Application of Machine Learning Techniques for Predicting Turbidity in Desalination Plants

UNIVERSIDAD
COMPLUTENSE
MADRID

Samuel Lozano Iglesias

Supervised by:

Dr. Ángel González Prieto & Dr. Miguel Ruiz García

Faculty of Mathematical Sciences

Complutense University of Madrid

Final Degree Project submitted to obtain the

Double Bachelor's Degree in Mathematics and Physics

June 2024

Abstract

Machine learning is an artificial intelligence technique widely used for the prediction of time-dependent events, which we use to address the prediction of water turbidity in desalination plants. To this end, we study one of the types of machine learning, supervised learning, highlighting the different objectives of the models for which it is applicable: regression and classification models. In addition, we present artificial neural networks and the mechanisms that allow them to learn, such as regularisation or optimisation algorithms. The characteristics of this type of networks are also detailed, differentiating between feedforward and recurrent networks, and we discuss universal approximation theorems that underpin their use. All these elements are applied to the turbidity problem, which is solved by means of two models, a regression model and a classification model. Both use the same type of recurrent neural network: long short-term memory. Finally, the comparison of the models reveals that the classification model shows a higher accuracy, despite providing less information, and is sufficient to be implemented in the desalination plant.

Key words: *machine learning*, *regression*, *classification*, *neural network*, *turbidity.*

Contents

Chapter 1

Introduction

Weather prediction has been a subject of study since the beginning of human civilization. In the most primitive eras, it was necessary to predict whether it would rain to adjust crops or hunting periods accordingly. Simple rules were established that related the sky's hues or cloud formations to weather changes. With the evolution of societies and the scientific method, empirical models capable of anticipating adverse weather events were developed in the mid-19th century, but it wasn't until the early 20th century that the first numerical predictive models emerged [32]. Today, weather prediction remains widely studied and improved with modern techniques.

The problem studied in this work is closely related to weather prediction and is of great importance for companies in the water treatment sector, such as ACCIONA. The question is: *Can the turbidity of water received at a desalination plant be predicted in advance?*

One of the tasks I participated in during my internship in the *Knowledge Applied to Business* department at ACCIONA was addressing this problem. Specifically, developing a model capable of estimating the turbidity level of water that would be collected at one of the company's plants several hours in advance. A model with these features is highly useful for the plant's operations as it allows avoiding issues caused by extreme turbidity spikes and performing maintenance tasks at times when production is not feasible.

Initially, the scientific approach leads to the assumption that the optimal model can be achieved by identifying the external variables involved and developing a set of rules to relate them to the desired prediction. This technique is not always fruitful, and in the case of turbidity, it may fail due to a lack of knowledge, technical limitations, or intrinsic uncertainty.

Once it is established that analytical modeling is not possible, the focus shifts to applying general prediction methods. This involves deducing the turbidity value without directly relating it to the external variables involved or even without knowing those variables. Predictive models are employed for this purpose.

Traditionally, predictive models have focused on studying time series: sets of data arranged chronologically and interrelated. This study is based on historical records (autoregressive models) or other independent variables that may affect the series (causal models). However, these models are not optimal for analyzing large volumes of data or data with many causal effects between variables. As artificial intelligence has advanced, prediction algorithms have adapted their techniques and developed models that have the ability to "learn": the so-called **machine learning**.

1.1 Machine Learning

Machine learning is a branch of artificial intelligence based on the concept of "learning": transforming experience into knowledge. To do this, machine learning models are trained with an initial set of data and then used to predict an outcome or perform a task. Since learning can take many forms, the characteristics of the models vary depending on the task to be solved, leading to different areas of machine learning.

Within the types of learning in machine learning theory, the focus of Chapter *2* is on one of them: **supervised learning**. This chapter describes the main linear models in supervised learning, based on their objective and complexity, highlighting regression and classification models.

On the other hand, to introduce the difference between the two main types of learning, supervised and unsupervised, consider the task of detecting anomalous values in a dataset. Supervised learning would involve training a model by studying the data while adding an indicator for whether the value is anomalous, so the model can produce the correct result. In contrast, unsupervised learning would train by receiving the data without such an indicator, requiring the model to group the data based on inferred common features and then detect the anomalous values.

In supervised learning, the dataset used for training contains sufficient and meaningful information to perform the required task. Thus, the goal of the model is to deduce that meaningful information even when it is absent, which occurs during a validation phase. Conversely, unsupervised learning has sufficient but not meaningful information, requiring the model to determine correlations between the data to detect differences. In this case, a training phase is not distinct from a subsequent validation phase, as the input data is not modified.

Lastly, there is an intermediate type of learning called reinforcement learning, where the model has meaningful information during training, but this feedback is not always immediately provided after making a prediction. Thus, the model must explore the data similarly to unsupervised learning, but it receives feedback after making predictions.

In conclusion, when seeking improvement through experience, i.e., learning, it makes sense to use machine learning algorithms rather than traditional prediction models to perform the tasks. These algorithms can be implemented using **artificial neural networks**: networks of nodes that simulate the effects of neurons in the brain. Among the types of neural networks, feedforward and recurrent networks are particularly notable, and these are developed in Chapter *3*.

Artificial neural networks have certain characteristics that must be properly defined, and their possibilities need to be explored, such as the distribution used for initialization, the activation function, or the loss function. Additionally, the third chapter discusses optimization algorithms and presents universal approximation theorems, which provide the theoretical foundation for using neural networks and indicate under what conditions a function can be approximated to a certain degree of precision using an artificial neural network.

It is important to highlight that choosing the right learning algorithm and neural network is crucial, as there is no optimal algorithm for all problems in a given set, but there is a better algorithm for each specific problem. This is demonstrated by the *No Free Lunch* theorems, presented by David H. Wolpert and William G. Macready in [42].

Therefore, developing a predictive model in line with the current paradigm involves understanding the problem at hand, selecting the most suitable neural network type to solve it, and adapting the learning process to the model to be implemented. This is precisely what has been done to predict water turbidity, by training two models: one for regression and another for classification.

The development and results of both models are presented in Chapter *4*, including a comparison of their advantages and disadvantages. It is worth noting that one of the models has proven to be accurate enough for ACCIONA to implement it in its desalination plant. This has enhanced the plant's planning capacity, directly impacting its productivity and, consequently, the water supply to nearby communities.

Chapter 2

Supervised Learning

Supervised learning is a type of machine learning that relies on learning by comparing predictions with the results that should have been obtained. In this way, a finite set of input data, commonly called a *dataset* or domain, X, is used, which contains different features relevant to solving the task. With it, the value of a finite set Y is estimated, which includes a label for each element of X. Finally, the model learns by minimizing the difference between the estimates and the labels.

Within supervised learning, there are two types of models: regression and classification models. Their structure and development are similar, but they differ in some features and, mainly, in their objective. Therefore, it is necessary to identify the type of model that should be used for a specific problem.

Regression models aim to approximate the value of a label given during training, although their value may not correspond to any correct label. This could be the case of a model that seeks to predict a person's height given certain input parameters, as the model could produce a predicted value that does not match any of the real heights used in training.

On the other hand, classification models aim to deduce which of the possible labels is correct, so the prediction is always one of the labels used in training. This could be the case of a model that seeks to predict whether a traffic light is red, yellow, or green, as the result will be one of the three classes.

Additionally, in algorithms that use supervised learning, two phases are distinguished: training and validation. Training is the phase already described: the model receives the domain X and the set of labels Y, with the goal of making estimates and comparing them with the real labels so that it learns through received *feedback*. However, in validation, the model receives a domain X_V with which it has never trained, and it does not receive its associated labels Y_V. The goal in this case is to predict the value of Y_V so that, by comparing the estimates, it can be verified that the model is capable of generalizing and that the estimation remains adequate when using data it has never been trained on.

Formally, the training set T is considered as a finite sequence of pairs $T = \{(\boldsymbol{x}^1, y^1), \ldots, (\boldsymbol{x}^N, y^N)\}$ in the space $X \times Y$. Furthermore, it is assumed that there exists an unknown function $f : X \to Y$, and an estimator of it, or prediction function, $\hat{f}$, is obtained, which is intended to be as similar to f as possible. It is also sought that $\hat{f}$ generalizes, meaning it maintains similarity with f in validation. The prediction function is of the form $\hat{f} : X \to \mathbb{R}$ if the model is regression and $\hat{f} : X \to Y$ if it is classification, although sometimes for classifications with K classes, prediction functions $\hat{f} : X \to [0,1]^K$ indicating probabilities are used, or even general functions $\hat{f} : X \to \mathbb{R}$. This prediction function is intended to be consistent [27].

Definición 2.0.1. *A prediction function is said to be consistent with the sample, or training set,* $T = \{(\boldsymbol{x}^1, y^1), \ldots, (\boldsymbol{x}^N, y^N)\}$*, if for each* $i \in \{1, \ldots, N\}$ *it holds that:*

$$\hat{f}(\boldsymbol{x}^i) = y^i =: f(\boldsymbol{x}^i)\,.$$

In summary, an error must be defined to indicate the accuracy of the prediction and the adequacy of the training, in order to minimize it as training progresses and, thereby, learn:

Definición 2.0.2. *Given a domain* X *of cardinality* N *and the set of correct labels* $Y \subseteq \mathbb{R}$*, the error of the prediction function* $\hat{f} : X \to \mathbb{R}$ *is defined as*

$$\mathcal{E}_{X,Y}(\hat{f}) := \frac{1}{N} \sum_{i=1}^{N} d\left(\hat{f}(\boldsymbol{x}^i), y^i\right)\,, \tag{2.1}$$

where $d(\cdot,\cdot) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ *is a distance defined in the metric space* $\mathbb{R}$.

As can be seen from the definition, the error depends on the chosen distance in Y, so there are different possible errors that are more or less suitable depending on the context and the algorithm to be used. An error notable for its simplicity is the empirical error, given by the discrete distance:

$$d(y^1, y^2) = \begin{cases} 0 & \text{if} \quad y^1 = y^2 \\ 1 & \text{if} \quad y^1 \neq y^2 \end{cases} \quad \Longrightarrow \quad \mathcal{E}_{X,Y}(\hat{f}) := \frac{|\{i \in \{1, \ldots, N\} : \hat{f}(\boldsymbol{x}^i) \neq y^i\}|}{N}\,.$$

This empirical error is not always appropriate, as it evaluates all incorrect predictions equally. Due to this, in practice, errors adapted to the structure of the considered labels Y are defined. Some examples of these errors are detailed below and are fundamental for the development of machine learning algorithms, as they aim to minimize them and learn from the *feedback* provided by the errors.

It is worth noting that for a *Machine Learning* model to be useful, not only must its training error be minimized, but its validation error must also be minimized. In fact, the factors that determine how good a machine learning algorithm is are its ability to reduce the training error ("ability to learn") and its ability to reduce the difference between the training error and the validation error ("ability to generalize") [12].

2.1 Multiple Linear Regression

Linear regression models are the simplest *Machine Learning* models that can be presented to infer a result that approximates the correct labels. In them, input data $\boldsymbol{x}^i \in \mathbb{R}^M$ with $i \in \{1, \dots, N\}$ are considered, where $X = \{\boldsymbol{x}^1, \dots, \boldsymbol{x}^N\}$ is the domain with N observations that depend on M predictor variables. This set is linearly combined through a series of weights $\tilde{\boldsymbol{w}} = (w_0, \dots, w_M)^T$ to obtain the prediction function $\hat{f} : X \to \mathbb{R}$. Let $\boldsymbol{x}^i = (x_1^i, \dots, x_M^i)^T$ be the vector of data associated with one of the observations, then:

$$\hat{f}(\boldsymbol{x}^i, \tilde{\boldsymbol{w}}) = w_0 + \sum_{j=1}^{M} w_j x_j^i \ . \tag{2.2}$$

It is worth noting that in this type of model, a positive sign of w_j represents that the j-th predictor variable and the correct labels have a direct correlation, while a negative sign of w_j indicates that the correlation is inverse [12].

To extend this model to a problem with greater complexity, M functions $\phi_1(\boldsymbol{x}), \dots, \phi_M(\boldsymbol{x})$ not necessarily linear can be used, which satisfy the relation:

$$\hat{f}(\boldsymbol{x}^i, \tilde{\boldsymbol{w}}) = w_0 + \sum_{j=1}^{M} w_j \phi_j(\boldsymbol{x}^i) = \sum_{j=0}^{M} w_j \phi_j(\boldsymbol{x}^i) \ ,$$

where it has been considered that $\phi_0(\boldsymbol{x}^i) \coloneqq 1$, for all $i \in \{1, \dots, N\}$. In reality, $D \neq M$ functions $\phi_1(\boldsymbol{x}), \dots, \phi_D(\boldsymbol{x})$ could have been used, not necessarily the same as the number of variables M, implying that the number of weights would be $D + 1$.

In any case, the prediction function $\hat{f}(\boldsymbol{x}^i, \tilde{\boldsymbol{w}})$ may not be linear if the functions $\phi_j(\boldsymbol{x}^i)$ are not, although the model remains a linear regression model, as the prediction function is linear in the weights $\tilde{\boldsymbol{w}}$. For $\phi_j(\boldsymbol{x}^i)$, different functions can be used, from polynomial to more sophisticated ones such as Gaussian functions [4]:

- Polynomial: $\quad \phi_j(\boldsymbol{x}^i) = \prod_{k=1}^{M} \left(x_k^i\right)^{k_j} \ , \quad$ for all $j \in \{1, \dots, M\}$.
- Gaussian: $\quad \phi_j(\boldsymbol{x}^i) = \prod_{k=1}^{M} \exp\left(-\frac{(x_k^i - \mu_k)^2}{2s^2}\right) \ , \quad$ for all $j \in \{1, \dots, M\}$.

2.1.1 Maximum Likelihood Parameters

As outlined in the previous section, by fixing weights $\tilde{\boldsymbol{w}} = (w_0, \ldots, w_M)$, it is possible to obtain a linear estimator of the form (2.2). This section explores what values the parameters $\tilde{\boldsymbol{w}}$ should take to minimize the difference between the regression and the correct label for each observation in the training set.

Assume that the underlying random process of the problem under study is a certain random variable $\mathcal{Y}$ that is normally distributed, with a fixed standard deviation σ but an unknown mean. That is, for all $\boldsymbol{x}^i \in \mathbb{R}^M$, it holds that $\mathcal{Y}(\boldsymbol{x}^i) \sim \mathcal{N}\left(f(\boldsymbol{x}^i), \sigma\right)$, where $f : \mathbb{R}^M \to \mathbb{R}$ is a function to be determined that conditions the mean of $\mathcal{Y}$. Thus, the labels $Y = \{y^1, \ldots, y^N\}$ are samples of the random variable $\mathcal{Y}$, i.e., $\mathcal{Y}(\boldsymbol{x}^1), \ldots, \mathcal{Y}(\boldsymbol{x}^N)$ with $X = \{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^N\}$ as the domain.

Minimizing the difference between the function f and the prediction function $\hat{f}_{\tilde{\boldsymbol{w}}}$ is equivalent to obtaining the weights that maximize the likelihood, where the subscript $\tilde{\boldsymbol{w}}$ has been added to emphasize the dependence of $\hat{f}$ on the weights. The likelihood function $\mathcal{L}(\tilde{\boldsymbol{w}})$ is defined as the product of the probability densities:

$$\mathcal{L}(\tilde{\boldsymbol{w}}) = \prod_{i=1}^{N} p\left(\hat{f}_{\tilde{\boldsymbol{w}}}(\boldsymbol{x}^i) = y^i\right) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} \cdot \exp\left(-\frac{\left(y^i - \hat{f}_{\tilde{\boldsymbol{w}}}(\boldsymbol{x}^i)\right)^2}{2\sigma^2}\right) . \tag{2.3}$$

With this, the following result is demonstrated:

Teorema 2.1.1. *Let a training set $T = \{(x_j^i, y^i) : i = 1, \ldots, N; j = 1, \ldots, M\}$, i.e., with N observations and M predictor variables, and $x_0^i = 1$, for all $i \in \{1, \ldots, N\}$. Taking $X = \{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^N\}$ and $Y = \{y^1, \ldots, y^N\}$, such that the labels y^i are samples $\mathcal{Y}(\boldsymbol{x}^i)$ of a random variable following a normal distribution with standard deviation σ and unknown mean $f(\boldsymbol{x}^i)$. If the mean is estimated by a linear prediction function:*

$$\hat{f}_{\tilde{\boldsymbol{w}}}(\boldsymbol{x}^i) = \sum_{j=0}^{M} w_j x_j^i = (\boldsymbol{x}^i)^T \cdot \tilde{\boldsymbol{w}} ,$$

then there exist maximum likelihood parameters $\tilde{\boldsymbol{w}}^*$ given by the solution to the linear system $(X^T X) \cdot \tilde{\boldsymbol{w}}^* = X^T Y$.

Proof. (Adapted from [30]). Considering the likelihood function $\mathcal{L}(\tilde{\boldsymbol{w}})$, defined in (2.3), the maximum likelihood parameters $\tilde{\boldsymbol{w}}^*$ are obtained by maximizing it. Thus, through a transformation to natural logarithms that does not alter the extrema of the function:

$$\begin{aligned}\log\left(\mathcal{L}(\tilde{\boldsymbol{w}})\right) &= \sum_{i=1}^{N} \log\left(\frac{1}{\sqrt{2\pi}\sigma} \cdot \exp\left(-\frac{\left(y^i - (\boldsymbol{x}^i)^T \cdot \tilde{\boldsymbol{w}}\right)^2}{2\sigma^2}\right)\right) = \\ &= N \cdot \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \sum_{i=1}^{N} \frac{\left(y^i - (\boldsymbol{x}^i)^T \cdot \tilde{\boldsymbol{w}}\right)^2}{2\sigma^2} = \\ &= N \cdot \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2} \cdot \left(Y - X^T \cdot \tilde{\boldsymbol{w}}\right)^T \cdot \left(Y - X^T \cdot \tilde{\boldsymbol{w}}\right) .\end{aligned}$$

Considering this relationship, it follows that:

$$\mathcal{E}(\tilde{\boldsymbol{w}}) := \log\left(\mathcal{L}(\tilde{\boldsymbol{w}})\right) = N \cdot \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2} \cdot \left(Y - X^T \cdot \tilde{\boldsymbol{w}}\right)^T \cdot \left(Y - X^T \cdot \tilde{\boldsymbol{w}}\right) . \tag{2.4}$$

Since the expression obtained in (2.4) is quadratic in $\tilde{\boldsymbol{w}}$, its optimization is immediate through the gradient of this expression, and there will be a global maximum, as the Hessian matrix $\nabla^2_{\tilde{\boldsymbol{w}}}\mathcal{E}(\tilde{\boldsymbol{w}}) = -X^T X/\sigma^2$ is negative semidefinite due to being a negated scalar product:

$$\begin{aligned}\frac{d\mathcal{E}}{d\tilde{\boldsymbol{w}}} &= \frac{-1}{2\sigma^2} \cdot \frac{d}{d\tilde{\boldsymbol{w}}}\left(\left(Y - X^T \cdot \tilde{\boldsymbol{w}}\right)^T \cdot \left(Y - X^T \cdot \tilde{\boldsymbol{w}}\right)\right) = \\ &= \frac{-1}{2\sigma^2} \cdot \left(Y^T Y - 2Y^T X \cdot \tilde{\boldsymbol{w}} + \tilde{\boldsymbol{w}}^T X^T X \cdot \tilde{\boldsymbol{w}}\right) = \frac{-1}{\sigma^2}\left(-Y^T X + \tilde{\boldsymbol{w}}^T X^T X\right) .\end{aligned}$$

Setting the above gradient to zero yields the sought linear system:

$$\frac{d\mathcal{E}}{d\tilde{\boldsymbol{w}}} = \frac{-1}{\sigma^2}\left(-Y^T X + \tilde{\boldsymbol{w}}^T X^T X\right) = \mathbf{0}^T \iff (\tilde{\boldsymbol{w}}^*)^T X^T X = Y^T X \iff (X^T X) \cdot \tilde{\boldsymbol{w}}^* = X^T Y .$$

■

Observación 2.1.2. *The generalization to the case where functions $\phi_j(\boldsymbol{x}^i)$ are considered instead of $\boldsymbol{x}^i$ is analogous and can be found in [9].*

In certain situations, it is interesting not to assume that the variance is known and to estimate the maximum likelihood variance $(\sigma^*)^2$, given a certain set of weights $\tilde{\boldsymbol{w}}$. To do this, following the development carried out in the previous proof:

$$\frac{\partial\mathcal{E}}{\partial\sigma^2} = \frac{\partial}{\partial\sigma^2}\left(N \cdot \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \sum_{i=1}^{N} \frac{\left(y^i - \boldsymbol{x}^T \cdot \tilde{\boldsymbol{w}}\right)^2}{2\sigma^2}\right) = -\frac{N}{2\sigma^2} + \frac{1}{2\sigma^4}\sum_{i=1}^{N}\left(y^i - (\boldsymbol{x}^i)^T \cdot \tilde{\boldsymbol{w}}\right)^2$$

Again, setting this to zero yields the maximum likelihood estimate of the variance, which is simply the average of the squared distances between the values of the prediction function $\hat{f}_{\tilde{\boldsymbol{w}}}(\boldsymbol{x}^i)$ and the labels y^i:

$$\frac{\partial\mathcal{E}}{\partial\sigma^2} = 0 \implies (\sigma^*)^2 = \frac{1}{N}\sum_{i=1}^{N}\left(y^i - (\boldsymbol{x}^i)^T \cdot \tilde{\boldsymbol{w}}\right)^2 \tag{2.5}$$

Observación 2.1.3. *In general, the probability distribution of the random variable $\mathcal{Y}$ associated with the process may not be normal. In such a case, it is replaced by the correct distribution, and other maximum likelihood parameters are obtained, as done in [29] for the Bernoulli distribution.*

2.1.2 Regularization

As seen, the quality of a *Machine Learning* model can be determined based on its error, which has been defined generally in (2.1). For this error, the function (2.4) can be considered, given by the negative natural logarithm of the likelihood (2.3) but ignoring the dependence on σ^2 as it is not a free parameter and disregarding the constant term, i.e.,

$$\mathcal{E}_{X,Y}(\tilde{\boldsymbol{w}}) = \frac{1}{2}||Y - X^T \cdot \tilde{\boldsymbol{w}}||^2 \,. \tag{2.6}$$

In a machine learning algorithm, this error tends to decrease by modifying the weights $\tilde{\boldsymbol{w}} = (w_0, \dots, w_M)^T$ during the training phase. Since the goal of a model is to also predict with data it has never trained on, i.e., to generalize, the definition of training error can be extended to the validation phase, and it can be sought to keep it low. Techniques with this purpose are called "regularization."

Once optimal weights $\tilde{\boldsymbol{w}}$ have been obtained for the training phase, an overfitting effect (*overfitting*, in English) is likely to occur. That is, the algorithm has a low training error but is unable to reduce the validation error: *the model learns but does not generalize.* A method to avoid this is to penalize the magnitude of the weights in a process called weight regularization [27].

Thus, a regularization parameter $\lambda \geq 0$, called weight decay, is introduced into the model's error, accompanied by the magnitude of the weights. For the quadratic error (2.6), the following expression is obtained:

$$\mathcal{E}_{X,Y}(\tilde{\boldsymbol{w}}) = \frac{1}{2}||Y - X^T \cdot \tilde{\boldsymbol{w}}||^2 + \frac{\lambda}{2} \cdot ||\tilde{\boldsymbol{w}}||^2 \,. \tag{2.7}$$

Observación 2.1.4. *The norm used can be different from the Euclidean norm, being* $||\cdot||_p$ *for any value of* $p \geq 1$. *In fact, lower values of* p *provide solutions with a greater number of parameters* $w_j = 0$, *but they can make models less stable [9].*

In this case, the optimal value $\tilde{\boldsymbol{w}}^*$ of the weights can be obtained by applying the gradient to equation (2.7), resulting in only a variation of the optimum for the unregularized case:

$$\frac{\partial \mathcal{E}}{\partial \tilde{\boldsymbol{w}}} = \frac{\partial}{\partial \tilde{\boldsymbol{w}}} \left(\frac{1}{2} \cdot \left[(Y - X^T \cdot \tilde{\boldsymbol{w}})^T \cdot (Y - X^T \cdot \tilde{\boldsymbol{w}}) + \lambda \tilde{\boldsymbol{w}}^T \cdot \tilde{\boldsymbol{w}} \right] \right) = -Y^T X + \tilde{\boldsymbol{w}}^T X^T X + \lambda \tilde{\boldsymbol{w}}^T$$

$$\Longrightarrow \frac{\partial \mathcal{E}}{\partial \tilde{\boldsymbol{w}}} = \mathbf{0}^T \iff (\tilde{\boldsymbol{w}}^*)^T \cdot \left(X^T X + \lambda I \right) = Y^T X \iff \left(\lambda I + X^T X \right) \cdot \tilde{\boldsymbol{w}}^* = X^T Y \,.$$

2.2 Multiple Linear Classification

Classification models, along with regression models, are the most common models within supervised learning. These are applied to problems where the labels belong to a discrete set of possible classes. Thus, the idea of these models is that they assign to each element in the training set a probability of belonging to each of the classes. In practice, most classification algorithms are based on partitioning the space $\mathbb{R}^M$ such that each label falls into a single partition [35].

Considering a domain $X \subseteq \mathbb{R}^M$ and a set of correct labels $Y \subseteq \mathbb{R}$, there exists a discrete set of K classes $\mathcal{C} = \{C_1, \ldots, C_K\}$ such that, for all $i \in \{1, \ldots, N\}$, $y^i \in C_k$ for some $k \in \{1, \ldots, K\}$. Assuming the usual case where the classes are disjoint, the space given by the input data can be divided into disjoint regions $R_1, \ldots, R_K$ called decision regions. The boundaries of these regions are called decision surfaces, and in the case of linear classification, they are hyperplanes of dimension $(M-1)$, where M is the dimension of X [12].

In general, two types of classification models should be distinguished:

- **Bayesian models**: infer, through modeling, the probability density of the data conditioned on a class, $p(\boldsymbol{x}^i|C_k)$, as well as the density of the different classes, $p(C_k)$. With this and Bayes' Theorem (see A.1 in the appendix chapter), the probability densities of the classes conditioned on the input data, the so-called *a posteriori* probability, $p(C_k|\boldsymbol{x}^i)$, are calculated:

$$p(C_k|\boldsymbol{x}^i) = \frac{p(\boldsymbol{x}^i|C_k)p(C_k)}{p(\boldsymbol{x}^i)} = \frac{p(\boldsymbol{x}^i|C_k)p(C_k)}{\sum_{l=1}^{K} p(\boldsymbol{x}^i|C_l)p(C_l)} . \tag{2.8}$$

With this probability, the class to which the input data $\boldsymbol{x}^i$ belongs is determined. In the simplest case with two classes C_1 and C_2, the *a posteriori* probability is of the form:

$$p(C_1|\boldsymbol{x}^i) = \frac{p(\boldsymbol{x}^i|C_1)p(C_1)}{\sum_{l=1}^{2} p(\boldsymbol{x}^i|C_l)p(C_l)} = \frac{1}{1 + \frac{p(\boldsymbol{x}^i|C_2)p(C_2)}{p(\boldsymbol{x}^i|C_1)p(C_1)}} = \frac{1}{1 + \exp\left(-\log\left(\frac{p(\boldsymbol{x}^i|C_1)p(C_1)}{p(\boldsymbol{x}^i|C_2)p(C_2)}\right)\right)} . \tag{2.9}$$

Observación 2.2.1. *As described in* Section *3.3.2, this a posteriori probability is nothing more than the logistic sigmoid function* (3.7) *with* $\alpha = 1$ *and* $a = \log\left(\frac{p(\boldsymbol{x}^i|C_1)p(C_1)}{p(\boldsymbol{x}^i|C_2)p(C_2)}\right)$.

For the case where there are $K > 2$ classes, the *a posteriori* probability density can be described by a multiclass generalization determined by equation (2.8):

$$p(C_k|\boldsymbol{x}^i) = \frac{p(\boldsymbol{x}^i|C_k)p(C_k)}{\sum_{l=1}^{K} p(\boldsymbol{x}^i|C_l)p(C_l)} = \frac{\exp\left(\log\left(p(\boldsymbol{x}^i|C_k)p(C_k)\right)\right)}{\sum_{l=1}^{K} \exp\left(\log\left(p(\boldsymbol{x}^i|C_l)p(C_l)\right)\right)} . \tag{2.10}$$

Observación 2.2.2. *Again, as described in* Section *3.3.2, this a posteriori probability is the softmax function* (3.8) *considering* $a_k = \log\left(p(\boldsymbol{x}|C_k)p(C_k)\right)$.

- **Discriminative models**: directly model the *a posteriori* probability, $p(C_k|\boldsymbol{x}^i)$, assigning a label directly to each input data $\boldsymbol{x}^i$. It is worth noting that in a discriminative model with K possible classes, it is common for the correct labels to be vectors $\boldsymbol{y}^1, \ldots, \boldsymbol{y}^N$, each of the form $\boldsymbol{y}^i = (y_1^i, \ldots, y_K^i)^T$ with $y_k^i \in \{0, 1\}$, for all $k \in \{1, \ldots, K\}$. That is, the correct label for each observation is a vector with a 1 in the position of the class to which it belongs and a 0 in the rest, representing the probability of belonging to each of the classes.

Directly modeling the *a posteriori* probability is more useful for simple practical cases than modeling the *a priori* probability and then obtaining the *a posteriori* probability through equation (2.8). For this reason, this work focuses solely on discriminative models. However, for a result describing maximum likelihood parameters in Bayesian models, see A.2 in the appendix chapter.

2.2.1 Support Vector Machines

Starting with an analysis similar to that proposed in SECTION *2.1*, the linear classification model is the simplest discriminative model that can be proposed. In it, the prediction function, now called a discriminant, is linear and generalizes to $\hat{f} : X \to \mathbb{R}$, where $\boldsymbol{w} = (w_1, \dots, w_M)^T$ is the weight vector and $\tilde{\boldsymbol{w}} = (w_0, \dots, w_M)^T$ is its extension:

$$\hat{f}(\boldsymbol{x}^i, \tilde{\boldsymbol{w}}) = w_0 + \sum_{j=1}^{M} w_j x_j^i = w_0 + (\boldsymbol{x}^i)^T \cdot \boldsymbol{w} \,, \tag{2.11}$$

where one of the N observations $X = \{\boldsymbol{x}^1, \dots, \boldsymbol{x}^N\}$ that make up the domain has been considered.

Also considering the simplest linear classification, into two classes with values $+1$ and -1, then class C_1 can be defined as the observations that satisfy $\hat{f}(\boldsymbol{x}^i) \geq 0$, and class C_2 as those that do not. Therefore, the decision surface is an M-dimensional generalization of the two-dimensional hyperplane given in figure 2.1 [4]:

$$H = \{\boldsymbol{x} \in \mathbb{R}^M : \hat{f}(\boldsymbol{x}) = 0\} \,.$$

In fact, in linear classification problems, it is necessary to use decision hyperplanes to define the concept of a linearly separable dataset:

Definición 2.2.3. *Let $X = \{\boldsymbol{x}^1, \dots, \boldsymbol{x}^N\}$ be a set of input data with $\boldsymbol{x}^i \in \mathbb{R}^M$, whose data is divided into K disjoint classes. Then X is said to be linearly separable if its decision regions can be separated by linear surfaces, i.e., hyperplanes of dimension $(M-1)$.*

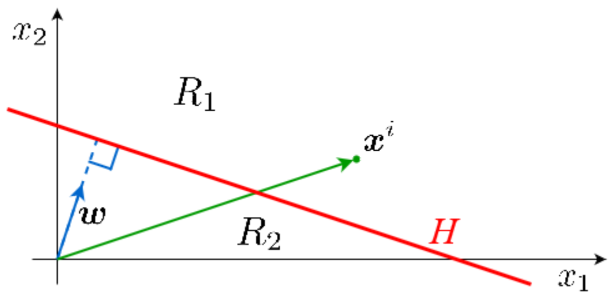


Figure 2.1: *Decision surface in a two-dimensional space with two decision regions, R_1 and R_2, separated by a decision surface, the plane H, and with $\boldsymbol{w}$ the weight vector.*

Observación 2.2.4. *This section deals exclusively with binary classification, and the generalization to a finite number K of classes is not trivial. First, an approach using $K-1$ two-class classifiers (the considered class versus the rest) leads to the creation of ambiguous regions [30]. The same occurs if a classifier formed by $K(K-1)/2$ two-class classifiers (pairs of possible classes) is considered. Therefore, the simplest approach is to generalize the prediction function: $\hat{f}_k(\boldsymbol{x}^i) = w_{k0} + (\boldsymbol{x}^i)^T \cdot \boldsymbol{w}_k$.*

Now, in the following proposition, the linear decision surface is characterized geometrically, i.e., its position and normal vector, as well as the distance of an arbitrary point to this hyperplane.

Proposición 2.2.5. *Given a linear classification model with two classes, the weight vector $\boldsymbol{w}$ is orthogonal to the decision surface, the term w_0 determines the position of this surface, and the perpendicular distance of an arbitrary point $\boldsymbol{x} \in \mathbb{R}^M$ to the surface is $|\hat{f}(\boldsymbol{x})|/||\boldsymbol{w}||$.*

Proof. (Adapted from [4]). Considering two points $\boldsymbol{x}^a$ and $\boldsymbol{x}^b$ on the decision surface $\hat{f}(\boldsymbol{x}) = 0$, then it holds that the vector $\boldsymbol{w}$ is orthogonal to both, hence it is orthogonal to the surface:

$$w_0 + (\boldsymbol{x}^a)^T \cdot \boldsymbol{w} = w_0 + (\boldsymbol{x}^b)^T \cdot \boldsymbol{w} = 0 \implies \big((\boldsymbol{x}^a)^T - (\boldsymbol{x}^b)^T\big) \cdot \boldsymbol{w} = 0\,.$$

Similarly, considering a point $\boldsymbol{x}^c$ on the surface, the perpendicular distance from the origin to this surface is given by:

$$d(O,H) = \frac{\big|(\boldsymbol{x}^c)^T \cdot \boldsymbol{w}\big|}{||\boldsymbol{w}||} = \frac{|-w_0|}{||\boldsymbol{w}||}\,, \quad \text{since } \hat{f}(\boldsymbol{x}) = w_0 + (\boldsymbol{x}^c)^T \cdot \boldsymbol{w} = 0\,.$$

Finally, the perpendicular distance between the hyperplane and the point $\boldsymbol{x}^d$, such that $\boldsymbol{x}^d \notin H$, can be deduced through the orthogonal projection $\boldsymbol{x}^d_\perp$ of $\boldsymbol{x}^d$ onto the surface H. In fact, the orthogonal projection $\boldsymbol{x}^d_\perp$ is obtained by displacing the point $\boldsymbol{x}^d$ a distance (with sign) $\varrho = \pm d(\boldsymbol{x}^d, H)$ in the direction normal to H. This direction is the unit vector $\boldsymbol{w}/||\boldsymbol{w}||$, and the sign of ϱ is taken positive if $\boldsymbol{x}^d$ is above the decision hyperplane and negative if $\boldsymbol{x}^d$ is below. Thus:

$$\boldsymbol{x}^d_\perp = \boldsymbol{x}^d - \frac{\boldsymbol{w}}{||\boldsymbol{w}||} \cdot \varrho \implies \varrho \cdot \frac{\boldsymbol{w}}{||\boldsymbol{w}||} = \boldsymbol{x}^d - \boldsymbol{x}^d_\perp\,.$$

Now, both sides are multiplied by $\boldsymbol{w}^T$ and w_0 is added and subtracted. Additionally, the relation (2.11) is considered, and since $||\boldsymbol{w}||^2 = \boldsymbol{w}^T\boldsymbol{w}$, the following is obtained:

$$\varrho \cdot \frac{\boldsymbol{w}^T\boldsymbol{w}}{||\boldsymbol{w}||} = w_0 + \boldsymbol{w}^T \cdot \boldsymbol{x}^d - w_0 - \boldsymbol{w}^T \cdot \boldsymbol{x}^d_\perp \implies \varrho = \frac{\hat{f}(\boldsymbol{x}^d) - \hat{f}(\boldsymbol{x}^d_\perp)}{||\boldsymbol{w}||}\,.$$

With this expression, since $\boldsymbol{x}^d_\perp \in H$, then $\hat{f}(\boldsymbol{x}^d_\perp) = 0$ and the desired result is achieved:

$$\pm d(\boldsymbol{x}^d, H) = \varrho = \frac{\hat{f}(\boldsymbol{x}^d)}{||\boldsymbol{w}||} \implies d(\boldsymbol{x}^d, H) = \frac{|\hat{f}(\boldsymbol{x}^d)|}{||\boldsymbol{w}||}\,.$$

■

With this concept of distance between the hyperplane and a point $\boldsymbol{x}$, the possibility arises of finding the hyperplane that "best" separates the classes. That is, given a linearly separable training set $T = \{(\boldsymbol{x}^1, y^1), \dots, (\boldsymbol{x}^N, y^N)\}$, the "best" separation is defined as the one that maximizes the distance between the decision surface and all points in the domain X, which will be called the margin. This is the basis of Support Vector Machines (SVM).

Definición 2.2.6. *Given a dataset $X = \{\boldsymbol{x}^1, \dots, \boldsymbol{x}^N\}$, the margin r in a support vector machine with two classes and estimator $\hat{f}$ is defined as:*

$$r = \min_{\boldsymbol{x} \in X} \{d(\boldsymbol{x}, H)\} \qquad \textit{with} \qquad H = \{\boldsymbol{x} \in \mathbb{R}^M : \hat{f}(\boldsymbol{x}) = 0\}\,.$$

Observación 2.2.7. *The concept of margin was used by Vladimir Vapnik and Alexey Chervonenkis to prove that learning is possible when the margin is "large" [39].*

First, since the weight vector $\boldsymbol{w}$ is only relevant in its direction, it can be assumed that $||\boldsymbol{w}|| = 1$, so the condition for support vector machines can be posed as an optimization problem, where the $y^i \in \{+1, -1\}$ are the correct labels:

$$\begin{aligned} \max \quad & \{r\} \\ \text{s.a} \quad & y^i \cdot \left(w_0 + (\boldsymbol{x}^i)^T \cdot \boldsymbol{w}\right) \geq r\,, \quad \text{for all } \; i \in \{1, \dots, N\} \\ & ||\boldsymbol{w}|| = 1 \\ & r > 0 \end{aligned} \tag{2.12}$$

Alternatively, a scaling factor can be defined to ensure that $\hat{f}(\boldsymbol{x}^a) = 1$ if $\boldsymbol{x}^a$ is the point closest to the hyperplane. Thus, by Proposition 2.2.5, the margin is $r = ||\boldsymbol{w}||^{-1}$, and the condition for support vector machines is to minimize the size of the weights, since maximizing $||\boldsymbol{w}||^{-1}$ is equivalent to minimizing $||\boldsymbol{w}||$. Ultimately, the optimization problem becomes:

$$\begin{aligned} \min \quad & \{||\boldsymbol{w}||\} \\ \text{s.a} \quad & y^i \cdot \left(w_0 + (\boldsymbol{x}^i)^T \cdot \boldsymbol{w}\right) \geq 1\,, \quad \text{for all } i \in \{1, \dots, N\} \end{aligned} \tag{2.13}$$

Both problems (2.12) and (2.13) determine the utility of support vector machines, which are responsible for obtaining the decision hyperplane that maximizes the margin r. Therefore, the two problems must be equivalent, as demonstrated in the following theorem:

Teorema 2.2.8. *Maximizing the margin r considering normalized weights is equivalent to minimizing the size of the weights if the data is scaled so that the margin is unitary, i.e., the optimization problems* (2.12) *and* (2.13) *are equivalent.*

Proof. (Extracted from [9]). Assuming (2.12), the equation can be reparameterized by a non-normalized vector $\boldsymbol{w}'$ such that $\boldsymbol{w} := \frac{\boldsymbol{w}'}{||\boldsymbol{w}'||}$. Thus, since $r > 0$, the following is obtained:

$$y^i \cdot \left(w_0 + (\boldsymbol{x}^i)^T \cdot \frac{\boldsymbol{w}'}{||\boldsymbol{w}'||} \right) \geq r \implies y^i \cdot \left(\frac{w_0}{r} + (\boldsymbol{x}^i)^T \cdot \frac{\boldsymbol{w}'}{r||\boldsymbol{w}'||} \right) \geq 1 \, .$$

Renaming the parameters, $w_0'' := \frac{w_0}{r}$ and $w'' := \frac{\boldsymbol{w}'}{r||\boldsymbol{w}'||}$, we obtain:

$$||\boldsymbol{w}''|| = \left|\left| \frac{\boldsymbol{w}'}{r||\boldsymbol{w}'||} \right|\right| = \frac{1}{r} \cdot \left|\left| \frac{\boldsymbol{w}'}{||\boldsymbol{w}'||} \right|\right| = \frac{1}{r} \implies r = \frac{1}{||\boldsymbol{w}''||} \, .$$

Substituting this into (2.12) and considering that maximizing $||\boldsymbol{w}''||^{-1}$ is equivalent to minimizing $||\boldsymbol{w}||$, we obtain the desired problem (2.13):

$$\min_{\boldsymbol{w}''} \{||\boldsymbol{w}''||\} \quad \text{subject to} \quad y^i \cdot (w_0'' + (\boldsymbol{x}^i)^T \cdot \boldsymbol{w}'') \geq 1 \quad \text{for all } i \in \{1, \ldots, N\} \, .$$

Conversely, assuming (2.13), we define $r := ||\boldsymbol{w}||^{-1}$, which is greater than 0. Therefore, we can multiply the imposed inequality by r:

$$y^i \cdot (w_0 + (\boldsymbol{x}^i)^T \cdot \boldsymbol{w}) \geq 1 \implies y^i \cdot \big(w_0 + (\boldsymbol{x}^i)^T \cdot \boldsymbol{w}\big) \cdot r \geq r \implies y^i \cdot \left(\frac{w_0}{||\boldsymbol{w}||} + (\boldsymbol{x}^i)^T \cdot \frac{\boldsymbol{w}}{||\boldsymbol{w}||} \right) \geq r \, .$$

Defining $w_0' := \frac{w_0}{||\boldsymbol{w}||}$ and $\boldsymbol{w}' := \frac{\boldsymbol{w}}{||\boldsymbol{w}||}$, then $||\boldsymbol{w}'|| = 1$ and we obtain the relation:

$$y^i \cdot \left(\frac{w_0}{||\boldsymbol{w}||} + (\boldsymbol{x}^i)^T \cdot \frac{\boldsymbol{w}}{||\boldsymbol{w}||} \right) \geq r \implies y^i \cdot \big(w_0' + (\boldsymbol{x}^i)^T \cdot \boldsymbol{w}'\big) \geq r \, .$$

Finally, minimizing $||\boldsymbol{w}||$ is equivalent to maximizing $||\boldsymbol{w}||^{-1}$, so with the above conditions, we obtain the desired problem (2.12):

$$\max_r \{r\} \quad \text{subject to} \quad y^i \cdot (w_0' + (\boldsymbol{x}^i)^T \cdot \boldsymbol{w}') \geq r \quad \text{with} \quad ||\boldsymbol{w}'|| = 1 \quad \text{and} \quad r > 0 \, .$$

■

2.2.2 Kernel Methods

The use of linear models for the classification of an input dataset is not, by any means, optimal. So much so that models of this type are often transformed by introducing a *kernel* function, which improves the performance of the classification algorithm. At the same time, using such functions allows extending non-linearly separable datasets to spaces where they are separable (see Example 2.2.10).

Given a training set $T = \{(\boldsymbol{x}^1, y^1), \ldots, (\boldsymbol{x}^N, y^N)\}$, the binary classification problem can be analyzed using support vector machines if the N points are linearly separable. Considering an M-dimensional space and two given classes, the goal is to verify that one of the possible partitions of the N points of T into two classes is linearly separable. Thus, the introduction of *kernel* functions is based on the fact that a partition of N points in an M-dimensional space is more likely to be linearly separable the larger M is.

To verify this claim, we define $C(N, M)$ as the number of partitions of N points that are linearly separable in an M-dimensional space, and we seek to show that $C(N, H) > C(N, M)$ if $H > M$, i.e., the number of linearly separable partitions increases with dimension. This is precisely what is demonstrated in the following Cover's Theorem [7]:

Teorema 2.2.9. *A classification problem of N points into two classes in a space of dimension M, with $N > M$, which is projected by a nonlinear transformation into another space of dimension H, with $H > M$, is more likely to be linearly separable than if projected into a space of dimension H with $H < M$.*

Proof. (Adapted from [28]). By induction, suppose we have the linearly separable partitions of N points and add one more:

1. If there exists a hyperplane that separated the N points and passes through the new point, then the hyperplane can be infinitesimally shifted in either of its two perpendicular directions to include the new point in one of the classes. Thus, for each separable partition of the N points that passes through the new point, there are two possible linearly separable partitions, with the new point in each of the two regions. The number of such partitions for N points is denoted as $C_1(N, M)$.

2. If a hyperplane that separated the N points does not pass through the new point, then the hyperplane continues to linearly separate the $N + 1$ points, and a partition for the $N + 1$ points is obtained from the partition of the N points by simply including the new point in the corresponding region. The number of such partitions for N points is denoted as $C_2(N, M)$.

Thus, to obtain $C(N+1, M)$, the linearly separable partitions of the first type must be counted twice, and those of the second type once. This is equivalent to considering that

$$C(N + 1, M) = 2 \cdot C_1(N, M) + C_2(N, M) = C(N, M) + C_1(N, M) \,.$$

In fact, $C_1(N, M) = C(N, M - 1)$, since restricting the hyperplane to pass through a point is equivalent to removing one degree of freedom, i.e., projecting the N points onto an $(M-1)$-dimensional space. Thus:

$$C(N + 1, M) = C(N, M) + C(N, M - 1) \,.$$

Now, this relation is applied recursively, and it is taken into account that $C(1, k) = 0$ if $k < 1$ and $C(1, k) = 2$ if $k \geq 1$.

- Applying the relation once:

$$C(N+1,M) = C(N,M) + C(N,M-1)\,.$$

- Applying the relation twice:

$$\begin{aligned} C(N+1,M) &= C(N-1,M) + C(N-1,M-1) + C(N-1,M-1) + C(N-1,M-2) = \\ &= C(N-1,M) + 2C(N-1,M-1) + C(N-1,M-2)\,. \end{aligned}$$

- Applying the relation three times:

$$\begin{aligned} C(N+1,M) &= C(N-2,M) + C(N-2,M-1) + 2C(N-2,M-1) + 2C(N-2,M-2) + \\ &\quad + C(N-2,M-2) + C(N-2,M-3) = \\ &= C(N-2,M) + 3C(N-2,M-1) + 3C(N-2,M-2) + C(N-2,M-3)\,. \end{aligned}$$

- Applying the relation N times:

$$C(N+1,M) = C(1,M) + N\cdot C(1,M-1) + \binom{N}{2} C(1,M-2) + \ldots + C(1,M-N)\,.$$

Ultimately, since $N > M$, we obtain:

$$C(N+1,M) = 2\cdot\sum_{j=0}^{M-1}\binom{N}{j}\,.$$

Therefore, for a fixed number of points $N+1$, the number of linearly separable partitions increases with the value of M, i.e., for $H > M$, there is a higher probability than for $H < M$ of finding the desired hyperplane. ■

With this result, the learning of the linear models described in SECTION *2.2* can be improved by extending them to higher-dimensional spaces through nonlinear functions. Thus, given a domain X of dimension M, a function $\phi : X \to \mathcal{H}$ is chosen, where $\mathcal{H}$ is called the feature space and is a Hilbert space, i.e., a vector space with an inner product and an associated norm, which is also complete with respect to that norm. This function allows transforming the initial training set $T = \{(\boldsymbol{x}^1, y^1), \ldots, (\boldsymbol{x}^N, y^N)\}$ into $T_\phi = \{(\phi(\boldsymbol{x}^1), y^1), \ldots, (\phi(\boldsymbol{x}^N), y^N)\}$, with which the prediction function, $\hat{f}$, is trained, and the prediction of the input data is obtained as $\hat{f}(\phi(\boldsymbol{x}^i))$. Additionally, the loss function is trivially transformed as $\mathcal{E}_{X,\phi}(\hat{f}) = \mathcal{E}_X(\hat{f}\circ\phi)$.

The function ϕ must be chosen appropriately, as not all extensions to higher-dimensional spaces are useful, as illustrated by the following example, represented in Figure 2.2.

Ejemplo 2.2.10. *The domain $X = \{-2, -1, 0, 1, 2\}$, where $x_3 = 0$ has the label 0 and the rest have the label 1, is not linearly separable by a hyperplane in $\mathbb{R}$, i.e., by a point. If it is extended to $\mathbb{R}^2$ via the function $\phi_1 : \mathbb{R} \to \mathbb{R}^2$ given by $\phi_1(x) = (x, x^2)$, then $X_{\phi_1} = \{\phi_1(-2), \phi_1(-1), \phi_1(0), \phi_1(1), \phi_1(2)\}$ is linearly separable, as a separating hyperplane in $\mathbb{R}^2$ would be the horizontal line passing through the point $(0, 1/2)$. However, the extension $\phi_2 : \mathbb{R} \to \mathbb{R}^2$ given by $\phi_1(x) = (x, x^3)$ would not be suitable, as it does not transform the set into a linearly separable one.*

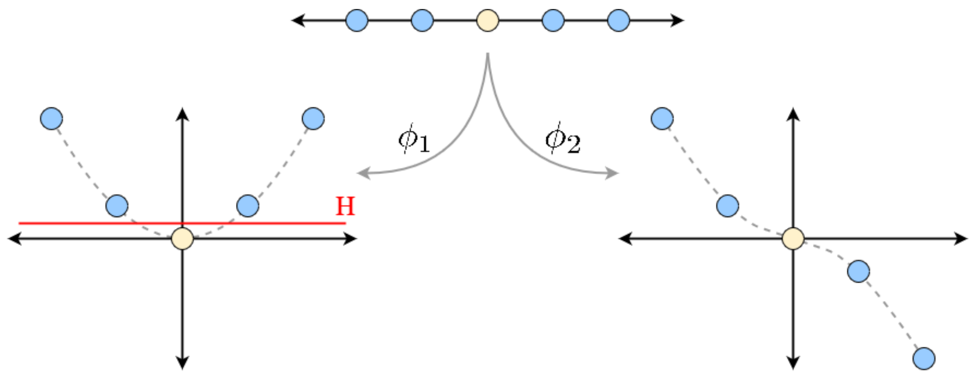


Figure 2.2: *Domain in $\mathbb{R}$ that is not linearly separable, which, when transformed into $\mathbb{R}^2$ by $\phi_1(x) = (x, x^2)$, becomes linearly separable, but by $\phi_2(x) = (x, x^3)$ it does not.*

Computationally, it is very costly to explicitly define the extension ϕ and perform operations with it in higher-dimensional spaces. Therefore, in practice, *kernel* functions are defined using the inner product in the considered Hilbert space:

$$\mathcal{K}(\boldsymbol{x}^i, \boldsymbol{x}^j) = \langle \phi(\boldsymbol{x}^i), \phi(\boldsymbol{x}^j) \rangle_{\mathcal{H}} \,. \tag{2.14}$$

Ejemplo 2.2.11. *If $\mathcal{H} = \mathbb{R}^H$, then: $\mathcal{K}(\boldsymbol{x}^i, \boldsymbol{x}^j) = \langle \phi(\boldsymbol{x}^i), \phi(\boldsymbol{x}^j) \rangle_{\mathbb{R}^H} = \phi(\boldsymbol{x}^i)^T \cdot \phi(\boldsymbol{x}^j)$.*

These *kernel* functions can be interpreted as similarity functions, understood as the similarity implicitly defined by the extensions, but they are simply functions $\mathcal{K} : X \times X \to \mathbb{R}$. An example of such functions are the polynomial kernels of order k, presented in 2.2.12.

Ejemplo 2.2.12. *Polynomial kernels of order k are functions $\mathcal{K}(\boldsymbol{x}^i, \boldsymbol{x}^j) = ((\boldsymbol{x}^i)^T \cdot \boldsymbol{x}^j)^k$. It can be easily proven that, indeed, there exist extensions $\phi : \mathbb{R}^M \to \mathbb{R}^H$ that satisfy the relation (2.14), with $H = M^k$:*

$$\begin{aligned}
\mathcal{K}(\boldsymbol{x}^i, \boldsymbol{x}^j) &= ((\boldsymbol{x}^i)^T \cdot \boldsymbol{x}^j)^k = ((\boldsymbol{x}^i)^T \cdot \boldsymbol{x}^j) \cdot \ldots \cdot ((\boldsymbol{x}^i)^T \cdot \boldsymbol{x}^j) = \\
&= \left(\sum_{l=1}^{M} x_l^i x_l^j \right) \cdot \ldots \cdot \left(\sum_{l=1}^{M} x_l^i x_l^j \right) = \sum_{J \in \{1,\ldots,M\}^k} \prod_{m=1}^{k} x_{J_m}^i x_{J_m}^j = \\
&= \sum_{J \in \{1,\ldots,M\}^k} \prod_{m=1}^{k} x_{J_m}^i \prod_{m=1}^{k} x_{J_m}^j \implies \phi(\boldsymbol{x}) = \prod_{m=1}^{k} x_{J_m} \quad \textit{with} \quad \phi : \mathbb{R}^M \to \mathbb{R}^{M^k} \,.
\end{aligned}$$

2.2.2.1 Kernel Methods in Support Vector Machines

As mentioned, the application of kernel methods to support vector machines not only improves the classification algorithm but also allows classifying sets that were previously unclassifiable. A necessary condition for a linear support vector machine to classify a training set $T = \{(\boldsymbol{x}^1, y^1), \dots, (\boldsymbol{x}^N, y^N)\}$ was that it be linearly separable, as indicated in SECTION *2.2.1.* With the introduction of functions $\phi : X \to \mathcal{H}$, this condition is no longer necessary, and the condition now imposed is that the training set $T_\phi = \{(\phi(\boldsymbol{x}^1), y^1), \dots, (\phi(\boldsymbol{x}^N), y^N)\}$ be linearly separable.

On the other hand, kernel functions $\mathcal{K}(\boldsymbol{x}^i, \boldsymbol{x}^j) = \langle \phi(\boldsymbol{x}^i), \phi(\boldsymbol{x}^j) \rangle_{\mathcal{H}}$ do not arise artificially but from considering the optimization problems (2.12) and (2.13) that define support vector machines. In particular, an adaptation of problem (2.13) is used, but considering the minimization of the square instead of the maximization of the inverse, i.e., an equivalent optimization problem given by[1]:

$$\begin{array}{ll} \min & \left\{ \frac{1}{2} ||\boldsymbol{w}||^2 \right\} \\ \text{s.a} & y^i \cdot (w_0 + \phi(\boldsymbol{x}^i)^T \cdot \boldsymbol{w}) \geq 1 \quad \text{for all } i \in \{1, \dots, N\} \end{array} \tag{2.15}$$

Proposición 2.2.13. *Given a binary classification problem with classes* $+1$ *and* -1*, and a training set* $T = \{(\boldsymbol{x}^1, y^1), \dots, (\boldsymbol{x}^N, y^N)\}$ *with* $\boldsymbol{x}^i \in \mathbb{R}^M$ *for all* $i \in \{1, \dots, N\}$*. Assuming an extension to a real space of finite dimension* $H > M$*,* $\phi : \mathbb{R}^M \to \mathbb{R}^H$*, then the prediction function* $\hat{f} : \mathbb{R}^M \to \mathbb{R}$ *obtained by the support vector machine is:*

$$\hat{f}(\boldsymbol{x}) = \frac{1}{|\mathcal{G}|} \cdot \left[\sum_{i \in \mathcal{G}} y^i - \sum_{i,j \in \mathcal{G}} u_j y^j \mathcal{K}(\boldsymbol{x}^j, \boldsymbol{x}^i) \right] + \sum_{i \in \mathcal{G}} u_i y^i \mathcal{K}(\boldsymbol{x}, \boldsymbol{x}^i) \,, \tag{2.16}$$

where $u_i \geq 0$ *for all* $i \in \{1, \dots, N\}$ *and* $\mathcal{G}$ *is the set of active constraints, i.e.,* $j \in \mathcal{G}$ *if* $u_j \neq 0$*. Additionally, the kernel* $\mathcal{K} : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}$ *is considered, with* $\mathcal{K}(\boldsymbol{x}^i, \boldsymbol{x}^j) = \langle \phi(\boldsymbol{x}^i), \phi(\boldsymbol{x}^j) \rangle_{\mathbb{R}^H}$*.*

Proof. The usual inner product in $\mathbb{R}^H$ is considered: $\langle \phi(\boldsymbol{x}^i), \phi(\boldsymbol{x}^j) \rangle_{\mathbb{R}^H} = \phi(\boldsymbol{x}^i)^T \cdot \phi(\boldsymbol{x}^j)$. Let the optimization problem (2.15) be considered, now taking into account the KKT conditions (Karush-Kuhn-Tucker, see A.3 in the appendix chapter), with $\boldsymbol{u} = (u_1, \dots, u_N) \in \mathbb{R}$, the Lagrangian function is of the form:

$$L(w_0, \boldsymbol{w}, \boldsymbol{u}) = \frac{1}{2} ||\boldsymbol{w}||^2 + \sum_{i=1}^{N} u_i \cdot \left[1 - y^i \cdot (w_0 + \phi(\boldsymbol{x}^i)^T \cdot \boldsymbol{w}) \right] \,.$$

Differentiating this relation with respect to $\boldsymbol{w}$ and w_0 and setting both to zero, two of the conditions to be satisfied for minimization are obtained:

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{N} u_i y^i \phi(\boldsymbol{x}^i) = 0 \implies \boldsymbol{w} = \sum_{i=1}^{N} u_i y^i \phi(\boldsymbol{x}^i) \,.$$

[1]A scaling factor of 1/2 is multiplied, which does not affect the optimization [26].

$$\frac{\partial L}{\partial w_0} = -\sum_{i=1}^{N} u_i y^i = 0 \implies \sum_{i=1}^{N} u_i y^i = 0\,.$$

Substituting these results into the prediction function (2.11) modified by $\phi(\boldsymbol{x}^i)$:

$$\hat{f}(\boldsymbol{x}) = w_0 + \phi(\boldsymbol{x})^T \cdot \boldsymbol{w} = w_0 + \phi(\boldsymbol{x})^T \cdot \sum_{i=1}^{N} u_i y^i \phi(\boldsymbol{x}^i) = w_0 + \sum_{i=1}^{N} u_i y^i \mathcal{K}(\boldsymbol{x}, \boldsymbol{x}^i)\,. \tag{2.17}$$

Additionally, since the relation $u_i \cdot \left[1 - y^i \cdot (w_0 + \phi(\boldsymbol{x}^i)^T \cdot \boldsymbol{w})\right] = 0$ must hold for all $i \in \{1, \dots, N\}$, the active constraints of $\mathcal{G}$, i.e., those j such that $u_j \neq 0$, will be those that satisfy $y^i \cdot (w_0 + \phi(\boldsymbol{x}^i)^T \cdot \boldsymbol{w}) = 1$, allowing the value of w_0 to be obtained:

$$y^i \cdot (w_0 + \phi(\boldsymbol{x}^i)^T \cdot \boldsymbol{w}) = 1 \implies (y^i)^2 \cdot (w_0 + \phi(\boldsymbol{x}^i)^T \cdot \boldsymbol{w}) = y^i \implies w_0 = y^i - \phi(\boldsymbol{x}^i)^T \cdot \boldsymbol{w} \implies$$
$$\implies |\mathcal{G}| \cdot w_0 = \sum_{i \in \mathcal{G}} y^i - \phi(\boldsymbol{x}^i)^T \cdot \boldsymbol{w}\,.$$

Thus, using the relation for $\boldsymbol{w}$ in terms of the u_i, we have:

$$w_0 = \frac{1}{|\mathcal{G}|} \cdot \left[\sum_{i \in \mathcal{G}} y^i - \sum_{i \in \mathcal{G}} \phi(\boldsymbol{x}^i)^T \boldsymbol{w}\right] = \frac{1}{|\mathcal{G}|} \cdot \left[\sum_{i \in \mathcal{G}} y^i - \sum_{i,j \in \mathcal{G}} u_j y^j \mathcal{K}(\boldsymbol{x}^j, \boldsymbol{x}^i)\right]\,.$$

Finally, the other summation in equation (2.17) can be transformed into the set of indices $\mathcal{G}$, since $u_j = 0$ if $j \notin \mathcal{G}$, resulting in the desired relation. ■

Observación 2.2.14. *Proposition 2.2.13 can be generalized to the case $\phi : \mathbb{R}^M \to \mathcal{H}$ with $\mathcal{H}$ a Hilbert space, also replacing the inner product with the corresponding one in $\mathcal{H}$. The proof of the generalized proposition can be found in [26].*

The modification of support vector machines using *kernel* functions is particularly useful because the prediction function $\hat{f}$ has a lower computational cost. This is because only the active constraints $u_j \neq 0$ are considered in its calculation, or equivalently, the training data that satisfy $y^j \cdot \big(w_0 + \phi(\boldsymbol{x}^j)^T \cdot \boldsymbol{w}\big) = 1$.

Ultimately, the optimization problem (2.15) is then transformed into the following optimization problem, which depends on the parameters $u_i \geq 0$ with $i \in \{1, \dots, N\}$ and whose solution can be carried out using the SMO algorithm [26]:

$$\begin{aligned} \max \quad & \left\{\sum_{i=1}^{N} u_i - \frac{1}{2} \sum_{i,j=1}^{N} u_i u_j y^i y^j \mathcal{K}(\boldsymbol{x}^i, \boldsymbol{x}^j)\right\} \\ \text{s.t.} \quad & \sum_{i=1}^{N} u_i y^i = 0 \\ & u_i \geq 0\,, \quad \text{for all } i \in \{1, \dots, N\} \end{aligned} \tag{2.18}$$

Finally, it is worth noting that not all *kernels* are valid because not all will lead to well-defined optimization problems. In fact, the appropriate *kernels* are those that satisfy the following Mercer's Theorem, whose proof can be found in [25].

Teorema 2.2.15. *A function $\mathcal{K} : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}$ is a valid kernel if and only if, for any $X = \{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^N\} \subseteq \mathbb{R}^M$, the matrix $K = \big(\mathcal{K}(\boldsymbol{x}^i, \boldsymbol{x}^j)\big)_{i,j=1}^N$ is symmetric and positive semidefinite.*

Chapter 3

Artificial Neural Networks

In order to emulate the functioning of the human brain, artificial neural networks (ANN) began to be developed computationally in the mid-20th century, with the work of Frank Rosenblatt standing out, as he implemented the first artificial neural network in [31].

An artificial neural network is a model designed to simulate the brain's processing of a specific task through a "learning" process that uses interconnected computational cells called artificial neurons. "Knowledge" is acquired through a learning algorithm, such as the supervised learning described in CHAPTER *2*, and is stored in the neurons through synaptic weights, which determine interneuron connections.

Artificial neural networks have fundamental properties that increase their utility compared to computer programs or individual neurons [13]:

- **Generalization**: Neural networks can produce results similar to those from the learning phase but with different input data, allowing them to make predictions.
- **Non-linearity**: Due to their structure and construction, neural networks enable the development of complex algorithms with non-linear properties that describe physical and real systems with great precision.
- **Adaptability**: The synaptic weights of the network can be modified as the model is retrained, allowing adaptation to non-stationary systems.

An artificial neuron, or node, is the fundamental unit that makes up an artificial neural network. It consists of a set of connections with synaptic weights, a linear combiner, and a not necessarily linear activation function that modifies the output value and limits its range. A bias may also be included to alter the output value of the linear combiner before applying the activation function.

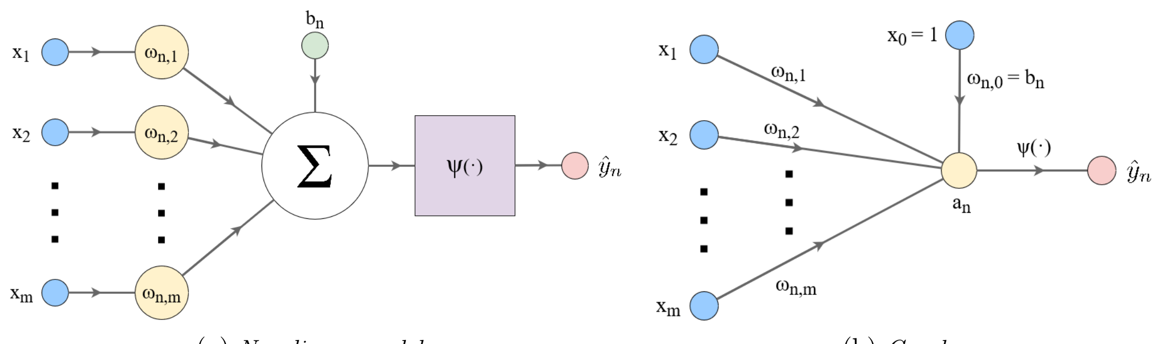
Formally, for a neuron n, there are input values $x_j \in \mathbb{R}$ with $j \in \{1, \dots, m\}$ and m the number of connections incident to the neuron. Additionally, the synaptic weights $w_{n,j} \in \mathbb{R}$ are associated with the neuron n and the input connection j, while the linear combiner produces the activation $a_n \in \mathbb{R}$, i.e., the output value before the activation function:

$$a_n = \sum_{j=1}^{m} w_{n,j} x_j \ .$$

A bias $b_n \in \mathbb{R}$ is added to this, and an activation function $\psi : \mathbb{R} \to \mathbb{R}$ is applied, so the output of the neuron $\hat{y}_n$ is (see Figure 3.1a):

$$\hat{y}_n = \psi(a_n + b_n) = \psi\left(\sum_{j=1}^{m} w_{n,j} x_j + b_n\right) \ . \tag{3.1}$$

In reality, the bias can be interpreted as an additional input, with value $x_0 = 1$ and synaptic weight $w_{n,0} = b_n$. Furthermore, this representation of the neuron is nothing more than a simple directed graph (see A.4 in the appendix chapter, which includes the graph theory concepts used) with $m + 3$ vertices, corresponding to the $m + 1$ input variables (including the bias), the operator vertex, and the output vertex (see Figure 3.1b). It has $m + 2$ arcs: $m + 1$ synaptic arcs with weights $w_{n,j}$ and the remaining activation arc that applies the activation function $\psi(\cdot)$.



(a) *Non-linear model.* (b) *Graph.*

Figure 3.1: *Representations of neuron n.*

With this interpretation of neurons as graphs, the concept of a complete neural network is reached, which can be simplified to a partially complete neural network if each neuron is considered as a single vertex and the activation arc is implied (see Figure 3.2). Additionally, in these networks, biases are included in the source nodes as input values for each neuron.

Definición 3.0.1. *A partially complete neural network is a directed graph $G = \{V, A\}$ with $|V|$ the number of vertices and $|A|$ the number of arcs, where $V = V_s \cup V_n$ with $|V_s|$ the number of source nodes and $|V_n|$ the number of neurons. Additionally, the vertices V_s have no predecessors but have at least one successor, and the vertices V_n must have at least one predecessor.*

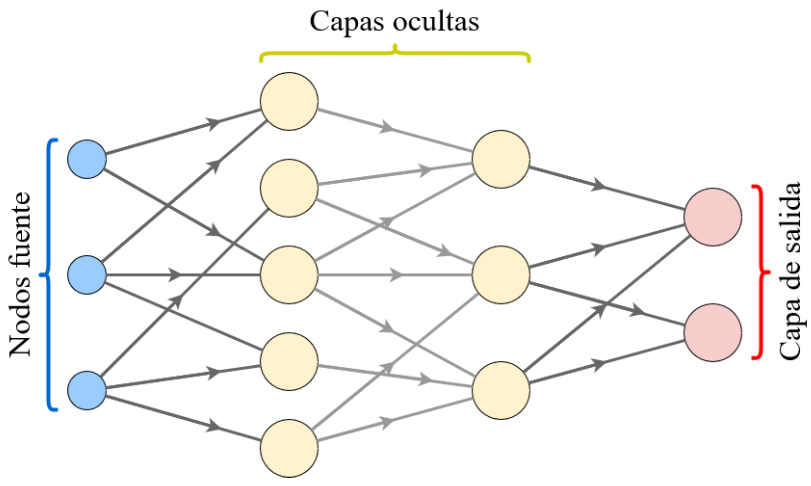


Figure 3.2: *Representation as a partially complete graph of a neural network.*

Observación 3.0.2. *The concept of a complete neural network has nothing to do with whether the graph representing it is complete or not. In fact, a complete neural network can also be defined as a directed graph $G = \{V, A\}$ that satisfies $V = V_s \cup V_{n,w} \cup V_{n,\psi}$ and $A = A_w \cup A_\psi$, with $|V_{n,w}| = |V_{n,\psi}|$, $|A_w|$ the number of synaptic arcs, and $|A_\psi|$ the number of activation arcs. Additionally, the vertices $V_{n,w}$ have a single successor belonging to $V_{n,2}$, and the vertices $V_{n,\psi}$ have a single predecessor belonging to $V_{n,w}$. The arcs incident to $V_{n,w}$ are synaptic arcs, and the arcs incident to $V_{n,\psi}$ are activation arcs.*

Generally, the V_s that are not biases belong to the input layer, the V_n that have no successors belong to the output layer, and the rest of the V_n belong to the hidden layers. The depth of the network, denoted as Z, is defined as the number of layers excluding the layer with the source nodes, while the width of the network is $\max_{z \in \{1,\dots,Z\}} |V_z|$. Finally, the size of the network is simply $|V|$, the number of neurons and source nodes. These parameters are referred to as the architecture of the neural network, and, as an example, the network in Figure 3.2 has a size of 13, a depth of 3, and a width of 5.

A layer is said to be fully connected (*fully connected layer*, in English) if each of its neurons is connected to all the neurons in its previous layer and all the neurons in its subsequent layer. If this occurs for all layers of the network, it is a fully connected neural network, i.e., *fully connected neural network*.

The use of this type of network is very common because it does not assume a specific arrangement or properties of the network. All input values affect all neurons and all output values, with the network itself deciding which connections are relevant and which are not. However, these networks have more difficult convergence compared to other networks where some connections are not possible, as they depend on a larger number of parameters [12].

The neural networks described as graphs can be classified into two types: those without cycles and those with cycles. A network like the one in Figure 3.2 is called feedforward, as its arcs only affect neurons in subsequent layers, whereas if cycles exist, i.e., there are arcs affecting neurons in previous layers, the network is recurrent.

Once the network has obtained output values, it is necessary to define a method to validate that the network has improved and "acquired knowledge." For this purpose, a total error of the network for a training session is defined, which is intended to be minimized: a loss function dependent on the synaptic weights is defined, which must be minimized and is closely related to the activation functions of the output layer [30]. This is nothing more than what was described in Chapter *2* as the error of supervised learning, but applied to the entire neural network.

Returning to the concepts developed for supervised learning, a training session is considered that aims to minimize the given loss function $\mathcal{E}(\boldsymbol{w})$, where $\boldsymbol{w}$ is a vector with the weights of the neural network. A local minimum of the loss function is a value $\boldsymbol{w}^*$ such that $\nabla\mathcal{E}(\boldsymbol{w}^*) = 0$, but in general, analytical solutions to this relationship cannot be found. Therefore, it is useful to resort to numerical methods of the form:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \boldsymbol{\Delta}^{(t)} , \quad \text{with } t = 1, 2, \dots \tag{3.2}$$

where t indicates the iterative step, $\boldsymbol{w}^{(0)}$ is an initial vector of weights conveniently chosen (see Section *3.3.1*), and $\boldsymbol{\Delta}^{(t)}$ is the modification of $\boldsymbol{w}^{(t)}$ in each iteration t.

Observación 3.0.3. *Although the goal of training a neural network is to find the global minimum of the loss function, in practice, it is sufficient to find several local minima and compare them, as the existence of a global minimum is not even guaranteed [4].*

3.1 Gradient Descent Optimization

The simplest way to iterate the relationship (3.2) is by considering $\boldsymbol{\Delta}^{(t)} = -\eta \cdot \nabla\mathcal{E}\left(\boldsymbol{w}^{(t)}\right)$, where $\eta > 0$ is the "learning rate." This simply involves taking a small step in the direction in which the error decreases, i.e., a step in the negative direction of the gradient. This technique is known as gradient descent, and it is not computationally efficient because, since the loss function is defined for a training set $T = \{(\boldsymbol{x}^1, y^1), \dots, (\boldsymbol{x}^N, y^N)\}$, the method requires evaluating the gradient of the loss function over the entire set.

Thus, for neural networks trained with large datasets T, i.e., $N \gg 1$, a stochastic gradient descent method is considered, which iterates over the weights for each element of T:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \cdot \nabla \mathcal{E}_i \left(\boldsymbol{w}^{(t)} \right) \quad \text{with} \quad \mathcal{E}(\boldsymbol{w}) = \sum_{i=1}^{N} \mathcal{E}_i(\boldsymbol{w}) \, , \tag{3.3}$$

where $\mathcal{E}_i(\boldsymbol{w})$ is the loss function that would be obtained if the training set $T_i = \{(\boldsymbol{x}^i, y^i)\}$ were considered for a given i. Equivalently, $\mathcal{E}_i(\boldsymbol{w})$ is the contribution of the input data $\boldsymbol{x}^i$ to the loss function $\mathcal{E}(\boldsymbol{w})$.

This modification simplifies the computational complexity of the optimization algorithm by requiring fewer operations per iteration. Additionally, the stochastic method is more effective for training sets with repeated or very similar elements and also prevents the optimization from getting stuck in a local minimum.

A modification to the stochastic gradient descent method involves introducing "batches," which are subsets of the training set T over which an iteration is performed. That is, in batch gradient descent, subsets $T_1, \ldots, T_B \subseteq T$ are considered, and equation (3.3) is modified so that in each iteration t, the loss function obtained for one of these batches, $\mathcal{E}_{T_b}(\boldsymbol{w})$, is used, with $b \in \{1, \ldots, B\}$.

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \cdot \nabla \mathcal{E}_{T_b} \left(\boldsymbol{w}^{(t)} \right) \quad \text{with} \quad T_b \subseteq T \, . \tag{3.4}$$

Observación 3.1.1. *Stochastic gradient descent can be understood as a type of batch gradient descent where there are N batches, each containing a single data point.*

The concept of a batch should not be confused with that of an "epoch." An epoch is a complete pass through the training set T, i.e., an epoch is considered complete when each $(\boldsymbol{x}^i, y^i) \subseteq T$ has been used in at least one iteration of the optimization algorithm.

Frequently, when training a model with a training set T, B batches $T_1, \ldots, T_B$ are considered, disjoint and such that $T_1 \cup \ldots \cup T_B = T$. Each of these batches is used in an iteration of the weights $\boldsymbol{w}$, not necessarily in order, resulting in weights $\boldsymbol{w}^{(t_1)}, \ldots, \boldsymbol{w}^{(t_B)}$. When all B iterations have been performed, it is said that an epoch ϵ has been completed. The iterative training process completes epochs $\epsilon_1, \ldots, \tilde{\epsilon}$ as it trains with the complete set T. Additionally, in each epoch, different batches can be used in an arbitrary order.

3.1.1 *Adam* Optimizer

The stochastic gradient descent method (3.3) has a main problem related to convergence: if a low learning rate is considered, the variation in synaptic weights is smaller and more stable, but convergence is slow; if a high learning rate is considered, the variation in synaptic weights is larger and more unstable, but convergence is faster.

Observación 3.1.2. *Recently, results have emerged supporting the potential long-term benefits of local instabilities, such as those by Cohen et al. [6]. To take advantage of them, a dynamic learning rate can be introduced, as presented in [33].*

Classically, efforts have been made to modify this method to achieve fast convergence while avoiding instability. For this purpose, a "momentum constant" $\alpha \in (0,1)$ is included, which modifies the iterative term by adding a weighting with the result from the previous iteration:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \boldsymbol{\Delta}^{(t)} \quad \text{with} \quad \boldsymbol{\Delta}^{(t)} = \alpha \boldsymbol{\Delta}^{(t-1)} - \eta \cdot \nabla \mathcal{E}_i \left(\boldsymbol{w}^{(t)} \right) . \tag{3.5}$$

Thus, the relationship can be written recursively as:

$$\boldsymbol{\Delta}^{(t)} = -\eta \sum_{s=1}^{t} \alpha^{t-s} \cdot \nabla \mathcal{E}_i \left(\boldsymbol{w}^{(t-s)} \right) .$$

This implies that if the gradient takes a certain sign in several consecutive iterations, the momentum causes the term $\boldsymbol{\Delta}^{(t)}$ to increase and accelerates the descent. On the other hand, if the gradient oscillates significantly in sign for nearby iterations, the momentum term induces stabilization, reducing the value of $\boldsymbol{\Delta}^{(t)}$.

The *Adam* method is a modification of the momentum method (3.5) that introduces parameters β_1 and β_2, close to 1, to control the decay of $\boldsymbol{\Delta}^{(t)}$ and its square $\left(\boldsymbol{\Delta}^{(t)}\right)^2$. These terms are called first and second-order moments, respectively.

This optimization method, presented by Diederick P. Kingma and Jimmy Lei Ba in [17], is widely used in the development of predictive models due to its fast convergence and adaptability, also being affected by a small parameter $\varepsilon \sim 10^{-8}$:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \cdot \frac{\dfrac{\boldsymbol{\Delta}^{(t)}}{1-\beta_1^t}}{\sqrt{\dfrac{\left(\boldsymbol{\Delta}^{(t)}\right)^2}{1-\beta_2^t}} + \varepsilon} \quad \text{with} \quad \begin{cases} \boldsymbol{\Delta}^{(t)} = \beta_1 \boldsymbol{\Delta}^{(t-1)} - (1-\beta_1) \nabla \mathcal{E}_i \left(\boldsymbol{w}^{(t)} \right) \\ \left(\boldsymbol{\Delta}^{(t)}\right)^2 = \beta_2 \left(\boldsymbol{\Delta}^{(t-1)}\right)^2 - (1-\beta_2) \left(\nabla \mathcal{E}_i \left(\boldsymbol{w}^{(t)} \right) \right)^2 \end{cases} .$$

3.2 Regularization by *Dropout*

Once the learning of a model built with a neural network has been optimized, the problem discussed in SECTION *2.1.2* arises: generalization. In that section, a general regularization technique for supervised learning models was described. This included a regularization parameter on the model's error, which, for a neural network, corresponds to the loss function, modifying the error so that the weights tend to be smaller.

Another common regularization technique is called *dropout*, which prevents overfitting by randomly eliminating a certain number of connections. The eliminated connections are different in each iteration, so this method reduces the likelihood of parameters learning only noise or specific details about the training data. However, it should be noted that applying these techniques introduces an additional error to the network that is not negligible.

In practice, to apply dropout techniques, a dropout layer can be considered, which sets a certain proportion c of the input values for that layer to 0, varying the values set to zero in each training iteration. In this way, for low proportions ($c \ll 1$), all connections train in some epoch, but not simultaneously, and the input values that are used are rescaled by multiplying them by $(1-c)^{-1}$ so that the sum of all input terms to the layer remains invariant.

It is worth noting that these dropout techniques by layer are more effective and less costly than other regularization techniques, such as weight decay, as demonstrated by Srivastava *et al.* in [37]. This implies that they have numerous applications and allow the development of more complex models while maintaining convergence.

3.3 Characteristics of a Neural Network

As expressed in the general description of a neural network, certain characteristics effectively determine the functioning and convergence of the network. Therefore, these characteristics must be determined accurately to ensure the network is suitable, and for this, it is necessary to understand the available options and which are optimal depending on the type of learning and network used. Among these characteristics, the following stand out: the initialization distribution, the activation function, and the loss function.

3.3.1 Initialization Distribution

To begin the iterative training process, it is necessary to define initial synaptic weights, $\boldsymbol{w}^{(t)}$, so that their optimization is as efficient as possible. For this purpose, there are various heuristic strategies that indicate initialization distributions that improve training performance.

It is worth noting that this task is very relevant, as the same model may or may not converge depending on the initial weights considered. Therefore, it is avoided that the initial weights are large, as in recurrent networks, this tends to cause instabilities, and it is also avoided that two neurons with the same activation function and connected to the same variable have the same initial values, as they would then tend to update in the same way constantly.

In general, weight initialization is usually random, determined by a Gaussian or uniform distribution. The uniform distribution is such that its density function takes the value $(b-a)^{-1}$ in the interval $[a, b]$ and is zero elsewhere. A common initialization distribution for a fully connected network with M input values and S output values is the uniform distribution: $U(-M^{-1/2}, M^{-1/2})$. The utility of this distribution was described by Xavier Glorot and Joshua Bengio in [11].

Furthermore, a normalized version of this initialization distribution is even more useful for highly interconnected networks, known as Glorot normal initialization (*Glorot normal*, in English) and determined by the uniform distribution:

$$w_{n,j} \sim U\left(-\sqrt{\frac{6}{M+S}}\,,\,\sqrt{\frac{6}{M+S}}\right) . \tag{3.6}$$

3.3.2 Activation Function

The primary activation function in the output layer for regression models is the identity function $\psi(a) = a$, but there are some activation functions that improve performance in different models, as they introduce non-linearities [13]:

- **Logistic sigmoid function**: This function takes values in the range $(0, 1)$ and, with $\alpha = 1$, is the usual function in the output layer for binary classification models. It is a smooth approximation of the Heaviside function $\psi(a)$, which takes the value 1 if $a \geq 0$ and 0 otherwise. In fact, the Heaviside function is recovered as the parameter $\alpha \to \infty$:

$$\sigma(a) = \frac{1}{1+\exp(-\alpha a)} . \tag{3.7}$$

  It can be observed that this expression is directly derived from Bayesian models for binary classification with supervised learning, described in SECTION *2.2*.

- ***Softmax* function**: This is a generalization of the logistic sigmoid function and also represents a smooth version of the maximum function [16]. As deduced in SECTION *2.2*, it makes sense to use it in the output layer for multiclass classification with K classes, where the logistic sigmoid function is recovered if $K = 2$. The main difference with the logistic sigmoid is that, in this case, the function considers the activations $\boldsymbol{a} = (a_1, \ldots, a_K)$ of the K neurons in that layer and produces a vector with the probabilities associated with each class:

$$\text{Softmax}(\boldsymbol{a}) = \left(\frac{\exp(a_1)}{\sum_{n=1}^{K}\exp(a_n)}\,,\,\ldots\,,\,\frac{\exp(a_K)}{\sum_{n=1}^{K}\exp(a_n)}\right)^T . \tag{3.8}$$

- **Rectified Linear Unit (ReLU)**: This is a modified and continuous version of the Heaviside function that minimizes errors in large neural networks and does not modify zero activations, so it is used in hidden layers. It is also notable for its use in the output layer for regression models with non-negative values [26]:

$$\mathrm{ReLU}(a) = \begin{cases} 0 & \text{if} \quad a < 0 \\ a & \text{if} \quad a \geq 0 \end{cases} . \tag{3.9}$$

3.3.3 Loss Function

The loss function of a neural network for a training set with N independent observations, $T = \{(\boldsymbol{x}^1, y^1), \ldots, (\boldsymbol{x}^N, y^N)\}$, can be defined in general as was done in (2.1), i.e., as the sum of the distances between the predicted and actual results for each data point $\boldsymbol{x}^i \in X$. However, for a study where the output layer of the neural network has a specific activation function ψ, the loss function is defined by it [4]:

- **Mean Squared Error**: For a regression problem, what was expressed in Section *2.1.1* for an arbitrary prediction function $\hat{f} : X \to \mathbb{R}$ is generalized. Then, the loss function is defined, up to a constant, as the negative natural logarithm of the maximum likelihood function (2.3), i.e., it is given by the generalization of the expression (2.6):

  $$\mathcal{E}_{X,Y}(\hat{f}) = \frac{1}{2} \sum_{i=1}^{N} \left(y^i - \hat{f}(\boldsymbol{x}^i) \right)^2 . \tag{3.10}$$

  This loss function is commonly known as the squared error, but it is more convenient to use the average over the number of training values, the mean squared error (MSE):

  $$\mathcal{E}_{X,Y}(\hat{f}) = \frac{1}{N} \sum_{i=1}^{N} \left(y^i - \hat{f}(\boldsymbol{x}^i) \right)^2 . \tag{3.11}$$

- **Cross-Entropy**: For a binary classification problem with two classes C_0 and C_1, where $y^i = 0$ or $y^i = 1$, respectively, the posterior probability of one of the classes, $p\left(C_1|\boldsymbol{x}^i\right)$, is given by equation (2.8). Then, it can be deduced that the probability of the other class is $p\left(C_0|\boldsymbol{x}^i\right) = 1 - p\left(C_1|\boldsymbol{x}^i\right)$. Furthermore, since $\hat{f}(\boldsymbol{x}^i) \in [0,1]$, as it is obtained from the logistic sigmoid activation function (3.7) of the output layer, $\hat{f}(\boldsymbol{x}^i)$ can be interpreted as the conditional probability $p\left(C_1|\boldsymbol{x}^i\right)$.

  From equation (2.3), the likelihood function is defined in this case as the product of independent Bernoulli distributions. This is because the distribution of a random variable $\mathcal{Y}$ with two discrete possibilities, 0 and 1, depending on a parameter $q \in (0,1)$, is a Bernoulli with $p\left(\mathcal{Y} = y\right) = q^y (1-q)^{1-y}$. Ultimately, the obtained likelihood function is:

  $$\mathcal{L}(\hat{f}) = \prod_{i=1}^{N} p\left(\hat{f}(\boldsymbol{x}^i) = y^i\right) = \prod_{i=1}^{N} \hat{f}(\boldsymbol{x}^i)^{y^i} \cdot \left(1 - \hat{f}(\boldsymbol{x}^i)\right)^{1-y^i} . \tag{3.12}$$

The cross-entropy loss function is defined, up to a constant, as the negative natural logarithm of the previous likelihood function:

$$\mathcal{E}_{X,Y}(\hat{f}) = -\sum_{i=1}^{N} \left[y^i \cdot \log\left(\hat{f}(\boldsymbol{x}^i)\right) + (1-y^i) \cdot \log\left(1-\hat{f}(\boldsymbol{x}^i)\right) \right] . \tag{3.13}$$

- **Categorical Cross-Entropy**: For a multiclass classification problem with K disjoint classes, i.e., mutually exclusive, the correct label $\boldsymbol{y}^i$ will be a vector of dimension K with all elements zero except for $y^i_k = 1$ if the label belongs to class C_k. Now, since the vector $\hat{\boldsymbol{f}}(\boldsymbol{x}^i) = \left(\hat{f}_1(\boldsymbol{x}^i), \dots, \hat{f}_K(\boldsymbol{x}^i)\right) \in [0,1]^K$ is determined by the *softmax* function, it is a vector of dimension K whose terms sum to 1 and are between 0 and 1, so it can be interpreted as a vector with the probabilities of the different classes. In this situation, the likelihood function is defined as:

$$\mathcal{L}(\hat{f}) = \prod_{i=1}^{N} p\left(\hat{\boldsymbol{f}}(\boldsymbol{x}^i) = \boldsymbol{y}^i\right) = \prod_{i=1}^{N}\prod_{k=1}^{K} p\left(\hat{f}_k(\boldsymbol{x}^i) = y^i_k\right) = \prod_{i=1}^{N}\prod_{k=1}^{K} \left[\hat{f}_k(\boldsymbol{x}^i)\right]^{y^i_k} . \tag{3.14}$$

Considering negative natural logarithms on both sides, the categorical cross-entropy loss function is obtained:

$$\mathcal{E}_{X,Y}(\hat{f}) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y^i_k \cdot \log\left(\hat{f}_k(\boldsymbol{x}^i)\right) . \tag{3.15}$$

In the case of imbalanced classes, i.e., when some classes have more elements than others, it is interesting to modify the loss function (3.15) with a balanced function. For this purpose, constant terms $W_1, \dots, W_K$ are added, which weight the contribution of each class to the overall loss function differently:

$$\mathcal{E}_{X,Y}(\hat{f}) = -\sum_{i=1}^{N}\sum_{k=1}^{K} W_k \cdot y^i_k \cdot \log\left(\hat{f}_k(\boldsymbol{x}^i)\right) . \tag{3.16}$$

Observación 3.3.1. *In practice, the weights $W_1, \dots, W_K$ should be larger the fewer the number of elements in the training set for a given class, as this overweights an error in its prediction and counteracts the imbalance.*

3.4 Feedforward Networks

As described at the beginning of this chapter, feedforward neural networks are graphs whose arcs only connect consecutive layers, i.e., they connect the input layer to the first hidden layer, the z-th hidden layer to the $(z+1)$-th, and the last hidden layer to the output layer. This implies that these networks cannot have cycles and that they only transmit information "forward" [26].

This type of network can be interpreted as a composition of different applications for each layer z of size $|V_z|$ of the network, $\xi^{(z)} : \mathbb{R}^{|V_{z-1}|} \to \mathbb{R}^{|V_z|}$ with $z \in \{1, \dots, Z\}$. In fact, the applications

$\xi^{(z)}$ are described by a composition of an affine application $\boldsymbol{a}^{(z)} : \mathbb{R}^{|V_{z-1}|} \to \mathbb{R}^{|V_z|}$ and an activation application $\Psi^{(z)} : \mathbb{R}^{|V_z|} \to \mathbb{R}^{|V_z|}$.

The affine application is such that $\boldsymbol{a}^{(z)}\left(\hat{\boldsymbol{y}}^{(z-1)}\right) = W^{(z)} \cdot \hat{\boldsymbol{y}}^{(z-1)} + \boldsymbol{b}^{(z)}$, where $\boldsymbol{b}^{(z)} \in \mathbb{R}^{|V_z|}$ are the biases of the z-th layer and $W^{(z)} = \left(w_{n,j}^{(z)}\right) \in \mathbb{R}^{|V_z|\times|V_{z-1}|}$ are the synaptic weights, while $\hat{\boldsymbol{y}}^{(z-1)}$ is the output of the $z-1$ layer, or equivalently, the input of the z layer. On the other hand, the activation application consists of activation functions $\psi_n^{(z)}$, with $n \in \{1, \dots, |V_z|\}$, real and not necessarily equal:

$$\Psi^{(z)}\left(\boldsymbol{a}^{(z)}\right) = \left(\psi_1^{(z)}\left(a_1^{(z)}\right), \dots, \psi_{|V_z|}^{(z)}\left(a_{|V_z|}^{(z)}\right)\right)^T .$$

Ultimately, for a neural network, its prediction function $\hat{f} : \mathbb{R}^M \to \mathbb{R}^S$ can be defined based on an input data vector $\boldsymbol{x} \in \mathbb{R}^M$:

$$\hat{f}(\boldsymbol{x}) = \xi^{(Z)} \circ \ldots \circ \xi^{(1)}(\boldsymbol{x}) = \left(\Psi^{(Z)} \circ \boldsymbol{a}^{(Z)}\right) \circ \ldots \circ \left(\Psi^{(1)} \circ \boldsymbol{a}^{(1)}\right)(\boldsymbol{x}) .$$

It is important to realize that, in fact, $\hat{f}(\boldsymbol{x}) = \hat{\boldsymbol{y}}^{(Z)}$, the vector of the output of the final layer, and, denoting $\hat{\boldsymbol{y}}^{(0)} := \boldsymbol{x}$, then for any layer z it holds that:

$$\hat{\boldsymbol{y}}^{(z)}(\boldsymbol{x}) = \xi^{(z)} \circ \ldots \circ \xi^{(1)}(\boldsymbol{x}) = \Psi^{(z)}\left(\boldsymbol{a}^{(z)}\left(\hat{\boldsymbol{y}}^{(z-1)}\right)\right) . \tag{3.17}$$

With a description of the feedforward network as a composition of functions, its utility for approximating other more complex functions must be proven. This is what the universal approximation theorems do, providing a foundation for this type of network by demonstrating that the approximation of functions using feedforward networks with the appropriate parameters is possible. It is worth noting that the theorems are not constructive; they show the existence of such parameters but not how to obtain them, so it is necessary to study a given predictive model in detail to attempt to reach its optimal parameters.

3.4.1 Universal Approximation Theorems

In 1989, George V. Cybenko proved a first universal approximation theorem in [8]. It states that a feedforward neural network with a linear output layer and at least one hidden layer with a sigmoidal activation function and sufficiently many hidden neurons can approximate any continuous function, where a sigmoidal function is a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ such that $\lim_{a\to\infty} \sigma(a) = 1$ and $\lim_{a\to-\infty} \sigma(a) = 0$. Therefore, neural networks are often referred to as "universal approximators."

Cybenko's theorem is the only universal approximation theorem proven in this work, and its statement is based on the concept of a dense set:

Definición 3.4.1. *A set $A \subseteq X$ is said to be dense in X if and only if the closure of A equals X, i.e., $\bar{A} = X$.*

With this, Cybenko's universal approximation theorem is stated as follows [8]:

Teorema 3.4.2. *Consider the set of feedforward neural networks with a single hidden layer with a continuous sigmoidal activation function, input values belonging to* $[0,1]^M$*, and output in* $\mathbb{R}$*. This set is dense in the vector space of continuous functions* $f : [0,1]^M \to \mathbb{R}$ *with respect to the infinity norm,* $||f||_\infty = \sup\{|f(x)| : \boldsymbol{x} \in [0,1]^M\}$.

Furthermore, the presented universal approximation theorem is equivalent to stating that, for every continuous function $f : [0,1]^M \to \mathbb{R}$ and every $\varepsilon > 0$, there exists an $n \in \mathbb{N}$, $\boldsymbol{w}_l \in \mathbb{R}^M$, and $a_l, b_l \in \mathbb{R}$ for every $l \in \{1, \dots, n\}$, such that:

$$\left|\left|f - \hat{f}\right|\right|_\infty < \varepsilon \quad \text{for} \quad \hat{f}(\boldsymbol{x}) = \sum_{l=1}^{n} a_l \sigma\left(\boldsymbol{x}^T \cdot \boldsymbol{w}_l + b_l\right) \quad \text{with} \quad \boldsymbol{x} \in [0,1]^M . \tag{3.18}$$

Observación 3.4.3. *In this situation, it is clear that, in fact, the output of a neural network with a hidden layer of arbitrary width* n *is being described, where the activation function of the output layer is the identity. Additionally, the vectors* $\boldsymbol{w}_i$ *are the weights of the hidden layer, the terms* b_i *are the biases, and the* a_i *are the weights of the output layer.*

The proof of this theorem requires considering the Hahn-Banach and Riesz representation theorems, as well as a lemma on continuous sigmoidal functions. These three results can be found in SECTION *A.5* of the appendix chapter. Additionally, it should be recalled what a Borel measure is: μ is a measure on the Borel σ-algebra. That is, a non-negative application that assigns 0 to the empty set and whose value for a countable union of disjoint sets is equal to the sum of the values for the sets. In fact, the Borel σ-algebra is the smallest σ-algebra that contains the open sets of the space, where a σ-algebra is a non-empty family of subsets that is closed under complements and countable unions.

Proof. (Extracted from [8]). Let $\mathcal{S}$ be the set of functions $\sum_{l=1}^{n} a_l \sigma\left(\boldsymbol{x}^T \cdot \boldsymbol{w}_l + b_l\right)$, and let $\mathcal{C}\left([0,1]^M\right)$ be the vector space of continuous functions on $[0,1]^M$, then $\mathcal{S} \subset \mathcal{C}\left([0,1]^M\right)$ and is a vector subspace. Thus, to prove that it is dense, it suffices to show that its closure $\bar{\mathcal{S}} = \mathcal{C}\left([0,1]^M\right)$.

Assuming that $\bar{\mathcal{S}} \neq \mathcal{C}\left([0,1]^M\right)$ and using the Hahn-Banach theorem A.5.1, there exists a bounded linear functional $L : \mathcal{C}\left([0,1]^M\right) \to \mathbb{R}$, such that $L \neq 0$ and $L(f) = 0$ if $f \in \bar{\mathcal{S}}$.

Now, by the Riesz representation theorem A.5.2, there exists a Borel measure μ on $[0,1]^M$, such that, for every function $f \in \mathcal{C}\left([0,1]^M\right)$, it holds that:

$$L(f) = \int_{[0,1]^M} f(\boldsymbol{x}) d\mu(\boldsymbol{x}) .$$

In particular, since the function $\hat{f}(\boldsymbol{x}) = \sigma\left(b + \boldsymbol{x}^T \cdot \boldsymbol{w}\right)$ belongs to $\mathcal{S} \subset \mathcal{C}\left([0,1]^M\right)$, then it holds that, for any $\boldsymbol{w} \in \mathbb{R}^M$ and $b \in \mathbb{R}$:

$$L(\hat{f}) = \int_{[0,1]^M} \hat{f}(\boldsymbol{x}) d\mu(\boldsymbol{x}) = \int_{[0,1]^M} \sigma\left(b + \boldsymbol{x}^T \cdot \boldsymbol{w}\right) d\mu(\boldsymbol{x}) = 0\,.$$

In this case, since $L \neq 0$, then $\mu \neq 0$, and it is verified that this is a contradiction. In fact, since σ is a continuous sigmoidal function, then, by lemma A.5.3, it holds that $\mu = 0$.

■

Later, in 1993, Moshe Leshno *et al.* proved a stronger universal approximation theorem in [19], providing a sufficient and necessary condition on the activation functions of feedforward networks to be universal approximators. In their publication, they stated that a feedforward network with an arbitrarily wide hidden layer, M source nodes in its input layer, and a single neuron in the output layer can approximate any continuous function with any degree of precision if and only if the activation functions of the network are not polynomial.

These results refer to the category of arbitrarily wide neural networks with bounded depth, while in 1999, Vitaly Maiorov and Allan Pinkus demonstrated in [22] that there exists a sigmoidal activation function such that neural networks with two hidden layers and bounded width are universal approximators.

Teorema 3.4.4. *There exists a sigmoidal activation function σ that is real, analytic, and strictly increasing and satisfies that: for any $\varepsilon > 0$ and $f \in [0,1]^M$, there exist real constants $\tilde{a}_l, a_{lm}, \tilde{b}_l, b_{lm}$ and vectors $\boldsymbol{w}_{lm} \in \mathbb{R}^M$ such that, for every $\boldsymbol{x} \in [0,1]^M$, it holds that:*

$$\left| f(\boldsymbol{x}) - \sum_{l=1}^{6M+3} \tilde{a}_l \sigma \left(\sum_{m=1}^{3M} a_{lm} \sigma\left(\boldsymbol{x}^T \cdot \boldsymbol{w}_{lm} + b_{lm}\right) + \tilde{b}_l \right) \right| < \varepsilon\,.$$

Observación 3.4.5. *It is clear that in Theorem 3.4.4, a neural network with two hidden layers and bounded width is described, where the width of each layer is $3M$ and $6M+3$, respectively, with M being the dimension of the input data $\boldsymbol{x} \in \mathbb{R}^M$. In fact, weights of the first hidden layer $\boldsymbol{w}_{lm}$, with biases b_{lm}, weights of the second hidden layer a_{lm}, with biases $\tilde{b}_l$, and weights of the output layer $\tilde{a}_l$ have been introduced.*

On the other hand, in 2017, Zhou Lu *et al.* proved in [21] that for neural networks with ReLU activation functions (3.9) and bounded width, a universal approximation result also holds. Also related to ReLU activation functions, in 2022, Zuowei Shen *et al.* characterized in [36] the depth and width necessary to approximate a function using neural networks with this type of activation function.

3.4.2 Backpropagation

The universal approximation results presented in the previous section affirm the existence of feedforward neural networks that can serve as universal approximators, but they do not provide a constructive method for obtaining them. Therefore, to develop a predictive model using a *feedforward* neural network, it must be trained through an iterative optimization algorithm, such as those described in SECTION *3.1*.

In the first part of this chapter, it has been established that the weight vector $\boldsymbol{w}$ must be updated iteratively based on the relation (3.2). Stochastic gradient descent is an optimization algorithm that allows this update by considering the gradient of the part of the loss function associated with the data $\boldsymbol{x}^i$ at each iteration t, i.e., $\nabla\mathcal{E}_i\left(\boldsymbol{w}^{(t)}\right)$. Thus, it becomes necessary to develop a method to optimize the calculation of this gradient: in feedforward networks, this is backpropagation.

Observación 3.4.6. *Backpropagation is not an optimization algorithm for the neural network that allows learning, like gradient descent, but a method for calculating the gradient needed to apply such algorithms [12].*

Considering the loss function for a data $\boldsymbol{x}$, $\mathcal{E}(\boldsymbol{w})$, its dependence on the weights $\boldsymbol{w}$ is through the activation functions. In particular, the dependency of $\mathcal{E}$ on the weight $w_{n,j}^{(z)}$ that connects neuron j from layer $z-1$ to neuron n from layer z is such that, using the chain rule:

$$\frac{\partial\mathcal{E}}{\partial w_{n,j}^{(z)}} = \frac{\partial\mathcal{E}}{\partial a_n^{(z)}} \cdot \frac{\partial a_n^{(z)}}{\partial w_{n,j}^{(z)}} = \frac{\partial\mathcal{E}}{\partial a_n^{(z)}} \cdot \hat{y}_j^{(z-1)} =: \delta_n^{(z)} \cdot \hat{y}_j^{(z-1)} . \tag{3.19}$$

Thus, the "error signals" $\delta_n^{(z)}$ of neuron n from layer z are defined, and it is considered that $\hat{y}_j^{(z-1)} = 1$ in the case where $w_{n,j}^{(z)} = b_n^{(z)}$ is the bias of neuron n from layer z. Additionally, it is assumed that $\hat{\boldsymbol{y}}^{(0)} = \boldsymbol{x}$, the input values.

Observación 3.4.7. *The calculation of these error signals is the interesting part of the method, as this is where the "backward propagation" occurs [30].*

Using the chain rule, it can be seen that the error signal of neuron n from layer z is fully determined by the error signals of neurons in layer $z+1$, along with the activation function of neuron n and the synaptic weights $w_{rn}^{(z+1)}$, where $r \in \{1, \dots, |V_{z+1}|\}$, connecting neuron n to neurons in the next layer:

$$\delta_n^{(z)} = \frac{\partial\mathcal{E}}{\partial a_n^{(z)}} = \sum_{r=1}^{|V_{z+1}|} \frac{\partial\mathcal{E}}{\partial a_r^{(z+1)}} \cdot \frac{\partial a_r^{(z+1)}}{\partial a_n^{(z)}} = \sum_{r=1}^{|V_{z+1}|} \delta_r^{(z+1)} \cdot w_{r,n}^{(z+1)} \cdot \left[\psi_n^{(z)}\left(a_n^{(z)}\right)\right]' . \tag{3.20}$$

In the last equality, we used that, by equation (3.17):

$$\frac{\partial a_r^{(z+1)}}{\partial a_n^{(z)}} = \frac{\partial}{\partial a_n^{(z)}} \left[\boldsymbol{w}_r^{(z+1)} \cdot \hat{\boldsymbol{y}}^{(z)} + b_r^{(z+1)} \right] = \frac{\partial}{\partial a_n^{(z)}} \left[\boldsymbol{w}_r^{(z+1)} \cdot \Psi^{(z)} \left(\boldsymbol{a}^{(z)} \right) + b_r^{(z+1)} \right] =$$
$$= w_{r,n}^{(z+1)} \cdot \left[\psi_n^{(z)} \left(a_n^{(z)} \right) \right]' .$$

This relation holds for any neuron n in a hidden layer z, but cannot be applied if $z = Z$, the output layer, as $z + 1$ is not a layer in the network. Therefore, for the output layer, it must be considered that $\delta_n^{(Z)}$ can be calculated from the loss function defined in the network and the corresponding activation function of the output layer. The derivation of the explicit expressions for the functions studied in this work can be found in A.6 in the appendix chapter.

Ultimately, the gradient of the loss function of a feedforward neural network is computed using backpropagation as follows [4]:

1. A data vector $\boldsymbol{x}$ is introduced and propagated through the various neurons using activation functions and the relation (3.1). This results in the final output values of the network $\hat{f}(\boldsymbol{x})$.
2. The error signals $\delta_k^{(Z)}$ of the neurons in the output layer are obtained, according to the corresponding relation for their loss function. These error signals are then backpropagated through all neurons in the hidden layers using relation (3.20).
3. The gradient of the loss function for an input $\boldsymbol{x}$ in an iteration is defined as the vector of partial derivatives (3.19), which includes the backpropagated error signals from the second step, and the outputs of the neurons obtained in the first step.

3.5 Recurrent Networks

The feedforward neural networks studied in Section *3.4* have a main limitation: they are limited to static tasks, i.e., they can approximate a static function between input and output data. However, in predictive models with temporal dependence, the tasks to be predicted can exhibit variations, so it is necessary to use neural networks that allow dynamic tasks to be developed.

To achieve this property, the training signals from previous stages must be reintroduced into the network through recurrent connections, so networks that include such connections are called recurrent neural networks (RNN). Additionally, the concept of "stage", E, is introduced as the training period, meaning the network starts training at time t_1 and ends at time $t_2 > t_1$, with the stage being the period $[t_1, t_2]$.

The crucial part of recurrent networks is the *feedback*, which allows for creating very different architectures for different specific cases. In these networks, the neurons of later layers are connected to neurons of earlier layers, forming cycles, or even to themselves (*feedback* self). Thus, it can be understood that the neural network has a "state" at each stage, and at the next stage, the neurons receive both the input values $\boldsymbol{x}^i$ and the values generated by the different neurons in the previous stage. In this way, RNNs will use "delays", denoted by τ^{-1}, that will allow modifying at which stage the different values should be introduced.

Suppose a simple feedforward neural network with two layers, consisting of M and S neurons, one for input and one for output, respectively. That is, it has input values $\boldsymbol{x} \in \mathbb{R}^M$ and output values $\hat{\boldsymbol{y}}^{(1)} \in \mathbb{R}^S$. Let the current stage be E. If we want to convert the previous network into a recurrent network, we can consider the inputs to the network as the input values $\boldsymbol{x}_{E-q}, \dots, \boldsymbol{x}_E$ and the output values $\hat{\boldsymbol{y}}^{(1)}_{E-p}, \dots, \hat{\boldsymbol{y}}^{(1)}_{E-1}$. To do this, we simply introduce $p+q$ delays τ^{-1}, as shown in Figure 3.3, resulting in a nonlinear autoregressive neural network with exogenous inputs (NARX). This recurrent network consists of $(p+q+1) \cdot M$ source nodes, associated with the input vector of the current stage and the input vectors from the q previous stages, as well as the output vectors from the p previous stages.

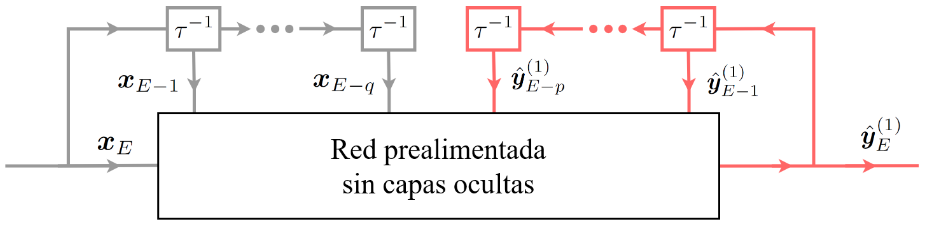
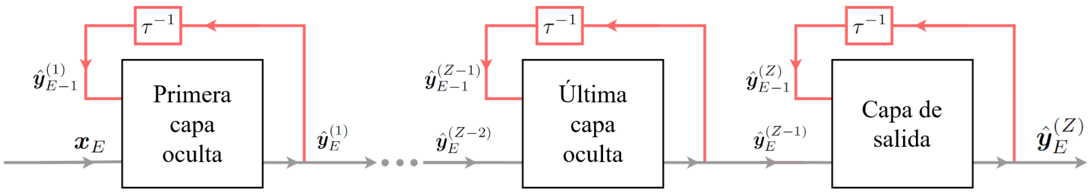


Figure 3.3: *NARX neural network, where the recurrent feedback connections are shown in red.*

As can be seen, the more layers and neurons are added to the feedforward network, the more possible recurrent neural networks can be obtained, simply by combining the *feedback* between the different layers [13].

A common example is the recurrent multilayer perceptrons (RMLP), where each layer z of the neural network has as input values at stage E the usual values from that stage $\hat{\boldsymbol{y}}^{(z-1)}_E$ and the values produced by that layer in the previous stage $\hat{\boldsymbol{y}}^{(z)}_{E-1}$, as shown in Figure 3.4. In this case, considering activation functions $\Psi^{(z)}$ and biases $\boldsymbol{b}^{(z)}$ for the different layers of size $|V_z|$, as well as usual synaptic weights $w_{n,j}$ from neuron j of layer $z-1$ to neuron n of layer z and recurrent weights $\tilde{w}_{n,k}$ from neuron j of layer z to neuron n of the same layer z, the output value for neuron n of layer z is:

$$\left(\hat{y}^{(z)}_n\right)_E = \psi^{(z)}_n \left(b^{(z)}_n + \sum_{j=1}^{|V_{z-1}|} w^{(z)}_{n,j} \left(\hat{y}^{(z-1)}_j\right)_E + \sum_{k=1}^{|V_z|} \tilde{w}^{(z)}_{n,k} \left(\hat{y}^{(z)}_j\right)_{E-1} \right). \tag{3.21}$$



τ^{-1}
τ^{-1}
τ^{-1}
$\hat{\boldsymbol{y}}_{E-1}^{(1)}$
$\boldsymbol{x}_E$
Primera capa oculta
$\hat{\boldsymbol{y}}_{E}^{(1)}$
$\hat{\boldsymbol{y}}_{E-1}^{(Z-1)}$
$\hat{\boldsymbol{y}}_{E}^{(Z-2)}$
Última capa oculta
$\hat{\boldsymbol{y}}_{E-1}^{(Z)}$
$\hat{\boldsymbol{y}}_{E}^{(Z-1)}$
Capa de salida
$\hat{\boldsymbol{y}}_{E}^{(Z)}$

Figure 3.4: *RMLP network, where the recurrent feedback connections are shown in red.*

Observación 3.5.1. *Neurons whose outputs are calculated with Equation* (3.21) *are called first-order neurons, and second-order neurons can be defined if their activation function input includes a multiplicative term of* $\left(\hat{y}_j^{(z-1)}\right)_E$ *with* $\left(\hat{y}_k^{(Z)}\right)_{E-1}$.

On the other hand, analogously to the universal approximation theorems for feedforward neural networks, recurrent networks also have theorems stating that RNNs are universal approximators.

In 2002, Xiaoou Li and Wen Yu proved in [20] that any nonlinear dynamic system can be approximated by a recurrent neural network with the desired degree of precision and no restrictions on the compactness of the state space, provided the network has an adequate number of hidden neurons. Later, in 2006, Anton M. Schäfer and Hans G. Zimmermann proved in [34] a similar result but using sigmoid activation functions.

3.5.1 Optimization Algorithms

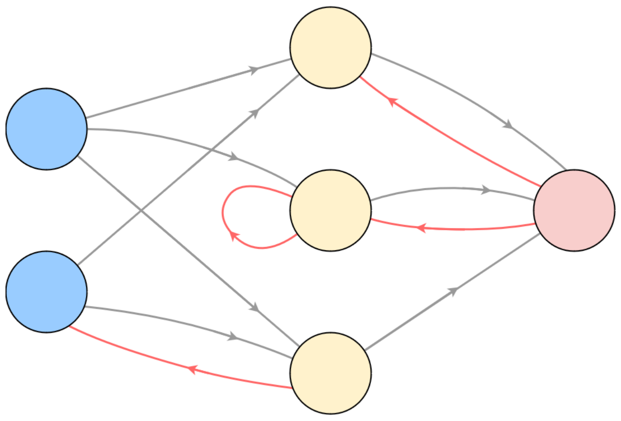
The typical optimization algorithms for recurrent networks with supervised learning are backpropagation through time (BPTT) and real-time recurrent learning (RTRL). These algorithms differ from the usual ones for feedforward networks because it is necessary to propagate information through the recurrent connections.

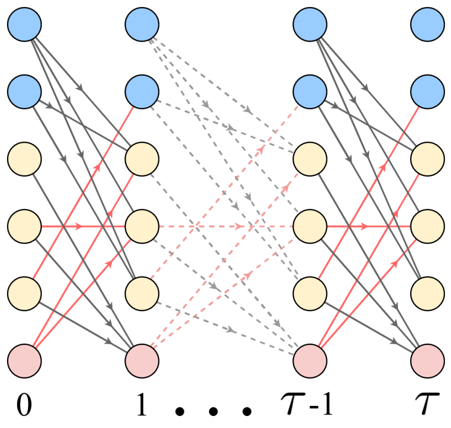
The BPTT algorithm is a generalization of the classic stochastic gradient descent algorithm along with the associated backpropagation. For this, a feedforward unfolded network is considered, formed by τ layers, where each layer contains the information of the recurrent neural network, and then the known algorithm is applied.

Thus, for training a stage, τ discrete time steps are considered, and the unfolded network is constructed as a network with τ layers and $|V_n|$ neurons per layer, where $|V_n|$ is the size of the recurrent network. Additionally, in each layer of the unfolded network, there is a copy of each neuron of the RNN, and the synaptic weights $w_{n,j}^{(z)}$ that connect neuron j of layer $z-1$ of the unfolded network to neuron n of layer z are copies of the synaptic weights connecting neurons n and j in the recurrent network [13]. A comparison can be seen in Figures 3.5a and 3.5b.

Observación 3.5.2. *An interesting aspect is that, by ordering the neurons appropriately, in the unfolded feedforward network, the ascending or horizontal arcs correspond to recurrent connections* *in the RNN, while the descending arcs in the unfolded network correspond to synaptic connections in the RNN. This is clearly illustrated in Figures 3.5a and 3.5b.*

Since each layer z of the unfolded network represents a time step of the recurrent network, the stochastic gradient descent algorithm presented in SECTION *3.1* is applied, also using the backpropagation described in SECTION *3.4.2* on the unfolded feedforward network. This allows optimizing both synaptic weights and recurrent weights, and the process is repeated for the next stage [40].


(a) *RNN architecture with size 6.*



(b) *Unfolded network of size* $6 \cdot (\tau + 1)$*.*

Figure 3.5: *Unfolding of a recurrent network for* τ *time steps.*

Observación 3.5.3. *The BPTT algorithm exponentially increases in computational complexity as the size of the recurrent neural network or the number of recurrent weights increases, so it is only useful for small RNNs.*

The BPTT algorithm relies on updating the synaptic weights using the stochastic gradient descent presented in SECTION *3.1*, also utilizing the backpropagation method described in SECTION *3.4.2*, since the recurrent network is unfolded into feedforward networks. On the other hand, the RTRL algorithm is said to optimize ”online” because it modifies synaptic weights as information propagates during the forward pass, so no unfolding is needed. Its development can be consulted in [41].

3.5.2 Vanishing Gradient

One of the main issues in recurrent neural networks trained with gradient descent-based optimization algorithms and backpropagation is the vanishing gradient. This occurs because the error signals backpropagated tend to grow or decay at each time step. If they grow excessively, they lead to oscillating and non-converging weights, while if they decay too much, the learning process slows down and no improvements are made [38].

The explanation is simple. Since the weights $w_{n,j}$ (including the recurrent $\tilde{w}_{n,j}$) are updated based on the relation (3.3) with a learning rate η and a stage between times t_1 and t_2, we have:

$$\Delta_{n,j} = -\eta \cdot \frac{\partial \mathcal{E}(t_1, t_2)}{\partial w_{n,j}} = -\eta \cdot \sum_{t=t_1}^{t_2} \delta_n(t) \hat{y}_j(t) \, .$$

The backpropagated error signal at time $t \in [t_1, t_2)$ for neuron n is:

$$\delta_n(t) = \psi_n'(a_n(t)) \cdot \sum_{r \in U_n} w_{r,n} \cdot \delta_r(t+1) \, , \tag{3.22}$$

where U_n is the set of neurons connected to neuron n via a weight.

Thus, the error signal given to an output layer neuron s at time t is backpropagated through the network for $t - t_1$ time steps until reaching an arbitrary neuron n. The error signal then modifies as follows:

$$\frac{\partial \delta_n(t_1)}{\partial \delta_s(t_2)} = \begin{cases} \psi_n'(a_n(t_1)) \cdot w_{s,n} & \text{if} \quad t_2 - t_1 = 1 \\ \psi_n'(a_n(t_1)) \cdot \left[\sum_{r \in U_n} \frac{\partial \delta_r(t_1+1)}{\partial \delta_s(t_2)} \cdot w_{r,n} \right] & \text{if} \quad t_2 - t_1 > 1 \end{cases} . \tag{3.23}$$

Expanding equation (3.23) over time and denoting by n_t a neuron at time t, we obtain the following expression:

$$\frac{\partial \delta_n(t_1)}{\partial \delta_s(t_2)} = \sum_{n_{t_1} \in U_n} \cdots \sum_{n_{t_2 - 1} \in U_n} \left(\prod_{t=t_1+1}^{t_2} \psi_{n_t}'(a_{n_t}(t_2 - t + t_1)) \cdot w_{n_t, n_{t-1}} \right) . \tag{3.24}$$

From this relation, it follows that if $\left|\psi_{n_t}'(a_{n_t}(t_2 - t + t_1)) \cdot w_{n_t, n_{t-1}}\right| > 1$ for all t, then the product grows exponentially, and the error signal explodes. On the other hand, if $\left|\psi_{n_t}'(a_{n_t}(t_2 - t + t_1)) \cdot w_{n_t, n_{t-1}}\right| < 1$ for all t, the error signal decays exponentially, leading to the disappearance of the global error signal. This is the so-called vanishing gradient phenomenon, which has been described, among others, in [38].

3.5.3 Long-Short Term Memory

A type of recurrent neural network that is particularly relevant due to its many practical applications is the Long Short-Term Memory (LSTM) network. These were proposed to mitigate the vanishing gradient problem, and in them, neurons form LSTM units, known as "memory cells", which consist of a constant error carousel (CEC), an input gate, and an output gate [14].

The theoretical foundation of these cells is based on creating neurons with a constant error signal flow. This is achieved by considering a neuron n with a single feedback loop to itself. Thus, its error signal at time t is simply, according to equation (3.22):

$$\delta_n(t) = \psi'_n(a_n(t)) \cdot w_{n,n} \cdot \delta_n(t+1) \,. \tag{3.25}$$

To ensure a constant error signal flow, i.e., $\delta_n(t) = \delta_n(t+1)$, the condition $\psi'_n(a_n(t)) \cdot w_{n,n} = 1$ must be satisfied. Integrating this, we obtain:

$$\psi_n(a_n(t)) = \frac{a_n(t)}{w_{n,n}} \,.$$

It follows that ψ_n must be linear and constant over time, so we can use the identity function as the activation function and set $w_{n,n} = 1$. These are the conditions for the CEC, as:

$$\hat{y}_n(t+1) = \psi_n(a_n(t+1)) = \psi_n(\hat{y}_n(t) \cdot w_{n,n}) = \hat{y}_n(t) \,.$$

In summary, considering a memory cell m_j as shown in figure 3.6, the input gate (In) controls the signals entering the memory cell, while the output gate (Out) protects other memory cells from disturbances originating in m_j. Furthermore, the cell includes a CEC between the previous activation function $\psi^{(\text{pre})}_{m_j}$ and the subsequent one $\psi^{(\text{pos})}_{m_j}$, and memory cells m_j are grouped into "memory blocks" m, which are sets of memory cells sharing an input and output gate.

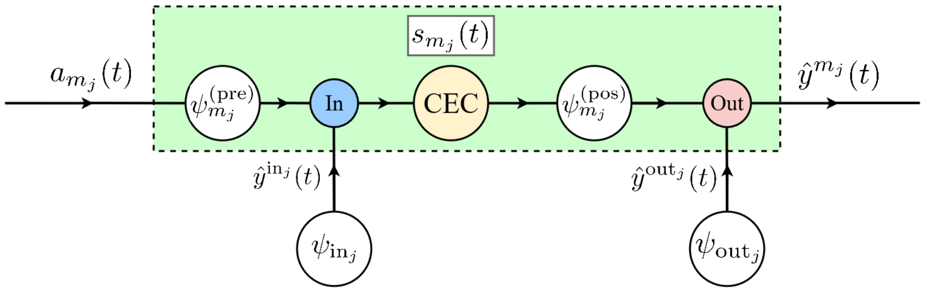

$s_{m_j}(t)$
$a_{m_j}(t)$
$\psi^{(\text{pre})}_{m_j}$
In
CEC
$\psi^{(\text{pos})}_{m_j}$
Out
$\hat{y}^{m_j}(t)$
$\hat{y}^{\text{in}_j}(t)$
$\hat{y}^{\text{out}_j}(t)$
ψ_{in_j}
ψ_{out_j}

Figure 3.6: *Architecture of a memory cell m_j.*

For time t, a memory cell receives three distinct types of values: those from the input gate, $\hat{y}^{\text{in}_j}(t)$, those from the output gate, $\hat{y}^{\text{out}_j}(t)$, and those entering directly into the cell, $a_{m_j}(t)$. Given the activation functions ψ_{in_j} and ψ_{out_j} for the input and output gates, respectively, usually logistic sigmoids, the values received are:

$$\begin{aligned}
\hat{y}^{\text{in}_j}(t) &= \psi_{\text{in}_j}\left(a_{\text{in}_j}(t)\right) = \psi_{\text{in}_j}\left(\sum_{r\in U} w_{\text{in}_j,r}\hat{y}_r(t-1) + \sum_{i\in I} w_{\text{in}_j,i}\hat{y}_i(t)\right) \\
\hat{y}^{\text{out}_j}(t) &= \psi_{\text{out}_j}\left(a_{\text{out}_j}(t)\right) = \psi_{\text{out}_j}\left(\sum_{r\in U} w_{\text{out}_j,r}\hat{y}_r(t-1) + \sum_{i\in I} w_{\text{out}_j,i}\hat{y}_i(t)\right) \\
a_{m_j}(t) &= \sum_{r\in U} w_{m_j,r}\hat{y}_r(t-1) + \sum_{i\in I} w_{m_j,i}\hat{y}_i(t)
\end{aligned} \tag{3.26}$$

Moreover, the "state" of the memory cell $s_{m_j}(t)$ can be defined as a function of its previous state and the multiplication of the input values through the input gate and direct ones, the latter multiplied by a previous activation function $\psi_{m_j}^{(\text{pre})}$:

$$s_{m_j}(t) = s_{m_j}(t-1) + \hat{y}^{\text{in}_j}(t) \cdot \psi_{m_j}^{(\text{pre})}\left(a_{m_j}(t)\right) \ , \tag{3.27}$$

where it is assumed that $s_{m_j}(0) = 0$ and $\psi_{m_j}^{(\text{pre})}(a) = \frac{4}{1-\exp(-a)} - 2$, which rescales to the interval $(-2, 2)$.

The output value of the memory cell, $\hat{y}^{m_j}(t)$, is obtained by multiplying the output gate value by the state of the cell $s_{m_j}(t)$, under the effect of a subsequent activation function $\psi_{m_j}^{(\text{pos})}$:

$$\hat{y}^{m_j}(t) = \hat{y}^{\text{out}_j}(t) \cdot \psi_{m_j}^{(\text{pos})}\left(s_{m_j}(t)\right) \ , \tag{3.28}$$

where $\psi_{m_j}^{(\text{pos})}(a) = \frac{2}{1-\exp(-a)} - 1$ rescales to the interval $(-1, 1)$.

In summary, LSTM networks are a type of recurrent neural networks formed by cells with gates that modify input and output signals. This structure allows the network to store and forget information from previous iterations at each step, making them particularly useful for modeling problems with long-term temporal dependencies.

Observación 3.5.4. *The optimization algorithm for LSTM networks is a combination of the BPTT and RTRL algorithms for recurrent networks, as detailed in [38]. The basis of this algorithm is that BPTT is used to train the components of the network that are after the cells, while RTRL is used for the components after the cells as well as for the cell components.*

Chapter 4

Predictive Model of Water Turbidity

Water turbidity is a physical variable that is difficult to describe analytically, as its dependence on other variables is not clear. Furthermore, being a physical parameter with a somewhat ambiguous definition complicates its analysis even further.

Definición 4.0.1. *Turbidity is the measure of the degree of transparency, or clarity, of a liquid. It is an optical characteristic measured by the amount of light scattered as it passes through the material, and its unit is NTU (from the English acronym, Nephelometric Turbidity Unit).*

On the other hand, the characterization and prediction of water turbidity are of vital importance in many circumstances, particularly in drinking water production. Turbidity events pose a problem for water desalination plants, as desalination mechanisms based on membranes and reverse osmosis cannot function with high turbidity levels[1], forcing plants to halt operations.

Thus, for the operational efficiency of a desalination plant, it is beneficial to anticipate these events and plan the shutdown and cleaning of the plants, optimizing their performance. This is the reason why the company ACCIONA became interested in developing a model with these characteristics, providing the necessary data and resources.

Ultimately, with the intention of testing all the concepts described in the previous chapters and due to ACCIONA's expressed interest, the following problem is posed: *Given the historical turbidity data collected in a desalination plant, is it possible to develop a predictive model that estimates turbidity in the plant two hours into the future?*

[1]See [1] for more information on reverse osmosis and the effect of turbidity on it.

4.1 Problem Definition and Considered Data

Once the problem is understood, the first challenge is to determine what results are desired from the prediction and, based on this, what variables are necessary.

Naturally, the first idea that arises is to predict the exact turbidity value in NTU at a future time τ, but after initial attempts, it was observed that this approach leads to low accuracy. Therefore, to improve prediction accuracy without losing excessive information, an alternative approach is proposed: predicting the probability that turbidity will be at a certain level at time τ. Throughout this chapter, both options are developed, comparing their similarities and differences, and presenting the results obtained with each approach. They will be referred to as **Model 1** (turbidity prediction) and **Model 2** (turbidity level prediction).

In any case, for both approaches, the turbidity value is necessary as an input variable, which was obtained from the ACCIONA-operated plant as a time series of NTU values for the period from 2017 to 2023. The time step between the considered data points is one minute, resulting in 3,680,640 turbidity data points. All of them are associated with different external variables that are expected to influence turbidity production.

4.1.1 Considered External Variables

Water turbidity has various origins and, therefore, different influencing parameters, including the presence of suspended matter and sediments related to erosion, organic matter such as plankton or other microscopic organisms, and algae growth.

These parameters are very difficult to measure directly, so other meteorological and biological variables that are related to them but measurable via satellites were studied. The nine selected external variables were obtained from the *Meteomatics* database[2] and are as follows:

- **Sea Current Speed (S.C.S.)**: Measured in km/h, ocean currents can transport sediments and microorganisms.
- **Sea Current Direction (S.C.D.)**: Measured from 0 to 360 degrees.
- **Wind Speed (W.S.)**: Measured in km/h, wind can transport dust and other materials that fall into the water, increasing turbidity.
- **Wind Direction (W.D.)**: Measured from 0 to 360 degrees.
- **Dust (D)**: Measured in μg/m^3, considering dust particles between 0.9 and 20 μm, which can directly impact turbidity.

[2]For more information, see `https://www.meteomatics.com/`.

- **Nitrogen Dioxide (NO_2)**: Measured in μg/m^3, NO_2 can serve as an indicator of organic matter present in the water.
- **Water Temperature (W.T.)**: Measured in degrees Celsius, temperature can accelerate the proliferation of microorganisms and algae.
- **Sulfur Dioxide (SO_2)**: Measured in μg/m^3, SO_2 can also indicate organic matter presence in water.
- **Salinity (S)**: Measured in μg/m^3, salinity can promote the development of algae and living organisms.

For all these variables, a spatial point one kilometer offshore from the desalination plant's water intake was considered. Data was collected from 2017 to 2023 with a time step of five minutes, resulting in 6,625,152 meteorological and biological data points.

It is useful to conduct a preliminary analysis of correlation coefficients to understand the relationships between these variables and turbidity. To this end, Pearson's correlation coefficient is used (see A.7 in the appendix chapter), and the results are presented in Figure 4.1. It can be observed that none of the correlation coefficients are significant, as all have an absolute value lower than 0.3 [5].

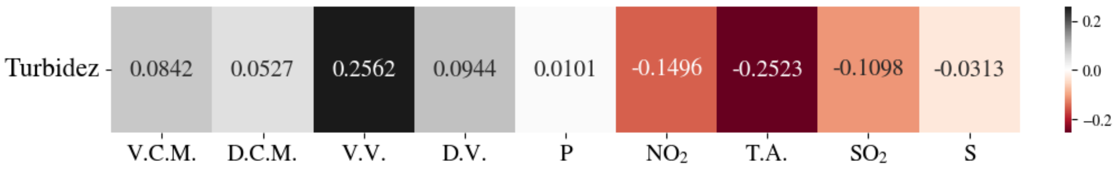
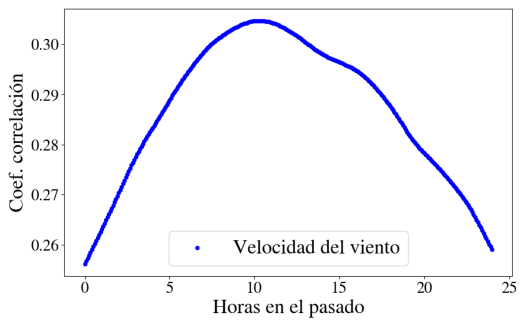


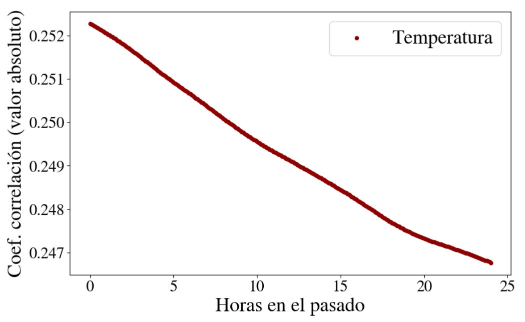
Figure 4.1: *Pearson correlation coefficients for turbidity with external variables.*

Additionally, in the search for correlations that could facilitate the development of the predictive model, attempts were made to relate the historical data of different variables with current turbidity. The procedure involves obtaining Pearson's correlation coefficient between turbidity at time t and external variables at time $t - t_1$, where $t_1 \in \mathbb{N}$ varies in the range of 0 to 288. Considering that t represents discrete time points with a five-minute step, this interval allows studying the external variables up to 24 hours before the considered turbidity value.

After performing this analysis, for each external variable, the results are plotted, comparing correlation coefficients against the past time step, and identifying the maximum correlation values. Two examples of these plots are shown in Figures 4.2a and 4.2b for wind speed and temperature, respectively.



(a) *Correlation with wind speed.*



(b) *Correlation with temperature.*

Figure 4.2: *Pearson correlation coefficients for turbidity with temporal lag.*

As it can be observed, some variables, such as wind speed, have a higher correlation with turbidity when a time lag is considered, while others, such as temperature, have a stronger correlation with turbidity in the present moment. Ultimately, this supports the need for a machine learning model capable of relating time series at different time instants: *it is necessary to use a model with short- and long-term memory.*

4.1.2 Necessary Data Adaptations for Prediction

Before developing the predictive model, three types of adaptations are required for the input data: temporal calibration, noise suppression, and classification into categories.

First, the considered turbidity data have a temporal spacing of one minute, while the external variables have a time interval of five minutes. To improve model accuracy, the turbidity data are averaged (arithmetic mean) over five-minute intervals and combined with the external variables, resulting in a *dataset* of 7,361,280 data points.

Mathematically, the sequence of turbidity data $\{(\tilde{t}^k, \tilde{x}^k)\}_{k=1}^{\tilde{N}}$ with $\tilde{N} = 3,680,640$ is transformed by:

$$x^i = \frac{1}{5}\sum_{j=0}^{4} \tilde{x}^{5i-j} \quad \text{and} \quad t^i = \tilde{t}^{5i}\,, \qquad \forall i \in \left\{1, ..., \frac{\tilde{N}}{5}\right\}\,,$$

resulting in a new sequence of turbidity data $\{(t^i, x^i)\}_{i=1}^{N}$ with $N = 736,128$. A two-dimensional matrix of size $736,128 \times 10$ is then considered, with time instants as rows and turbidity along with nine external variables as columns.

On the other hand, the received turbidity data contained anomalous values that were not related to actual high turbidity events, referred to as *noise*. This was identified because real turbidity fluctuations cannot be exponential, yet the studied data contained very short time intervals with large fluctuations. In summary, noise has been carefully defined and corrected for model implementation.

Definición 4.1.1. *A turbidity value x^i at an instant t^i, i.e., (t^i, x^i), is considered noise if its deviation from the average of the last 10 non-noise values exceeds 10 times that average. That is, noise $\mathcal{R}$ in the dataset is defined as $i \in \mathcal{R}$ if*

$$|x^i - \bar{x}^i| < 10 \cdot \bar{x}^i \qquad \textit{where} \qquad \bar{x}^i = \frac{1}{10}\sum_{j=1}^{10} x^{i_j} \ ,$$

where $i_{j'} < i_j < i$ if $j' > j$ and such that $\forall i_k \in (i_{j'}, i)$ with $i_k \neq i_j$, $i_k \in \mathcal{R}$.

Once the noise set $\mathcal{R}$ is identified, it is removed by linearly interpolating the closest non-noise values for each noisy value, i.e.,

$$\forall i \in \mathcal{R} \quad \Longrightarrow \quad x^i = x^{j_1} + \frac{x^{j_2} - x^{j_1}}{j_2 - j_1} \cdot (i - j_1) \ ,$$

where $j_1 < i$ such that $\forall j' \in (j_1, i)$, $j' \in \mathcal{R}$ and $j_2 > i$ such that $\forall j' \in (i, j_2)$, $j' \in \mathcal{R}$. An example of this procedure can be seen in Figure 4.3.

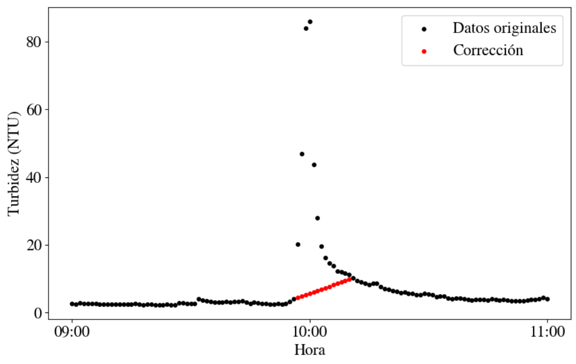


Figure 4.3: *Transformation of turbidity data to remove noise.*

Finally, for the predictive model focused solely on forecasting turbidity levels at a given instant τ (Model 2), three different turbidity classes with values 0, 1, and 2 are defined, and a column is added to the *dataset* with the class associated with the actual turbidity value x_i at each instant t_i:

- **Class 0**: Low turbidity level, with $x \in [0, a_1)$. In this case, $a_1 = 15$ NTU was used.
- **Class 1**: Medium turbidity level, with $x \in [a_1, a_2)$. In this case, $a_2 = 40$ NTU was used.
- **Class 2**: High turbidity level, with $x \in [a_2, \infty)$. In this case, the upper limit is 100 NTU due to sensor saturation.

This transformation can be seen in Figure 4.4, which shows in black the exact turbidity value at each time instant, and with colored bands representing the three classes: green for Class 0, yellow for Class 1, and red for Class 2. At a glance, it is evident that the number of data points belonging to each class is not balanced (97.2% in Class 0, 2.3% in Class 1, and 0.5% in Class 2), meaning different weights will need to be introduced to compensate for this imbalance.

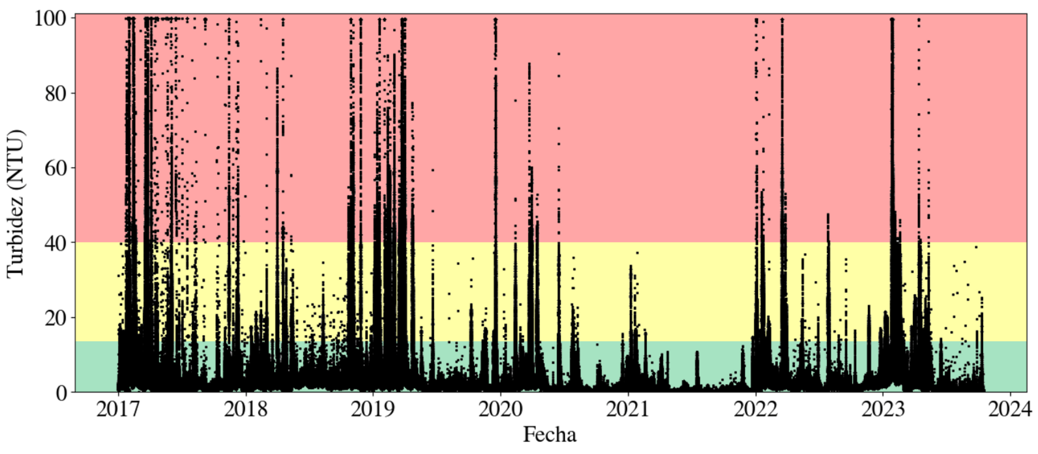


Figure 4.4: *Classification into three classes (displayed in green, yellow, and red) of the turbidity data (in black) used for model development.*

4.2 Model Characteristics

Once the data has been preprocessed, the next step is to decide on the main features that will indicate the type of model to be used. This includes: initialization, number of layers, layer size, type of network, activation functions, optimizer, and loss function.

The initialization in both models is performed using the normalized initialization distribution introduced by Xavier Glorot and Yoshua Bengio, which has been described in SECTION *3.3.1*. This initialization method was chosen due to its adaptability to LSTM networks and its impact on improving network performance.

On the other hand, given the need to use a neural network capable of relating multiple variables with their temporal history, the network used for both models is an LSTM network. This type of network, which has been discussed in detail in SECTION *3.5.3*, is a type of recurrent neural network (RNN) that avoids the loss of information that could be relevant in the long term, which is key in the problem being addressed due to the low number of turbidity events used for training and their temporal separation.

Additionally, for both models, neural networks with 1,259,235 parameters distributed across six layers have been used, with the size increasing in the two central layers and decreasing at the extremes. Between each pair of layers, a *dropout* layer with an increasing proportion in the intermediate layers has been introduced, which, as explained in SECTION *3.2*, reduces the risk of overfitting in the network. The reason for the variability in the layers is related to better model convergence and avoiding issues due to computationally intractable loss functions.

The activation function used in the layers is the Rectified Linear Unit (ReLU), presented in SECTION *3.3.2*, due to its optimality for maintaining a true zero value and its computational simplicity, which improves performance.

The chosen optimizer is *Adam*, which, as described in SECTION *3.1.1*, adapts to the training of large volumes of data, considering first and second-order moments in gradient descent, which increase the convergence rate and avoid instabilities.

Finally, although all the described features are common to both models, they differ in their last layer and loss function:

- For Model 1, the loss function used is the mean squared error (3.11), which, as studied in SECTION *3.3.3*, allows optimizing the obtained result through its difference with the real value of turbidity at a given time. Additionally, the last layer chosen is a fully connected dense layer with the identity as the activation function, adapted to the developed regression model.
- For Model 2, the loss function used is categorical cross-entropy (3.15). In this case, it is an appropriate loss function because it is a multiclass classification problem, and, as discussed in SECTION *3.3.3*, it allows the use of more than two classes and assigns a probability to each, which is the value compared with the real result. Similarly, the last layer used is also a fully connected dense layer but with a *softmax* activation function to adapt it to the developed classification model.

4.2.1 Weighting of Different Classes

Given the complexity of the input data and the significant difference in the volume of data in each of the classes (see Figure 4.4), in Model 2, a variable weight has been assigned to each class and for each epoch in the loss function, as indicated in equation (3.16). This has allowed for better model convergence and greater accuracy in the obtained results. The weighting of the classes has not been constant throughout all training epochs, and this decision is supported by the following argument:

> Initially, a sharp weighting was considered to excessively penalize the incorrect prediction of the less represented classes, to achieve an overprediction of them and better model convergence. Later, this weighting was gradually reduced and adapted to the appropriate balance of the classes based on the amount of data in each, ending with a training epoch where the weighting is inversely proportional to the number of events in each class.

In summary, let $\epsilon \in \{1, \ldots, \tilde{\epsilon}\}$ be the considered training epoch, the weight of each class has been, knowing that the number of data points in class 0 is N_0, in class 1 is N_1, and in class 2 is N_2, satisfying $N_0 \gg N_1 > N_2$:

$$\begin{aligned} W_0 &= \frac{N_0 + N_1 + N_2}{N_0} \cdot \exp\left(-\frac{\tilde{\epsilon} - \epsilon}{\tilde{\epsilon}}\right) , \\ W_1 &= \frac{N_0 + N_1 + N_2}{N_1} \cdot \exp\left(\frac{\tilde{\epsilon} - \epsilon}{2\tilde{\epsilon}}\right) , \\ W_2 &= \frac{N_0 + N_1 + N_2}{N_2} \cdot \exp\left(\frac{\tilde{\epsilon} - \epsilon}{\tilde{\epsilon}}\right) . \end{aligned} \tag{4.1}$$

4.3 Obtained Results

In this section, in addition to presenting the results of the two developed models, the differences and similarities found in the training, validation, and predictive capacity of both will be discussed. For both models, training will be performed with the period 2017-2023 (630,908 records) and validation with the period 2023-2024 (82,393 records).

4.3.1 Model 1

The main limitation of Model 1 is its sensitivity to the data and the difficult convergence it presents. Given that 97.2% of the normalized input data is less than 0.1, the model hardly deviates from a stationary prediction at low turbidity values, and when it does approach other types of predictions, after one or two training epochs, it moves away from that trend to return to the stationary one. Thus, most models trained in this way consist of a constant prediction of null turbidity.

On the other hand, since only 2.8% of the data significantly affects the loss function when it is in the stationary solution, the accuracy values for erroneous models are higher than for models that tend to other types of results. This implies that the usual metric for defining accuracy is incorrect, so the following relationship has been considered as accuracy:

$$\text{Accuracy} = 1 - \frac{1}{N} \sum_{i=1}^{N} \frac{|y^i - \hat{f}(\boldsymbol{x}^i)|^2}{y^i} , \tag{4.2}$$

where $\hat{f}(\boldsymbol{x}^i)$ are the predicted values and y^i are the real values, which to be considered in the sum must satisfy $y^i \geq a_1$, with a_1 being the value of class 1 in Model 2. This is, roughly speaking, the accuracy of the model in predicting intense turbidity peaks based on the average value of these events, which are of practical interest.

In this way, after training the model for 100 epochs, in batches of 1000 samples, and adapting the learning rate in different epochs to facilitate convergence, the results shown in Figures 4.5 and 4.6 have been obtained for training and validation, respectively.

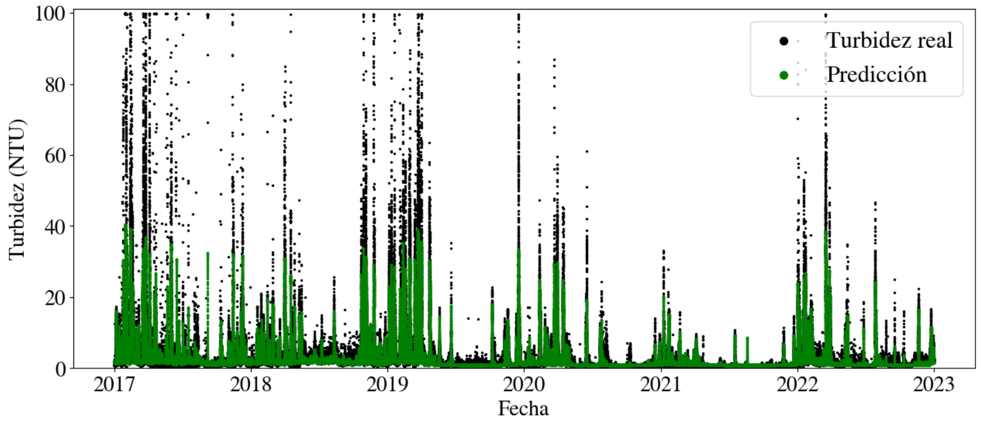


Figure 4.5: *Predictions of Model 1 for the training period.*

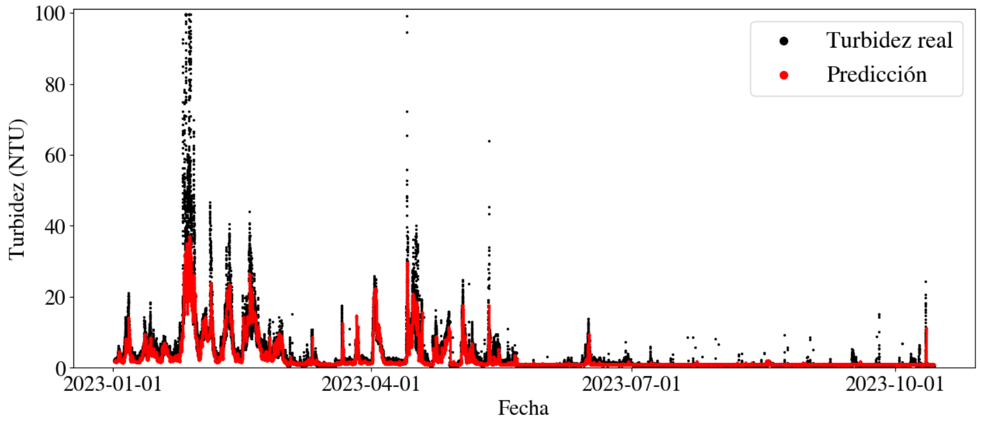


Figure 4.6: *Predictions of Model 1 for the validation period.*

It is verified that, with the accuracy defined in (4.2), the results have been those shown in Table 4.1. The accuracy of the model is not acceptable at a practical level, as it does not manage to approach the real turbidity values during periods of high turbidity. In fact, it is shown that addressing the turbidity problem through its exact prediction is complex and imprecise, necessitating an alternative approach, such as that of Model 2.

	Training	Validation
Accuracy	0.553	0.496

Table 4.1: *Accuracy obtained with Model 1, given by equation* (4.2).

4.3.2 Model 2

For Model 2, the main difficulty encountered has been its convergence. Prior to introducing class weighting, the network exhibited characteristics similar to those of Model 1, making it very unlikely to produce a prediction different from class 0, as it is the majority class. However, once the weights were introduced, the problem became convergence, as during training, the model either diverged, with loss functions tending to infinity, or fell into the stationary solution of class 0. Nevertheless, once this issue was resolved, a model was achieved that, after various training sessions, remains in a stable solution different from the one assigning class 0 in every instance.

In this case, since classification errors are not equivalent, the definition of accuracy is important. Clearly, classifying a class 0 event as class 2, or vice versa, is less acceptable than classifying a class 1 event as class 2, rendering the usual definition of accuracy as the proportion of correctly classified events meaningless. In this case, accuracy is defined as 1 minus the average of the absolute difference between the correct class and the predicted class, which penalizes extreme incorrect classifications more heavily. Thus:

$$\text{Accuracy} = 1 - \frac{1}{N}\sum_{i=1}^{N} |y^i - \hat{f}(\boldsymbol{x}^i)| \,, \tag{4.3}$$

where y^i is the correct class and $\hat{f}(\boldsymbol{x}^i) \in \{0, 1, 2\}$ is the predicted class.

In this case, considering the loss function with the weights (4.1), the model is trained for 100 epochs, in batches of 1000 samples, and adapting the learning rate, thus arriving at the results shown in Figures 4.7 and 4.8, for training and validation, respectively.

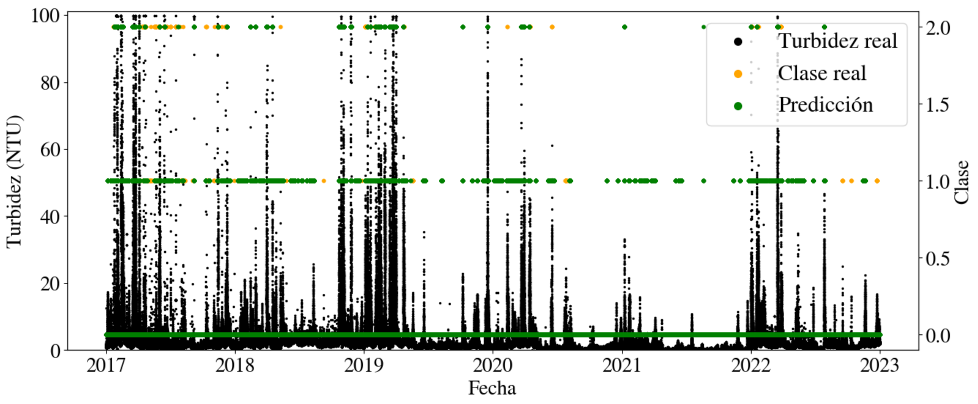


Figure 4.7: *Predictions of Model 2 for the training period.*

As can be seen, the accuracies shown in Table 4.2 for Model 2 significantly improve over the accuracy of Model 1, making the classification model more accurate and reliable. Thus, it is worth noting that, since an exact turbidity value is not predicted, the decision was made to reduce the exactness of the results in favor of improving their accuracy.

	Training	Validation
Accuracy	0.951	0.938

Table 4.2: *Accuracy obtained with Model 2, given by equation* (4.3).

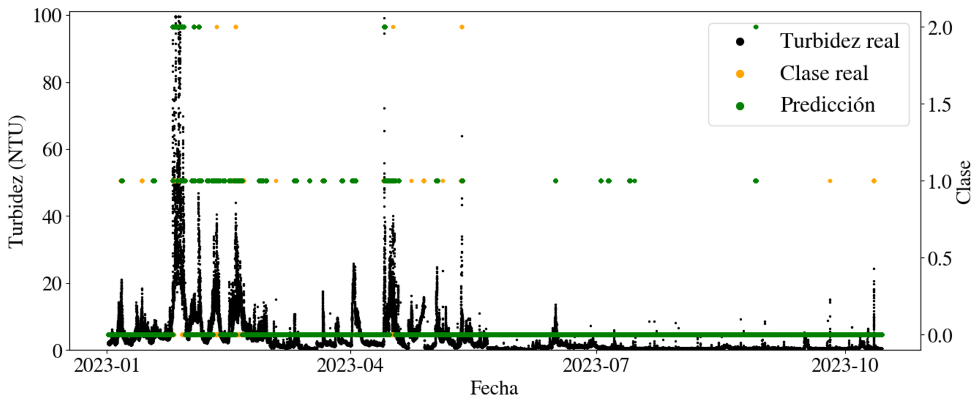


Figure 4.8: *Predictions of Model 2 for the validation period.*

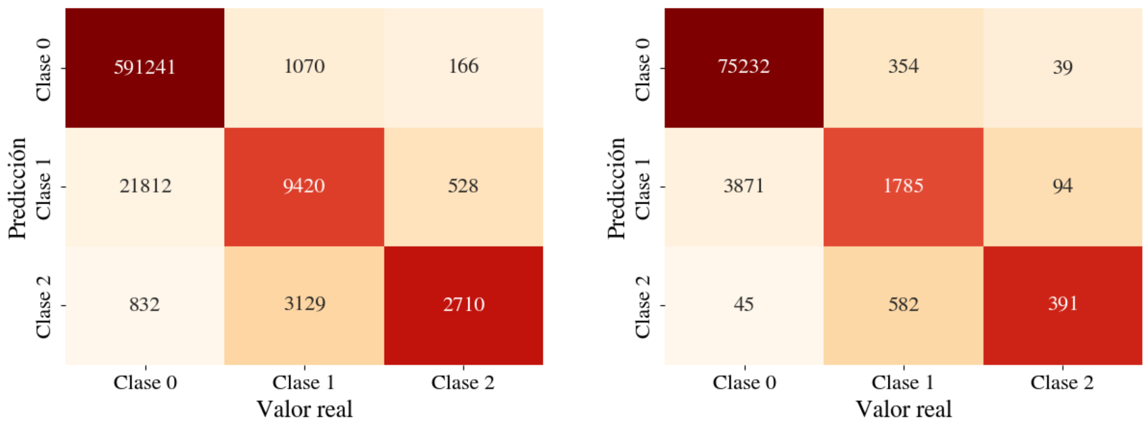
In fact, to verify the real performance of the predictive model, an alternative accuracy can be defined that only considers real turbidity events classified as class 1 or 2, i.e., modifying equation (4.3) with y_i such that $y_i \in \{1, 2\}$. For this alternative accuracy, the results shown in Table 4.3 are obtained, indicating that, indeed, the presented model is adequate and successfully predicts turbidity events.

	Training	Validation
Accuracy	0.703	0.659

Table 4.3: *Alternative accuracy (only high turbidity) obtained with Model 2.*

On the other hand, in Figures 4.9a and 4.9b, the confusion matrix (see A.8 in the appendix chapter for more information on confusion matrices) for the model's results can be observed, for training and validation, respectively. Among many other conclusions, these confusion matrices reveal that 87.9% of turbidity peaks are predicted at least as intermediate peaks, and only 4.4% of the times the model predicts an intense peak, there was actually no peak.

Due to these and other benefits that this predictive turbidity model provides for the company ACCIONA, it is currently being implemented as a preliminary model in one of its desalination plants, with the intention of making it a useful daily tool.



(a) *Training period.* (b) *Validation period.*

Figure 4.9: *Confusion matrices associated with Model 2.*

Chapter 5

Conclusions

As outlined in this document and supported by the literature, machine learning and neural networks encompass a wide range of methods to address both real-world and abstract problems. The possibilities are so vast that, at times, it becomes difficult to decide which type of network and learning approach is most suitable for a particular problem.

Although results have been deduced claiming the universality of neural networks as approximators, their proofs are not constructive, which leads to the problem of identifying how to optimize prediction models so they can properly estimate the case at hand. Additionally, it must be understood how to ensure that the model generalizes and makes accurate predictions on unseen input data. This is precisely what has been attempted, and achieved, in predicting water turbidity in desalination plants.

Since the problem posed by ACCIONA was not trivial and involved finding temporal correlations between different variables and turbidity, the use of feedforward networks was not feasible. Therefore, two models involving recurrent neural networks, specifically those with short- and long-term memory, were developed and compared to provide information on future turbidity levels. It was found that prioritizing accuracy in prediction over obtaining more informative results—i.e., prioritizing a model that predicts the turbidity class rather than the exact value—is more convenient and useful for solving this particular problem.

Finally, a real-world problem related to predicting weather events was successfully addressed, albeit in a simpler form due to the fewer variables involved. Given the benefits of the developed model, it has been implemented in the desalination plant, which will be useful for its operation and have a direct impact on a crucial part of people's lives: the **water supply**.

Appendix A

Definitions and Relevant Theorems

The following definitions and complementary theorems to the development presented in the work are provided, which may be of interest to consult.

A.1 Bayes' Theorem

Bayes' theorem relates the conditional probability of an event in a hypothesis with the prior probability and the posterior probability [18]:

Teorema A.1.1. *Let $\{A_1, \ldots, A_n\}$ be a set of mutually exclusive and exhaustive events such that $P(A_i) \neq 0$, $\forall i \in \{1, \ldots, N\}$. For an arbitrary event B, we have:*

$$P(A_j|B) = \frac{P(A_j) \cdot P(B|A_j)}{\sum_{i=1}^{N} P(A_i) \cdot P(B|A_i)} \quad \textit{for all } j \in \{1, \ldots, n\}.$$

Proof. Since $P(A_i \cap B) = P(A_i) \cdot P(B|A_i) = P(B) \cdot P(A_i|B) = P(B \cap A_i)$ for all $i \in \{1, \ldots, N\}$, by the total probability of exhaustive and mutually exclusive sets:

$$P(B) = \sum_{i=1}^{N} P(B \cap A_i) = \sum_{i=1}^{N} P(A_i) \cdot P(B|A_i) \quad \Longrightarrow$$

$$\Longrightarrow \quad P(A_j|B) = \frac{P(A_j) \cdot P(B|A_j)}{P(B)} = \frac{P(A_j) \cdot P(B|A_j)}{\sum_{i=1}^{N} P(A_i) \cdot P(B|A_i)}.$$

■

A.2 Bayesian Models of Linear Classification

In Bayesian classification models for K classes, the goal is not to directly obtain the posterior probability $p(C_k|\boldsymbol{x}^i)$, but rather to infer the class-conditional probability densities, $p(\boldsymbol{x}^i|C_k)$, and the probabilities of the different classes, $p(C_k)$. With these probabilities, the posterior probabilities sought for classification can be obtained analytically using equation (2.8).

In general, if the training set T is sufficiently large, the probability of the class $p(C_k)$ is estimated as the fraction of training points whose correct label belongs to class k, as demonstrated in theorem A.2.2 below. On the other hand, the conditional probability densities can be studied in an analogous way to the approach in SECTION *2.1.1.*

In this case, the distribution followed by these probabilities is a multivariate Gaussian distribution, i.e., a generalization to a K-dimensional space of the normal distribution, where we have a random vector $\boldsymbol{x}$, a mean vector $\boldsymbol{\mu}$, and the covariance matrix Σ [4]:

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu},\Sigma) = \frac{1}{(2\pi)^{D/2}|\Sigma|^{1/2}} \cdot \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right) .$$

In particular, the covariance matrix is the generalization of the variance concept for a scalar random variable to the vector of random variables. This matrix is square and includes the covariance between the elements of the vector [15]:

Definición A.2.1. *Let $X = (X_1,\dots,X_n)^T$ be a vector of random variables, then the covariance matrix Σ of dimension $n \times n$ is such that:*

$$\Sigma_{ij} = Cov(X_i,X_j) = E\left[(X_i - E(X_i))(X_j - E(X_j)\right] ,$$

where $E(X)$ is the expected value of the random variable X.

Teorema A.2.2. *Given a training set $T = \{(\boldsymbol{x}^1,y^1),\dots,(\boldsymbol{x}^N,y^N)\}$, consider a Bayesian model with two classes, C_0 and C_1, such that $y^i = 0$ for class C_0 and $y^i = 1$ for class C_1. Assuming that the probability densities of the classes are $p(C_0) = \rho$ and $p(C_1) = 1-\rho$, and the class-conditional probability densities follow Gaussian distributions with a shared covariance matrix $\boldsymbol{\Sigma}$, then:*

- *The maximum likelihood estimate for ρ is the fraction of points in class C_0 relative to the total number of points:*
$$\rho^* = \frac{N_0}{N} = \frac{N_0}{N_0+N_1} .$$

- *The maximum likelihood estimate for the means $\boldsymbol{\mu}_0$ and $\boldsymbol{\mu}_1$ of the class-conditional probability densities are, respectively, the mean of the input data associated with each class:*
$$\boldsymbol{\mu}_0 = \frac{1}{N_0}\sum_{i=1}^{N}(1-y^i)\boldsymbol{x}^i \quad \text{and} \quad \boldsymbol{\mu}_1 = \frac{1}{N_1}\sum_{i=1}^{N} y^i\boldsymbol{x}^i .$$

Proof. [4]. Given an input data point $\boldsymbol{x}^i$ from class C_0, we have that $y^i = 0$ and thus:

$$p(\boldsymbol{x}^i, C_0) = p(C_0)p(\boldsymbol{x}^i|C_0) = \rho \cdot \mathcal{N}(\boldsymbol{x}^i|\boldsymbol{\mu}_0,\boldsymbol{\Sigma}) .$$

On the other hand, for class C_1 and $y^i = 1$, we have:

$$p(\boldsymbol{x}^i, C_1) = p(C_1)p(\boldsymbol{x}^i|C_1) = (1-\rho)\cdot\mathcal{N}(\boldsymbol{x}^i|\boldsymbol{\mu}_1, \boldsymbol{\Sigma})\,.$$

Thus, the likelihood function is, where $Y = (y_1, \dots, y_N)^T$:

$$\mathcal{L}(\rho, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1) = p(Y|\rho, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) = \prod_{i=1}^{N} \left(\rho\cdot\mathcal{N}(\boldsymbol{x}^i|\boldsymbol{\mu}_0, \boldsymbol{\Sigma})\right)^{1-y^i}\cdot\left([1-\rho]\cdot\mathcal{N}(\boldsymbol{x}^i|\boldsymbol{\mu}_1, \boldsymbol{\Sigma})\right)^{y^i}\,.$$

Considering the natural logarithm of the likelihood function and maximizing it:

$$\begin{aligned}\mathcal{E}(\rho, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1) &= \log\left(p(Y|\rho, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1, \boldsymbol{\Sigma})\right) = \\ &= \sum_{i=1}^{N}(1-y^i)\cdot\left[\log(\rho) + \log\left(\mathcal{N}(\boldsymbol{x}^i|\boldsymbol{\mu}_0, \boldsymbol{\Sigma})\right)\right] + y^i\cdot\left[\log(1-\rho) + \log\left(\mathcal{N}(\boldsymbol{x}^i|\boldsymbol{\mu}_1, \boldsymbol{\Sigma})\right)\right]\end{aligned}$$

$$\frac{\partial\mathcal{E}}{\partial\rho} = \sum_{i=1}^{N}\frac{1-y^i}{\rho} - \frac{y^i}{1-\rho} = \sum_{i=1}^{N}\frac{1-y^i-\rho}{\rho(1-\rho)} = \frac{-1}{\rho(\rho-1)}\left(N\rho - \sum_{i=1}^{N}(1-y^i)\right) = 0\,.$$

From this, we directly deduce that:

$$\rho^* = \frac{1}{N}\sum_{i=1}^{N}(1-y^i) = \frac{N_0}{N}\,.$$

Now, considering maximization with respect to $\boldsymbol{\mu}_0$:

$$\begin{aligned}\frac{\partial\mathcal{E}}{\partial\boldsymbol{\mu}_0} &= \frac{\partial}{\partial\boldsymbol{\mu}_0}\left(\sum_{i=1}^{N} y^i\log\left(\mathcal{N}(\boldsymbol{x}^i|\boldsymbol{\mu}_i, \boldsymbol{\Sigma})\right)\right) = \frac{\partial}{\partial\boldsymbol{\mu}_0}\left(-\sum_{i=1}^{N}\frac{y^i}{2}(\boldsymbol{x}-\boldsymbol{\mu}_0)^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_0)\right) = \\ &= \sum_{i=1}^{N} y^i\left(\boldsymbol{\mu}_0^T\boldsymbol{\Sigma}^{-1}\boldsymbol{x}^i - \boldsymbol{\mu}_0^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_0\right) = -\boldsymbol{\mu}_0^T\boldsymbol{\Sigma}^{-1}\cdot\left(N_1\boldsymbol{\mu}_0 - \sum_{i=1}^{N} y^i\boldsymbol{x}^i\right) = 0 \quad\Longrightarrow \\ &\Longrightarrow\quad \boldsymbol{\mu}_0^* = \frac{1}{N_1}\sum_{i=1}^{N} y^i\boldsymbol{x}^i \qquad \text{and, similarly,}\quad \boldsymbol{\mu}_1^* = \frac{1}{N_1}\sum_{i=1}^{N}(1-y^i)\boldsymbol{x}^i\,.\end{aligned}$$

Finally, the desired result is obtained. ■

A.3 Karush-Kuhn-Tucker Conditions

The Karush-Kuhn-Tucker (KKT) conditions impose certain characteristics, i.e. necessary conditions, on the local minima of a function f and N constraints g_i. In particular, they are based on the following theorem, whose proof can be found in [23]:

Teorema A.3.1. *Let an optimization problem with $\boldsymbol{x} \in \mathbb{R}^M$ be given by*

$$\begin{aligned} \min \quad & \{f(\boldsymbol{x})\} \\ s.t. \quad & g_i(\boldsymbol{x}) \leq 0\,, \quad \forall\, i \in \{1, \dots, N\} \end{aligned}$$

and define $\mathcal{G}(\boldsymbol{x})$ as the set of indices of the active constraints, i.e. $g_i(\boldsymbol{x}) = 0$. If the functions f, g_i with $i \in \mathcal{G}(\boldsymbol{x})$ are differentiable and the g_i with $i \notin \mathcal{G}(\boldsymbol{x})$ are continuous, and furthermore the gradients $\nabla g_i(\boldsymbol{x})$ with $i \in \mathcal{G}(\boldsymbol{x})$ are linearly independent, then, if $\tilde{\boldsymbol{x}}$ is a local minimum of f and is feasible, i.e. satisfies the conditions $g_i(\tilde{\boldsymbol{x}}) \leq 0$, there exist scalars $u_i \geq 0$ with $i \in \{1, \dots, N\}$ such that the Lagrangian function $L(\tilde{\boldsymbol{x}}) = f(\tilde{\boldsymbol{x}}) + \sum_{i=1}^{N} u_i g_i(\tilde{\boldsymbol{x}})$ satisfies:

$$\begin{aligned} & \frac{\partial L}{\partial \tilde{x}_j} = 0\,, \quad \forall\, j \in \{1, \dots, M\} \\ & u_i g_i(\tilde{\boldsymbol{x}}) = 0\,, \quad \forall\, i \in \{1, \dots, N\}\,. \end{aligned}$$

It is worth noting that, if $\boldsymbol{x}_0$ is a point satisfying the above conditions, i.e. the KKT conditions, and the functions f, g_i with $i \in \mathcal{G}(\boldsymbol{x}_0)$ are convex, then $\boldsymbol{x}_0$ is a global minimum. The proof can be found in [23].

Observación A.3.2. *A function $f : \mathbb{R}^M \to \mathbb{R}$ is convex if for any $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ it satisfies that $f(\lambda \boldsymbol{x}_1 + (1 - \lambda \boldsymbol{x}_2) \leq \lambda \cdot f(\boldsymbol{x}_1) + (1 - \lambda) \cdot f(\boldsymbol{x}_2)$, for all $\lambda \in [0, 1]$.*

A.4 Graph Theory

In graph theory, certain concepts are useful when representing neural networks [24]:

Definición A.4.1. *A directed graph G is a pair $G = \{V, A\}$ where V is a set of vertices and A is a set of ordered pairs (i, j) with $i, j \in V$, called arcs; the vertex i is called the initial vertex, and the vertex j is called the final vertex. An arc of the form (i, i) is called a loop.*

A directed graph G is said to be simple if it has no loops or repeated arcs. Also, the set of successors of a vertex i is defined as $\Gamma^+(i) = \{j : (i, j) \in A\}$, and the set of predecessors as $\Gamma^-(i) = \{j : (j, i) \in A\}$.

Definición A.4.2. *If there exists at least one arc connecting each pair of vertices, the graph is complete.*

A.5 Results Used in the Proof of Cybenko's Universal Approximation Theorem

The proof of Cybenko's universal approximation theorem uses three important results which are shown in order: the Hahn-Banach theorem, the Riesz representation theorem, and a lemma about sigmoid continuous functions.

Teorema A.5.1. *Let S be a subspace of a normed vector space X and $\boldsymbol{x}_0 \in X$. Then, $\boldsymbol{x}_0 \in \bar{S}$ if and only if there does not exist a bounded linear functional L defined on X such that $L(\boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in S$ but $L(\boldsymbol{x}_0) \neq 0$.*

Proof. The proof can be found in [10]. ■

Teorema A.5.2. *Let Y be a locally compact Hausdorff space, such as $[0,1]^M$. Then, for every bounded linear functional L defined on $\mathcal{C}(Y)$, there exists a Borel measure μ such that, for every $f \in \mathcal{C}(Y)$, we have:*

$$L(f) = \int_Y f(\boldsymbol{x}) d\mu(\boldsymbol{x}) \, .$$

Proof. The proof can be found in [2]. ■

Lema A.5.3. *A continuous sigmoid function $\sigma : \mathbb{R} \to \mathbb{R}$ satisfies that, given μ a Borel measure over $[0,1]^M$, the measure is null, i.e., $\mu = 0$, if the following holds for all $\boldsymbol{w} \in \mathbb{R}^M$ and $b \in \mathbb{R}$:*

$$\int_{[0,1]^M} \sigma(\boldsymbol{x}^T \cdot \boldsymbol{w} + b) d\mu(\boldsymbol{x}) = 0 \, .$$

Proof. The proof can be found in [26]. ■

A.6 Error Signals for the Output Layer

The error signals of the neurons in the output layer, associated with the loss functions described in this document for one iteration, are, using the relation (3.17):

- **Mean Squared Error (MSE)** (3.11), with $\hat{f}(\boldsymbol{x}) = \hat{y}^{(Z)} \in \mathbb{R}$, so $|V_Z| = 1$:

$$\begin{aligned} \delta^{(Z)} &= \frac{\partial \mathcal{E}}{\partial a^{(Z)}} = \frac{\partial}{\partial a^{(Z)}} \left[\frac{1}{N} \left(y - \hat{f}(\boldsymbol{x}) \right)^2 \right] = \\ &= \frac{\partial}{\partial a^{(Z)}} \left[\frac{1}{N} \left(y - \hat{y}^{(Z)} \right)^2 \right] = \\ &= \frac{\partial}{\partial a^{(Z)}} \left[\frac{1}{N} \left(y - \psi^{(Z)} \left(a^{(Z)} \right) \right)^2 \right] = -\frac{2}{N} \cdot \left(y - \psi^{(Z)} \left(a^{(Z)} \right) \right) \cdot \left[\psi^{(Z)} \left(a^{(Z)} \right) \right]' \, . \end{aligned}$$

Expressing the relation in terms of the network output, $\hat{f}(\boldsymbol{x})$, since for a regression model with mean squared error, the activation function of the last layer is the identity $\psi^{(Z)}(a^{(Z)}) = a^{(Z)}$, then $\left[\psi^{(Z)}(a^Z)\right]' = 1$ and we get:

$$\delta^{(Z)} = \frac{2}{N} \cdot \left(\hat{f}(\boldsymbol{x}) - y \right) \, . \tag{A.1}$$

- **Cross-Entropy** (3.13), with $\hat{f}(\boldsymbol{x}) = \hat{y}^{(Z)} \in [0,1]$, so $|V_Z| = 1$:

$$\begin{aligned}
\delta^{(Z)} &= \frac{\partial \mathcal{E}}{\partial a^{(Z)}} = \frac{\partial}{\partial a^{(Z)}} \left[-y \cdot \log\left(\hat{f}(\boldsymbol{x})\right) - (1-y) \cdot \log\left(1 - \hat{f}(\boldsymbol{x})\right) \right] = \\
&= \frac{\partial}{\partial a^{(Z)}} \left[-y \cdot \log\left(\psi^{(Z)}\left(a^{(Z)}\right)\right) - (1-y) \cdot \log\left(1 - \psi^{(Z)}\left(a^{(Z)}\right)\right) \right] = \\
&= -y \cdot \frac{\left[\psi^{(Z)}\left(a^{(Z)}\right)\right]'}{\psi^{(Z)}\left(a^{(Z)}\right)} + (1-y) \cdot \frac{\left[\psi^{(Z)}\left(a^{(Z)}\right)\right]'}{1 - \psi^{(Z)}\left(a^{(Z)}\right)} = \\
&= \frac{\psi^{(Z)}\left(a^{(Z)}\right) - y}{\left[\psi^{(Z)}\left(a^{(Z)}\right)\right] \cdot \left[1 - \psi^{(Z)}\left(a^{(Z)}\right)\right]} \cdot \left[\psi^{(Z)}\left(a^{(Z)}\right)\right]' .
\end{aligned}$$

Now, since the activation function of the last layer is the logistic sigmoid function (3.7) with $\alpha = 1$, we have:

$$\begin{aligned}
\left[\psi^{(Z)}\left(a^{(Z)}\right)\right]' &= \left[\sigma\left(a^{(Z)}\right)\right]' = \left[\frac{1}{1 + \exp\left(-a^{(Z)}\right)}\right]' = \frac{\exp\left(-a^{(Z)}\right)}{\left[1 + \exp\left(-a^{(Z)}\right)\right]^2} = \\
&= \frac{1}{1 + \exp\left(-a^{(Z)}\right)} \cdot \left[1 - \frac{1}{1 + \exp\left(-a^{(Z)}\right)}\right] = \sigma\left(a^{(Z)}\right) \cdot \left[1 - \sigma\left(a^{(Z)}\right)\right] = \\
&= \left[\psi^{(Z)}\left(a^{(Z)}\right)\right] \cdot \left[1 - \psi^{(Z)}\left(a^{(Z)}\right)\right] .
\end{aligned}$$

Thus, for cross-entropy, we get:

$$\delta^{(Z)} = \hat{f}(\boldsymbol{x}) - y . \tag{A.2}$$

- **Categorical Cross-Entropy** (3.15), with $\hat{f}(\boldsymbol{x}) = \hat{\boldsymbol{y}}^{(Z)} \in [0,1]^K$, thus $|V_Z| = K$:

$$\begin{aligned}
\delta_n^{(Z)} &= \frac{\partial \mathcal{E}}{\partial a_n^{(Z)}} = \frac{\partial}{\partial a_n^{(Z)}} \left[-\sum_{k=1}^{K} y_k \cdot \log\left(\hat{f}_k(\boldsymbol{x})\right) \right] = \\
&= \frac{\partial}{\partial a_n^{(Z)}} \left[-\sum_{k=1}^{K} y_k \cdot \log\left(\psi_k^{(Z)}\left(a_k^{(Z)}\right)\right) \right] = -y_n \cdot \frac{\left[\psi_n^{(Z)}\left(a_n^{(Z)}\right)\right]'}{\psi_n^{(Z)}\left(a_n^{(Z)}\right)} .
\end{aligned}$$

By deriving the activation function of the last layer, which in this case is the n-th term of the *softmax* function (3.8):

$$\begin{aligned}
\left[\psi_n^{(Z)}\left(a_n^{(Z)}\right)\right]' &= \left[\frac{\exp\left(a_n^{(Z)}\right)}{\sum_{k=1}^{K} \exp\left(a_k^{(Z)}\right)}\right]' = \frac{\exp\left(a_n^{(Z)}\right)}{\sum_{k=1}^{K} \exp\left(a_k^{(Z)}\right)} \left[1 - \frac{\exp\left(a_n^{(Z)}\right)}{\sum_{k=1}^{K} \exp\left(a_k^{(Z)}\right)}\right] = \\
&= \left[\psi_n^{(Z)}\left(a_n^{(Z)}\right)\right] \cdot \left[1 - \psi_n^{(Z)}\left(a_n^{(Z)}\right)\right] .
\end{aligned}$$

Thus, the error signal for the n-th neuron in the output layer is obtained as:

$$\delta_n^{(Z)} = -y_n \cdot \left[1 - \hat{f}_n(\boldsymbol{x})\right] . \tag{A.3}$$

A.7 Pearson Correlation Coefficient

The Pearson correlation coefficient between two random variables measures their linear dependency, or, in other words, the degree of covariation between them. For two samples, the interpretation is analogous: the degree of linear relationship between them.

Definición A.7.1. *Let $\{x_i\}_{i=1}^N$ and $\{y_i\}_{i=1}^N$ be two data samples of size N, the sample Pearson correlation coefficient, r_{xy}, is defined as*

$$r_{xy} = \frac{\sum_{i=1}^N (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^N (x_i - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^N (y_i - \bar{y})^2}} ,$$

where $\bar{x}$ and $\bar{y}$ are the sample means: $\bar{x} = \frac{1}{N}\sum_{i=1}^N x_i$, $\bar{y} = \frac{1}{N}\sum_{i=1}^N y_i$.

This correlation coefficient can vary between -1 and 1, with both extremes indicating perfect linear dependence (inverse or direct, respectively). On the other hand, when $r_{xy} = 0$, it indicates no linear dependence between the samples [15].

A.8 Confusion Matrix

Confusion matrices are tools that allow visualizing the accuracy and performance of a predictive model in a classification task. They show a square matrix with the number of correct and incorrect cases predicted based on the true label.

Definición A.8.1. *A confusion matrix is a square matrix of size $K \times K$ that includes the number of predictions for the K classes in the rows, considering the respective true value of each data point in the K columns.*

Thus, a confusion matrix includes correct predictions (terms in the diagonal), false positives (terms in the lower triangular part), and false negatives (terms in the upper triangular part). This can be seen in Figure A.1. It is worth noting that this matrix can also be represented as its transpose.

In general, the color scale presented in the matrix corresponds to the performance percentage by rows, that is, it represents the percentage of predictions that assigned a particular class and were correct or incorrect. However, in cases with highly imbalanced classes, the values displayed are not percentages but absolute records [3].

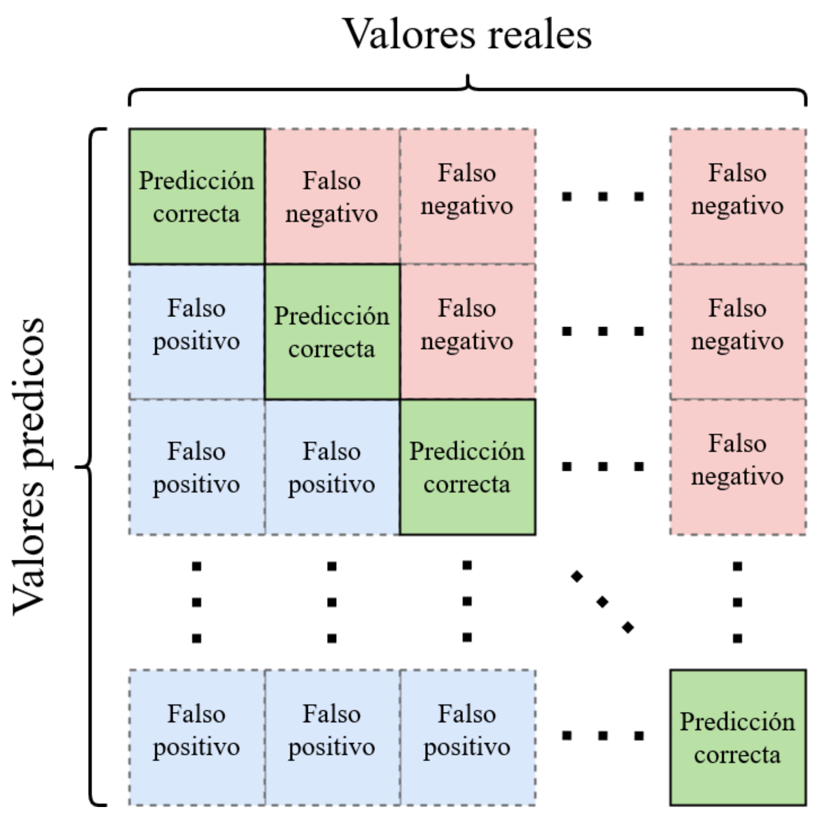


Figure A.1: *Typical confusion matrix in a classification model.*

Bibliography

[1] A. Altaee, M. Hafiz, A. Al Hawari, R. Alfahel, M. Hassan, S. Zaidi, and A. Yasir. Impact of High Turbidity on Reverse Osmosis: Evaluation of Pretreatment Processes. *Desalination and Water Treatment*, 208:96–103, 08 2020.

[2] G. Bachman and L. Narici. *Functional Analysis.* Dover Publications, Mineola, New York, second edition, 2000.

[3] E. Beauxis-Aussalet and L. Hardman. Visualization of Confusion Matrix for Non-Expert Users. 2014.

[4] C. M. Bishop. *Pattern Recognition and Machine Learning.* Information Science and Statistics. Springer, Berlin, Germany, August 17 2006.

[5] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Routledge, New York, 2nd edition, 1988.

[6] J. M. Cohen, S. Kaur, Y. Li, J. Z. Kolter, and A. Talwalkar. Gradient descent on neural networks typically occurs at the edge of stability, 2022.

[7] T. M. Cover. Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition. *IEEE Transactions on Electronic Computers*, EC-14(3):326–334, 1965.

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

[9] M. P. Deisenroth, A. A. Faisal, and C. S. Ong. *Mathematics for Machine Learning.* Cambridge University Press, Cambridge, UK, April 2020.

[10] R. E. Edwards. *Functional Analysis: Theory and Applications.* Dover Publications, 1995.

[11] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and M. Titterington, editors, *Proceedings of the Thirteenth International*

Conference on Artificial Intelligence and Statistics, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[12] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[13] S. Haykin. *Neural Networks and Learning Machines.* Pearson, Upper Saddle River, NJ, 3rd edition, November 2008.

[14] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[15] R. V. Hogg, J. W. McKean, and A. T. Craig. *Introduction to Mathematical Statistics.* Pearson, 8th edition, 2019.

[16] S. Kim. Mathematical Foundations of Machine Learning, Enero 2024.

[17] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization, 2017.

[18] P. M. Lee. *Bayesian Statistics: An Introduction.* Wiley, 4th edition, 2012.

[19] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

[20] X. Li and W. Yu. Dynamic system identification via recurrent multilayer perceptrons. *Information Sciences*, 147(1):45–63, 2002.

[21] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang. The expressive power of neural networks: A view from the width. In *Advances in Neural Information Processing Systems*, volume 30, pages 6231–6239. Curran Associates, 2017.

[22] V. Maiorov and A. Pinkus. Lower bounds for approximation by mlp neural networks. *Neurocomputing*, 25(1):81–91, 1999.

[23] P. M. Menéndez. *Condiciones de Karush-Kuhn-Tucker.* Universidad Complutense de Madrid, 2024.

[24] P. M. Menéndez. *Teoría de Grafos.* Universidad Complutense de Madrid, 2024.

[25] J. Mercer. Functions of positive and negative type and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society A*, 209(441–458), 1909.

[26] A. V. Morales. *Apuntes de GC.* Universidad Complutense de Madrid, 2018.

[27] N. J. Nilsson. *Introduction to Machine Learning.* Robotics Laboratory, Department of Computer Science, Stanford University, Stanford, CA, November 3 1998. E-mail: nilsson@cs.stanford.edu.

[28] E. Orhan. Cover's Function Counting Theorem, December 16 2014. eorhan@cns.nyu.edu.

[29] F. Ortega, A. González-Prieto, and J. Bobadilla. Providing reliability in recommender systems through bernoulli matrix factorization. *Information Sciences*, 553:110–128, Apr. 2021.

[30] B. D. Ripley. *Pattern Recognition and Neural Networks.* Cambridge University Press, 1996.

[31] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[32] J. M. V. Rubio. Origen y desarrollos actuales de la predicción meteorológica. *Encuentros Multidisciplinares*, 2013.

[33] M. Ruiz-Garcia, G. Zhang, S. S. Schoenholz, and A. J. Liu. Tilting the playing field: Dynamical loss functions for machine learning, 2021.

[34] A. M. Schäfer and H. G. Zimmermann. Recurrent neural networks are universal approximators. *International Journal of Neural Systems*, 17(04):253–263, 2007. PMID: 17696290.

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

[36] Z. Shen, H. Yang, and S. Zhang. Optimal approximation rate of ReLU networks in terms of width and depth. *Journal de Mathématiques Pures et Appliquées*, 157:101–135, 2022.

[37] N. Srivastava, G. Hinton, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014.

[38] R. C. Staudemeyer and E. R. Morris. Understanding LSTM - a tutorial into Long Short-Term Memory Recurrent Neural Networks. *ArXiv*, abs/1909.09586, 2019.

[39] V. N. Vapnik. *The Nature of Statistical Learning Theory.* Springer, New York, 2000.

[40] P. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[41] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

[42] D. Wolpert and W. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.