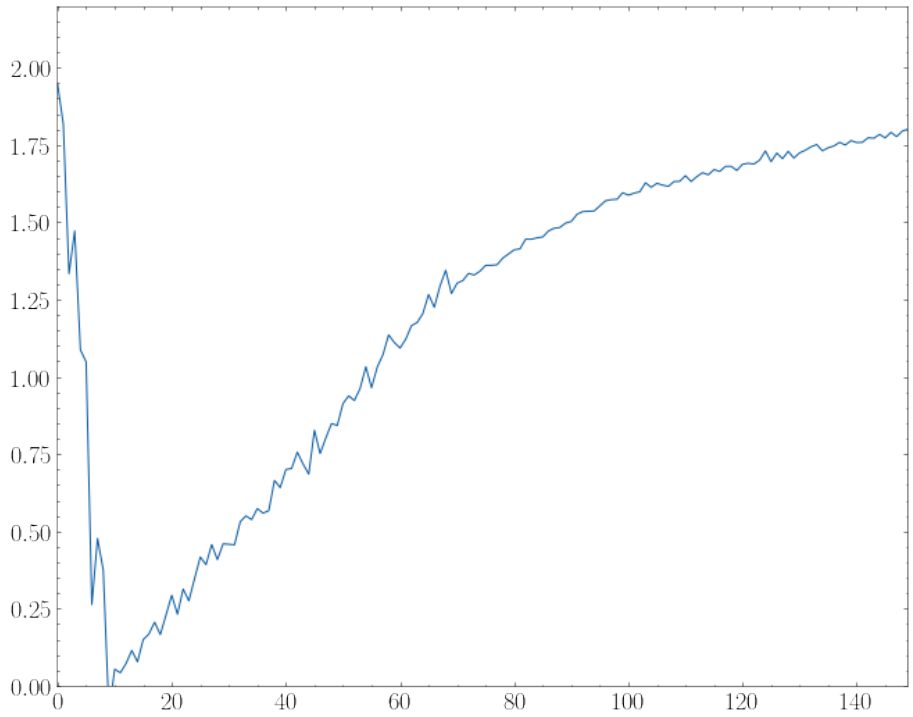


Figure 26: Example of overfitting in a linear regression using an error as a metric.

At the beginning of the training both metrics stop, until they reach approximately epoch 10, at which point the validation metric can be seen to gradually worsen. It is at this point that you can see that the network has an overfitting problem. Depending on the metric used, the overfitting can appear before or after, but it is a more complex issue that will not be explained in this paper.

There are still several techniques that will be explained below to avoid overfitting:

- Regularization L1 and L2: These are two methods that calculate a value known as a penalty that is added to the error and thus can penalise parameters with large values. If a neuron has large values it may be a sign that the neuron is trying to memorise. In fact, it is considered bad practice to let a small set of neurons in the network bear the greatest responsibility, i.e., their associated parameters are very large.

  On the one hand, you have L1 which is the sum of all the absolute values of w and b. It is a linear penalty since the associated function is directly proportional to the parameters. The penalty L2 is the sum of all parameters w and b squared. It is a non-linear function and penalises with greater intensity the large values, unlike L1 that penalises much more in proportion to the small values because it is linear. This causes the model to become invariant to small input values and variant only to large values. Therefore, L1 is rarely used unless it is in combination with L2. Both functions are dependent on a value λ, with this value you can dictate how much impact L1 and L2 will have on the final error. Mathematically:

$$L_{1w} = \lambda \sum_m |w_m| \qquad L_{2w} = \lambda \sum_m w_m^2 \tag{80}$$

$$L_{1b} = \lambda \sum_n |b_n| \qquad L_{2b} = \lambda \sum_n b_n^2 \tag{81}$$

The final error is given by the simple sum of the error calculated by the cost function and L1 with L2:

$$c_T = c + L_{1w} + L_{1b} + L_{2w} + L_{2b} \tag{82}$$

As regularization is used for the calculation of c, it is necessary to know its derivative in order to use back-propagation. The cost derivative is as follows:

$$\frac{\partial c_T}{\partial w^{L_i}} = c' + L'_{1w} + L'_{1b} + L'_{2w} + L'_{2b} \tag{83}$$

The L1 and L2 derivatives are as follows:

$$L'_{1w} = \lambda \begin{cases} 1, & \text{si } w_m > 1 \\ -1, & \text{si } w_m < 1 \end{cases} \qquad L'_{2w} = 2\lambda w_m \tag{84}$$

$$L'_{1b} = \lambda \begin{cases} 1, & \text{si } b_n > 1 \\ -1, & \text{si } b_n < 1 \end{cases} \qquad L'_{2b} = 2\lambda b_n \tag{85}$$

- Dropout : Using this technique, in each iteration of the training the network is modified. Let's suppose that you have an input x and the desired value y. Ordinarily, you would train by propagating x forward through the network, and then propagating it backward to determine the contribution to the gradient. With dropout, this process is modified. It starts by randomly (and temporarily) removing a percentage of the hidden neurons in the network, while leaving the incoming and outgoing neurons unchanged. The neurons that have been deleted will be "ghost" neurons for that epoch:

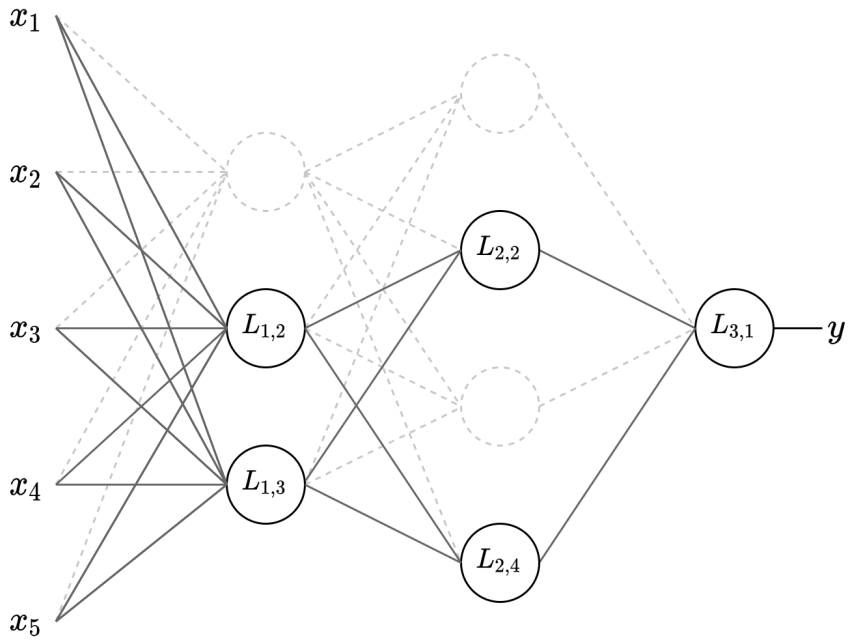


Figure 27: Dropout applied to a network with 50% probability

Once the process of forward propagation and backpropagation has been completed, a new epoch will begin by randomly eliminating a set of neurons. In short, for each iteration, a subset of the neurons will be eliminated according to a given percentage.

By repeating this process throughout the training phase, the network will learn W-matrixes that will have been learned under conditions where a subset of the hidden neurons was removed. When the whole network is actually executed, it will mean that the number of active neurons will be higher. Therefore, it will be compensated by reducing the proportional part with those that were entered. For example, using a percentage equal to 50, the values of W will be half of what the gradient has calculated.

- Selection of an iterative learning rate: This technique is based on the study of various learning rates depending on the epoch of the training. A set of values is usually given in an orderly fashion between a minimum and a maximum. Each value of this interval will be a learning rate for different epochs and it will be possible to study their behaviour and see if the descent of the gradient is capable of converging.

  If it can converge, the results obtained will be those expected, making the cost function progressively smaller. On the contrary, if the learning rate is very low or very high, it will cause that the descent of the gradient cannot converge resulting in the result of the cost function being irregular and generally worsening the model.

2.6 Recurrent Neural Network

2.6.1 Basic notions

Recurrent Neural Network (RNN) is a type of network architecture that is applied to a multitude of problems, especially NLP problems such as voice recognition or text translation, but it is also used for problems in which the data is arranged in time and has a certain relationship, such as in the case of stock market prediction or object detection in videos.

In summary, RNN are networks are able to to work with sequential data models. An example using sequential data might be the following: A picture is shown with a circle drawn and you are asked in which direction the circle is going. Any answer to this question will be random and without any criteria. However, if several photos are shown and it is explained that they are photos taken before the first photo and in these photos, you can see that the circle follows a trajectory, then the answer to the question will be trivial because it is easy to predict where the circle will go.

Figure 28: Graphical example of a single piece of data vs. a sequence.

In this way you can see the importance of having temporary information and how this can be used to solve predictions. Sequential information is present in the world in different forms: audio (bit sequence), text (sequence of characters or words) or values that change over time.

The way RNN works is thanks to a concept known as sequential memory. Sequential memory is the ability to reproduce sequences of words, numbers, letters, symbols... of a human. For that reason, it is easy to say the alphabet and yet more difficult to say the alphabet backwards.

Using some kind of loop within the network, this sequential memory can be emulated so that a network architecture can use previous information to process the calculation of a new value. This information that is passed using the loop is known as the hidden state (h) and contains information from the previous input values.

When reference is made to an RNN, we are talking about a network that at least one of the layers is recurrent and therefore has some kind of loop. That is, there can be an RNN with only one recurrent layer or an RNN with two recurrent layers or any kind of combination with at least one recurrent layer. Several examples are shown below:

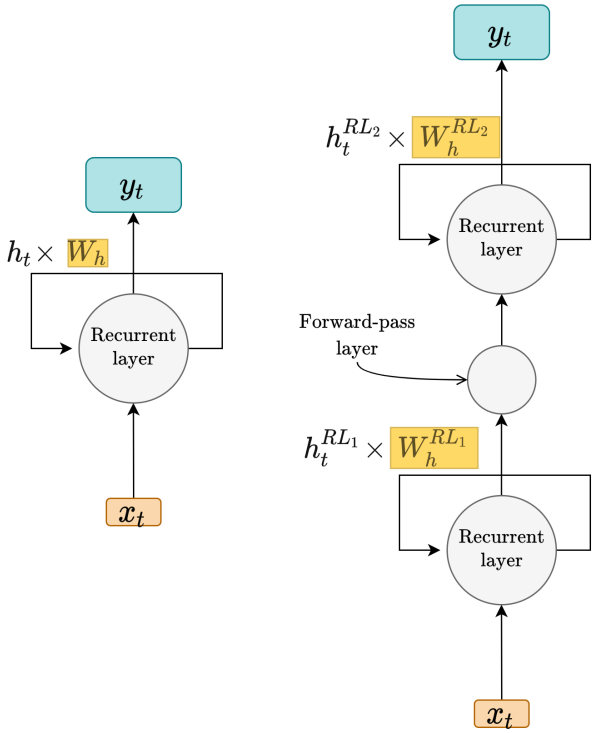


Figure 29: Several examples of RNN

From now on, instead of representing the loop with a reflexive connection, it will be done using the horizontal axis for each of the iterations and in such a way that a linked network chain remains. This is the most common way to see the RNN represented.

In the previous diagram, a new matrix has been introduced: W_h. This matrix is the proper weight matrix for vector h. Furthermore, as there can be several recurrent layers, to differentiate W_h and h the index RL_i will be used, where i is the number of the recurrent layer.

2.6.2 Types

There are three different types of RNN:

- Many to One: From a sequence generate a vector. For example, rate from one to ten opinions on products written by customers.
- Many to many: From sequence to generate a sequence. For example, translate text from one language to another.
- One to many: From a single vector a sequence is generated. For example, a network specialized in describing the elements of an image.

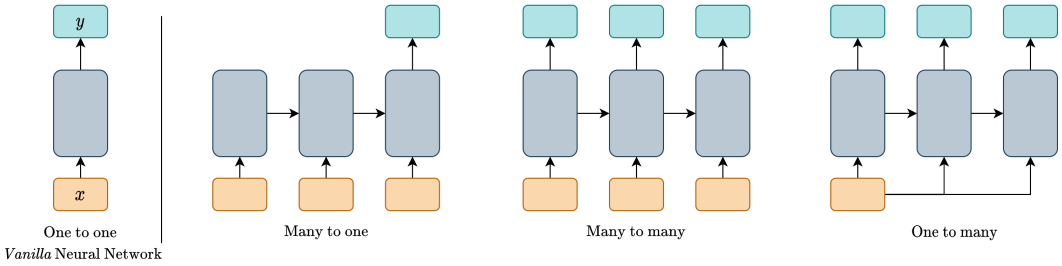


Figure 30: Operational representation of RNN types

2.6.3 Forward pass algorithm in RNN

The operation of the forward pass in an RNN is similar to a normal layer. A basic recurrent layer usually has only one layer with the activation function such as $tanh$. The vector h_t is a vector calculated by this activation function but that not only receives as parameter x, but also receives the previous hidden state h_{t-1}. In the same way, the vector h_{t+1} will have information of both h_t and x_{t+1}. As you can see, the hidden state of each layer is passed from layer to layer. Graphically:

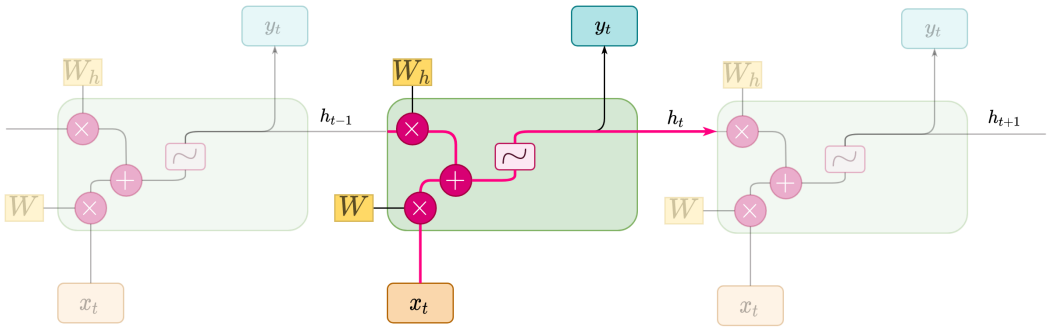


Figure 31: Functioning of an RNN.

For each iteration, the presence of information from previous h is reduced, causing the gradient to fade, a concept that will be explained later. This information flow can be better seen with the following diagram:

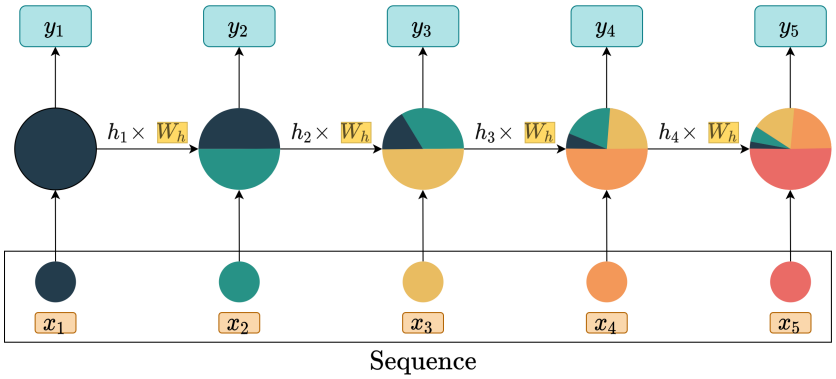


Figure 32: Structure of a recurring layer

The recurrent layers, as with the basic layers of a network, will have an associated matrix of weights (although they do not have biases b). This matrix will be represented with W_h leaving W for basic layer weight matrices. There can be multiple recurrent layers each one with an independent W_h matrix, that is why the RL_i index is used as explained in figure 29.

The calculation of y is similar to the calculation in a normal neural network. Starting from the equation explained in equation 35:

$$y = a(W^{L_l} \cdot (a(W^{L_{l-1}} \cdot ... \cdot (a(W^{L_1} \cdot x')^{L_1})) \; ... \; ^{L_{l-1}}))^{L_l} \tag{86}$$

Simplifying this calculation to a single layer:

$$\begin{aligned} z &= W^{L_l} \cdot y^{L_i} \\ y &= a(z) \end{aligned} \tag{87}$$

If it is a recurrent layer, a new summation has to be added to the operation, but we want it to continue to have the same properties of the activation function, so in reality, what is going to be modified is the calculation of z:

$$\begin{aligned} z &= W^{L_l} \cdot y^{L_i} + W_h \cdot h \\ y &= a(z) \end{aligned} \tag{88}$$

For example, if you have a recurrent network made up of five layers, where the first and third layers are recurrent, the calculation of final y is as follows:

$$y = a(W^{L_5} \cdot a(W^{L_4} \cdot a(W^{L_3} \cdot a(W^{L_2} \cdot a(W^{L_1} \cdot x' + W_h^{RL_1} \cdot h_1))^{L_2} + W_h^{RL_2} \cdot h_2)^{L_3})^{L_4})^{L_5} \tag{89}$$

The calculation of z will depend on whether the recurring layer is the first one:

$$z^{L_1} = W^{L_1} \cdot x' + W_h \cdot h \tag{90}$$

Otherwise, if the recurring layer is not the first one:

$$z^{L_i} = W^{L_i} \cdot y^{L_{i-1}} + W_h \cdot h \tag{91}$$

2.6.4 Backpropagation algorithm and descent of the gradient in RNN

The calculation of the error in this type of architectures is the same as in the forward-pass networks, simply applying any cost function using as two arguments: the predicted value and the real value as explained in the section 2.5.2. All calculated errors will be added up and saved in C. This error will be used to calculate the gradient of all the neurons. To do this it will be necessary to back-propagate the error. This error will be back-propagated in the same way as explained in the section 2.5.4 for the non-recurrent layers.

For layers that apply some kind of loop, a new derivative must be added. To retropropagate the error, it is necessary to divide this error by proportional parts for each neuron. If, when adding the loop, a new connection is being added to itself, the error will have to be divided into two parts: one part for the previous layers and another for that same layer. Below you can see a diagram representing how the error is backpropagated. In summary, the error is back-propagated both in the network and in time, which is why this variant is called back-propagation through time.

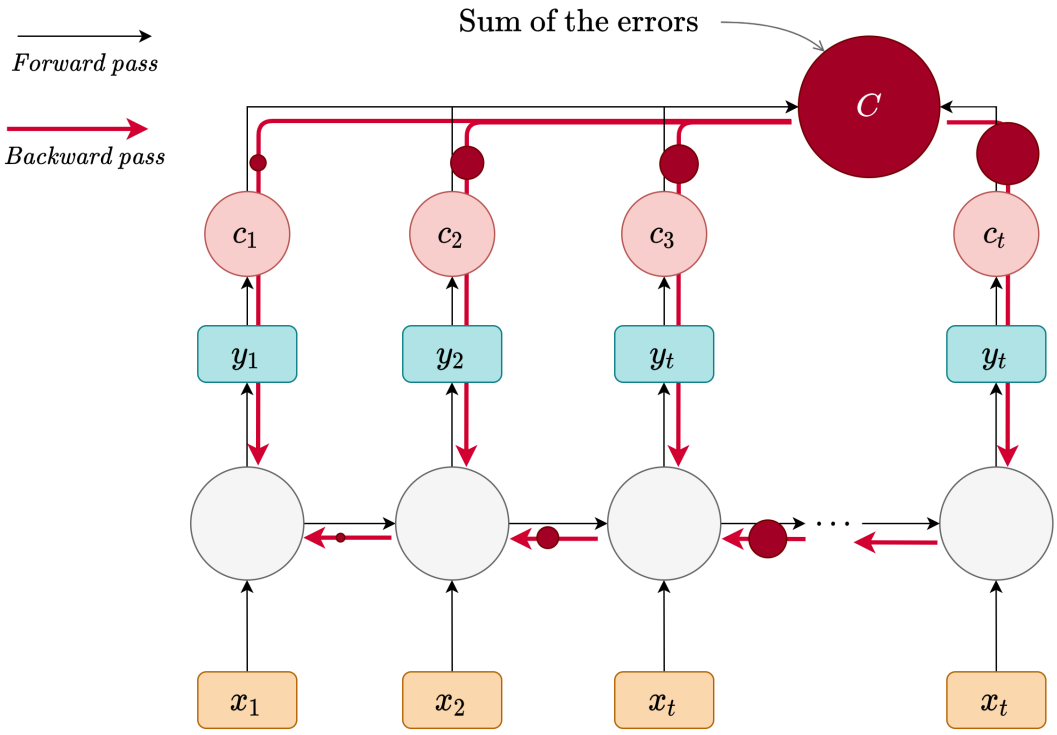


Figure 33: Backpropagation over time in RNN

As can be seen, the error calculated C from all the c_i, flow backwards in time. For the first part of the backpropagation calculation the same equations 66 and 67 can be used. In these equations the cost is calculated with respect to w and with respect to b. The cost must be C and not c_i. For the recurrent layers, a new derivative must be calculated: $\frac{\partial c}{\partial W_H}$.

Applying the chain-rule as explained in section 2.5.4.

$$\frac{\partial C}{\partial W_h} = \frac{\partial C}{\partial z^{L_i}} \cdot \frac{\partial z^{L_i}}{\partial W_h} \tag{92}$$

And as in the gradient calculation made for w and for b in equation 55, the gradient of W_H must be calculated in the same way:

$$\nabla f_{w_h} = \left(\frac{\partial C}{\partial w_h^{RL_1}}, \quad \frac{\partial C}{\partial w_h^{RL_2}}, \quad \cdots, \quad \frac{\partial C}{\partial w_h^{RL_n}} \right) \tag{93}$$

A detail to be emphasised is that w_h is being used and not W_H, since the latter refers to the weight matrix of the recurrent layer. The gradient must be calculated independently for each neuron, as is the case with w and W. This concept is explained in more detail in section 2.4.4. Once all ∇f_h have been calculated for each neuron, the weight matrix w_H can be updated:

$$H^* = H - \eta * \nabla f_{w_h} \tag{94}$$

It is important to first perform the backpropagation calculation and then perform the gradient calculation.

2.6.5 Problems of descending gradient in recurrent "vanilla" layers

As can be seen in figure 32, for each time step, the W_H matrix has to be multiplied by H. The updating of the recurrent layer is given by a non-linear activation function. This causes the calculation of the gradient, which is the derivative of the cost with respect to the parameters that form the network, including all the initial states. It requires many repeated multiplications of these weights and also the repeated use of the derivatives of the activation function. And this can be problematic:

- Explosive gradient: It occurs when many of the values involved in the process of repetitive multiplication of matrices are greater than 1 causing that for each iteration, the gradient is larger and larger and therefore the updating of the parameters: w^*, b^* and w_H^* tends to infinity. There is a solution to this problem and it is to apply what is known as gradient trimming that basically normalises the values so that they are not so big.

- Fading of the gradient: It occurs when many of the values involved in the process of repetitive multiplication of matrices are small causing that for each iteration, the gradient becomes smaller and smaller and therefore the update of the parameters w^*, b^* y w_H^* tend to 0. This is one of the main problems when training recurrent networks.

An example of this problem can be seen by selecting a number from 0 to 1 and iteratively multiplying itself. You can see that this value for each iteration becomes smaller and smaller and eventually, it will fade away. This causes the error to be much more difficult to propagate further into the past and so a network would be programmed to learn the near past. This does not mean that it is a problem, there may be times when this type of architecture is needed. Graphically, this fading of information can be seen in figure 33, where for each iteration you have the current x information and some information from the previous iterations.

This error can be avoided if:

- By choosing ReLU as the activation function: This helps to ensure that the derivative value of $a()$ does not become small but only when $x > 0$.
- Initialising the parameters in a non-random method: For example initialising the matrix with the identity matrix.
- Designing the network architecture in an optimal way: Using a more complex layer such as LSTM, GRU... that is more effective in tracking long term memory by controlling what information passes and which is used to update its internal state. Specifically, it is the concept of a door.

2.6.6 LSTM layers

The Long-Short Term Memory (LSTM) are the most used recurrent layers for its capacity to maintain long term dependencies and are widely used in deep learning communities.

The key behind LSTM are the gates that allow the neuron to actively add or remove information. It consists of several activation functions σ and a $tanh$. The trigger function σ results in the output value being a value between 0 and 1. Intuitively one can think of these functions as aiming to capture and retain information between 0, or nothing, and 1, everything.

Below is a diagram of how an LSTM gate works.

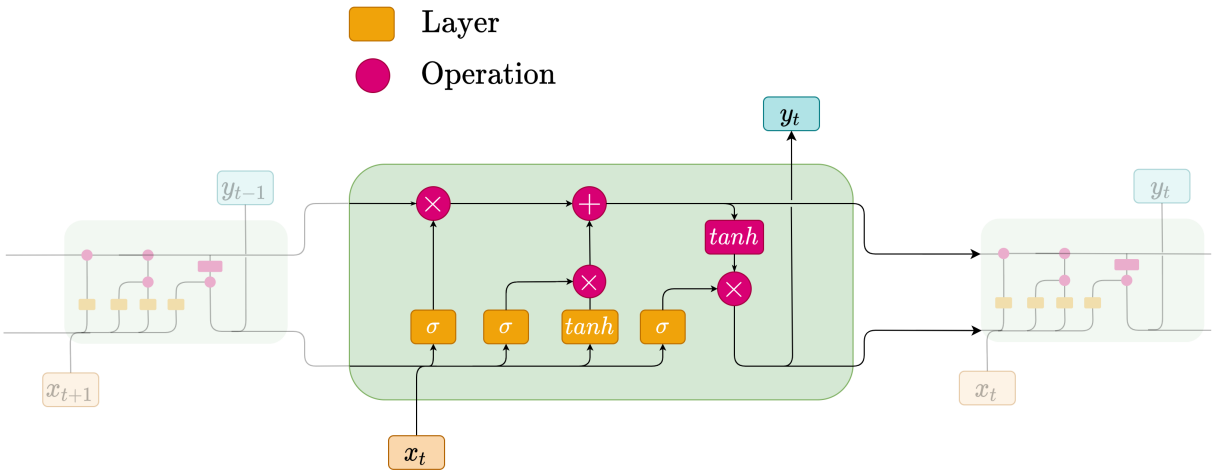


Figure 34: Behaviour of an LSTM

LSTMs are basically formed by four different steps:

1. Forgetting irrelevant information: In the first output of this part is the first function σ together with the operation of the product to scale between the output of σ and the previous state h_{t-1}.

2. Save new information: It consists of the second function σ and the $tanh$ function next to it.

3. Use the two previous steps to calculate a new internal state h_t: It is the part represented by the top line and calculate c_t. This vector is usually considered as the LSTM memory.

4. Compute output value y_t: Formed by the last function σ and by the $tanh$ function. It controls which information is important and saves the information about the current status.

With this gate design, a recursive neuron can be designed to allow the flow of the gradients uninterruptedly and prevent them from fading.

In summary, an LSTM neuron allows both memory and internal state to be separated into two parts allowing the network to learn long-term dependencies in one sequence. It does this by using gates to control the flow of information: forgetting irrelevant information, saving important information, selectively updating the internal state and saving part of the information. All this allowing the flow of the gradients to be unaltered to avoid the fading of the gradient.

2.7 Papers and similar projects

In this section, several projects will be analysed, studying both their implementation and the results obtained. The results of this work have been useful for us to compare our

models and results.

2.7.1 Development of a station-level demand prediction and visualisation tool to support bike-sharing systems' operators

In this work [39], a process for predicting the demand for bicycles and a tool for visualising it is created for the city of Thessaloniki, Greece. Different models are used, including *XGBoost*, *Random Forest* and Neural Network, among others. The data available is as follows:

- Geographical information: Station coordinates, therefore using two attributes: latitude and longitude.
- Temporary information expressed in: Time, day of the week, month, and year. The first three are mapped to a tuple ($sine$ and $cosine$).
- Work calendar information: Using two variables: one to identify whether it is a working day or not and another to identify whether it is a Sunday or not.
- Weather information: Using both current weather information and short-term forecasts. This type of information includes average temperature, wind speed, amount of clouds and precipitation.

For the neural network, a dense forward-pass network with 2 hidden layers of 20 and 8 neurons respectively and a single neuron output layer has been used. For the metrics they have used: MAE, MSE, RMSE and RMSLE. The dataset covers the years 2014 to 2018 and data between 00:00 and 6:00 are discarded, justifying that most stations are not automated and operate only during the rest of the time. The results are as follows:

Métrica	Valor
MAE	0.89
MSE	2.66
RMSE	0.64
RMSLE	0.49

This project is a sample of a good use case for a neural network, using the predictions to display them in a web tool so that users and the company can make decisions. As far as the neural network is concerned, I think it could have been done much better because the network used is very simple and the hyperparameters used are the default ones of sci-kit learn and the results are regular and can be improved.

2.7.2 Predicting station-level hourly demand in a large-scale bike-sharing network: A graph convolutional neural network approach

This study proposes a convolutional [40] neural network model that can learn hidden heterogeneous correlations by pairs between stations to predict the hourly demand at the

station level on a large scale.

Two architectures of the GCNN-DDGF model are explored: The CNN-DDGF model that contains the convolution and forward blocks, and GCNNrec-DDGF also contains an LSTM block to capture the time dependencies in the demand series. In addition, four types of GCNN models are proposed whose adjacency matrices are based on various shared bicycle system data, including the spatial distance matrix (SD), the demand matrix (SD), the average trip duration matrix and the demand correlation matrix. These six types of GCNN models and seven other reference models are built and compared in a *Citi Bike* data set for New York City that includes 272 stations and over 28 million transactions from 2013 to 2016.

The results show that the GCNNrecDDGF has the best performance in terms of the RMSE, the MAE and the coefficient of determination (R2), followed by the GCNNregDDGF. They outperform the other models. It is found to capture some information similar to the details embedded in the SD, DE and DC matrices. More importantly, it also uncovers hidden heterogeneous correlations in pairs between stations that are not revealed by any of those matrices.

Métrica	Valor
MAE	1.44
RMSE	2.46
R^2	0.67

2.7.3 Kaggle competition: Shared bikes demand forecasting

The Kaggle platform is a platform where those interested in Data Science can share blogs, knowledge, and datasets among other resources. In addition, the platform itself or companies that want to hire new personnel who are experts in Data Science frequently hold competitions where any person or team in the world can participate, which is why it is common to see multiple competitions active on Kaggle with different problems related to Big Data and its datasets.

During the summer of 2020, a competition took place [41] that attempted to solve the same problem that has been proposed in this paper: predicting the number of trips that will be made from each of the stations in a city. In this case, the competition used a dataset that included: city (there were multiple cities), time in hours, number of bicycles in an hour, a multitude of variables in relation to meteorology (20 variables) and an associated work schedule.

The metric used to measure the models delivered by the participants was the RMSLE and the best results obtained were the following:

Métrica	RMSLE
Guillermo Alejandro Chacon	0.18721
Maria Garcia	0.18973
Eleanor Manley	0.19837
Milan Goetz	0.20110
Luisa Runge	0.20162
Paula San Roman Bueno	0.20209
Diego Cuartas	0.20498
Valentina Premoli	0.20569

3 DEVELOPMENT

3.1 Problem definition

This work aims to study recurrent neural networks. However, it does not only focus on the theoretical section but also on the practical one. Therefore, a problem has been sought to be solved as an example. The problem which has been attempted to solve is the prediction of the demand of shared bicycles (bikesharing) in the city of Chicago. As a solution is to be carried out using supervised learning, a huge amount of data is needed beforehand. Chicago has been chosen for just that reason, because of the ease of access to public data needed to be able to train on the network without any restrictions or licences. In particular, the dataset offered by the company *Divvy* has been downloaded [42].

In addition, the Chicago City Council offers interactive maps [43] which have been of great help in studying the different properties of the stations, as well as a multitude of data providers, both meteorological [44] and of other types, which can also be added to the dataset and thus improve the precision of the model, but which, due to lack of time and the results obtained, could not be carried out as explained in section 8.

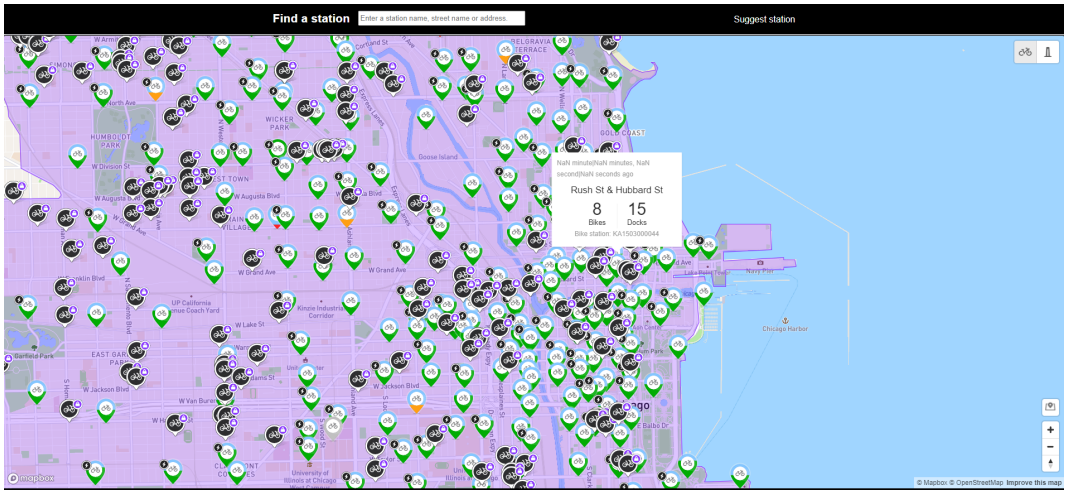


Figure 35: Interactive map of the bike rental station network in Chicago [43]

3.2 Used tools and libraries

The language used for the development of this library has been Python 3.6 [45]. This language is a default option together with R in the Artificial Intelligence field. With Python, it has been used a set of libraries that make this language the favourite option to develop this kind of projects. The libraries used in this project have been:

- Tensorflow [30]: it one of the most important Deep Learning platform nowadays. This Google-based library goes beyond Artificial Intelligence, but its flexibility and large community of developers has positioned it as the leading tool in the Deep Learning sector.

- Keras [46]: Keras is an open source library written in Python, which is mainly based on the work of François Chollet, a Google developer, in the framework of the ONEIROS project. The aim of the library is to accelerate the creation of neural networks: to this end, Keras does not function as a stand-alone framework, but as an intuitive user interface (API) that allows access to and development of various self-learning frameworks. Keras-compatible frameworks include TensorFlow.

- Numpy [47]: Numpy is the premier library for scientific computing in Python. It has many integrated functions of N-dimensional matrix calculation, as well as the Fourier transform, multiple functions of linear algebra and several randomness functions.

- Pandas [48]: It is an open source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

- Matplotlib [49]: It is a library for the generation of graphics from data contained in lists or arrays in the Python programming language and its mathematical extension NumPy. It provides an API, pylab, designed to remind to MATLAB. The graphic theme called SciencePlots [50] has been used.

3.3 Data pre-processing

As mentioned in the previous section, a public dataset offered by the company *Divvy* has been used [42]. This dataset contains the history of all the journeys recorded from 2004 to 2019 in the city of Chicago. This data is offered divided into parts: either by quarters or by months. The final dataset contains $26,328,986$ trips in the entire network with a total of 633 stations. For each trip, there are 12 labels associated with them. A sample of the data collected in the dataset can be seen in Appendix B.

For the development of this project, four different modules have been created. Each of these modules will result in a data structure stored in CSV except for the window generator module which will contain information that will be useful for the next module. The main reason why the first two modules result in the creation of a file is to save unnecessary computation time each time a model is to be trained. As training the models requires a multitude of tests with different configurations, it is a process that would have been repeated continuously and whose result would always have been the same. On the other hand, the window generator, a module that will be explained later, being a module that is instantaneous and that also generates different results depending on the model to be developed, does not save the result in any file.

Figure 36: Project structure divided by modules.

During the following sections each of these modules will be explained in detail including diagrams and code.

3.3.1 Feature selection - Dataset connection and variable filter

The original dataset [42] is divided into years and each year into multiple parts. Therefore, the first task of all has been the union of each of these parts into a single file. The file format offered by Divvy is in a CSV as usual, and therefore the pandas `read_csv` function that converts a CSV to a `DataFrame` has been mainly used. On the other hand, each part had its own nomenclature of columns so it has been necessary to create a single nomenclature to work with a single file and therefore some columns have been renamed. Once the renaming of columns has been carried out, the `DataFrame` will be attached to another one that will contain all the data and which will be used for the following steps.

> "A `DataFrame` is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it as a spreadsheet or a SQL table. It is generally the most commonly used object in the pandas library". [48]

An example of `DataFrame` that has been seen before could be table 2. To be able to initialise a `DataFrame` it can be done from a Python dictionary, where each key is the `string` that identifies the column and each value is a vector with the same amount of values that rows want to have. Another way to initialise a `DataFrame` is by using functions like `read_csv()` , `read_h5()` or `read_pickle()` .

The code used is something like the following:

```
import pandas as pd

# Path to the csv files
csv_filenames = ["trips-2014-Q1", "trips-2014-Q2-Q3",
                 "trips-2014-Q4", "trips-2015-Q1-Q2",
                 # ...
                 "trips-2019-Q3-Q4"]
csv_paths = [f"/path/to/{file}.csv" for file in csv_filenames]

# This will be the final DataFrame
df = pd.DataFrame()

# For every CSV do
for path in csv_paths:

    # CSV to DataFrame
    df_temp = pd.read_csv(path)

    # Parse date as datetime which is always first column
    df_temp[cols[0]] = pd.to_datetime(
        df_temp[cols[0]],
        format='%Y-%m-%d %H:%M:%S',
```

```
        infer_datetime_format=True)

    # Rename the columns depending on their
    # current names with a external function
    df_temp = rename_columns(df_temp)

    # Merge the DataFrames
    df = pd.concat([df, df_temp], join='outer')

# ...
```

Once the `df` variable is obtained, which collects all the trips, a filtering of the columns is necessary. At this point, the `DataFrame` contains 12 values associated with each trip.

```
df.columns

'''
Index(['trip_id', 'start_time', 'end_time', 'bikeid',
       'tripduration', 'from_station_id',
       'from_station_name', 'to_station_id',
       'to_station_name', 'usertype', 'gender',
       'birthyear'],
      dtype='object')
'''
```

Only two are selected from this `DataFrame` :

- `start_time` : This data is necessary because we want to predict the use of bicycles based on temporary information. This value, given in seconds using the format "Unix Time" [51] allows to identify at what date and time the trip started among other data. Thanks to this, different patterns can be identified, as these vary during the time of year. There are not the same patterns during the winter as during the summer, or even, it is not the same at 2 in the morning as at 2 in the afternoon.
- `from_station_id` : The models to be built will predict the number of trips that will start at each station in Chicago independently. This value will therefore be used to calculate the number of trips started at an interval at each station. A similar task would have been to predict the number of trips that end at each station for an interval and thus control the flow of bicycles through the station network. If that had been the case, it would have been necessary to make use of `to_station_id` and repeat the same process explained from this point on.

The other attributes are also useful for other types of problems such as predicting the most likely destination based on the station where the journey starts, or obtaining which connections are the most travelled. However, it is outside the objectives that have been set for this work so only the previously mentioned columns will be used.

The resulting dataset is therefore the same as the original dataset but with only the `start_time` and `from_station_id` columns. In the code this is easy to implement by adding only the following line:

```
# ...

# Filter the columns that we need
cols = ["start_time", "from_station_id"]
df = df[cols]

# Save the DataFrame in a CSV
df.to_csv("trips.csv", index=False)
```

More columns could have been used to help predict other than those already filtered, however, due to lack of time and because the results obtained were already quite good, it has not been necessary to add them. This is discussed further in section 8 below.

Graphically this module does the following:

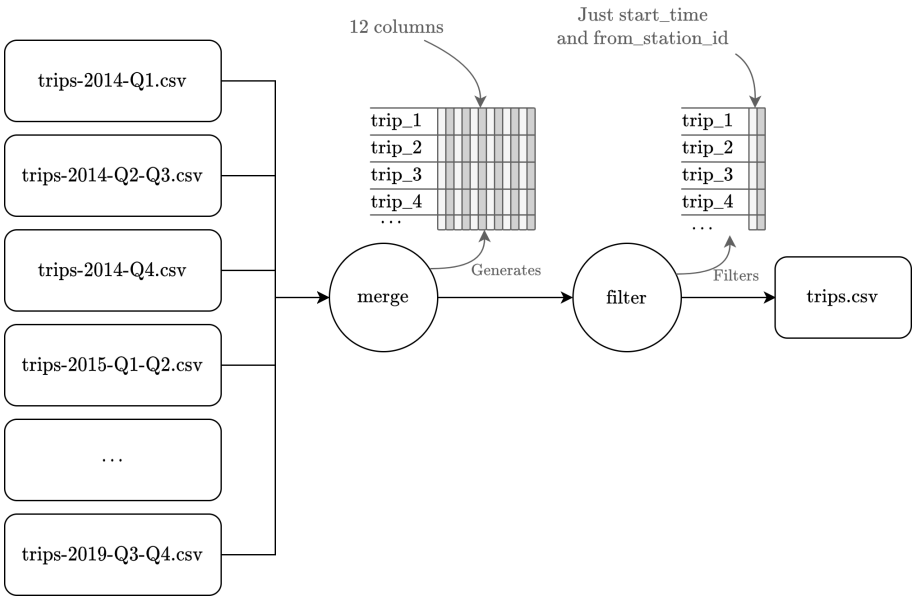


Figure 37: Structure of the feature-selection module

3.3.2 Definition of inputs and outputs of the neural networks

Before continuing to explain the steps carried out, the structure of the dataset needed to train the neural networks must be defined. One of the main requirements in neural network design is the correct specification of the input and output vector of the neural network. The number of variables and the type of variables that will be used in the network is one of the main keys for the model to work satisfactorily. In this section both the structure of the input vector and the structure of the output vector will be explained with graphic examples.

The models worked with will have values in the input vector representing the date and time represented with the following attributes: `hour` (h), `day_of_week` (d) and `month` (m). To do this, it is necessary that the trips are grouped by intervals, thus creating new

columns(*quantity_*j), where j is an integer that represents the station identifier within the network. These columns will be referenced with a q. In total there are 633 stations in the city of Chicago. So for each interval, the neural network will have 636 input values. You can see graphic examples in figure 38 and 39. The size of the final dataset will be `n_intervals` $\times$(`n_stations` $+3$).

The most basic thing would be to create a model that takes a single interval as input and predicts a single interval represented by the output vector. For this model, the network input would be 636, formed by $[h, d, m, q_j], j \; \forall \; [1, 633]$, where h, d and m are the three variables that provide information about the interval and $q_j, j \in [1, 633]]$ are values that represent the number of trips initiated for station j in that interval. The output would be a vector of 633 elements that would represent the prediction of the interval immediately after the input interval and would have the same order as the input vector *quantity_*$j, j \in [1, 633]$.

If you want to use more intervals in the model, the input vector will have a number of elements multiple of 636. For example, if you want to use 5 intervals as input vector, then the network will have a first layer with $636 \times 5 = 3180$ units or neurons. Similarly, if you want the output of the network to be a vector containing the prediction for all the stations in the network, the size of the resulting vector will be a multiple of 633.

In Autoregressive (AR) models, despite being models capable of predicting multiple intervals if necessary, the output of these models will always be an interval. This is because it is necessary to calculate all the intervals independently because the AR model uses its own prediction as an input vector. More information on this can be seen in section 4.3 and 4.2.4.

Below, several examples of different network configurations using different combinations for the number of input and output intervals can be seen graphically:

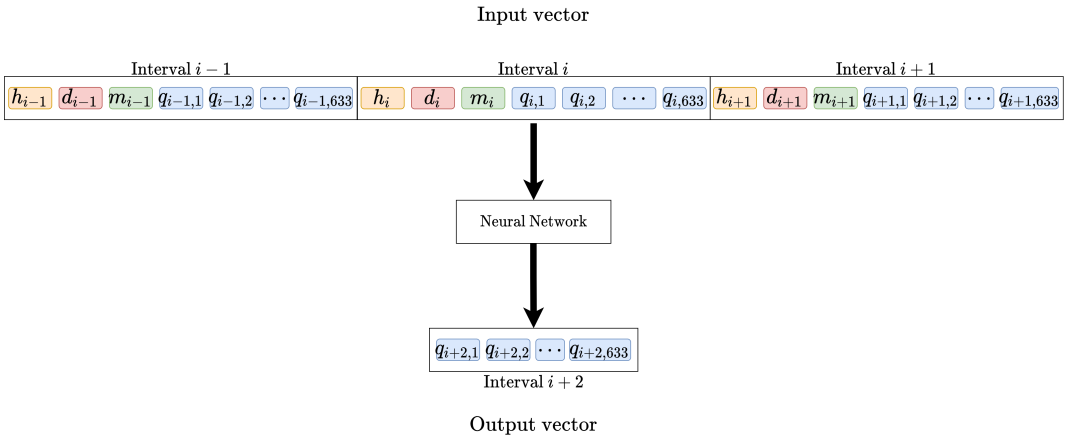


Figure 38: Example of a neural network using 3 input intervals and predicting 1 interval.

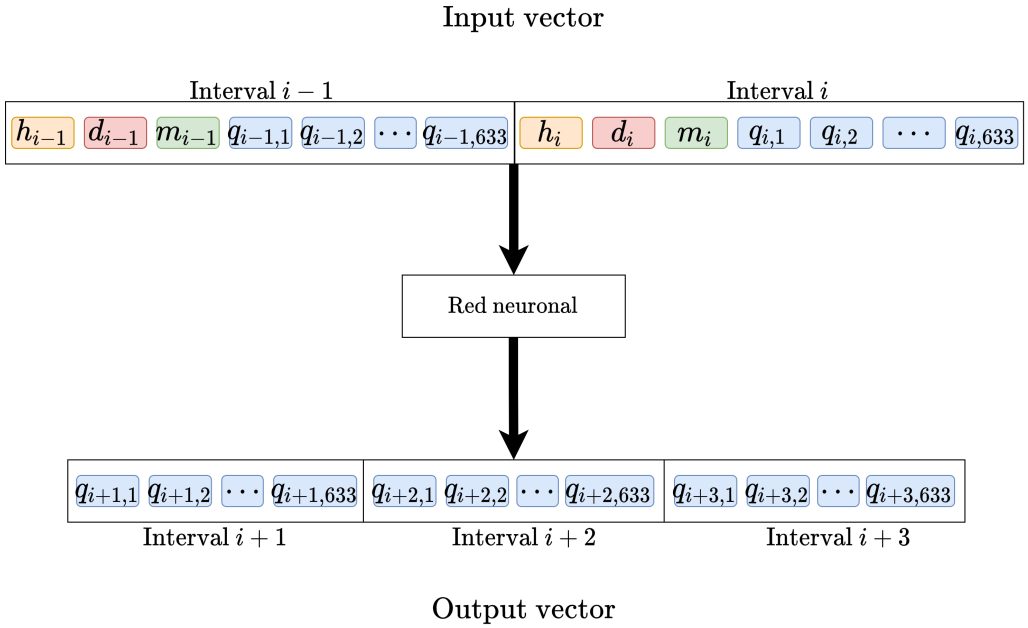


Figure 39: Example of a neural network using 2 input intervals and predicting 3 intervals.

This way of structuring data and models is very flexible. It allows the code to be easily adapted to other datasets and is not unique to the city of Chicago. In addition, it also allows you to work with a smaller group of stations if you wish. As each column is a station, simply by filtering through the columns that are of interest, the models could be trained with these columns. It would also be possible to work by communities, grouping the stations according to a series of rules and thus be able to study the patterns of a neighbourhood and not of each station in a particular way as is done in this work.

3.3.3 Modifying the dataset for the neural network

Once the dataset to be achieved has been explained along with the motivation, this section will explain the steps followed to achieve this result. The dataset at this point has as many rows as there have been trips between 2014 and 2019. For each row, there are two attributes: `from_station_id` and `start_time` .

In this module we want to modify this dataset to obtain another one containing the vectors that can be used as input vectors in the network. That is, we want to create a dataset that contains in each row the intervals and in each column the data of that interval: hour, `day_of_week` , `month` and `quantity_j` . When all the steps of this module have been completed, the data will be saved in a file called `intervals.csv` and a sample of this data can be seen in Appendix C.

First, the code of this module, loads the data of the previous module from the CSV `trips.csv` file:

```
import pandas as pd

# Read CSV as DataFrame an use datetime as index
df = pd.read_csv("/path/to/trips.csv", index="start_time")
```

Once the `DataFrame` has been loaded, the following steps have been carried out:

1. Grouping interval trips: This step aims to group the trips that start at each of the stations into intervals. Different sizes of intervals have been studied, such as 15 or 30 minute intervals, but finally, 1 hour has been chosen because the number of rows is considerably reduced and it is still a valid interval and not very wide.

   The different trips have been grouped in intervals according to the `from_station_id` and `start_time` columns. As an example, suppose you have the following `DataFrame` in the `df` variable:

start_time	*from_station_id*
10:10 - 12/2/2018	48
10:28 - 12/2/2018	15
10:56 - 12/2/2018	15
11:03 - 15/2/2018	15
11:12 - 15/2/2018	48
11:15 - 15/2/2018	15
11:18 - 15/2/2018	15
11:31 - 15/2/2018	15
11:44 - 15/2/2018	48
11:49 - 15/2/2018	15
22:00 - 16/2/2018	15
22:15 - 16/2/2018	48
22:37 - 16/2/2018	193
22:56 - 16/2/2018	48

Table 3: Example of dataset stored in `trips.csv`

The code used to group the trips by intervals and by station has been the following:

```
INTERVAL = "1H"  # It could be also 15Min

df = df.groupby('from_station_id') \
       .resample(INTERVAL, on='start_time') \
       .size() \    # Resampling using the sum rule
       .to_frame()  # Converts it to DataFrame
```

After executing these lines of code and using the example in table 3 you can see that the result is as follows:

start_time	*from_station_id*	*quantity*
10:00 - 12/2/2018	15	2
10:00 - 12/2/2018	48	1
10:00 - 12/2/2018	193	0
11:00 - 15/2/2018	15	5
11:00 - 15/2/2018	48	2
11:00 - 15/2/2018	193	0
22:00 - 16/2/2018	15	1
22:00 - 16/2/2018	48	2
22:00 - 16/2/2018	193	1

Table 4: Example of the dataset grouped by season and by interval.

2. Pivot by intervals: This step aims to group all the intervals into one alone, thus increasing the number of columns and reducing the number of rows. In other words, you want each row to represent an interval and each column to represent a season. Using the same example as in the previous step, the final `DataFrame` would look like this.

start_time	*quantity_15*	*quantity_48*	*quantity_193*
10:00 - 12/2/2018	2	1	0
11:00 - 15/2/2018	5	2	0
22:00 - 16/2/2018	1	2	1

Table 5: Example of a dataset containing the intervals.

Primarily, the code used for this step makes use of the `pivot()` pandas function:

```
# Prepare the new columns names for each station
df["quantity_index"] = "quantity_" + \
                        df["from_station_id"].astype("str")

# We don't need from_station_id anymore
df = df.drop(columns=["from_station_id"])

# Make the pivot around quantity_index column and
# set the value of the column the same value as
# quantity from before saving quantity from
df = df.pivot(columns='quantity_index', values='quantity')

# If station any interval didn't have any trips for
# a station, then fill it with 0
return df.fillna(0)
```

If you compare table 3 with table 6 you can see that the amount of data has been reduced considerably and only the necessary information that the network will need has been selected.

3. Get the time variable: The `start_time` column is a Series with timestamps and therefore has more information than necessary (year, month, day, hour, minute, second...). The only temporary information needed is a subset of these values. Specifically, the values that have been thought to be useful are: the time, the day of the week (it is not the same on a Monday as on a Saturday) and the month (it is not the same on a winter month as on a summer month). These are the simplest values that have been used, but this does not mean that other data that provide more information can't be used, such as: day of the month, year or the minute.

   Using the variables in this way and not using the timestamp directly to the network, the model can learn patterns based on simpler variables such as the time, day of the week or month, and not with a 64-bit number that barely provides any information since the network would take them as ascending values without any criteria.

   The following lines of code have been used to create the new columns with the attributes explained above:

```
# Index and start_time column is the same

df['hour'] = df.index.hour                # Number between 0-23
df['day_of_week'] = df.index.dayofweek    # Number between 0-6
df['month'] = df.index.month              # Number between 1-12
```

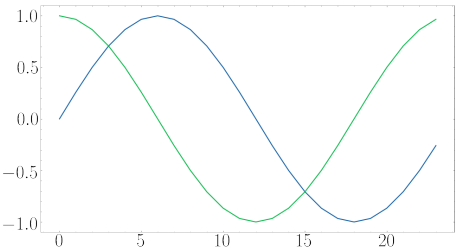
   At the end of this step there will therefore be the following columns for each of the intervals: `start_time` , `hour` , `day_of_week` , `month` and one column for each station. The column `start_time` is not deleted because it is the column used as an index and helps to understand the information you are working with. But it will not be used by the neural network.

   Using the same example as the previous steps, the table would look like this:

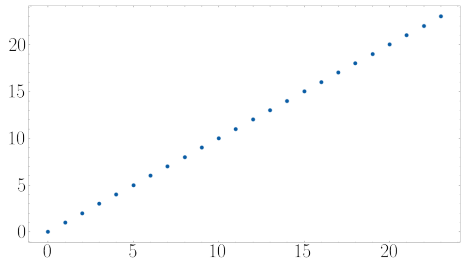
start_time	*quantity_15*	*quantity_48*	*quantity_193*	*hour*	*day_of_week*	*month*
10:00 - 12/2/2018	2	1	0	10	0	2
11:00 - 15/2/2018	5	2	0	11	3	2
22:00 - 16/2/2018	1	2	1	22	4	2

Table 6: Example of a final dataset to be used by the neural network.

   At this point, the possibility of storing the information of the temporal variables in the form of a signal with values between the ranges $[-1, 1]$ using sine and cosine [52] was also studied. This allows the model to have access to the temporal information with continuous and not discrete values as it can be seen in figure 40 below. The results obtained with both techniques were similar, so for simplicity we leave it with discrete values.



((a)) Time with continuous values from -1 to 1



((b)) Time with discrete values from 0 to 23

Figure 40: Possible ways of representing the hour variable.

The `DataFrame` obtained by this module is saved in a CSV file called `intervals.csv` which will be used by the following modules whose content will have a similar structure to that shown in table 6.

```
# Save df in a CSV file
df.to_csv("path/to/intervals.csv")
```

3.4 Window Generator

The first step of this module is to load the `DataFrame` generated in the previous step and divide it into three parts: training, validation and test. Specifically, the 2017 data has been used for the training, the 2018 data for the validation and the 2019 data for the test. The lines of code used to make this division are shown below:

```
from datime import datetime


def narrow_down_dataset(df, from_date, to_date):
    sub_df = df[(df.index >= from_date) & (df.index <= to_date)]

    # Just making sure intervals are sorted
    sub_df = sub_df.sort_values(by="start_time")

    return sub_df


'''
Splits the df into 3 parts depending on the year of the input
'''
def split_dataset(df, train_year=2017, validation_year=2018,
                  test_year=2019):
    dfs = []

    for year in [train_year, validation_year, test_year]:
        # 1st of Jan at 00:00
        from_date = datetime(year, 1, 1, 0)
```

```
        # 31st of Dec at 23:59
        to_date = datetime(year, 12, 31, 23, 59, 59)

        dfs.append(narrow_down_dataset(df, from_date, to_date))

    return dfs

df = pd.read_csv("/path/to/intervals.csv")
train_df, val_df, test_df = split_dataset(df)
```

These three variables (`train_df`, `val_df` and `test_df`) will be used for the different phases of neural network training as explained in section 2.5.1.

The window generator module is based on the code of an official tutorial from the Tensorflow Library [53]. Basically this generator allows to group the intervals in a variable way and allows to adjust the model to the needs that are needed. The models in this work will make a set of predictions based on a window of consecutive samples of the data. Mainly the window can be adjusted with three variables:

- Number of input intervals: Allows you to set the number of intervals that the model will have as input values. In this work, as explained in the previous section, an hour-sized interval is being used. So if you want the model to have data with a previous day to predict, we will indicate to the model that we want to have 24 previous intervals. It must be emphasised that it is extremely important that the data in the window are ordered and that they are contiguous intervals.

- Number of output intervals (labels): Allows you to set the number of intervals you want to predict, i.e., the number of intervals that the model will return. In this work, for each model, different configurations have been tested, the results of which will be shown later. It should be noted that the higher this value is and therefore the greater the number of predictions that the model will make the less accurate the model is expected to be. These are called labels, since they are the intervals that will be used to predict the quantity labels.

- Offset: It is a variable that allows establishing a space between the first input interval and the first label interval. This is useful because you can prepare models that, given 24 input intervals, try to foresee the same interval but from a future day.

Below are some examples of windows with different configurations:

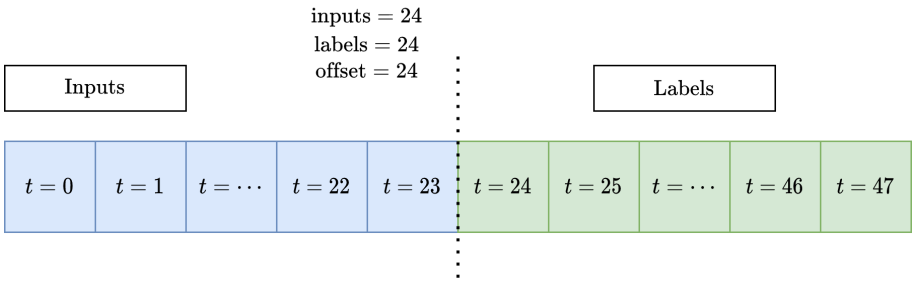


Figure 41: Window with a multiple inputs and multiple outputs.

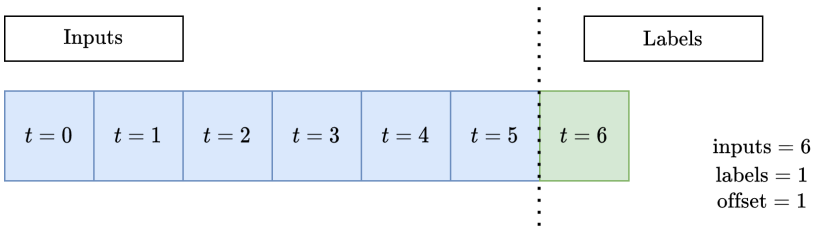


Figure 42: Window with a single output interval.

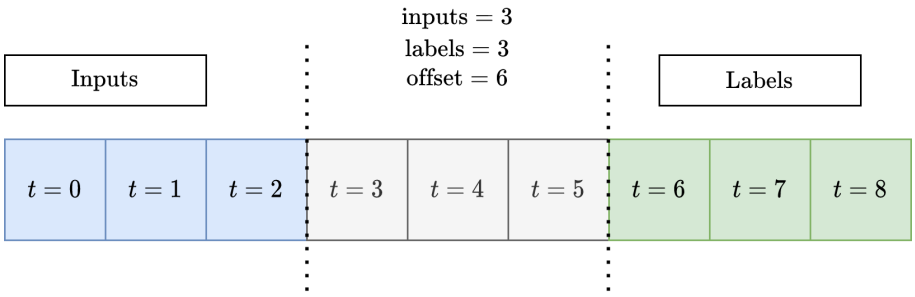


Figure 43: Window with offset.

Each box represents an interval. The blue colour represents that they will be intervals that the model will use as input, each blue box has an associated vector with `hour`, `day_of_week`, `month`, and the number of trips started for all stations in that interval. All the vectors that represent the input intervals, as explained in section 3.3.2, will be transformed into a single dimensional vector. The green boxes on the other hand represent the intervals that the neural networks will predict. For each interval, a vector will be created with as many elements as stations in the system, being each value the prediction for each station.

Predictions on the other hand can be made in two different ways:

- Single prediction: Given a set of input data, all predictions will be made from these data in a single iteration.

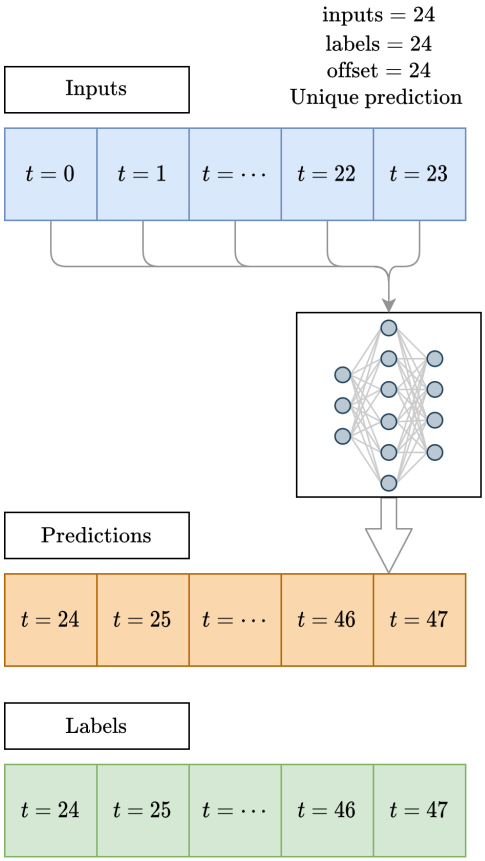


Figure 44: Unique predictions.

- Auto-regressive prediction : The prediction of the nearest interval in the future will be made first. This prediction will be used together with a state vector in order to calculate the next prediction. Viewed graphically:

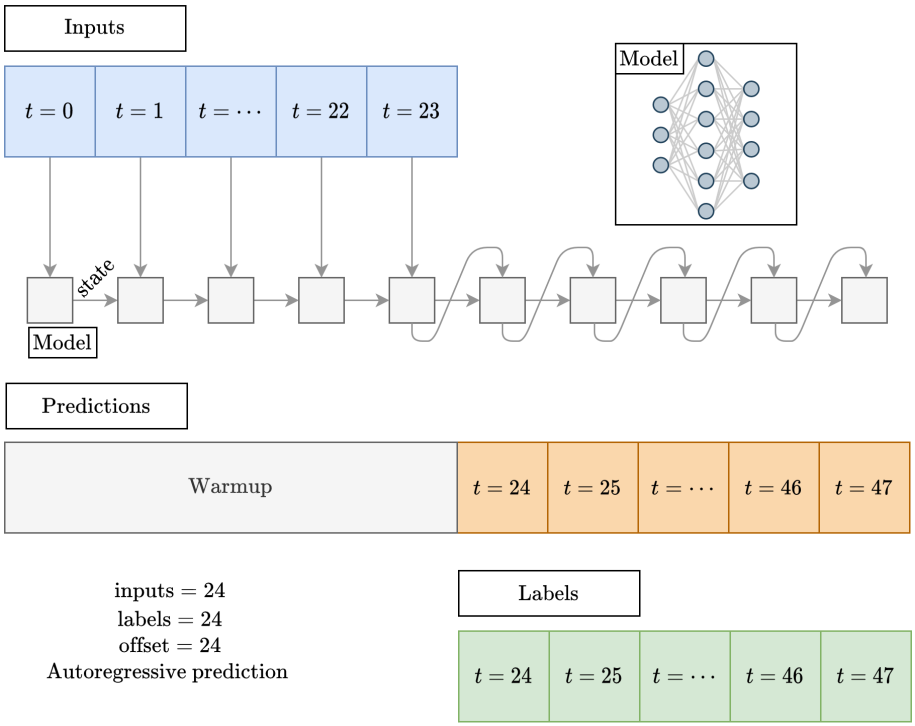


Figure 45: Autoregressive prediction.

Once you have the windows, you must make the window slide and thus generate new vectors that will serve for training. Let us say, for example, that you have a dataset with 7 intervals. You want to train a model that receives 3 input intervals and 1 output interval. Therefore, the model will be trained with a batch of size 4:

Figure 46: Behaviour of a sliding window..

3.4.1 Window Generator Code

For this module, a class has been generated to create objects and it is based largely on the official Tensorflow tutorial [30] that has been modified to meet the needs of the project.

The code used can be seen in Appendix D.

All the methods used in the class are explained in detail below:

- Indexes and offsets: The `__init__` method (Python constructor) includes all the necessary logic for input and label indexes. It also takes the training, validation and test `DataFrames` as input. These will be converted to `tf.data.Datasets` later. Finally, it receives the index of the column where the temporary information is located. This index is a value equal to -3, because as you can see in table 6, the last three columns are the ones that contain information about the time, day of the week and month.
- Splitting the window between input and label: Given a list of consecutive intervals, the `split_window()` method will turn into an input window and a label window.
- Creation of `tf.data.Datasets` : The `make_dataset()` method will take an interval `DataFrame` and convert it into a pair `tf.data.Dataset` (`input_window` , `label_window`). Size 32 batches have been used.
- The `WindowGenerator` class contains the training, validation and test `DataFrames`. At the end, the properties to access them are added, which are of type `tf.data.Datasets` that make use of the previous method `make_dataset`.

4 DEVELOPED MODELS

This section explains the models that have been developed. A base model and four neural networks with four different architectures have been created. The results of these models can be seen in section 5.

4.1 Baseline model

Before defining the neural networks in this work, a study of other similar work and results has been carried out and can be seen in section 2.7. This allows us to use the results of other projects as a reference point, knowing whether the development of this practice is improving the results obtained by other projects or not. In addition to having the points of reference of other projects, a base model has been defined.

A base model is a trivial model that is usually developed when you want to solve a prediction or classification problem in order to know if the other models to be developed will improve the result. In this project, the base model developed does not have logic and simply returns the value that existed one week before as a prediction.

Below you can see graphically the model's predictions with respect to the real values:

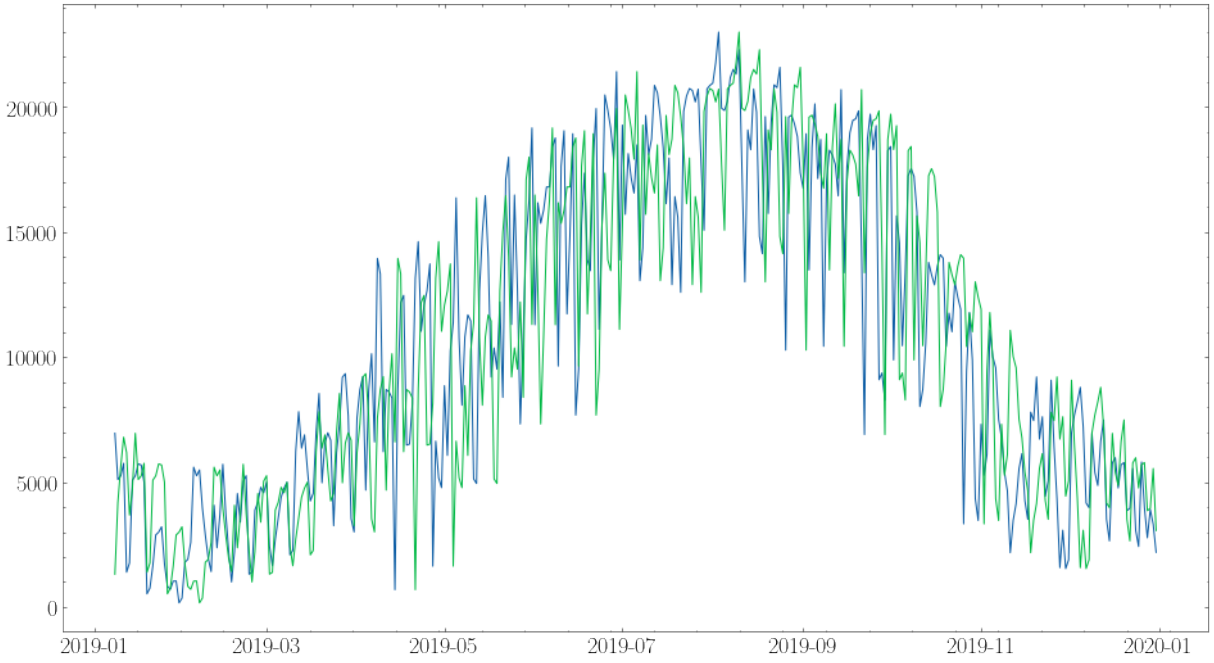


Figure 47: Predictions of the basic model

It can be seen that the predictions are the same real values but displaced by a week. Once this model has been developed, the metrics shown in section 5 have been calculated together with the rest of the results of the other models.

4.2 Models made with neural networks

4.2.1 Dense model

This model is a forward-pass type single layer model. It has two other layers, but they do not implement logic associated with the neural network itself, but are layers of translation from matrix to vector and vice versa as explained below. Graphically, the network that has been defined is the following:

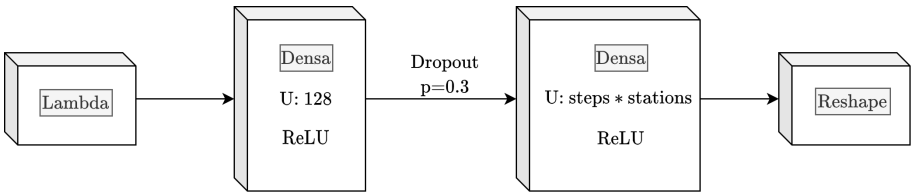


Figure 48: Dense model

The first hidden layer has 128 units. The number of layers and the number of neurons in a layer are parameters that have been set in a pseudo-random way. Different combinations have been made trying different amounts of units and 2 dense layers with 128 units in the first one is the combination that has obtained the best results. The second hidden layer has the same amount of units that the model has to return, which is a value calculated by multiplying the number of stations with which it is working, 633, by the number of intervals to be calculated. For example, if you want to calculate a single interval for all the stations, the vector of that layer will be 633 elements, if you want to calculate two intervals, the number of elements of the vector will be $1266 = 2 \times 633$ and so on. This layer is common to all the models that will be explained from now on.

The last layer is a Reshape-type layer. This layer has neither an activation function nor parameters to adjust because all it does is convert the vector of the previous layer into a more understandable matrix and with which you can work more comfortably with the dataset where each row will be an interval and each column will be a station. The values of this matrix will therefore be the prediction that the model returns for the interval and station defined by its row and column in which they are found.

Following the same logic, the first layer is of the Lambda type and is in charge of performing the opposite task, given a matrix to obtain a vector. Lambda layers are layers in which the user defines a function to be executed.

The code for this model has been defined as shown below:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Reshape, Dense, Dropout, Lambda

# `steps` is a variable which has the number of intervals to be predict
steps = 0

# `stations` is the number of stations in the bike network
stations = 0
```

```
dense_model = Sequential([
    # Matrix to vector
    Lambda(lambda x: x[:, -1:, :]),

    # Input layer
    Dense(128, activation="relu"),

    Dropout(0.3),

    # Ouput layer
    Dense(steps * stations),

    # Vector to matrix
    Reshape([steps, stations])
])
```

The results of the model can be seen together with the other results in section 5.

4.2.2 Basic recurrent model

This model is the same as the dense model but with the addition of a new simple recurrent layer explained in section 2.6.1. The simple recurrent layer is a layer that allows us to use information from other previous intervals that will be used for the final prediction. This recurrent layer has 100 units and graphically the model can be represented as follows:

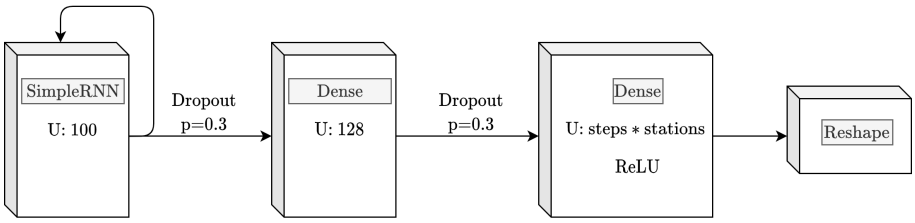


Figure 49: Basic recurrent model

As you can see it is an extension of the dense model explained in the previous section but adding a new *SimpleRNN* type layer. The code of this model has been defined as shown below:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Reshape, Dense, Dropout, Lambda, SimpleRNN

# `steps` is a variable which has the number of intervals to be predict
steps = 0

# `stations` is the number of stations in the bike network
stations = 0

rnn_model = Sequential([
    # RNN layer
    SimpleRNN(100, return_sequences=True),

    Dropout(0.3),
    Dense(128),
    Dropout(0.3),

    # Ouput layer
```

```
    Dense(steps * stations),

    # Vector to matrix
    Reshape([steps, stations])
])
```

The results of the model can be seen together with the other results in section 5.

4.2.3 LSTM model

This model is the same as the basic recurrent model but replacing the recurrent layer with an LSTM layer which is more complex and allows the network to learn to use more information from the past as explained in section 2.6.6. This recurrent layer has 128 units and graphically the model can be represented as follows:

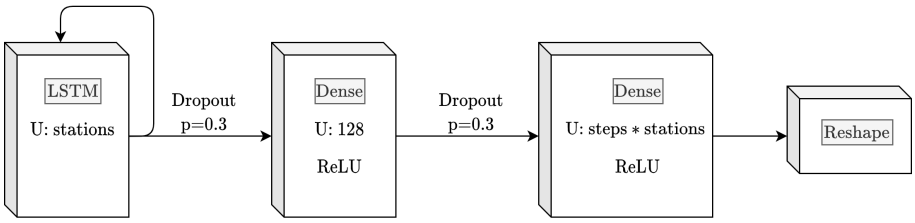


Figure 50: Recurring model LSTM

As you can see it is the same model but replacing the *SimpleRNN* layer by a LSTM. The code of this model has been defined as shown below:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Reshape, Dense, Dropout, Lambda, LSTM

# `steps` is a variable which has the number of intervals to be predict
steps = 0

# `stations` is the number of stations in the bike network
stations = 0

lstm_model = Sequential([
    # LSTM layer
    LSTM(stations, return_sequences=False),

    Dropout(0.3),
    Dense(128, activation="relu"),

    # Output layer
    Dropout(0.3),
    Dense(steps * stations, activation="relu"),

    # Vector to matrix
    Reshape([steps, stations])
])
```

The results of the model can be seen together with the other results in section 5.

4.2.4 Autoregressive (AR) model

An AR model is another type of recurrent neural network. The peculiarity of this network is that the result obtained is used as an input value for the calculation of the next iteration. The graphically generated model is as follows:

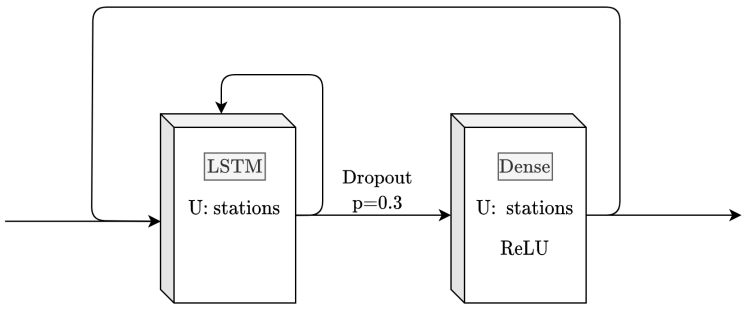


Figure 51: Recurrent AR model

The model has an LSTM layer along with a dense layer. The output of this layer which is the prediction of the model is reintroduced into the model to predict the rest of the predictions.

Tensorflow does not have a model in its tool library that meets the needs that were being sought and for that reason, this model has had to be implemented using another technique. A new class has been created that inherits from `tf.keras.Model` and which has three methods: `__init__` , `warmup` and `call` . In this way it is possible to generate models that can have unique architectures different from the pass-forward ones. For the implementation, the documentation of Tensorflow [30] and Keras [46] has been followed. The code developed for this class can be read in Appendix E.

To create a new AR model simply generate a new instance as follows:

```
AutoRegressive(lstm_units=stations, steps=steps, stations=stations)
```

The results of the model can be seen together with the other results in section 5.

4.3 Windows

As explained in section D, predictions can be generated based on the input arguments. That is, you can create a model that, given an X number of intervals, can predict a Y number of intervals. We wanted to compare different adjustments that can be made to the windows and see the results obtained. In short, a different model was created for each of the window sizes.

From here on, reference will be made to the window configuration with tuples, where the first term is the size of input intervals and the second element is the size of intervals that the model predicts. For example given the following tuple 12-5, it means that the

model developed uses a window with 12 input intervals and 5 output intervals.

A total of 15 different window configurations have been used: 3-1, 5-1, 8-1, 8-3, 8-5, 12-1, 12-3, 12-5, 24-1, 24-3, 24-5, 36-8, 36-12, 48-12, 48-24. As 4 different neuronal architectures have been developed, 60 models plus the basic model have been developed at the end.

4.4 Training, validation and testing of neural networks

Once the models have been created, a function has been coded to automate the process of training, validation and testing of each of the models, in addition to obtaining the metrics that will be needed to study and compare the different models. The code can be read in Appendix F.

The code is divided into three distinct functions:

- The `compile_and_fit()` function, given a network architecture model (dense, RNN, LSTM or AR), will compile such a model using Huber's loss as a cost function and Adam's optimizer for gradient descent. The learning rate used has been a value between 0.001 and 0.0001 depending on the model. To obtain this value, it has been made use of the technique explained in section 2.5.7 in which the learning rate is calculated in an iterative way and studying which values make the descent of the gradient converge in a faster and constant way.

  The optimiser used was Adam's optimiser. This decision has been made with the same process as the decision of how many layers and how many neurons to use in the network, on a trial and error basis. Surely, there is a combination that obtains better results, but from the experience of the results obtained with the different comparisons, the models usually do not improve excessively by this type of decisions, it is much more important other hyper parameters like the activation function to use, an optimal learning rate or the use of Dropout between layers.

  Another technique to avoid the appearance of overfitting is the use of a function called `EarlyStopping` that will stop the learning process if it considers that the model is starting to memorise and not learn or if after a `patience` equal to 10 epochs the model has not learned considerably from the past. This is done from `callbacks` , which are functions that are executed after executing an epoch. Another `callback` used has been a function that in real time draws the graphs of how the learning process is going with the metrics with the training dataset as the validation dataset.

  The cost function used will be explained in the next section along with the metrics used to measure model performance.

- The `compile_and_fit()` function only works with one model and one specific window size. On the other hand, there is the `model_generator()`) function that, given a model, is in charge of creating all the windows with the different sizes and training the model for each of the windows. As explained previously, in total 15 different window sizes have been used, so a call to the `model_generator()` function causes 15 different models to be trained and therefore the time of this process usually takes between 2 and 5 hours depending on the model being trained.

- For each model trained and in order not to waste the results, the function `get_metrics()` is in charge of computing this data and saving it in a CSV that will be used by other functions later on to display the results and draw different graphs.

5 RESULTS

This section will show all the results obtained with the different models. In total 4 models have been developed (1 baseline model and 3 neural networks). These models have been iterated from a baseline model to more complex models. This improvement process is reflected in the metrics obtained.

5.1 Metrics used

Six different metrics have been used to measure the performance of the models. In addition, one of these metrics, Huber's loss (see section 2.5.2) has been used as a cost function.

5.1.1 Loss of Huber

As explained in section 2.5.2, this loss function does not prioritise outliers but treats all data similarly. This is very useful, for the problem we want to solve.

In the dataset that is being used, a data that does not follow the pattern (outlier) does not mean that it is an error that should be considered as can happen in other types of problems. For example, a bicycle station right in front of a university is in great demand on a Thursday at 5pm, while the rest of the Thursday has not been in such demand for that interval. This is due to a single event, e.g., an examination at that faculty on that day. As this is a valid data and not an error as such, the network will not immediately learn that there will be such demand on Thursdays, but it will take multiple days of repetition for the network to begin to adjust to this data and so the loss of Huber is well suited to this type of problem. The equation for Huber's loss is as follows:

$$\text{Huber} = \begin{cases} \text{MSE} & \text{if } |\hat{y} - y| \leq \delta, \\ \text{MAE} & \text{e.o.c.} \end{cases} \tag{95}$$

For the implementation of the code, Keras' own library implementation has been used:

```
from tf.losses import Huber
```

5.1.2 MAE

The mean absolute error (MAE) is one of the most basic and fastest metrics to calculate. It is calculated using the average of the differences between the actual values and the predicted values as shown below:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{96}$$

To make use of this equation in Keras, it can be imported as follows:

```
from tf.metrics import MeanAbsoluteError
```

5.1.3 MSE y RMSE

These two metrics indicate basically the same. The MSE is a value that is obtained by calculating the average between all the values obtained by applying the square to the difference between the real value and the predicted value as shown below:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{97}$$

On the other hand, RMSE is the square root of the RMSE as can be seen below:

$$\text{RMSE} = \sqrt{MSE} \tag{98}$$

To be able to make MSE and RMSE calculations, they can be imported from Tensorflow.

```
from tf.losses import MeanSquaredError
from tf.metrics import RootMeanSquaredError
```

5.1.4 MSLE y RMSLE

These two metrics are used because there are many projects, including the Kaggle competition (see section 2.7.3) that use them, so that the results can be compared. The explanation of MSLE can be seen in section 2.5.2. RMSLE is the same as MSLE but with a square root:

$$\begin{aligned} \text{MSLE} &= \frac{1}{n}\sum_{i=1}^{n}\left(\log\left(\frac{y_i+1}{\hat{y}_i+1}\right)\right)^2 \\ \text{RMSLE} &= \sqrt{\text{MSLE}} \end{aligned} \tag{99}$$

In order to be able to do MSLE calculations, the Keras function can be imported. This function also allows us to define the RMSLE function as shown below:

```
from tf.keras.losses import MeanSquaredLogarithmicError
def RootMeanSquaredLogarithmicError(y_true, y_pred):
    msle = MeanSquaredLogarithmicError(y_true, y_pred)
    return tf.keras.backend.sqrt(msle)
```

5.2 Results of the baseline model

The baseline model, as explained in section 4.1, returns the number of trips that started for an interval the previous week for which you want to predict the data. Calculating all the metrics, the results of this model are as follows:

Metrics	val	test
MSLE	0.221549	0.239364
MSE	3.383871	3.699979
MAE	0.603720	0.647458
HUBER	0.462739	0.499985
RMSE	1.913998	1.914635
RMSLE	0.470690	0.489248

Table 7: Metrics obtained from the baseline model.

These values are the ones that will be used as a reference to be able to compare the models that are developed and check that an improvement is being achieved.

5.3 Neural networks

Below are the results of all the models with the different windows that have been trained to study how they behave. All the models have the same hyperparameters that have been explained in section 2.2.1. The columns represent the models used, which all have the metrics obtained for both the validation and the testing datasets. The rows are divided both by the metrics used and by the window size for each model. In total 60 models have been developed (15 different window sizes for 4 neural network models):

	Windows	Dense		RNN		LSTM		AR	
		val	test	val	test	val	test	val	test
	(3, 1)	0.344	0.380	0.334	0.374	0.474	0.508	0.290	0.321
	(5, 1)	0.375	0.411	0.347	0.388	0.476	0.510	0.296	0.330
	(8, 1)	0.410	0.445	0.352	0.389	0.477	0.512	0.291	0.324
	(8, 3)	0.348	0.380	0.331	0.369	0.293	0.329	0.295	0.331
	(8, 5)	0.373	0.411	0.346	0.385	0.301	0.342	0.303	0.342
	(12, 1)	0.435	0.472	0.378	0.418	0.472	0.506	0.288	0.320
	(12, 3)	0.344	0.379	0.334	0.370	0.294	0.330	0.302	0.338
HUBER	(12, 5)	0.371	0.409	0.346	0.386	0.303	0.340	0.307	0.344
	(24, 1)	0.460	0.495	0.434	0.474	0.472	0.507	0.290	0.323
	(24, 3)	0.353	0.386	0.332	0.370	0.295	0.331	0.300	0.335
	(24, 5)	0.363	0.401	0.347	0.386	0.301	0.341	0.311	0.348
	(36, 8)	0.355	0.399	0.358	0.397	0.299	0.338	0.317	0.358
	(36, 12)	0.362	0.403	0.373	0.409	0.306	0.348	0.335	0.385
	(48, 12)	0.367	0.408	0.370	0.409	0.306	0.347	0.327	0.369
	(48, 24)	0.374	0.414	0.385	0.423	0.311	0.353	0.337	0.374

	Windows	Dense		RNN		LSTM		AR	
		val	test	val	test	val	test	val	test
	(3, 1)	0.136	0.154	0.130	0.149	0.235	0.253	0.115	0.131
	(5, 1)	0.153	0.171	0.140	0.160	0.240	0.258	0.117	0.134
	(8, 1)	0.177	0.195	0.143	0.162	0.242	0.261	0.115	0.131
	(8, 3)	0.134	0.151	0.129	0.148	0.114	0.131	0.118	0.135
	(8, 5)	0.149	0.170	0.136	0.156	0.117	0.135	0.123	0.142
	(12, 1)	0.198	0.217	0.157	0.177	0.233	0.251	0.117	0.132
	(12, 3)	0.132	0.150	0.129	0.147	0.114	0.131	0.123	0.140
MSLE	(12, 5)	0.148	0.169	0.136	0.157	0.118	0.135	0.124	0.142
	(24, 1)	0.217	0.235	0.194	0.217	0.234	0.252	0.118	0.134
	(24, 3)	0.137	0.154	0.130	0.149	0.115	0.132	0.122	0.139
	(24, 5)	0.144	0.164	0.138	0.157	0.117	0.136	0.129	0.148
	(36, 8)	0.144	0.167	0.142	0.163	0.119	0.138	0.128	0.148
	(36, 12)	0.148	0.169	0.148	0.166	0.121	0.141	0.138	0.162
	(48, 12)	0.150	0.172	0.150	0.170	0.121	0.141	0.135	0.156
	(48, 24)	0.154	0.175	0.155	0.175	0.122	0.142	0.139	0.158

	Windows	Dense		RNN		LSTM		AR	
		val	test	val	test	val	test	val	test
	(3, 1)	2.518	2.802	2.513	2.831	4.958	5.302	1.623	1.793
	(5, 1)	3.228	3.526	2.678	3.022	5.017	5.375	1.684	1.899
	(8, 1)	3.885	4.144	2.755	3.018	5.028	5.392	1.622	1.826
	(8, 3)	2.792	3.012	2.454	2.736	1.740	2.025	1.692	1.976
	(8, 5)	3.185	3.550	2.730	3.031	1.890	2.284	1.839	2.155
	(12, 1)	4.278	4.593	3.237	3.514	5.017	5.358	1.582	1.785
	(12, 3)	2.746	3.017	2.557	2.830	1.784	2.081	1.810	2.052
MSE	(12, 5)	3.171	3.510	2.692	2.998	1.929	2.240	1.895	2.187
	(24, 1)	4.653	4.974	4.393	4.749	5.032	5.389	1.604	1.818
	(24, 3)	2.831	3.069	2.448	2.724	1.751	2.038	1.749	2.002
	(24, 5)	2.990	3.330	2.701	2.980	1.872	2.222	2.122	2.403
	(36, 8)	2.724	3.146	2.928	3.220	1.743	2.111	2.014	2.404
	(36, 12)	2.880	3.225	3.210	3.475	1.870	2.272	2.327	2.842
	(48, 12)	2.965	3.333	3.098	3.372	1.855	2.246	2.093	2.476
	(48, 24)	3.081	3.431	3.377	3.653	1.950	2.350	2.357	2.649

	Windows	Dense		RNN		LSTM		AR	
		val	test	val	test	val	test	val	test
	(3, 1)	0.508	0.550	0.519	0.565	0.699	0.738	0.468	0.504
	(5, 1)	0.546	0.587	0.543	0.592	0.674	0.713	0.475	0.515
	(8, 1)	0.594	0.636	0.542	0.586	0.672	0.711	0.469	0.508
	(8, 3)	0.524	0.563	0.521	0.565	0.469	0.512	0.455	0.498
	(8, 5)	0.546	0.591	0.535	0.581	0.477	0.524	0.461	0.506
	(12, 1)	0.642	0.685	0.561	0.609	0.702	0.741	0.444	0.481
	(12, 3)	0.519	0.560	0.520	0.562	0.469	0.512	0.458	0.501
MAE	(12, 5)	0.545	0.590	0.537	0.583	0.478	0.522	0.463	0.506
	(24, 1)	0.677	0.718	0.629	0.674	0.704	0.743	0.446	0.485
	(24, 3)	0.526	0.564	0.522	0.567	0.473	0.515	0.453	0.494
	(24, 5)	0.536	0.580	0.539	0.585	0.477	0.523	0.467	0.511
	(36, 8)	0.548	0.599	0.551	0.598	0.477	0.523	0.475	0.523
	(36, 12)	0.556	0.604	0.562	0.606	0.485	0.534	0.488	0.545
	(48, 12)	0.559	0.608	0.571	0.617	0.485	0.533	0.479	0.530
	(48, 24)	0.570	0.618	0.582	0.626	0.491	0.539	0.490	0.534

	Windows	Dense		RNN		LSTM		AR	
		val	test	val	test	val	test	val	test
	(3, 1)	1.587	1.674	1.585	1.683	2.225	2.303	1.273	1.339
	(5, 1)	1.797	1.877	1.637	1.739	2.241	2.319	1.297	1.378
	(8, 1)	1.972	2.037	1.660	1.738	2.244	2.322	1.274	1.351
	(8, 3)	1.671	1.735	1.567	1.655	1.318	1.424	1.301	1.406
	(8, 5)	1.785	1.885	1.653	1.740	1.375	1.511	1.356	1.468
	(12, 1)	2.070	2.143	1.796	1.876	2.238	2.317	1.258	1.337
	(12, 3)	1.659	1.738	1.598	1.679	1.336	1.441	1.346	1.434
RMSE	(12, 5)	1.782	1.876	1.642	1.732	1.389	1.497	1.377	1.478
	(24, 1)	2.157	2.230	2.096	2.179	2.243	2.321	1.266	1.348
	(24, 3)	1.682	1.752	1.565	1.651	1.323	1.428	1.323	1.415
	(24, 5)	1.729	1.825	1.643	1.726	1.368	1.491	1.457	1.550
	(36, 8)	1.649	1.772	1.712	1.795	1.320	1.453	1.419	1.550
	(36, 12)	1.697	1.798	1.792	1.865	1.366	1.508	1.526	1.685
	(48, 12)	1.722	1.826	1.760	1.836	1.362	1.499	1.447	1.574
	(48, 24)	1.755	1.853	1.837	1.911	1.397	1.533	1.535	1.627

	Windows	Dense		RNN		LSTM		AR	
		val	test	val	test	val	test	val	test
RMSLE	(3, 1)	0.367	0.391	0.360	0.385	0.484	0.501	0.339	0.360
	(5, 1)	0.390	0.413	0.373	0.399	0.488	0.506	0.342	0.365
	(8, 1)	0.420	0.440	0.377	0.401	0.490	0.509	0.339	0.361
	(8, 3)	0.365	0.388	0.359	0.384	0.337	0.362	0.343	0.367
	(8, 5)	0.385	0.411	0.368	0.394	0.341	0.367	0.350	0.376
	(12, 1)	0.444	0.465	0.395	0.420	0.481	0.500	0.341	0.363
	(12, 3)	0.362	0.387	0.358	0.383	0.337	0.361	0.350	0.374
	(12, 5)	0.384	0.410	0.369	0.395	0.342	0.367	0.351	0.376
	(24, 1)	0.464	0.484	0.439	0.464	0.482	0.501	0.343	0.366
	(24, 3)	0.369	0.392	0.360	0.385	0.339	0.363	0.349	0.372
	(24, 5)	0.378	0.404	0.370	0.396	0.342	0.368	0.359	0.384
	(36, 8)	0.379	0.408	0.377	0.403	0.344	0.370	0.357	0.385
	(36, 12)	0.384	0.411	0.384	0.407	0.347	0.375	0.371	0.402
	(48, 12)	0.387	0.414	0.387	0.412	0.347	0.375	0.367	0.395
	(48, 24)	0.392	0.418	0.394	0.418	0.350	0.377	0.372	0.397

Graphs have also been developed to make it easier to visualise the metrics using the testing dataset. In all the graphs there is a horizontal line representing the metric obtained by the baseline model:

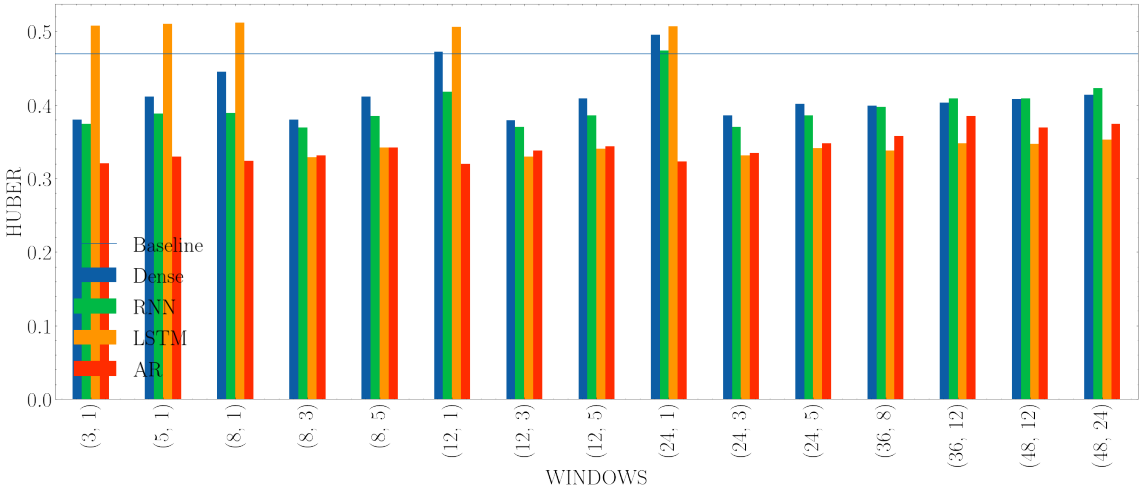


Figure 52: Results using Huber's loss

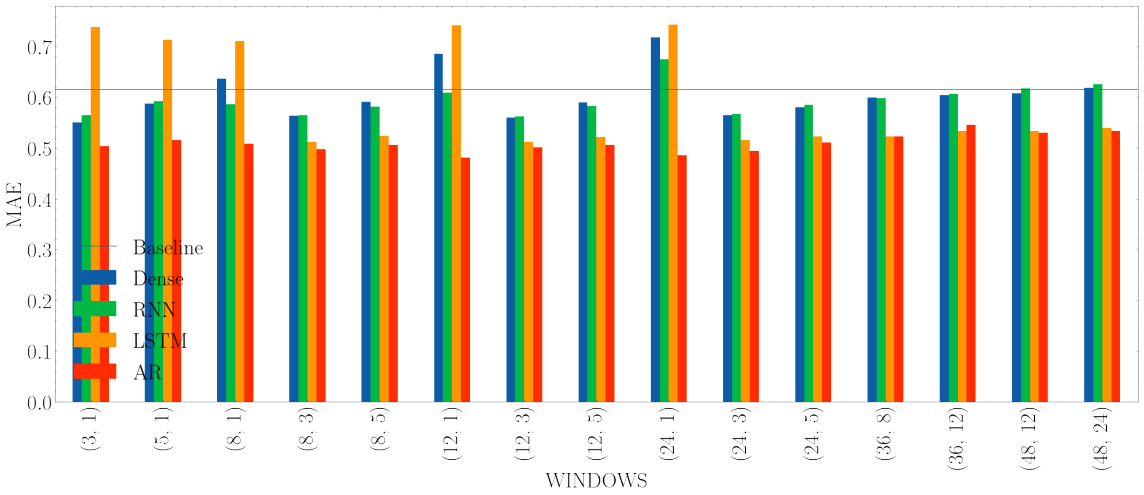


Figure 53: Metrics using MAE

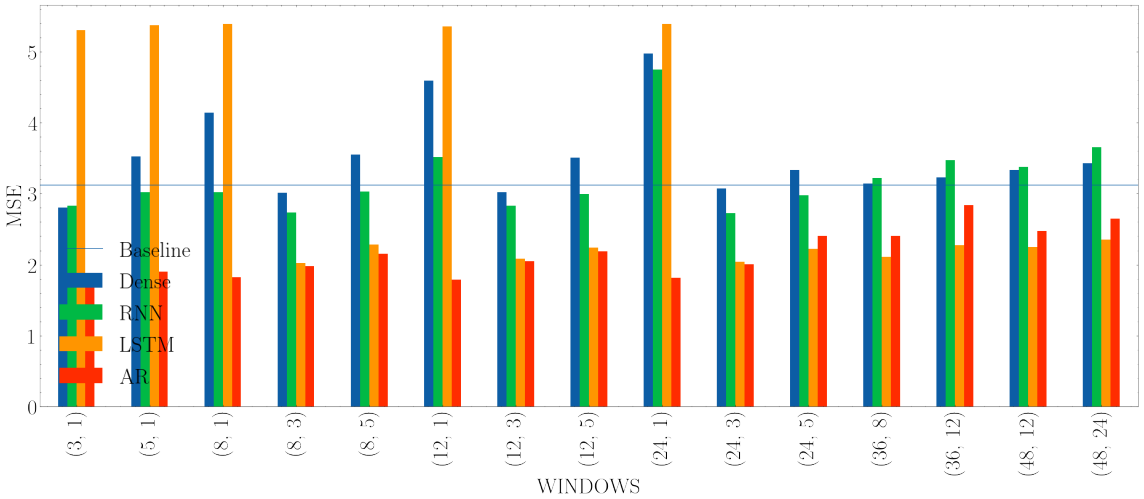


Figure 54: Metrics using MSE

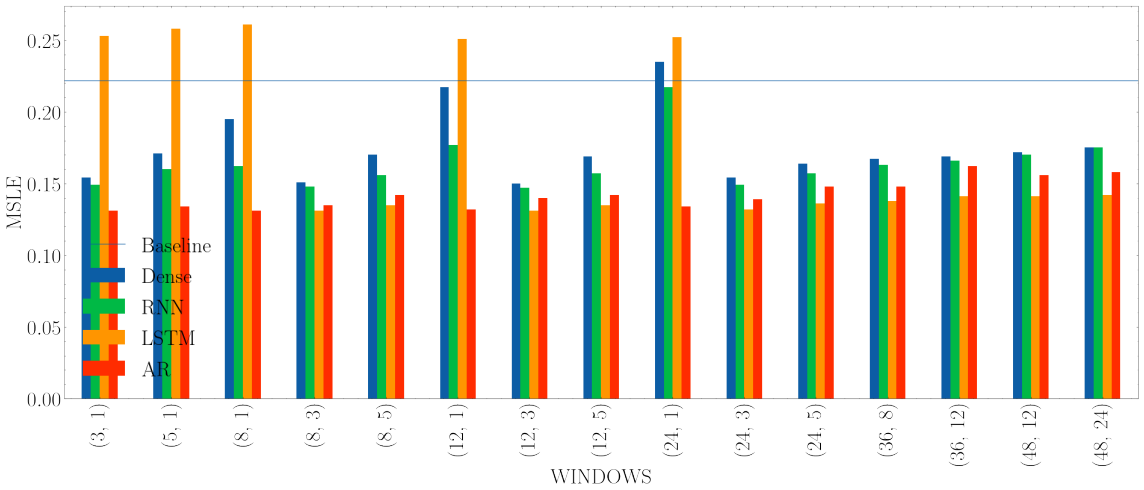


Figure 55: Metrics using MSLE

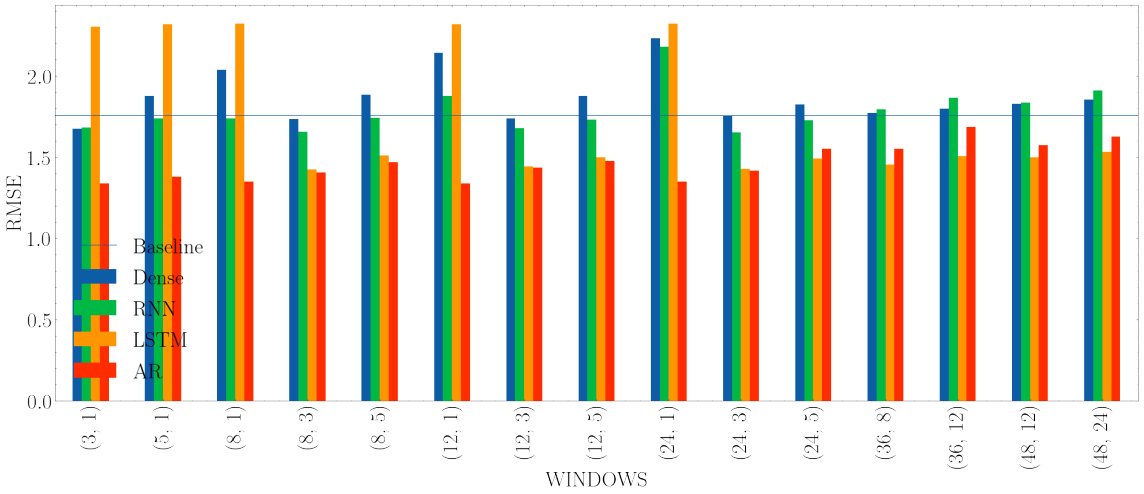


Figure 56: Metrics using RMSE

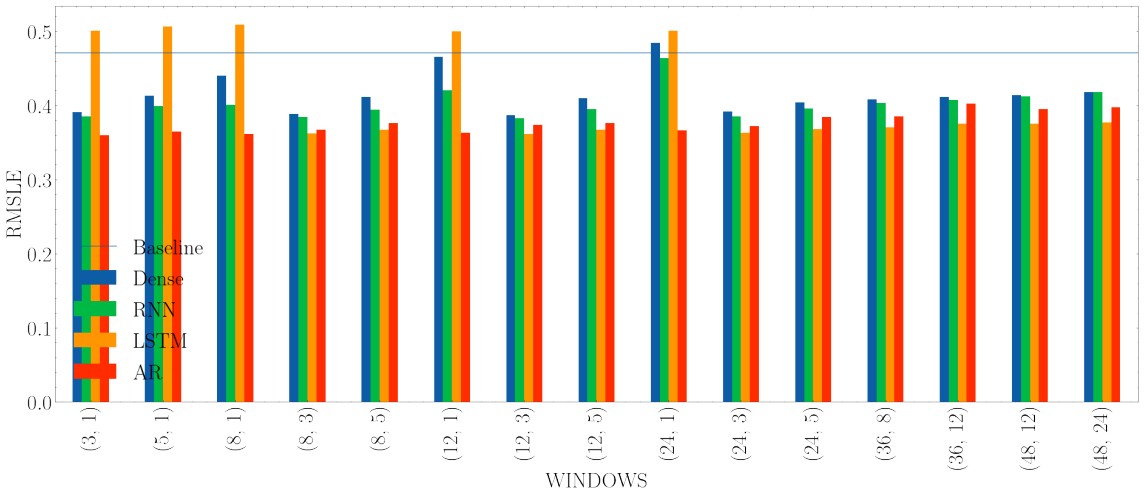


Figure 57: Metrics using RMSLE

6 PROBLMES ENCOUNTERED

During the development of this work we have not encountered different problems both in the development of neural networks and in the pre-processing necessary to create eldataset

6.1 Structure of the dataset

Every neural network requires a well-defined architecture before it can be created. Among other parameters, it is necessary to define both the input and output with which the model will work. At the beginning of the development, when training the first test networks, we obtained very bad values for what we were waiting for. This was a consequence of the poor definitionhttps://www.overleaf.com/project/600b5b96eebd9b7f42454677 that had been carried out to train the networks.

The mistake was mainly in what I was contemplating from the beginning that a network could accept a matrix as input, which is not true. A network always has to work with vectors for the variables x and y.

Later, I developed a structure that made more sense and with which the network obtained better results. The problem with this solution was the training time when trying to train a basic network. It was unfeasible to continue with this data structure.

Finally, after several meetings with the tutor and with the help of a PhD student we concluded that the best way to train the neural network was with the data structure explained earlier in this paper. This solution, despite being very intuitive and easy to understand, has taken me a great deal of time to come up with.

6.2 Development of pre-processing

In order to develop a data structure with which the models of the neural networks can be generated, it is necessary to restructure the dataset we use to a dataset with which the necessary windows can be generated.

It is necessary to use some kind of library to facilitate this process. In this work we have decided to use `DataFrame` of pandas together with other libraries like Numpy or Tensorflow, which were my first time using them but I have been able to adapt easily.

Despite this, I have encountered more problems than expected when using these libraries and they have slowed down my development. In addition, many of the functions that have been developed I was not sure if they performed the task that was entrusted to them and therefore, I have also spent some time to write some unitary tests to ensure that the code was good.

Many of these problems have their origin in working with more than 3 dimensions. From that point on, it is much more complex for the human brain to be able to treat structures of this style and it is important to carry out a study and a pseudo-code of what you want to do previously. I will certainly try to apply this to future projects: spending more time on design.

6.3 Overfitting problem

The problem of overfitting is a common problem when training neural networks. As explained in section 2.5.7, overfitting appears after a period of time in which the network has already learned and begins to memorize the data, causing it not to adapt to new situations correctly and may even worsen the results of the model.

For that reason, I have dedicated part of my work to studying how to avoid overfitting. Initially I used the strategy of working with regulators, which are equations with which the weights of a network are changed more slowly. But this solution has not had the expected results. What was achieved was that overfitting appeared but in much more advanced training iterations, so I dismissed this use of regulators.

Finally, two techniques have been used in the current models to avoid overfitting. Firstly, a technique has been used by which a learning rate is chosen with which the model converges when calculating the gradient. This technique is based on using several values of the learning rate and studying how the descent of the gradient behaves with respect to it. In second place, the technique of using Dropout (see section 2.5.7) has been used, which basically activates and deactivates connections between neurons in the network in a random way so that none is dispensable, that is to say, that it contains a great deal of knowledge regarding the patterns that are to be predicted.

7 CONCLUSIONS

7.1 Study of the results

The graphs presented in the previous section show the metrics obtained using the test `DataFrame`. Using this data, it reflects how good the models are at data you have never seen before and how they would behave in a production environment. In general, all the results are better than the results with the base model.

If you compare the architectures with each other, you can see the result we expected. The dense models are the worst of the four, even though their results are quite good in themselves. By using an architecture that can also learn from the past, the results can be slightly improved compared to the dense model. These improvements are noticeable in the models that use small windows. On the contrary, the larger the window size, the SRNN start to perform slightly worse than the dense ones. The reason for this pattern can be found in the fact that these models have too much information and cannot synthesize information from the past. This is the main problem of SRNN that was discussed in Section 2.6.1.

As far as LSTM architectures are concerned, it is worth mentioning that the calculated results generate an exception in that all models perform better than the baseline model. In particular, when LSTM architectures are used for a single interval prediction, that is, the window sizes are (3, 1), (5, 1), (8, 1), (12, 1) or (24, 1), the results obtained are abysmal.

Finally, the Autoregressive model is the model that generates the best results and the most regular to the change of configuration of window sizes and there is little difference between the models that predict less (3, 1) versus the models that cover more intervals to predict ((48, 12) or (48, 24)) being this the model that obtains better results in general.

Another aspect to mention and which is common to most models is that as the number of intervals to be predicted increases, the error also increases slightly to a degree that depends on the model being evaluated. This increase is so small that it could be concluded that the results obtained for all the windows are the same for all the configurations. Therefore, it would be the same to use a window size (8, 3) as a window (36, 12); the metrics are practically the same.

7.2 Acquired knowledge

During the development of this work, I have been able to acquire different knowledge. Before starting the project, my main personal objective with this project was to be able to understand the mathematics behind a neural network and how the Backpropagation algorithm works on a practical level. Without a doubt, I have not only been able to learn these concepts but also to learn how to use different tools and libraries to work with neural networks.

It is true that it is not necessary to understand the functioning of a neural network in order to use one, but it is also true that limited knowledge does not lead to the best results. For that reason, I decided to embark on this project and I sincerely believe that I have exceeded my expectations and have learned some fascinating new branches of computer science that I hope in the future I will be able to continue studying and using in practice.

Obviously, wanting to learn all the basics about neural networks doesn't mean you want to develop the whole practical side again when libraries like Keras or Tensorflow already offer simplified use of them. That's why, during the development of this project, I had to research some of these libraries and I was pleasantly surprised to find the amount of facilities they offer despite knowing only a small part of them. Along with my development and learning within the AI, I will continue to use and improve my skills with these tools or even investigate other similar ones such as PyTorch.

Finally, I would like to mention that the development of this document has been carried out entirely with LaTeXand has been a great decision due to the facilities it provides, since the user only has to worry about capturing information. Trivial tasks such as index control, page numbering or bibliography management are completely alien to the user.

On a personal level, this project has helped me to lay the foundations of what research and development activity is all about, with you being the owner of your time and the work to be done, and not following a statement and showing the results in a specific way. I have managed to resolve doubts that were raised before the project was carried out and now I understand more about the work and innovations that are published every day. At the same time many more doubts have arisen and I hope to continue resolving these doubts in the future by continuing this research work.

7.3 Conclusions regarding AI

Without doubt, the results obtained are generally much better than the baseline model and show that the work done has not been in vain. It is understandable that neural networks have become the most popular family of Machine Learning algorithms in the last decade. They have a great potential to solve different kinds of problems with good results. But neural networks in turn have a cost and that is that the human being is not able to interpret how the weights of the neural networks are adjusted in order to study their behaviour, why decisions and values are calculated and optimise the models even further. This deficiency can be important depending on which problem is being solved. In the project presented in this thesis, these adjustments are indifferent, since we do not want to interpret how the model has been adjusted for any reason: it simply works.

But there are other problems that may need to be addressed in order to achieve the desired results. A clear example is the accident that occurred in 2018 [54] due to both a human error and an error in the neural network driving the car. Although the final respon-

sibility was that of the driver who, at the time of the accident, was paying attention to the mobile phone instead of the road and was the person most responsible and whose main task was to supervise the decisions of the autonomous vehicle, it shows that this type of accident will continue to occur in the future, a future in which there may be no steering wheels and the user may not be able to interact with the control of the vehicle directly [55]. It is at this point that certain legal doubts arise as to how to act and what judicial decision to take in the event of an accident.

This question is closely related to another concept that perhaps much of society assumes, and that is that AI has the effect of destroying many jobs in their entirety. And this is not entirely true. Perhaps many processes will be replaced by this type of algorithm but there will always be someone in charge of supervising these decisions and there will always be someone responsible for the results who has obtained an AI. So, at the same time as jobs are destroyed, others emerge but in smaller numbers.

According to some experts the number of jobs that will be destroyed by the rise of Artificial Intelligence is 30% [56] and the creation of new jobs will only be 8%, this technological revolution is unprecedented and new political and legal measures are urgently needed for its adoption. Furthermore, is society still ready and willing to integrate this type of technology? AI offers a great number of tools that will help humanity in the development of its activities and in the discovery of new inventions, concepts and scientific advances, and the great challenge is therefore not only its implementation but also in the education of people so that they can live with it on a daily basis and understand that AI is not an enemy but an ally.

But the potential of Machine Learning and Artificial Intelligence lies not only in its technologies but also in its users. If we trust (in essence) the way societies are currently managed, there is no reason not to trust ourselves to do good with these technologies. And if we can suspend presentiment and accept that history warns us not to play God with powerful technologies, but that they are merely instructive, then we are likely to be freed from unnecessary anxiety about their use.

Artificial Intelligence and Machine Learning are products of both science and myth. The idea that machines could think and perform tasks just like humans is thousands of years old. Bringing cognitive systems into machines is not new either, or together with the ability of models to outperform humans in specific fields, we are facing a new technological revolution.

Most of the AI scenarios of the future are hypothetical, but AI raises existential questions for us. It shows that where science stops, philosophy and spirituality begin again, giving the importance that the fields of the humanities once had in carrying out thought and conclusions that are difficult to reproduce by any artificial system.

8 FUTURE WORK

Much of this project has been in the theoretical study of the concept of neural networks while the practical application of what has been learned has been the other part of the time dedicated. In this practical part, mainly, time has been invested in learning about the libraries used in the industry such as Tensorflow and Pandas, as well as in understanding the problem and studying different models and their behaviour. With this work it has therefore been demonstrated that the best model that has worked has been the AR one.

As part of the future work, it is proposed to improve this model to minimise all possible metrics and propose a model that obtains similar or better results than the current state of the art. This could be done by further studying both the network parameters and the learning rate, the optimiser or the error. But, adding more information to this could be more useful, since the more data, the more accurate the model will be. For example, you could use variables such as whether it is a holiday or the weather conditions. And this is one of the main differences between the data set in this project and others. The model had only three variables about the temporal information: `hour`, `day_of_month`and `month`. Therefore, there is no doubt that by adding these variables that provide information about the patterns of bicycle rentals, the model would improve considerably.

Another aspect of this project to be improved would be the exploitation of the results in some kind of tool useful to the company or to the users. The trained models have simply been used to compute the metrics and then discarded as there were no plans to use them in production. Conversely, if there had been a project using the models, they could have been kept on file and used in different tools. For example, a web application that tells the user the forecast of bicycles available for a certain time or software that allows logistics managers to relocate bicycles more efficiently. These and other applications could use as a main module some neural network model explained in this project.

On the other hand, more variables could be used containing important information such as the work calendar or variables related to meteorology, which influence the behaviour patterns of the bicycle network.

For example, on a non-working Monday, the number of bicycles rented in a park will be greater than any other Monday in another week. Or, if one day the weather conditions are not favourable for cycling, the number of bicycles rented will be considerably reduced. These types of variables are some examples that could be added to the dataset to improve the accuracy of the model but for lack of time and for simplicity they have not been added. In addition, the results obtained have been considered to be quite good in themselves and therefore it has not been necessary to invest time in this section although it would be a good study how these models could be improved with such changes.

Finally, it would be interesting to study if it is feasible to use this project but using other datasets and thus prove its portability.

9 REFERENCES

[1] M. of National Development and H. VÁTI Nonprofit Ltd., "The territorial state and perspectives of the european union", 2011. [Online]. Available: https://ec.europa.eu/regional_policy/sources/policy/what/territorial-cohesion/territorial_state_and_perspective_2011.pdf.

[2] A. Infanzon, "Descifrando enigma ayer, analítica y big data hoy", 2015. [Online]. Available: https://www.forbes.com.mx/descifrando-enigma-ayer-analitica-y-big-data-hoy/.

[3] P. Das, K. Acharjee, P. Das, and V. Prasad, "Voice recognition system: Speech-to-text", *Journal of Applied and Fundamental Sciences*, vol. 1, pp. 2395–5562, Nov. 2015.

[4] N. Li, S. Liu, Y. Liu, S. Zhao, and M. Liu, "Neural speech synthesis with transformer network", *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 6706–6713, Jul. 2019. DOI: 10.1609/aaai.v33i01.33016706.

[5] A. Gautam and S. Mohan, "A review of research in multi-robot systems", Aug. 2012. DOI: 10.1109/ICIInfS.2012.6304778.

[6] W. Zhang, T. Mei, H. Liang, B. Li, J. Huang, Z. Xu, Y. Ding, and W. Liu, "Research and development of automatic driving system for intelligent vehicles", *Advances in Intelligent Systems and Computing*, vol. 215, pp. 675–684, Sep. 2014. DOI: 10.1007/978-3-642-37835-5_58.

[7] Y. Jing, Y. Yang, Z. Feng, J. Ye, Y. Yu, and M. Song, *Neural style transfer: A review*, May 2017.

[8] J. Haugeland, "Artificial intelligence: The very idea (mit press, cambridge, ma, 1985); 287 pp.", *Artificial Intelligence*, vol. 29, pp. 349–353, Sep. 1986.

[9] R. Bellman, *An Introduction to Artificial Intelligence: Can Computers Think?* Boyd Fraser Publishing Company, 1978.

[10] E. Charniak and D. McDermott, *Introduction to Artificial Intelligence*. Addison Wesley, 1985.

[11] P. Winston, *Artificial Intelligence*. Addison-Wesley Publishing Company, 1976.

[12] R. Kurzweil, *The Age of Intelligent Machines*. MIT Press, 1990.

[13] E. Rich and K. Knight, *Artificial Intelligence*. 1991.

[14] N. J. Nilsson, *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998, ISBN: 978-1-55860-467-4.

[15] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[16] S. Harnad, "The turing test is not a trick: Turing indistinguishability is a scientific criterion", *SIGART Bull.*, vol. 3, no. 4, pp. 9–10, Oct. 1992, ISSN: 0163-5719. DOI: 10.1145/141420.141422. [Online]. Available: https://doi.org/10.1145/141420.141422.

[17] D. Poole, A. Mackworth, and R. Goebel, *Computational Intelligence: A Logical Approach*. Jan. 1998, ISBN: 978-0-19-510270-3.

[18] I. Salian, "Supervize me: What's the difference between supervised, unsupervised, semi-supervised and reinforcement learning?", 2018. [Online]. Available: https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/#:~:text=In%5C%20a%5C%20supervised%5C%20learning%5C%20model,and%5C%20patterns%5C%20on%5C%20its%5C%20own..

[19] P. Flach, *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge, 2012.

[20] R. Logan and M. Tandoc, "Thinking in patterns and the pattern of human thought as contrasted with ai data processing", *Information*, vol. 9, Apr. 2018. DOI: 10.3390/info9040083.

[21] T. Kühne, "What is a model?", Jan. 2004.

[22] H. Kinsley and D. Kukiela, "Neural network from scratch in python", 2020.

[23] S. Ramón y Cajal, "Sobre las fibras nerviosas de la capa molecular del cerebro", *Histología Normal y Patológica*, Aug. 1888.

[24] H. Dale and O. Loewi, "The chemical transmission of nerve action", 1936. DOI: https://www.nobelprize.org/prizes/medicine/1936/loewi/lecture/.

[25] D. R. y De Robertis (h), *Biologia celular y molecular*. El Ateneo, 1975.

[26] C. R. Noback and R. J.Demarest, *The human nervous system: Basic principles of neurobiology*, 10th ed. Mc Graw Hill, 1981.

[27] J. Fuster, "Cortex and memory: Emergence of a new paradigm", *Journal of Cognitive Neuroscience*, vol. 21, Jul. 2009. DOI: 10.1162/jocn.2009.21280.

[28] S. C. Kleene, "Representation of events in nerve nets and finite automata", Dec. 1951.

[29] M. A. Nielsen, "Neural networks and deep learning", 2015.

[30] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: http://tensorflow.org/.

[31] C. Pascual, "Tutorial: Understanding Regression Error Metrics in Python", 2018. [Online]. Available: https://www.dataquest.io/blog/understanding-regression-error-metrics/.

[32] G. Viswanathan, "Huber Error", 2020. [Online]. Available: https://medium.com/@gobiviswaml/huber-error-loss-functions-3f2ac015cd45.

[33] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[34] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-propagating Errors", *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. DOI: 10.1038/323533a0. [Online]. Available: http://www.nature.com/articles/323533a0.

[35] J. Kiefer and J. Wolfowitz, "Stochastic estimation of the maximum of a regression function", *Annals of Mathematical Statistics*, vol. 23, pp. 462–466, 1952.

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *CoRR*, vol. abs/1412.6980, 2015.

[37] M. D. Zeiler, "Adadelta: An adaptive learning rate method", *ArXiv*, vol. abs/1212.5701, 2012.

[38] J. C. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, 2010.

[39] N. Boufidis, A. Nikiforiadis, K. Chrysostomou, and G. Aifadopoulou, "Development of a station-level demand prediction and visualization tool to support bike-sharing systems' operators", *Transportation Research Procedia*, vol. 47, pp. 51–58, 2020, 22nd EURO Working Group on Transportation Meeting, EWGT 2019, 18th – 20th September 2019, Barcelona, Spain, ISSN: 2352-1465. DOI: https://doi.org/10.1016/j.trpro.2020.03.072. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352146520302593.

[40] L. Lin, Z. He, and S. Peeta, "Predicting station-level hourly demand in a large-scale bike-sharing network: A graph convolutional neural network approach", *Transportation Research Part C: Emerging Technologies*, vol. 97, pp. 258–276, 2018, ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2018.10.011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0968090X18300974.

[41] Kaggle, *Shared bikes demand forecasting competition*, 2020. [Online]. Available: https://www.kaggle.com/c/shared-bikes-demand-forecasting/.

[42] *Divvy trip history data*, 2020. [Online]. Available: https://divvy-tripdata.s3.amazonaws.com/index.html.

[43] *Divvy station map | chicago city | data portal*, 2020. [Online]. Available: https://data.cityofchicago.org/Transportation/Divvy-station-map/89n3-p56s.

[44] *Weather history download chicago*, 2020. [Online]. Available: https://www.meteoblue.com/en/weather/archive/export/chicago_united-states-of-america_4887398.

[45] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009, ISBN: 1441412697.

[46] F. Chollet *et al.* (2015). "Keras", [Online]. Available: https://github.com/fchollet/keras.

[47] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy", *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[48] T. pandas development team, *Pandas-dev/pandas: Pandas*, version latest, Feb. 2020. DOI: 10.5281/zenodo.3509134. [Online]. Available: https://doi.org/10.5281/zenodo.3509134.

[49] J. D. Hunter, "Matplotlib: A 2d graphics environment", *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[50] J. D. Garrett, "SciencePlots (v1.0.6)", Feb. 2021. DOI: 10.5281/zenodo.4106650. [Online]. Available: http://doi.org/10.5281/zenodo.4106650.

[51] R. Matthew Neil;Stones, *The Linux Environment*. Indiana, US, 2008, ISBN: 978-0-470-14762-7.

[52] Reddit, *How do you represent time-of-day in artificial neural networks?*, 2014. [Online]. Available: https://www.reddit.com/r/MachineLearning/comments/1utxnk/how_do_you_represent_timeofday_in_artificial/.

[53] *Time series forecasting*, 2020. [Online]. Available: https://www.tensorflow.org/tutorials/structured_data/time_series.

[54] T. GRIGGS and D. WAKABAYASHI, "How a self-driving uber killed a pedestrian in arizona", *The New York Times*, Mar. 2018. [Online]. Available: https://www.nytimes.com/interactive/2018/03/20/us/self-driving-uber-pedestrian-killed.html.

[55] A. J. Hawkins, "Gm will make an autonomous car without steering wheel or pedals by 2019", *The Verge*, Jan. 2018. [Online]. Available: https://www.theverge.com/2018/1/12/16880978/gm-autonomous-car-2019-detroit-auto-show-2018.

[56] "Jobs lost, jobs gained: What the future of work will mean for jobs, skills, and wages", *McKinsey Global Institute*, Jul. 2020. [Online]. Available: https://www.mckinsey.com/featured-insights/future-of-work/jobs-lost-jobs-gained-what-the-future-of-work-will-mean-for-jobs-skills-and-wages#.

A MATHEMATICAL VARIABLES

Symbol	Description
L_i	Neural Network Layer i
$L_{i,j}$	Neuron j of the layer i of the neural network
x	Model Input Vector
x'	$[[1] \oplus x]^T$
b	Bias of a neuron
$b^{L_{i,j}}$	Neuron Bias j in the layer i
w	Weight vector
$w*$	New weight vector after calculation with ∇f
$w^{L_{i,j}}$	Weight vector for the neuron j of the layer i
W^{L_i}	matrix of the i layer also formed by the b values
$W*$	New weight matrix after calculation with ∇f
z	Dot product between w and y or w and x
z^{L_i}	Dot product between w^{L_i} and yL_{i-1} or w^{L_1} and x
$a()$	Activation function
$a()^{L_i}$	Activation function for the layer i
$a'()$	Activation function derivative
$a'()^{L_i}$	Activation function derivative for the i layer
y	Model Output Vector
y^{L_i}	Output vector for the layer L_i
$\hat{y}$	Real Vector
$\hat{y}_i$	Actual vector that is in the i position of the dataset
$c()$	Cost function
$c'()$	Cost function derivative
∇f	Gradient vector
∇f_b	Gradient vector value for b
∇f_w	Subvector of the gradient vector for w
∂	Partial derivative
$\frac{\partial a}{\partial b}$	Partial derivative of a with respect to b
η	Learning Rate
$\oplus$	Concatenation of vectors

B SAMPLE OF THE ORIGINAL DIVVY DATASET

start_time	end_time	tripduration	from_station_id	from_station_name	to_station_id	to_station_name	gender	birthyear
2019-09-23 16:48:09	2019-09-23 17:01:57	828.0	44	State St & Randolph St	21	Aberdeen St & Jackson Blvd	NaN	NaN
2019-08-19 18:18:41	2019-08-19 18:29:20	639.0	287	Franklin St & Monroe St	110	Dearborn St & Erie St	Male	1990.0
2019-01-08 08:42:22	2019-01-08 08:53:58	696.0	66	Clinton St & Lake St	161	Rush St & Superior St	Male	1971.0
2019-05-18 13:27:52	2019-05-18 13:53:41	1,549.0	94	Clark St & Armitage Ave	35	Streeter Dr & Grand Ave	Male	1988.0
2019-03-01 07:04:37	2019-03-01 07:12:09	452.0	174	Canal St & Madison St	48	Larrabee St & Kingsbury St	Male	1966.0
2019-03-13 18:13:48	2019-03-13 18:17:25	217.0	210	Ashland Ave & Division St	130	Damen Ave & Division St	Male	1995.0
2019-08-11 20:05:13	2019-08-11 20:14:25	552.0	93	Sheffield Ave & Willow St	143	Sedgwick St & Webster Ave	Female	1995.0
2019-01-24 16:50:10	2019-01-24 17:21:20	1,870.0	525	Glenwood Ave & Touhy Ave	243	Lincoln Ave & Sunnyside Ave	Female	1986.0
2019-07-27 13:19:12	2019-07-27 13:33:34	862.0	72	Wabash Ave & 16th St	197	Michigan Ave & Madison St	Male	1995.0
2019-07-31 21:52:44	2019-07-31 22:22:41	1,796.0	52	Michigan Ave & Lake St	58	Marshfield Ave & Cortland St	NaN	NaN
2019-05-14 10:38:31	2019-05-14 10:44:45	374.0	236	Sedgwick St & Schiller St	48	Larrabee St & Kingsbury St	Male	1986.0
2019-06-16 19:25:32	2019-06-16 19:40:59	927.0	85	Michigan Ave & Oak St	224	Halsted St & Willow St	Male	1988.0
2019-06-10 12:18:33	2019-06-10 12:24:33	360.0	38	Clark St & Lake St	197	Michigan Ave & Madison St	Male	1993.0
2019-12-12 09:55:38	2019-12-12 10:21:01	1,522.0	296	Broadway & Belmont Ave	69	Damen Ave & Pierce Ave	NaN	NaN
2019-09-09 09:07:44	2019-09-09 09:25:01	1,036.0	26	McClurg Ct & Illinois St	620	Orleans St & Chestnut St (NEXT Apts)	Female	1992.0
2019-07-04 09:15:21	2019-07-04 10:26:40	4,279.0	99	Lake Shore Dr & Ohio St	6	Dusable Harbor	Female	1992.0
2019-11-26 08:21:39	2019-11-26 08:34:49	790.0	376	Artesian Ave & Hubbard St	217	Elizabeth (May) St & Fulton St	Female	1980.0
2019-07-22 08:09:05	2019-07-22 08:16:59	473.0	138	Clybourn Ave & Division St	133	Kingsbury St & Kinzie St	Male	1984.0
2019-05-10 18:43:10	2019-05-10 18:51:40	510.0	99	Lake Shore Dr & Ohio St	180	Ritchie Ct & Banks St	NaN	NaN
2019-04-05 09:38:21	2019-04-05 09:48:30	609.0	46	Wells St & Walton St	287	Franklin St & Monroe St	Male	1992.0
2019-07-22 16:50:47	2019-07-22 17:01:03	616.0	91	Clinton St & Washington Blvd	19	Throop (Loomis) St & Taylor St	Male	1997.0
2019-06-03 21:09:36	2019-06-03 21:41:00	1,884.0	276	California Ave & North Ave	316	Damen Ave & Sunnyside Ave	Male	1976.0
2019-08-01 19:20:36	2019-08-01 19:40:00	1,164.0	130	Damen Ave & Division St	124	Damen Ave & Cullerton St	Female	1990.0
2019-07-17 16:24:52	2019-07-17 16:39:45	893.0	157	Lake Shore Dr & Wellington Ave	268	Lake Shore Dr & North Blvd	Male	1997.0
2019-10-09 15:58:44	2019-10-09 16:09:00	616.0	129	Blue Island Ave & 18th St	281	Western Ave & 24th St	Male	1988.0
2019-04-08 11:34:37	2019-04-08 11:46:25	708.0	25	Michigan Ave & Pearson St	196	Cityfront Plaza Dr & Pioneer Ct	Male	1965.0
2019-06-11 17:40:58	2019-06-11 18:13:44	1,966.0	43	Michigan Ave & Washington St	254	Pine Grove Ave & Irving Park Rd	NaN	NaN
2019-05-25 05:47:49	2019-05-25 06:04:43	1,014.0	66	Clinton St & Lake St	97	Field Museum	Male	1976.0
2019-08-18 11:29:24	2019-08-18 11:51:34	1,330.0	199	Wabash Ave & Grand Ave	161	Rush St & Superior St	Female	1964.0
2019-09-10 08:31:13	2019-09-10 08:37:56	403.0	85	Michigan Ave & Oak St	194	Wabash Ave & Wacker Pl	Male	1993.0
2019-10-25 15:59:12	2019-10-25 16:07:03	471.0	211	St. Clair St & Erie St	636	Orleans St & Hubbard St	Female	1981.0
2019-08-13 16:01:17	2019-08-13 16:05:56	279.0	121	Blackstone Ave & Hyde Park Blvd	418	Ellis Ave & 53rd St	Male	1966.0
2019-07-08 20:05:44	2019-07-08 20:13:38	473.0	156	Clark St & Wellington Ave	131	Lincoln Ave & Belmont Ave	Male	1984.0
2019-08-11 13:15:22	2019-08-11 13:18:36	194.0	94	Clark St & Armitage Ave	289	Wells St & Concord Ln	Male	1990.0
2019-03-12 13:50:21	2019-03-12 13:54:03	222.0	32	Racine Ave & Congress Pkwy	22	May St & Taylor St	Male	1993.0
2019-12-07 08:29:06	2019-12-07 08:46:15	1,028.0	141	Clark St & Lincoln Ave	347	Ashland Ave & Grace St	Male	1991.0
2019-10-03 18:58:44	2019-10-03 19:22:45	1,440.0	365	Halsted St & North Branch St	117	Wilton Ave & Belmont Ave	Male	1987.0
2019-12-29 10:48:05	2019-12-29 10:58:48	642.0	299	Halsted St & Roscoe St	220	Clark St & Drummond Pl	Male	1983.0
2019-07-07 00:04:09	2019-07-07 00:24:27	1,218.0	291	Wells St & Evergreen Ave	7	Field Blvd & South Water St	Male	1989.0
2019-11-06 08:27:59	2019-11-06 08:33:24	325.0	190	Southport Ave & Wrightwood Ave	67	Sheffield Ave & Fullerton Ave	Female	1993.0
2019-04-13 19:10:09	2019-04-13 19:16:08	359.0	627	LaSalle Dr & Huron St (*)	23	Orleans St & Elm St (*)	Female	1976.0
2019-09-11 07:52:42	2019-09-11 08:05:43	781.0	331	Halsted St & Clybourn Ave	48	Larrabee St & Kingsbury St	Male	1979.0
2019-07-10 18:28:05	2019-07-10 18:57:29	1,764.0	268	Lake Shore Dr & North Blvd	29	Noble St & Milwaukee Ave	Male	1996.0
2019-08-09 10:48:54	2019-08-09 12:06:15	4,640.0	623	Michigan Ave & 8th St	268	Lake Shore Dr & North Blvd	NaN	NaN
2019-08-08 05:39:47	2019-08-08 05:45:06	319.0	283	LaSalle St & Jackson Blvd	47	State St & Kinzie St	Male	1975.0
2019-06-07 17:48:11	2019-06-07 18:04:44	993.0	144	Larrabee St & Webster Ave	230	Lincoln Ave & Roscoe St	Male	1989.0
2019-07-30 07:51:29	2019-07-30 08:06:40	910.0	13	Wilton Ave & Diversey Pkwy	359	Larrabee St & Division St	Female	1991.0
2019-11-29 14:14:22	2019-12-03 20:45:36	369,074.0	343	Racine Ave & Wrightwood Ave	67	Sheffield Ave & Fullerton Ave	Female	1960.0
2019-11-22 09:42:12	2019-11-22 10:16:53	2,080.0	131	Lincoln Ave & Belmont Ave	283	LaSalle St & Jackson Blvd	Male	1976.0
2019-07-25 16:56:04	2019-07-25 16:58:51	166.0	36	Franklin St & Jackson Blvd	68	Clinton St & Tilden St	Male	1968.0
2019-04-13 09:13:59	2019-04-13 09:16:34	155.0	229	Southport Ave & Roscoe St	227	Southport Ave & Waveland Ave	Female	1984.0
2019-08-25 11:03:24	2019-08-25 11:37:46	2,062.0	90	Millennium Park	35	Streeter Dr & Grand Ave	NaN	NaN
2019-04-05 16:46:32	2019-04-05 17:00:32	840.0	68	Clinton St & Tilden St	138	Clybourn Ave & Division St	Male	1990.0
2019-11-24 20:03:18	2019-11-24 20:08:39	320.0	419	Lake Park Ave & 53rd St	322	Kimbark Ave & 53rd St	Male	1995.0
2019-06-20 18:16:05	2019-06-20 18:23:50	465.0	287	Franklin St & Monroe St	90	Millennium Park	Male	1981.0
2019-06-14 16:29:10	2019-06-14 16:40:44	694.0	182	Wells St & Elm St	329	Lake Shore Dr & Diversey Pkwy	Male	1992.0
2019-04-26 14:36:59	2019-04-26 15:31:05	3,246.0	577	Stony Island Ave & South Chicago Ave	11	Jeffery Blvd & 71st St	Female	1992.0
2019-06-10 17:36:08	2019-06-10 17:50:58	890.0	283	LaSalle St & Jackson Blvd	182	Wells St & Elm St	Male	1996.0
2019-08-18 14:41:50	2019-08-18 14:51:00	549.0	330	Lincoln Ave & Addison St	166	Ashland Ave & Wrightwood Ave	Male	1988.0
2019-05-03 06:07:25	2019-05-03 06:16:04	519.0	23	Orleans St & Elm St (*)	164	Franklin St & Lake St	Male	1977.0

Table 8: Random samples from the original Divvy dataset

[1]The columns are omitted: *bikeid*, *tripid* y *usertype*

C SAMPLE OF THE INTERVAL DATASET

start_time	hour	day_of_week	month	quantity_1	quantity_2	quantity_3	quantity_4	quantity_631	quantity_632
2019-09-22 03:00:00	3	5	9	0.0	0.0	0.0	0.0	0.0	0.0
2017-01-10 00:00:00	0	2	1	0.0	0.0	1.0	0.0	0.0	0.0
2019-12-01 01:00:00	1	5	12	1.0	0.0	0.0	0.0	0.0	0.0
2018-01-02 17:00:00	17	1	1	0.0	0.0	0.0	0.0	0.0	0.0
2018-11-01 14:00:00	14	3	11	0.0	0.0	4.0	1.0	0.0	0.0
2017-01-23 03:00:00	3	1	1	0.0	0.0	0.0	0.0	1.0	0.0
2018-11-12 16:00:00	16	7	11	0.0	0.0	0.0	0.0	0.0	1.0
2019-12-25 12:00:00	12	1	12	0.0	0.0	0.0	0.0	0.0	0.0
2019-02-01 00:00:00	0	3	2	0.0	0.0	0.0	0.0	0.0	0.0
2017-03-04 08:00:00	8	6	3	0.0	0.0	0.0	1.0	0.0	0.0
2018-03-25 20:00:00	20	6	3	1.0	0.0	0.0	0.0	1.0	2.0
2019-06-08 07:00:00	7	4	6	3.0	4.0	0.0	4.0	0.0	0.0
2017-09-30 19:00:00	19	6	9	2.0	8.0	2.0	1.0	2.0	0.0
2018-08-22 21:00:00	21	2	8	0.0	0.0	2.0	1.0	3.0	0.0
2017-02-22 08:00:00	8	3	2	0.0	0.0	0.0	0.0	0.0	1.0
2017-07-05 18:00:00	18	3	7	1.0	3.0	17.0	10.0	1.0	3.0
2019-07-21 11:00:00	11	5	7	0.0	1.0	5.0	5.0	0.0	0.0
2019-07-27 00:00:00	0	4	7	1.0	0.0	0.0	0.0	0.0	0.0
2018-12-09 21:00:00	21	6	12	0.0	0.0	0.0	0.0	0.0	0.0
2018-04-16 06:00:00	6	7	4	1.0	0.0	0.0	0.0	0.0	0.0
2019-02-27 13:00:00	13	1	2	0.0	1.0	3.0	0.0	0.0	0.0
2019-10-25 02:00:00	2	3	10	0.0	0.0	0.0	0.0	0.0	0.0
2017-12-26 18:00:00	18	2	12	0.0	0.0	0.0	0.0	0.0	0.0
2017-10-29 07:00:00	7	7	10	2.0	0.0	0.0	0.0	0.0	1.0
2018-01-20 11:00:00	11	5	1	0.0	0.0	1.0	1.0	2.0	0.0
2018-06-08 03:00:00	3	4	6	0.0	0.0	0.0	0.0	0.0	0.0
2019-10-29 06:00:00	6	7	10	0.0	0.0	0.0	0.0	0.0	0.0
2019-11-25 11:00:00	11	6	11	3.0	0.0	2.0	8.0	0.0	0.0
2017-06-06 09:00:00	9	2	6	1.0	3.0	1.0	10.0	2.0	2.0
2018-03-16 22:00:00	22	4	3	0.0	3.0	0.0	0.0	0.0	0.0
2019-11-06 10:00:00	10	1	11	0.0	4.0	0.0	0.0	0.0	0.0
2017-06-11 17:00:00	17	7	6	1.0	11.0	21.0	10.0	4.0	3.0
2018-07-21 18:00:00	18	5	7	0.0	0.0	3.0	2.0	0.0	0.0
2019-05-25 11:00:00	11	4	5	0.0	0.0	6.0	0.0	1.0	1.0
2018-03-02 01:00:00	1	4	3	5.0	0.0	0.0	0.0	0.0	0.0
2018-12-21 06:00:00	6	4	12	0.0	0.0	0.0	0.0	0.0	0.0
2019-01-07 06:00:00	6	6	1	0.0	0.0	0.0	0.0	0.0	0.0
2018-05-04 13:00:00	13	4	5	1.0	0.0	5.0	2.0	3.0	3.0
2019-01-25 09:00:00	9	3	1	0.0	0.0	0.0	0.0	0.0	0.0
2019-11-24 19:00:00	19	5	11	0.0	0.0	0.0	0.0	0.0	0.0
2018-10-18 16:00:00	16	3	10	1.0	2.0	14.0	5.0	0.0	0.0
2018-07-14 23:00:00	23	5	7	1.0	1.0	6.0	0.0	0.0	0.0
2017-08-10 02:00:00	2	4	8	0.0	0.0	0.0	0.0	0.0	1.0
2017-04-13 00:00:00	0	4	4	0.0	0.0	0.0	0.0	3.0	0.0
2017-03-23 20:00:00	20	4	3	0.0	0.0	0.0	0.0	0.0	0.0
2018-09-04 16:00:00	16	1	9	3.0	21.0	20.0	10.0	2.0	2.0
2019-04-11 09:00:00	9	2	4	0.0	0.0	0.0	1.0	0.0	0.0
2019-05-25 01:00:00	1	4	5	0.0	0.0	0.0	0.0	0.0	0.0
2018-01-03 11:00:00	11	2	1	0.0	1.0	1.0	0.0	4.0	0.0
2018-04-19 00:00:00	0	3	4	4.0	0.0	0.0	0.0	0.0	0.0
2019-09-27 16:00:00	16	3	9	0.0	6.0	14.0	7.0	0.0	2.0
2017-02-16 01:00:00	1	4	2	0.0	0.0	0.0	0.0	0.0	0.0
2017-05-09 19:00:00	19	2	5	0.0	2.0	1.0	3.0	0.0	1.0
2019-11-26 01:00:00	1	7	11	1.0	0.0	0.0	0.0	1.0	0.0
2018-08-12 05:00:00	5	6	8	0.0	0.0	0.0	0.0	0.0	0.0

[2]

Table 9: Random samples from the dataset divided by intervals

[2]All stations whose id is between 5 and 631 are omitted

D WINDOWGENERATOR CLASS CODE

```
import tensorflow as tf
import numpy as np


class WindowGenerator():
    def __init__(self, input_width, label_width, shift,
                 train_df, val_df, test_df=None,
                 label_columns_index=-3):

        # Store the datasets
        self.train_df = train_df
        self.val_df = val_df
        self.test_df = test_df

        # Gets the indexes of the label column.
        self.label_columns_index = label_columns_index
        self.label_columns = train_df.columns[:label_columns_index]
                                      .tolist()
        self.label_columns_indices = {name: i for i, name in
                                      enumerate(self.label_columns)}

        # Dict of name of column as key and index as value
        self.column_indices = {name: i for i, name in
                               enumerate(train_df.columns)}

        # Work out the window parameters.
        self.input_width = input_width
        self.label_width = label_width
        self.shift = shift

        # Handle the indexes and offsets as shown in the diagrams
        # above.
        self.total_window_size = input_width + shift

        self.input_slice = slice(0, input_width)
        self.input_indices = np.arange(self.total_window_size)[
            self.input_slice]

        self.label_start = self.total_window_size - self.label_width
        self.labels_slice = slice(self.label_start, None)
        self.label_indices = np.arange(self.total_window_size)[
            self.labels_slice]

    def split_window(self, features):
        inputs = features[:, self.input_slice, :]
        labels = features[:, self.labels_slice, :]

        if self.label_columns is not None:
            labels = tf.stack(
                [labels[:, :, self.column_indices[name]]
                    for name in self.label_columns],
                axis=-1)

        # Slicing doesn't preserve static shape information, so set
        # the shapes manually. This way the `tf.data.Datasets` are
        # easier to inspect.
        inputs.set_shape([None, self.input_width, None])
        labels.set_shape([None, self.label_width, None])
        return inputs, labels

    def make_dataset(self, data):
        if data is None:
            print("Data is None")
            return
```

```
    data = np.array(data, dtype=np.float32)
    ds = tf.keras.preprocessing.timeseries_dataset_from_array(
        data=data,
        targets=None,
        sequence_length=self.total_window_size,
        sequence_stride=1,
        shuffle=True,
        batch_size=32,)

    ds = ds.map(self.split_window)

    return ds

@property
def train(self):
  return self.make_dataset(self.train_df)

@property
def val(self):
  return self.make_dataset(self.val_df)

@property
def test(self):
  return self.make_dataset(self.test_df)
```

3

[3] Code based on the official Tensorflow tutorial *"Time series forecasting"* [30]

E MODEL CODE

```
import tensorflow as tf
from tensorflow.keras.layers import LSTMCell, RNN, Dense

class AutoRegressive(tf.keras.Model):
    def __init__(self, lstm_units, steps, stations):
        super().__init__()
        self.steps = steps
        self.stations = stations
        self.lstm_units = lstm_units

        # First LSTM layer
        self.lstm_cell = LSTMCell(lstm_units)
        # Also wrap the LSTMCell in an RNN to simplify the `warmup` method.
        self.lstm_rnn = RNN(self.lstm_cell, return_state=True)

        # Ouput layer
        self.dense = Dense(stations, activation="relu")

    def warmup(self, inputs):
        # inputs.shape => (batch, time, features)
        # x.shape => (batch, lstm_units)
        x, *state = self.lstm_rnn(inputs)

        # predictions.shape => (batch, features)
        prediction = self.dense(x)
        return prediction, state

    def call(self, inputs, training=None):
        # Use a TensorArray to capture dynamically unrolled outputs.
        predictions = []

        # Initialize the lstm state
        prediction, state = self.warmup(inputs)

        # Insert the first prediction
        predictions.append(prediction)

        # Run the rest of the prediction steps
        for n in range(1, self.steps):
            # Use the last prediction as input.
            x = prediction

            # Concat with timestamp variables
            x = tf.concat([x, inputs[:,-1,-3:]], 1)

            # Execute one lstm step.
            x, state = self.lstm_cell(x, states=state, training=training)

            # Convert the lstm output to a prediction.
            prediction = self.dense(x)

            # Add the prediction to the output
            predictions.append(prediction)

        # predictions.shape => (time, batch, features)
        predictions = tf.stack(predictions)

        # predictions.shape => (batch, time, features)
        predictions = tf.transpose(predictions, [1, 0, 2])

        return predictions
```

4

[4]Code based on the official Tensorflow tutorial *"Time series forecasting"* [30]

F CODE USED TO COMPILE, TRAIN AND EVALUATE MODELS

```
import tensorflow as tf
import pandas as pd

from tf.keras.callbacks import EarlyStopping
from livelossplot import PlotLossesKeras



'''
Compiles and train model

@param model which will be training
@param window with the Datasets used
@param lr (learning rate) of the model
@param max_epoch for training
@param should_stop if detects overfitting the stops with a patience of 10
'''
def compile_and_fit(model, window, lr, max_epochs=150, should_stop=False):
    model.compile(
        loss=tf.losses.Huber(),
        optimizer=tf.optimizers.Adam(lr=lr),
        metrics=[MeanSquaredLogarithmicError(), MeanSquaredError(), MeanAbsoluteError(),
                RootMeanSquaredError(), RootMeanSquaredLogarithmicError])

    # Define the callbacks for the training proccess
    # PlotLossesKeras plots training and validation for every epoch in real time
    # early_stopping allows to stop if detects overfitting
    cbs = [PlotLossesKeras()] + ([EarlyStopping(monitor='val_loss',
                                                patience=10,
                                                mode='min')] if should_stop else [])

    model.fit(window.train, epochs=max_epochs,
                        validation_data=window.val,
                        callbacks=cbs,
                        verbose=2)

'''
Computes metrics and returns it as DataFrame
'''
def get_metrics(model, window):
    train_metrics = model.evaluate(window.train)
    val_metrics = model.evaluate(window.val)
    test_metrics = model.evaluate(window.test)

    metrics_names = model.metrics_names
    metrics_names[0] = "huber"
    metrics = pd.DataFrame({
        "names": metrics_names,
        "train": train_metrics,
        "val": val_metrics,
        "test": test_metrics,
    })

    return metrics


windows_sizes=[(3, 1), (5, 1), (8, 1), (8, 3), (8, 5), (12, 1),
               (12, 3), (12, 5), (24, 1), (24, 3), (24, 5)]

'''
Given a model it will train with different windows sizes

@param model which will be training
```

```
@param max_epoch for training
@param lr (learning rate) of the model
@param windows_sizes a list of tuples containing window sizes as (input, output)
'''
def model_generator(model, lr, max_epochs=150, windows_sizes=windows_sizes):
    # Loads and splits the DataFrames
    df = load_dataset()
    train_df, val_df, test_df = split_dataset(df)

    # For every windo
    for w in windows_sizes:
        # Create the windoe
        input_width, steps = r
        window = WindowGenerator(input_width=input_width,
                                 label_width=steps,
                                 shift=steps,
                                 train_df=train_df,
                                 val_df=val_df,
                                 test_df=test_df)


        # Train model
        compile_and_fit(
            model, window, lr=lr, should_stop=True, max_epochs=max_epochs)

        # Save metrics
        metrics = get_metrics(model, window)
        filename = "/path/to/results/{model.get_name()}" + \
                    "-{input_width}-{steps}.csv"
        metrics.to_csv(filename)
```

ACRONYMS

AI Artificial Intelligence. iii, 5–7, 9, 28, 40, 63, 98, 99

AR Autoregressive. iii, iv, vi, 3, 4, 68, 83, 84, 88–91, 97, 100

CO_2 Carbon dioxide. 1

CSV Comma-Separated Values. 64, 65, 69, 73

DL Deep Learning. 40, 63

IOC In Other Case. 26, 45

LR Learning Rate. 38, 107

LSTM Long-Short Term Memory. ii, iii, v, vi, 3, 4, 58, 59, 61, 82–84, 88–91, 97

MAE Mean Absolute Error. iii, v, vi, 31–34, 48, 60, 61, 86, 88, 90, 92

MAPE Mean Absolute Percentage Error. v, 34

ML Machine Learning. 7, 9, 16, 20, 98, 99

MPE Mean Percentage Error. v, 34, 35

MSE Mean Squared Error. iii, v, vi, 32, 33, 36, 37, 48, 60, 86–89, 92

MSLE Mean Squared Logarithmic Error. iii, vi, 33, 87–89, 93

NLP Natural Language Processing. 7, 35, 52

NN Neural Network. v, 8, 22, 28, 60

OLS Ordinary Least Square. 36, 37

PSS Product Service System. 1

ReLU Rectified Linear Unit. 17, 19, 58

RMSE Root Mean Squared Error. iii, vi, 32, 33, 60, 61, 87, 88, 90, 93

RMSLE Root Mean Squared Logarithmic Error. iii, vi, 60–62, 87, 88, 91, 94

RMSProp Root Mean Square Propagation. 47

RNN Recurrent Neural Network. ii, v, 3, 52–54, 56, 84, 88–91

SGD Stochastic gradient descent. 46

SQL Structured Query Language. 65

SRNN Simple Recurrent Neural Network. 97

XOR Exclusive Or. v, vii, 13–15