Data Driven Initialization for Machine Learning Classification Models

Author
David Santiago López Jaimes

Work presented as a requirement to opt for the title of Applied Mathematics and Computer Science

Director, Tutor
Alexander Caicedo Dorado

School of Engineering, Science and Technology
Applied Mathematics and Computer Science
Rosario University

Bogotá - Colombia
2022

Abstract

Thanks to the great advance technology has had, the increase in computer resources and the strong impact that the era of "big data" has had on society, artificial intelligence has become a highly studied and used area. Machine learning is a branch of artificial intelligence which main objective is to build models that are capable of learning from a set of data, without the need to be explicitly programmed. These models use tools from different branches of mathematics, such as statistics and linear algebra, to identify patterns and relationships between a set of data.

Regarding machine learning, it allows us to generate models that are capable of classifying a set of data based on its intrinsic characteristics and its relationship with an objective variable. These models are widely used in real-life problems, such as classifying a bank transaction as malicious or normal, determining with a certain probability whether a tumor is malignant or benign, estimating a person's credit risk, among others.

Most of these classification models learn through the use of gradient descent or its variations. This is an iterative algorithm which allows finding the parameters θ of the model that minimize a cost function and allow an adequate classification. These parameters are initialized randomly. However, there are several limitations when training these models. The real data is in considerably large dimensions, and it is difficult to know the shape of the cost surface that is generated with it. This causes the models to require a lot of care, and a large amount of time and computational resources for their training. On the other hand, due that in most cases the cost function is not convex as it normally happens in neural networks, it is possible that when initializing the weights randomly, the algorithm stalls because it was initialized in a flat region of the cost function, or that it initializes in a very rough region and does not converge to an appropriate minimum.

This is way the present study aims to propose an initialization strategy for classification problems that initialize the models in an appropriate region of the cost function in order to improve its convergence rate and produce better results in faster training times. We aim to propose a new deterministic initialization strategy for the logistic regression (linear classifier), the multinomial logistic regression and the classical neural networks (fully connected feed-forward) for classification problems.

We proposed an initialization strategy based on the properties of the statistical distribution of the data on which the models are trained. For the logistic regression and the multinomial logistic regression we propose to initialize the models with a characteristic vector of the data distribution of each class, such as its mean or median. In the fully connected feed-forward neural networks we propose to use prototype data of each one of the classes. These prototype data are not the most representative data of the entire class distribution, but in this case, they are data that map and linearize the separation boundary with the other classes.

A benchmark for the initialization proposal was made using various real datasets for classification tasks from the UCI and Kaggle repositories. We also tested the proposed initializations with different toy examples.

In the logistic regression, we compared the behavior of the model using random initialization and using the proposed initialization. For fully connected feed-forward neural networks, we compared the behavior of the neural networks using the proposed initialization and the state of the art initializations for these models,

Xavier's and He's initialization. In both cases, we were able to successfully initialize the models reducing it's required training time and making the learning algorithm start in a better region of the cost function. Codes and tables are available at https://bitbucket.org/davidsantiago1011/thesis-ml/.

In this way, we proposed new initialization strategies for the multinomial logistic regression and the neural network models for classification problems. The logistic regression initialization is based on statistical estimators of the data distribution and distance metrics, particularly the mean and the euclidean distance between different scalar products. The neural networks strategy is based on the decision boundary linearization using prototype data of each one of the classes. We have seen that our approach works very well for all the tested datasets, considerably reducing the computational resources required for the training of these models and increasing their performance.

Acknowledgements

First of all, I want to thank my tutor Alexander Caicedo-Dorado, for introducing and guiding me through the fields of computer science that now are my motivation. I thank him for his unconditional support, not only in the development of this thesis, but also throughout my undergraduate program. I want to thank Professor Santiago Alferez for his constant support, interest and collaboration in this thesis. I also want to thank Professor Margot Salas for her life lessons and for believing in me since the very beginning. I am thankful for the compassion and empathy they always showed through this process, even in the hardest times, always believing in the things that I am capable of achieving, I won't let you down.

I feel very fortunate to have had the opportunity to share a classroom with excellent teachers who not only shared their knowledge with me, but also motivated me to achieve excellence throughout my career. I want to thank my classmates, for being an academic and emotional support throughout this process, without them the path would have been much more complicated.

Finally, I want to thank my family for always believing in me, supporting me in the most difficult moments of this life stage, and teaching me that I set my own limits. This achievement is dedicated to you.

Contents

A mi familia, por motivarme y enseñarme el verdadero valor de la vida.

Chapter 1

Introduction

1.1 Machine Learning

Artificial intelligence is an area of computer science that aims at developing systems that can mimic human behaviour in various tasks and environments. Some fields of Artificial intelligence study how systems can learn form experience, what does it mean to develop cognitive abilities, as well as to study sensations and feelings. The field of artificial Intelligence was founded in the Dartmouth workshop in 1956, where several distinguish scientist gather together to discuss topics related to different aspects of learning and intelligence. Artificial Intelligence was heavily explored by the mathematician Alan Turing in his publication "Computing Machinery and Intelligence" [1]. Though the expectations for the development of this field were huge, it took several decades for Artificial intelligence to produce a huge impact. Nowadays, thanks to the advancement of technology, the increase in computing resources and, above all, the great impact that the era of "big data" has had on society artificial intelligence has become more popular.

One of the main branches of artificial intelligence is machine learning, which main objective is to develop models that are capable of learning from data, without the need to be programmed in an explicit way [2]. Currently, these models are used in a large number of applications ranging from recommendation systems on streaming platforms, to image classification to detect malignant tumors in medicine. The truth is that machine learning has become such a popular topic that it is not only the basis for the success of many startups, but it is also the subject of study in various scientific fields. This success is mainly due to the fact that machine learning is a multidisciplinary field which foundations had been, and are being, developed in areas such as mathematics, statistics, linear algebra and topology.

Machine learning models can be categorized into supervised, unsupervised and self-supervised learning. Supervised learning assumes that there is a ”supervisor” variable that knows the exact value of the target that is intended to be estimated, in the case of regression models, or the classes of data that are available for training these models, in the case of classification models. Meanwhile, unsupervised learning aims at learning characteristics of the data from patterns that are obtained heuristically, without knowing this information a priori. Self-suppervised learning consists of learning through information obtained from evaluations of the system’s output, in this context self-suppervised learning is known as reinforcement learning when the learning process is updated based on from trial and error through interactions of the system with its environment [3].

Therefore, if we focus on the way the different models learn, it is possible to say that supervised learning models learn by adjusting their internal parameters via an error signal between the known and the estimated output, unsupervised learning models adjust their parameters by learning from the intrinsic information of the data, and reinforcement learning models use a reinforcement function to adjust their local parameters [3].

Supervised learning problems can be categorized into regression and classification problems. In both cases, there is a series of observations with a certain number of predictors, regressors or descriptor variables, and the aim is to build a model that can predict the value of a specific target from those observations. The main difference is that in regression problems the target is a continuous value and in the case of classification it is a categorical value [1].

Predictive models of company's shares are a good example of regression problems. These models are widely used today by various organizations because they seek to predict the value of company's shares based on various factors, including the values of shares in the past. Consequently, although regression models are very popular, classification models are not far behind, in the field of medicine these models have had a great impact because they are widely used to classify all types of images, specifically, one of its main uses is the classification of tumors, in which various images of tumors are processed to build a model that is capable of determining whether a tumor is malignant or benign as well as their location.

In the case of unsupervised learning models, one of their main topics under study is clustering, which consists in grouping the observations using a fixed number of clusters based on the similarity of their characteristics. Clustering has the ability to reduce the complexity of the data by grouping it into small clusters, where the members of each cluster share similar characteristics [4]. Unlike supervised learning problems, in this case there is no target or objective variable, but the models learn the intrinsic characteristics of the data. Clustering has a large number of applications in various areas such as environmental engineering for the exploratory analysis of meteorological data [4], medicine for the recognition of new Alzheimer's patterns [5] and economics to understand patterns in the global economy [6].

1.2 Training of Machine Learning Models

In this section we will discuss different aspects related to the training of machine learning models. The discussion will be centered around gradient descent, more specifically to the algorithms itself, the influence of some hyper-parameters, as well as the initialization strategies.

1.2.1 Gradient Descent

The main goal of machine learning, in general, can be expressed in terms of optimizing an objective function $J(\theta)$ over a domain $\theta \in \Omega$ [7]. In this context learning constitute the process of finding the parameters that optimize the objective function. There are various learning algorithms, some more complex than others, however,

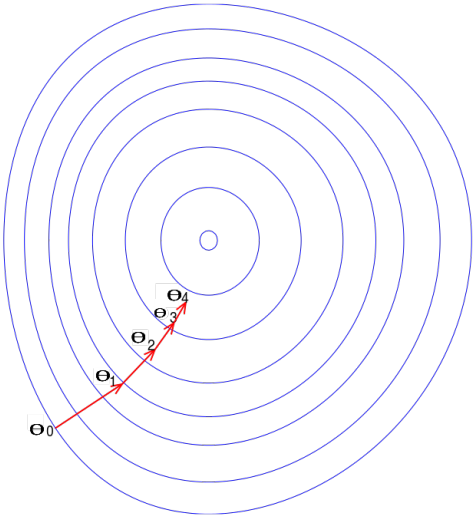

FIGURE 1.1: Illustration of gradient descent iterative procedure.
Credit: wikipedia

one of the most popular and flexible is gradient descent; an iterative algorithm that seeks to minimize a continuously differentiable function, which is called the cost function $J(\theta)$, that depends on the parameters $\theta \in \mathbb{R}^p$ of the model [8].

This method is originally attributed to the mathematician, engineer and physicist Augustin-Louis Cauchy who initially proposed it in 1847 [8]; and as expected in iterative methods, the algorithm updates the parameters θ in each iteration based on the fact that if a multi-variable function $G : D \to K$ is defined and differentiable in a neighborhood of a point $\theta \in D$, then G increases faster if it goes from θ in the direction of the gradient of G evaluated at θ [1]. Therefore, this method seeks to minimize the cost function $J(\theta)$, which by hypothesis is defined and differentiable in the entire real domain. This minimization occurs by updating the parameters θ in the opposite direction to the gradient of the cost function, $\nabla_\theta J(\theta)$.

Gradient descent starts with some randomly chosen initial parameters, $\boldsymbol{\theta}_0$, and iteratively updates them as shown in equation (1.1) below:

$$\begin{aligned} \theta &= \theta + \Delta\theta, \\ \Delta\theta &= -\eta \frac{\partial J}{\partial \theta}; \end{aligned} \tag{1.1}$$

where $\eta \in \mathbb{R}_+$ is a hyperparameter of the model, known as the learning rate. Thus, according to equation (1.1), a sequence of parameters $\{\boldsymbol{\theta}^0, \boldsymbol{\theta}^1, ...\}$ is computed, which, with a proper setting of the algorithm hyperparameters, will converge to the optimal parameters, $\hat{\boldsymbol{\theta}}$; i.e. the parameters that produce a minimum of the cost function $J(\theta)$. Figure 1.1 shows the graphical interpretation of gradient descent using level curves from the cost function.

Algorithm 1 shows the pseudocode for gradient descent. It can be seen that the algorithm starts in a random point and iterates until convergence.

Input: The training set $\{x_i, y_i\}_{i=1}^n$ with $x_i \in \mathbb{R}$, $y_i \in \mathbb{N}_+$, $\forall i \in \{1, 2, \ldots, n\}$, the learning rate η, the activation function h and the cost function J.
Process:
Initialize randomly the parameters θ
Repeat until convergence
  Shuffle $\{x_i, y_i\}_{i=1}^n$
  for *all* $x_j, y_j \in \{x_i, y_i\}_{i=1}^n$ **do**
    $\theta = \theta - \eta \nabla_\theta J(h(x_i \theta^T), y_i)$
  end
Output: The parameters θ

Algorithm 1: Stochastic Gradient Descent Algorithm.

1.2.2 Optimizers

Due to the popularity and effectiveness of gradient descent as a learning algorithm for various machine learning models, and to the availability of large datasets, gradient descent has been used as the based algorithm for the development of different optimizers, which main task is to learn the model parameters, θ.

Stochastic gradient descent, SGD, is perhaps the most naive implementation of Gradient descent. In a nutshell SGD update the parameters by taking a random sample from the observations and repeat the process until convergence. Though this strategy is fastest in the computations needed to update the parameters, it takes more time to converge to the minimum. The update formula for SGD is presented in equation (1.2).

$$\theta := \theta - \eta \nabla_\theta J(\theta; \mathbf{x}^{(i)}; \mathbf{y}^{(i)}). \tag{1.2}$$

SGD is fast in its computation, however, the path that it follows to reach the minima is *"messy"* due to the changes in direction caused by the evaluation of the gradients using individual samples [9]. Apart from this SGD has problems when it reaches inflexion points or local minima in the cost function surface.

In order to solve the problems due to inflexion points as well as local minima, the SGD algorithm was modified by including a *momentum* term. The inclusion of momentum accelerates SGD in the optimal direction. As mentioned before, SGD performs well over rough surfaces but it can present problems near local minima and inflexion points [9]. SGD with momentum adds, at each iteration, a fraction γ of the update vector from the previous iteration as shown below:

$$\begin{aligned} \mathbf{v_t} &= \gamma \mathbf{v}_{t-1} + \eta \nabla_\theta J(\theta), \\ \theta &:= \theta - \mathbf{v}_t. \end{aligned} \tag{1.3}$$

The term $\mathbf{v}_t$ in equation (1.3) can be seen as a term that keeps information concerning the previous states for the gradient in the cost function. This term has the particularity that it increases for dimensions whose gradients point in the same directions and decreases for dimensions whose gradients change direction, which penalizes dimensions that have an abrupt change in its gradient to obtain a better convergence rate, this term also helps to escape from local minima as well as saddle

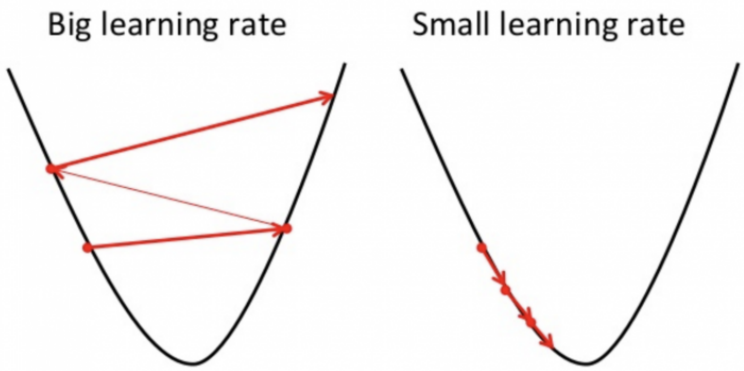

FIGURE 1.2: Illustration of a poor choice in gradient descent learning rate. In the left figure the case when the learning rate is too large, causing the algorithm to diverge, whilst in the right figure the case when the learning rate is too small, which causes the algorithm to take a long time to converge. *Credit: researchgate*

points in the cost function [9].

Apart from SGD and SGD+Momentum we can also find **ADAGrad** [10], which penalizes the update in the directions where the gradient of the cost function is larger. This optimizer now uses mini-batch of data, instead of using random individual samples from the observations, this helps to smooth the direction of the parameter updates. **ADAGrad** helps to identify a more precise direction that points towards the minimum, by dividing by the magnitude of the cumulative gradient in each direction. However, **ADAGrad** has problems with its convergence rate, since it can decrease too fast the relative learning rate producing a slow convergence. To solve this problem one may keep a weighted average of the gradients of the functions, mitigating the effect of the exploding cumulative gradient, the optimizer using this strategy is called **RMSProp** [11]. Additionally, when the momentum and the correction term for the direction used in **ADAGrad** are combined we obtained the optimizer **ADAM** [12].

1.2.3 Learning Rate

In all the different optimizers, the initial value for the learning rate, η, has to be selected. The learning rate is a hyper-parameter that represents the size of the learning step, i.e it indicates how much the parameters are moved in the opposite direction of the gradient in each iteration, in search for the optimal parameters $\boldsymbol{\theta}$.

The learning rate is one of the most important hyperparameters, which affects directly the convergence of the algorithm. If we have a learning rate that is too small the algorithm will take a long time to converge to the minimum; in contrast, a large learning rate will cause the algorithm to diverge. An illustration of this effect is depicted in figure 1.2 [1].

Apart form the learning rate, there are many other factors such as data preprocessing, regularization, initialization of learning algorithms, among others, which

affect the convergence of the algorithms. The initialization of the parameters has become a major research object in recent years and it is the focus of study for this thesis.

1.2.4 Initialization

Historically most machine learning models are initialized by means of small random values from a Gaussian distribution or an uniform distribution, regardless of the learning algorithm. As second option, in some models as logistic regression, the parameters can be initialized to zero. However, there are other more advanced techniques among them **Xavier Initialization** and **He Initialization** techniques are popular nowadays due to their efficiency and efficacy in the training of neural networks.

Xavier Initialization technique was proposed by Xavier Glorot and Yoshua Bengio in the paper "Understanding the Difficulty of Training Deep Feedforward Neural Networks" [13]. This technique consists of initializing the parameters with random values in the interval $[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$ from an uniform distribution U, where n is the size of data (number of features). There is a variation of this technique called **Xavier Normalized Initialization** which, like **Xavier Initialization**, consists of using random values obtain from a normal distribution, but bounded by the interval $[-\frac{\sqrt{6}}{\sqrt{n+m}}, \frac{\sqrt{6}}{\sqrt{n+m}}]$ where m is the number of output data from the model. This technique, according to various experiments, has a better convergence rate for more complex models such as neural networks.

Xavier initialization techniques were originally designed for linear activation functions but were extended, with satisfactory results, to the non-linear “Sigmoid" and "Tanh" functions. However, for the case of the non-linear activation function “ReLU”, several problems were evident [14]. Therefore, the **He Initialization** technique emerged, which was originally proposed by Kaiming He in his paper "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification" [15]. This technique consists of initializing the parameters with random values from a Gaussian distribution $G(0; \sqrt{2/n})$ with a mean of 0.0 and a standard deviation of $\sqrt{2/n}$ where n is the size of data (number of features). As it can be seen its initialization strategy is inspired in Xavier initialization technique, however, since half of the distribution in the activation function ReLU will be negative, then He compensated this by including the factor of 2 in its normalization.

Even though Xavier and He are among the state of the art when considering initialization strategies for neural networks, they do not use information concerning the input data distribution. This information can be useful when training the models. How to use the information contained in the data in order to properly initialize classification models in machine learning will be the central topic for this work.

1.3 Problem Statement

Most machine learning algorithms learn by means of gradient descent or its variations. This means that the algorithm initialize the parameters randomly, and iteratively update the parameters in order to reach a minima in the cost function. The shape of the cost function is embedded in a high dimensional space, and it depends

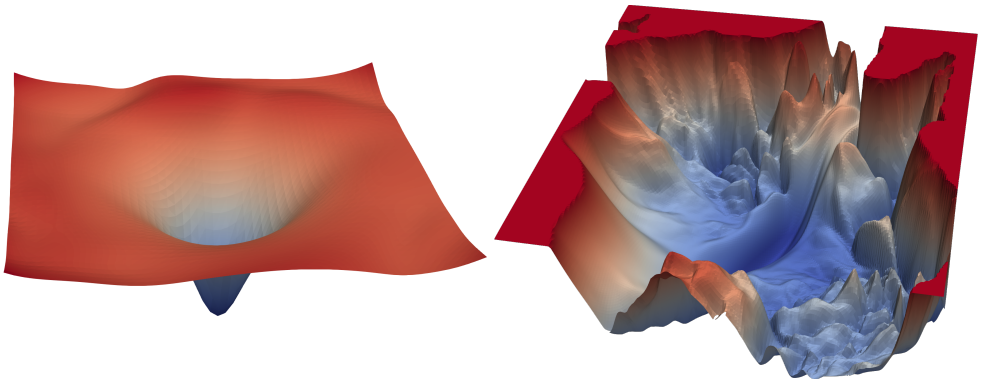

FIGURE 1.3: Flat and rough surfaces produced by cost functions

not only on the parameters, but also on the architecture of the neural network, and the input data. Therefore, it is extremely complicated to know the behaviour of the cost function in different regions, which complicates the training of these models. In addition since the cost function is often a non-convex function with many local minima and saddle points, it is possible that by randomly initializing the weights to cause the learning algorithm to get *"stuck"*, so that, in its training phase, the model can never reach an appropriate local minimum. Therefore, a random initialization can lead to flat areas of the cost function during training, and difficult convergence to an optimal minimum; or the initialization can lead to very rough areas that contain many local minima, which causes the algorithm to converge to minima that are not optimal. The visualization of the cost function surfaces is problematic, however, in [16] the authors developed a method to visualize in 3D the behaviour of these surfaces. Figure 1.3 presents some of the cost surfaces that can be generated by neural networks.

In this thesis we propose a new initialization strategy which takes into consideration the distribution of the input data by classes. We hypothesise that his initialization localize the model in a more convenient region in the cost function, in such a way that the training time is reduced and it converges to a more stable minima. We explore this initialization technique for logistic, multinomial and NN classifiers.

1.4 Objectives

The main objective of this thesis is to develop a strategy for the initialization of the θ parameters in various machine learning classification models, namely logistic and multinomial regression, as well as feed-forward NN. Which instead of being based on randomness takes into account statistical properties of the training data.

In order to reach the main objective for this thesis we propose the following specific objectives:

1. To develop a data base strategy for the initialization of the parameters for logistic regression.
2. To extend this strategy for multinomial regression.

3. To extend the developed strategy to classification models using classical artificial neural networks.

4. To evaluate the developed strategies using a benchmark with different datasets.

1.5 Outline

The rest of this thesis is organized as follows:

- Logistic regression is described in **chapter 2.** For this model we present the cost function that is used in order to train the model as well as its relation with neural networks. In this chapter we also include the proposed data driven initialization for logistic regression and its extension to multinomial regression. This strategy will be the base for initialization of neural networks.

- **Chapter 3** discusses classical fully connected feed-forward neural networks, their training algorithms -Backpropagation and gradient descent-, as well as the main challenges these models face during training. Here we also introduce the extension of the initialization technique presented in Chapter 2 for NN.

- **Chapter 4** is dedicated to describe and present results from the benchmarking of the proposed algorithms.This chapter explains the datasets used and compares the results for logistic regression models included in the scikit-learn library of Python, with the results provided by training using our initialization strategy. We also present the results provided by the random initialization of neural networks and the ones obtained with our proposal.

- In **chapter 5** we discuss the results presented in chapter 4, and we contrast our strategy with other works found in the literature.

- Finally, **chapter 6** summarizes the conclusions and discusses possible future work directions.

Chapter 2

Data Driven Initialization for logistic regression

This chapter describes the logistic regression model as a solution to binary classification problems. In addition, we describe the extension of classical logistic regression to multi-class classification problems, by means of the *one vs rest* strategy; and also, we present this extension using multinomial logistic regression model. At the end of the chapter we present the data driven strategy for initialization of logistics models.

2.1 History of the Logistic Regression

Classical logistic regression is a statistical method that aims to model a categorical dependent variable, from a data set composed of various independent regressors, in order to describe the relationship between those regressors and the response variable [17].

Logistic regression, in the field of classification problems, can be seen as a linear classifier, which is originally attributed to the American statistician and physicist Joseph Berkson in 1944. Although he was not who originally proposed it, he developed it in such a way that it became a general logistic model in different areas, which became a more general alternative to the *probit* model; a model based on regression that was proposed in 1934 and was the most popular at that time. However, this model had the limitation that the dependent variable was strictly binary, therefore, the flexibility of the *logit* model marked a revolution in various areas of statistics [18].

After 1944, logistic regression had a large number of refinements, such as the one from David Cox (1958), which become, not only one of the most used models in the area of machine learning for solving classification problems, but also, one of the most solid basis of many other more complex models, such as neural networks [19].

Logistic Regression has had a great impact in different fields, for instance, just to name a few examples in the areas of sociology and its research fields such as socio-sanitary [20], in medicine with the classification of tumors as venign or malignant [21], in the area of economics with the stability of global economic factors analysis [22] and even in research on various areas of mathematics.

2.2 Logistic Regression Model

The logistic regression model is a linear classification model, whose main objective is to determine the probability that an observation belongs to one class or another, based on a finite set of training data. The output of the model, in the binary case, in which the sample space Ω is constituted by the two different classes $\{0, 1\}$ can be interpreted as the probability of success of the response event.

In this sense, logistic regression models the target variables as a binomial probability distribution function, which is represented as the repetition of simulated events based on a Bernoulli distribution. Likewise, the model assumes the existence of a linear relationship between the predictors and the log-odds (the logarithm of the odds) of the probability of success of an observation $X = x$.

$$\log \frac{P(\text{success}|X = x)}{1 - P(\text{success}|X = x)} = \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle \,, \tag{2.1}$$

with $\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle$ the dot product between the parameters vector $\boldsymbol{\theta}$ and the observation $\boldsymbol{x}$. Therefore, the model uses a composition of a sigmoid function $\phi_{\text{sig}} : \mathbf{R} \to [0, 1]$ and a linear function $L_{\boldsymbol{\theta}} : \mathbf{R}^d \to \mathbf{R}$ to constitute its probabilistic output (in the interval $[0, 1]$) denoted as $h_{\boldsymbol{\theta}}(\boldsymbol{x})$, where $\boldsymbol{x}$ is the training set embedded in a d-dimensional space.

In particular, this model uses the logistic function, which is statistically represented as the cumulative distribution function of the logistic distribution. This distribution, due to its high kurtosis as shown in figure 2.1, has had a great impact on various mathematical models of exponential growth such as population growth, epidemiological propagation and even the absorption capacity of different materials [23].

On the other hand, as a linear function, this model uses the scalar product $L_{\boldsymbol{\theta}}(\boldsymbol{x}) = \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle$ between it's parameters and the set of regressors. The function shown in figure 2.2 is established as the activation function of the model for binary classification problems, and it is precisely this that allows logistic regression to model the relationship between a set of predictors and the probability occurrence of an event [18]:

$$h_{\boldsymbol{\theta}} = \phi_{\text{sig}} \circ L_{\boldsymbol{\theta}} = \left\{ \boldsymbol{x} \to \phi_{\text{sig}}(\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle) = \frac{1}{1 + e^{-\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle}} \;\middle|\; \theta \in \mathbf{R}^d \right\}. \tag{2.2}$$

The logistic function has the shape of S due to its two horizontal asymptotes located at $y = 0$ and $y = 1$, its point of inflection in $x = 0$ and monotonic growth, which is due to the non-negativity of its first derivative. This, as will be seen later, will be of great influence for training the model. However, the main reason why the logistic model uses the *sigmoid* function is that it performs a smooth projection of a space, i.e. the product between the regressors of the data and the parameters of the model is mapped to a probabilistic space [25].

It is worth mentioning that although the logistic regression, in itself, is not linear, it performs a linearization through the *sigmoid* function, which produces hyperplanes for the decision boundaries. This is interesting, because linear decision boundaries cannot always guarantee correct division of classes for certain data structures.

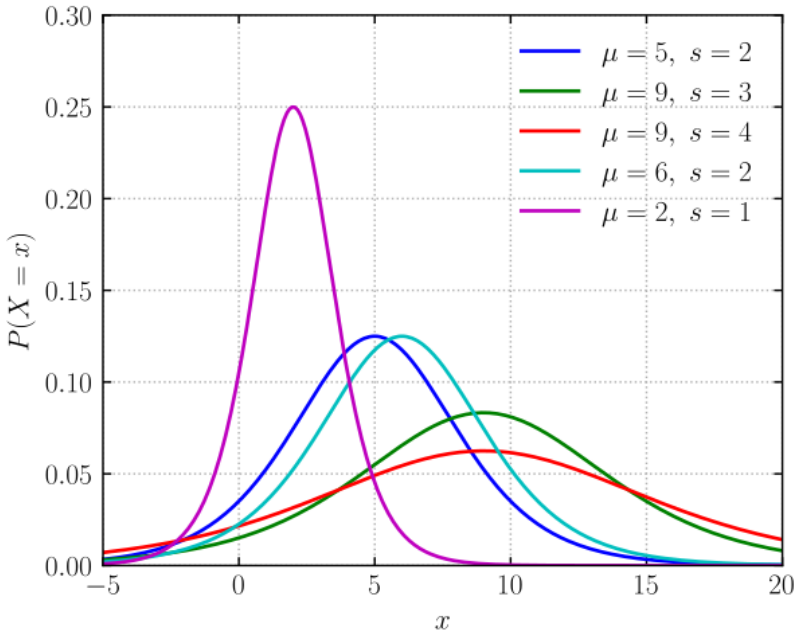

FIGURE 2.1: Illustration of the logistic probabilistic distribution function for different parameters. This distribution has a high kurtosis - statistical measure that determines the data concentration around the center of the frequency distribution [24] - this allows it to increase the probability of having extreme values ($\pm 3\sigma$ from the mean), which is very useful in population models. *Credit: wikipedia*

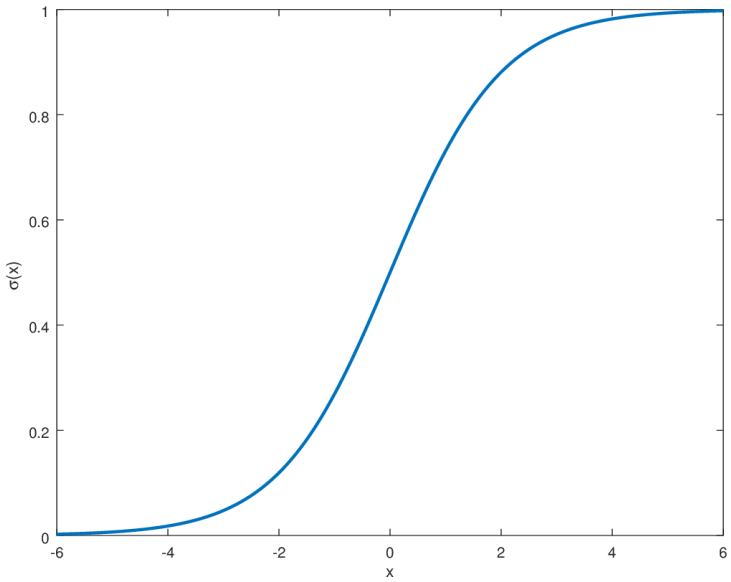

FIGURE 2.2: Sigmoid function. *Credit: github*

Therefore, to create nonlinear classification models the input regressors should be transformed, or more complex models such as neural networks should be considered.

2.3 Cost function for logistic regression models

To train logistic regression models we use gradient descent, which minimize a cost function $J(\boldsymbol{\theta})$. Therefore, the training problem is reduced to finding a cost function that characterizes the error of predicting $h_{\boldsymbol{\theta}}(\boldsymbol{x}) \in [0,1]$ given that the class to which an observation $\boldsymbol{x}$ belongs to is $\{0\}$ or $\{1\}$.

Let us consider a classification problem in which there are n observations, each with p predictors, and there are $1, 2, ..., K$ different classes. Suppose that the data is strictly labeled, that is, that every observation belongs to one and only one of the existing K classes.

Since we have been talking about binary classification problems, first, let us suppose that $K = 2$ with the labels $\{0, 1\}$. Before going into detail with the cost function, it is important to emphasize that, due to the characteristics of *sigmoid* function presented in figure 2.2, the prediction threshold is normally located at 0.5, that is, if $h_{\boldsymbol{\theta}}(\boldsymbol{x}) < 0.5$ the observation is predicted to belong to class $y = 0$, otherwise the model output is $y = 1$. This is mainly due to the fact that the function *sigmoid*, as mentioned before, presents a monotonic growth centered at $y = 0.5$.

Since a large output of the model $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ represents a large probability of success in the target event, we want a cost function that penalizes the model when its output $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ is very large, but the class to which the observation $\boldsymbol{x}$ belongs is $y = 0$. Similarly, the cost function must penalize when $1 - h_{\boldsymbol{\theta}}(\boldsymbol{x})$ is very large, but the class of $\boldsymbol{x}$ is $y = 1$.

Therefore, let's consider the binary Cross-Entropy/Log Loss function in equation 2.3 defined as:

$$J(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log p(y_i) - (1 - y_i) \log \left[1 - h_{\boldsymbol{\theta}}(\boldsymbol{x})\right] \tag{2.3}$$

where $y_i \in \{0, 1\}$ is the class to which the i-th observation belongs to, and $p(y_i)$ is the predicted probability of success for the same observation.

If we analyse the binary Cross-Entropy function presented in equation 2.3, we realize that for each observation belonging to the i-th class, the model predicted logarithmic probability that the observation actually belongs to the i-th class $\log [p(y = i)]$ is added to the loss. Therefore, if the model predicted probability $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ is very high, but the real class of the observation is $y = 0$, the sum to the loss will be very negative and in turn, the model will be penalized.

On the other hand, if we consider that the activation function has an exponential growth, and therefore, the output of the model will also be affected by it, it makes sense, in the cost function to penalize with the logarithmic probability, instead of

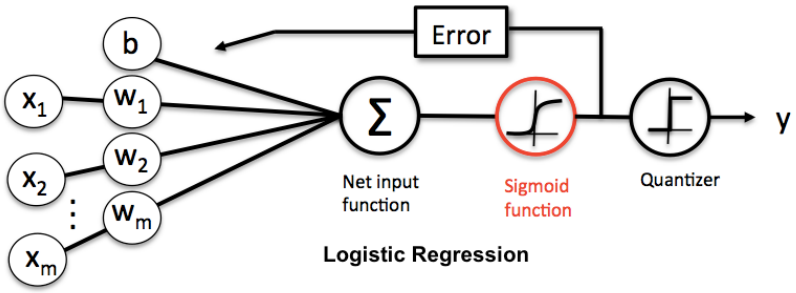

FIGURE 2.3: Logistic Regression Schematic. *Credit: mlxtend*

with the probability alone, since the logarithm function has a monotonically increasing behavior. Consequently, the function binary Cross-Entropy/Log Loss presented in equation 2.3 is a very good option to represent the logistic regression cost function [25].

Thus, the logistic regression problem can be written as:

$$\begin{aligned} \underset{\boldsymbol{\theta}\in\mathbf{R}^d}{\operatorname{argmin}}\, J(\boldsymbol{\theta}) &= -\frac{1}{m}\sum_{i=1}^{m} y_i \log\left[h_{\boldsymbol{\theta}}(\boldsymbol{x_i})\right] - (1-y_i)\log\left[1-h_{\boldsymbol{\theta}}(\boldsymbol{x_i})\right] \\ \text{s.t.} \quad h_{\boldsymbol{\theta}}(\boldsymbol{x}) &= \frac{1}{1+e^{-\boldsymbol{\theta}^T\boldsymbol{x}}}, \end{aligned} \tag{2.4}$$

where m is the size of the training set and h_θ is the output of the model. It is worth mentioning that in the equation 2.4, we assume that the vector $\boldsymbol{x_i}$ includes the constant term 1 to accommodate the bias θ_0.

The binary Cross-Entropy/Log Loss function in equation 2.3 has various advantages that are reflected in learning the model, it not only represents the average error between the prediction and the real class in a binary space, becoming a measure of uncertainty associated with the probability distribution, but it is also a convex function, which indicates that every local minimum is itself a global minimum. This allows a better rate of convergence, since there will be no problem of stagnation in non-optimal minimums.

The figure 2.3 illustrates the logistic regression model, which use gradient descent as the learning algorithm, the *sigmoid* function as the activation function, and the binary cross entropy function as the cost function.

2.4 Extension to multi-class classification problems

Logistic regression was originally proposed as a model for binary classification problems, however, there are currently a large number of problems that are not binary. To address multi-class classification we can use the *one vs rest* strategy.

The *one vs rest* strategy is a way to generalize binary classifiers to multi-class problems. This method arises from the need for multinomial classifiers and from

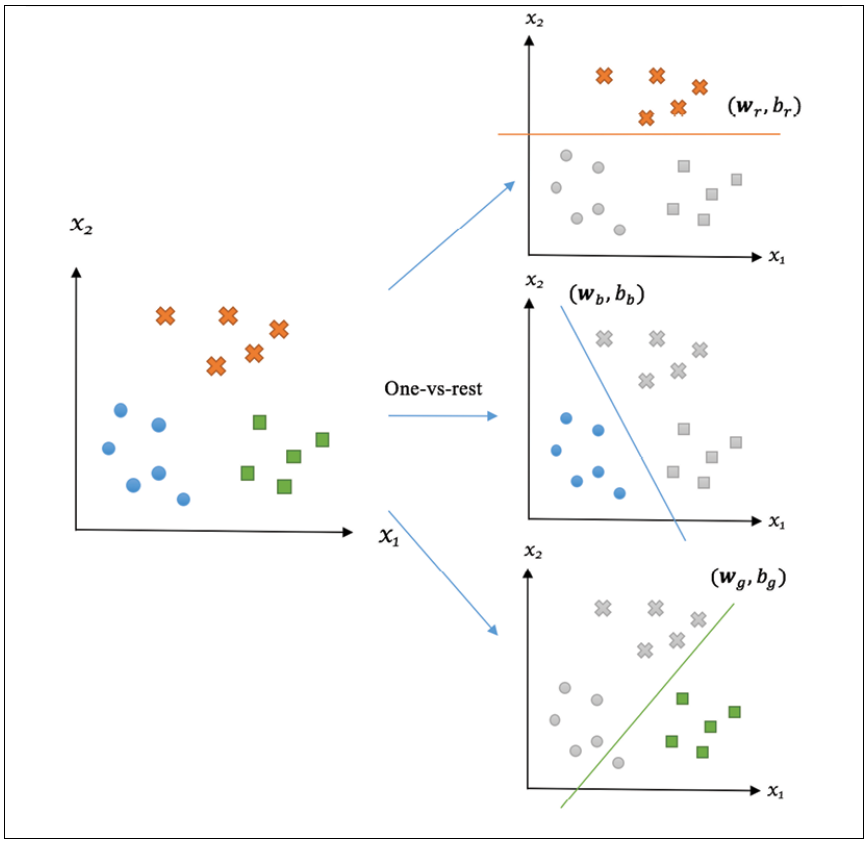

FIGURE 2.4: Illustration of the *one vs rest* strategy.

the fact that many classification algorithms were designed natively for binary classification problems.

One vs rest consists in transforming a classification problem with $K > 2$ classes into K binary classification problems, in which, for each of the classes $k \in \{1, ..., K\}$, a binary classifier is made taking $y = k$ as one class $y = 1$ and the rest $y = \{1, ..., K\} - \{k\}$ as $y = 0$, as shown in figure 2.4. The final result of the model would be the class that produces the maximum probability of success among all classifiers as expressed in equation 2.5.

$$\hat{y}_i = \underset{k \in \{1,...,K\}}{\operatorname{argmax}} \, h_\theta^k(\boldsymbol{x_i}) \tag{2.5}$$

Although the *one vs rest* strategy is widely used today and, is even one of the methods used to implement logistic regression in the sklearn library from Python. This method has several pitfalls. First of all, it requires the creation of as many models as the number of classes, each one with the same number of observations to evaluate, which can become very problematic for problems with a large number of classes [26].

Also, unbalanced distributions are likely to occur in each of the binary classifiers, because the class $y = 0$ has all the observations that belong to the set $\{1, ..., K\} - \{k\}$, this induces a bias in the model towards the majority class. However, there are several studies that defend the *one vs rest* strategy for multi-class classification

problems [27].

2.5 Multinomial logistic regression

We have already seen that the *one vs rest* strategy for the use of binary classifiers in multi-class classification problems, present various problems that considerably affect the performance of the model. This is why, in 1989, the statisticians Hosmer and Lemeshow proposed the Multinomial logistic regression model [28].

The Multinomial logistic regression model fits the predict probability distribution to a multinomial probability distribution, which actually is a generalization of the Binomial probabilistic distribution, in order to natively support multi-class classification problems [28].

In multinomial logistic regression, it's necessary to change the model output to a vector of probabilities in order to consider each class. The learning algorithm is changed to stochastic gradient descent, which is just a gradient descent applied to individual observations from a big sample, and the activation function is changed to the *softmax* function, which is defined as:

$$\text{softmax}_{z_i} = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \tag{2.6}$$

The *softmax* function turns all the inputs into positive values in order to distribute probabilities for each output node. We also need to change the loss function to the cross-entropy loss, which is a generalization of the binary cross-entropy. The cross entropy function maximizes the probability of the scoring vectors $p(y_i^b)\,\forall b \in \{1, ..., K\}$, in order to consider the error of predicting $\hat{y}_i = b$ when there exist more than two labels [29]. Equation 2.7 presents the cost function associated to multinomial logistic regression.

$$CE_\theta = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_i^{(k)} \log p(y_i^{(k)}) \tag{2.7}$$

where K is the total number of classes, $y_i^{(k)}$ is the kth component of the ith observation vector probabilities and $p(y_i^{(k)})$ is the predicted probability for the ith observation belonging to the kth class.

In this sense, the Multinomial logistic regression model presented in figure 2.5 not only establish a new concept for muti-class classifiers, it is also de basis for more complex models such as neural networks.

2.6 Data driven initialization for logistic regression parameters θ

Let's remember that the logistic regression goal is to find a set of parameters $\boldsymbol{\theta}$ that minimizes the cross-entropy cost function as indicated in equation 2.8, which actually constitutes the learning process of the model:

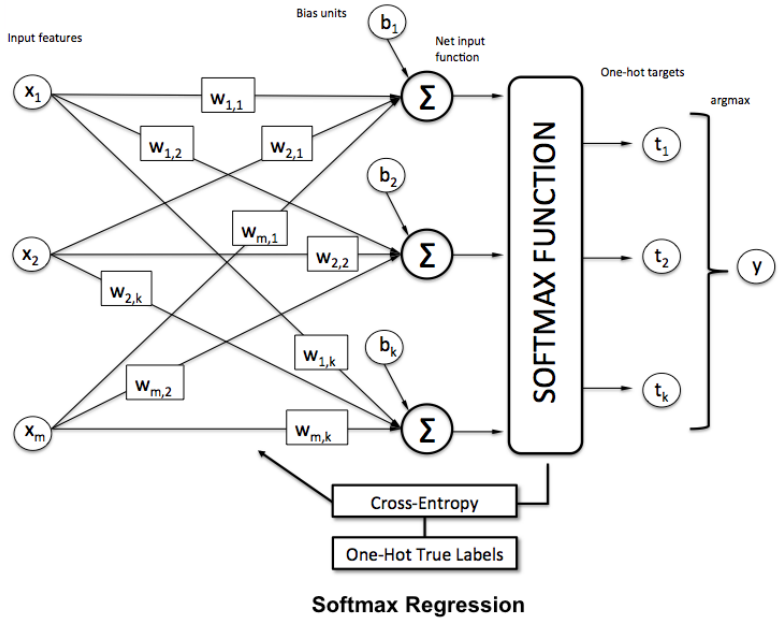

FIGURE 2.5: Multinomial Logistic Regression Schematic. *Credit: mlxtend*

$$\begin{aligned} \underset{\boldsymbol{\theta}}{\text{argmin}} \quad & \sum_{i=1}^{m} y_i \log\left(h_\theta(\mathbf{x}_i)\right) + (1-y_i)\log\left(1-h_\theta(\mathbf{x}_i)\right) \\ \text{s.t.} \quad & h_\theta(\mathbf{x}_i) = \frac{1}{1+e^{-(\theta_0+\boldsymbol{\theta}^{\mathrm{T}}\mathbf{x}_i)}}, \end{aligned} \tag{2.8}$$

where $\mathbf{x}_i \in \mathbf{R}^p \; \forall i : \{1,\ldots,m\}$, whit p the dimension of the input data and m the number of observations. As we mentioned before, for a binary classification problem, the output of the logistic regression model can be interpreted as the probability that the input data $\mathbf{x}_i$ belongs to the class $y = 1$, $P(\mathbf{x}|y=1)$.

The goal of the logistic regression can be seen as maximize $h_\theta(\mathbf{x}_i)$ where $\mathbf{x}_i \in C_1$ and minimize $h_{\boldsymbol{\theta}}(\mathbf{x}_i)$ if $\boldsymbol{x_i} \in C_2$ with $C_1 \rightarrow y = 1$ and $C_1 \rightarrow y = 0$. This concept is presented in figure 2.6 where some observations are fitted to a sigmoid function. From equation 2.8 it can be seen that by maximizing the dot product $\boldsymbol{\theta}^{\mathrm{T}}\mathbf{x}_k^{(1)}$ the $P\left(\mathbf{x}_k^{(1)}|y=1\right)$ is also maximized, where $\mathbf{x}_k^{(1)} \in C_1 \; \forall k : \{1,\ldots,m_1\}$, with C_1 the set of all observations that belong to the class $y = 1$, and m_1 the number of elements in C_1.

Therefore, one initialization value for the parameters in the logistic regression problem can be derived by finding the set of parameters $\boldsymbol{\theta}$ that maximizes the mean value of the dot products $\boldsymbol{\theta}^{\mathrm{T}}\mathbf{x}_k^{(1)} \; \forall k : \{1,\ldots,m_1\}$.

Let's consider the normalized dataset:

$$\hat{\mathbf{x}}_\mathbf{i} = \Sigma_\mathbf{X}^{-1}(\mathbf{x_i} - \mu_\mathbf{X}), \tag{2.9}$$

where $\hat{\mathbf{x}}_\mathbf{i}$ is the normalized observation i, and $\mu_\mathbf{X}$ and $\Sigma_\mathbf{X}$ are the mean value and the co-variance matrix for the complete dataset. If we want to find the parameter

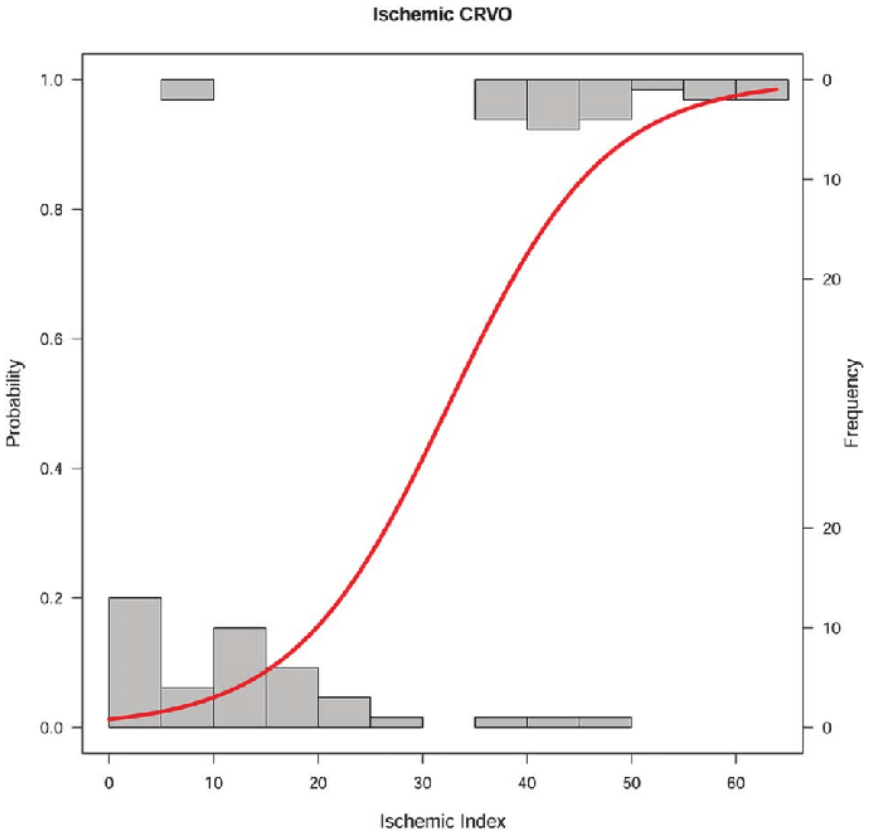

FIGURE 2.6: Illustration of the sigmoid function goal in the logistic regression model. *Credit: researchgate*

vector $\boldsymbol{\theta}$ that maximised the mean of the dot product with respect to $\boldsymbol{x_k^{(1)}}$, this leads to the following problem:

$$\begin{aligned} \underset{\boldsymbol{\theta}}{\text{argmax}} \quad & \frac{1}{m_1} \sum_{i=1}^{m_1} \boldsymbol{\theta}^{\mathrm{T}} \hat{\mathbf{x}}_i^{(1)} \\ \text{s.t.} \quad & \boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{\theta} = 1, \end{aligned} \tag{2.10}$$

where the constrain $\boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{\theta} = 1$ is used to mitigate the effect of the amplitude of the parameters on the dot product. By using the Lagrange multipliers, the problem reduces to:

$$\underset{\boldsymbol{\theta}}{\text{argmax}} \quad \mathcal{L}(\boldsymbol{\theta}, \lambda) = \frac{1}{m_1} \sum_{i=1}^{m_1} \boldsymbol{\theta}^{\mathrm{T}} \hat{\mathbf{x}}_i^{(1)} - \lambda(\boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{\theta} - 1), \tag{2.11}$$

where λ is the Lagrange multiplier. Applying the conditions for optimality we obtain:

$$\begin{aligned} \frac{\partial \mathcal{L}(\boldsymbol{\theta}, \lambda)}{\partial \boldsymbol{\theta}} &= \frac{1}{m_1} \sum_{i=1}^{m_1} \hat{\mathbf{x}}_i^{(1)} - 2\lambda \boldsymbol{\theta} = 0, \\ \frac{\partial \mathcal{L}(\boldsymbol{\theta}, \lambda)}{\partial \boldsymbol{\lambda}} &= \boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{\theta} - 1 = 0. \end{aligned} \tag{2.12}$$

The solution to this problem is given by:

$$\begin{aligned} \boldsymbol{\theta} &= \frac{1}{2\lambda}\mu_{\hat{\mathbf{x}}^{(1)}}, \\ \lambda &= \frac{1}{2}\mu_{\hat{\mathbf{x}}^{(1)}}^{\mathrm{T}}\mu_{\hat{\mathbf{x}}^{(1)}}; \end{aligned} \tag{2.13}$$

where $\mu_{\hat{\mathbf{x}}^{(1)}}$ is the mean value of the data points that belong to the class $y = 1$. Since we are not interested in the scaling factor but only in the direction of the parameters that maximizes this dot product, we can conclude that the parameters $\boldsymbol{\theta}$ that maximizes the mean value of their dot product with the data that belong to the class $y = 1$ are given by:

$$\boldsymbol{\theta} = \mu_{\hat{\mathbf{x}}^{(1)}} = \Sigma_{\mathbf{X}}^{-1}\left(\frac{1}{m_1}\sum_{i=1}^{m_1}\left(\mathbf{x}_i^{(1)} - \mu_{\mathbf{X}}\right)\right). \tag{2.14}$$

This result is evident, since $\mu_{\hat{\mathbf{x}}^{(1)}}$ is a representative point of the data that belong to the class $y = 1$, thus, by using $\boldsymbol{\theta} = \mu_{\hat{\mathbf{x}}^{(1)}}$ as initial parameters, $P\left(\mathbf{x}_k^{(1)}|y = 1\right)$ is maximized, which actually is the goal of logistic regression.

Since the scalar product $\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{x}$ is a measurement of similarity between the vectors $\boldsymbol{\theta}$ and $\boldsymbol{x}$, it make sense to use as initialization parameters a vector $\boldsymbol{\theta}$ that is a prototype vector for the distribution of the input data $\mathbb{X}$. If the initial parameters are not similar to most observations of the class $y = 1$, then the $P\left(\mathbf{x}_k^{(1)}|y = 1\right)$ for initial parameters is not adequate which will affect the performance of the learning algorithm.

If we consider a classification problem where $\{x_1^{(1)}, x_2^{(1)}, \ldots, x_{m_1}^{(1)}\}$ is the data that belongs to class $y = 1$ and exists an atypical observation $x_n^{(1)}$, with large values far away from the distribution of the data, then the sample mean $\mu_{\boldsymbol{x}^{(1)}}$ is largely affected. To mitigate this problem, we consider the normalized dataset, however, the distance between $\{x_1^{(1)}, x_2^{(1)}, \ldots, x_{m_1}^{(1)}\} - \{x_n^{(1)}\}$ and $x_n^{(1)}$ remains large even after normalizing the data. Therefore, in order to make the initialization of the parameters more robust to problems which have data with outliers, the median of the normalized data of the class $y = 1$ can be considered as initial parameters of the logistic regression instead of the mean:

$$\boldsymbol{\theta} = Me_1 = \text{median}(\hat{\boldsymbol{x}}^{(\mathbf{1})}) \tag{2.15}$$

Hence, the median Me_1 of the class $y = 1$ could be a more robust approach to initialize the parameters of the logistic regression.

In summary, we propose to use the mean value of the data points that belong to class $y = 1$ as the initial values for $\boldsymbol{\theta}$. In the case that the data contains large outliers, we propose to use the median value instead. By using the mean, we guarantee that gradient descent will start in a proper initial region of the cost function, where the dot product between the data $\boldsymbol{x}_k^{(\mathbf{1})}$ and $\boldsymbol{\theta}$ is maximized.

2.7 Data driven initialization for the bias θ_0 in logistic regression

In the previous section we discuss a strategy to find the parameters $\boldsymbol{\theta}$ that best represents the separating hyperplane. However, this hyperplane should be offset by the bias term, θ_0. Here we discuss three possible methodologies to find this parameter. The bias is a crucial parameter that determines the proper position for the classification boundaries of the model [30].

Let's consider the projection of the data points along the direction of $\boldsymbol{\theta}$, given by $\mathbf{p}_1 = \boldsymbol{\theta}^{\mathrm{T}}\mathbf{x}^{(1)}$ and $\mathbf{p}_2 = \boldsymbol{\theta}^{\mathrm{T}}\mathbf{x}^{(2)}$, where $\mathbf{x}^{(1)}$ are the observations that belong to class $y = 1$, and $\mathbf{x}^{(1)}$ are the observations that belong to class $y = 0$. Here we can consider two main cases, non overlapping classes, and overlapping classes.

2.7.1 Computation of the bias in non-overlapping classes

If the observations for the different classes are not overlapping, then we have that $\mathbf{p}_1 > \mathbf{p}_2$ for all the observations. In this case, we can define a value of the bias that maximizes the margin between the classes. This bias is given by:

$$\theta_0 = \frac{1}{2}\left(\min\left(p_1\right) + \max\left(p_2\right)\right).$$

This value of θ_0 sits just in the middle of the extreme values of the projections of the data over the parameters vector $\boldsymbol{\theta}$.

2.7.2 Computation of the bias in overlapping classes

In case that the classes are overlapping, then we have that some values of $\mathbf{p}_2 > \min\left(\mathbf{p}_1\right)$, and $\mathbf{p}_1 < \max\left(\mathbf{p}_2\right)$. In this case we should select θ_0 such that missclassifications are minimized. We proposed two strategist for this. The first strategy is based on a weighted mean of the overlapped projections; the second strategy attempt to obtained quadratic models for some statistics on each one of the projected values, such that the intercept of these models will provide the value for the bias.

Bias estimation in overlapped classes: weighted mean

Let´s considered $m^{(1)}_{\text{overlap}}$, and $m^{(2)}_{\text{overlap}}$, the number of observations where the projections $\mathbf{p}_1 < \max\left(\mathbf{p}_2\right)$ and $\mathbf{p}_2 > \min\left(\mathbf{p}_1\right)$, respectively. In this case, we could take the man value for the overlapped components to estimate the bias; however, this approach is susceptible to be biased in presence of unbalanced dataset. In order to mitigate this problem we can estimate the bias term as a weighed mean for both distributions, such that:

$$\theta_0 = \frac{m^{(2)}_{\text{overlap}} \sum\limits_{i \in (\mathbf{p}_1)_i < \max(\mathbf{p}_2)} (\mathbf{p}_1)_i + m^{(1)}_{\text{overlap}} \sum\limits_{i \in (\mathbf{p}_2)_i > \min(\mathbf{p}_1)} (\mathbf{p}_2)_i}{m^{(1)}_{\text{overlap}} + m^{(2)}_{\text{overlap}}},$$

where the elements $(\mathbf{p}_1)_i$ y $(\mathbf{p}_2)_i$ are the projections that fulfill the condition described in the sums.

Bias estimation in overlapped classes: quadratic approach

Another way to find the bias is to model, using a quadratic approximation, the cumulative probability distribution for the elements of the projections, for each class, that are overlapped. For instance, let's consider $\hat{\mathbf{p}}_1 : \{\mathbf{p}_1 | \mathbf{p}_1 < \max(\mathbf{p}_2)\}$ and $\hat{\mathbf{p}}_2 : \{\mathbf{p}_2 | \mathbf{p}_2 > \min(\mathbf{p}_1)\}$, for $\hat{\mathbf{p}}_1$ we know that the $\min(\hat{\mathbf{p}}_1)$ is equivalent to the 0-th percentile of the distribution, while $\max(\hat{\mathbf{p}}_2)$ is equivalent to the 100-th percentile. Whilst, for $\hat{\mathbf{p}}_2$ we know that the $\max(\hat{\mathbf{p}}_2)$ is equivalent to the 100-th percentile of the distribution, while $\min(\hat{\mathbf{p}}_1)$ is equivalent to the 0-th percentile. Therefore, if we model using a quadratic approximation the cumulative distribution for $\hat{\mathbf{p}}_1$ and $\hat{\mathbf{p}}_2$ they will cross. The value when they cross is selected as the bias. This approach is also robust against unbalanced data, since it takes into account the distribution of the overlapped samples. In order to fit these values to a quadratic we need another point of the distribution, this point can be used using the median. With these three points we can fit the quadratic, we obtain the following set of equations:

$$V_{\mathbf{p}_1} = a_1 c^2 + a_2 c + a_3; \tag{2.16}$$
$$V_{\mathbf{p}_2} = b_1 c^2 + b_2 c + b_3, \tag{2.17}$$
$$\tag{2.18}$$

where $V_{\mathbf{p}_1}$ is the value of the variable $\mathbf{p}_1$ for a given percentile c, likewise $V_{\mathbf{p}_1}$. Given the normalized values for the percentiles $0, 0.5$ and 1. The following set of equations is established:

$$\begin{bmatrix} \min(\mathbf{p}_1) \\ \text{med}(\hat{\mathbf{p}}_1) \\ \max(\mathbf{p}_2) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0.25 & 0.5 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \quad \text{and} \quad \begin{bmatrix} \max(\mathbf{p}_2) \\ \text{med}(\hat{\mathbf{p}}_2) \\ \min(\mathbf{p}_1) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0.25 & 0.5 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Solving these equations we get:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 2 & -4 & 2 \\ -3 & 4 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \min(\mathbf{p}_1) \\ \text{med}(\hat{\mathbf{p}}_1) \\ \max(\mathbf{p}_2) \end{bmatrix}, \quad \text{and} \quad \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2 & -4 & 2 \\ -3 & 4 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \max(\mathbf{p}_2) \\ \text{med}(\hat{\mathbf{p}}_2) \\ \min(\mathbf{p}_1) \end{bmatrix}.$$

If we want to find the intercept we can calculate the value of c, for which both quadratic functions meet, and evaluate this value on any of the models for the cumulative distributions. This value for c is given by:

$$c_{\text{intercept}} = \frac{-(a_2 - b_2) \pm \sqrt{(a_2 - b_2) - 4(a_1 - b_1)(a_3 - b_3)}}{2(a_1 - b_1)}.$$

We choose the value of $c_{\text{intercept}}$ such that $0 \leq c_{\text{intercept}} \leq 1$. We obtain the bias replacing $c_{\text{intercept}}$ in any of the equations in (2.16).

Now we have the complete algorithm for the initialization of logistic and multinomial logistic regression models. In the next chapter we will extend this model to Feed-Forward Neural Networks.

Chapter 3

Data Driven Initialization for neural networks

In this chapter we introduce an extension of the initialization algorithm for the classical feed-forward neural networks. The chapter starts with a general introduction to the neural network model, its training algorithm, as well as their challenges. It finalizes with a description of our proposed methodology for initialization.

3.1 The Biological Neuron

Neurons are the fundamental units of the brain and nervous system. This cells are responsible of receiving sensory input, process the information and then transform the input electrical signals to produce an adequate response [31].

The brain is made up of a large number of neurons that work together to form what is known as a neural network. The neural network, through the constant process of receiving information from the environment, processing it and transmitting electrical signals, is what allows living beings to generate reactions such as moving muscles, perceiving aromas, feeling pain, as well as generate thoughts.

In figure 3.1 we can see the biological components of a neuron. The neurons are divided in 3 main components, the *dendrites*, the *soma* and the *axon*. The dendrites are the ones in charge of receiving the electrical stimulus transmitted by other neurons. The soma is not only the body of the neuron, which contains the nucleus, but

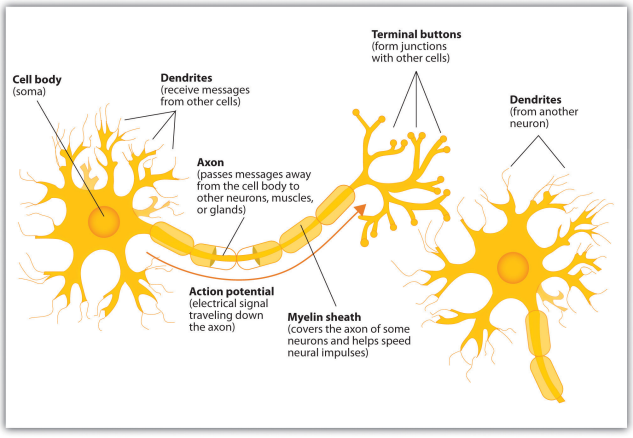

FIGURE 3.1: The components of a biological neuron. *Credit: wikimedia*

is also in charge of processing the electrical impulse received by the dendrites and communicating it to the axon. The axon is responsible for transporting the received impulse to the synaptic endings. These terminations, through the synapse, are responsible for transmitting the electrical impulse generated by the neuron to the other neurons of the neural network, by means of neurotransmitters. Thus, there exists a communication link between the synaptic terminations and the dendrites of other neurons [32].

3.2 A computational model for the Brain

The brain is one of the most complex and studied learning systems in various areas around the world. It's behavior is sophisticatedly structured, which makes its understanding extremely complicated. However, there is a great amount of studies that investigate the modeling of the brain.

The "One Learning Algorithm Hypothesis" of Andrew Ng (Google Brain), according to which the brains of higher-level animals and humans process and perceive sensory data such as vision, sound, and haptics with the same architecture abstract algorithmic [33].

This hypothesis was taken up by the electrical and computer engineering specialist Christos N. Mavridis and the HyNet director Dr.John S. Baras in their article "Towards the One Learning Algorithm Hypothesis: A System-theoretic Approach" [33]. Mavridis and Baras investigated the structure of a data-agnostic learning architecture that resembles the "one learning algorithm", using different principles from mathematics in order to establish a theoretical approach to this theory. Many researchers have tried to develop computational models for the brain behaviour. Here we will discuss some of the most important ones.

3.3 The McCulloch-Pitts Neuron

The McCulloch-Pitt Neuron is the first computational model of a neuron, which was proposed by the neuroscientist MuCulloch an the logician Walter Pitts in 1943. This first model is based on the fact that the nervous system contains many electrical cycles, in which there are points that regenerate the excitation of neurons and others that discharge the neurons excitation. This happens asynchronously for each neuron of the entire network but a dependency is created between the excitation of the neuron and its excitation in past times. [34].

The McCulloch-Pitts neuron model (MCP) makes the following physical assumptions in order to establish its computational representation:

- The activity of the neuron is an "all-or-nothing" process, i.e. both the input signal and the output signal are binary $\{0, 1\}$.
- The neuron is excited at a certain level at any time, and this level is independent of previous activity and the position of the neuron. This means that the model will have an associated fixed threshold μ that will determine the degree of excitability of the neuron.

- The only significant delay within the nervous system is synaptic delay.
- The structure of the network does not change over time
- Inhibitory inputs have an absolute veto over the output, that is, if an inhibitory signal reaches the neuron, the output of the neuron will always be 0 regardless of the rest of the incoming signals.

Therefore, MCP assumes that the output of the neuron is sum of its inputs with a weights ω, which are equal for each input predictor:

$$g(x_1, x_2, \ldots, x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} \omega x_i,$$

where $\omega \in \mathbb{R}_+$. Likewise, since the second assumption tells that when the number of excitatory signals equals or exceeds this threshold, the neuron will be excited and will generate a signal with a value of 1 that will be transmitted to the next or following neurons in the network, the output of this neuron passes through a thresholding function, such that:

$$y = f_\mu(g(\mathbf{x})) = \begin{cases} 1 & \text{if } g(\mathbf{x}) \geq \mu \\ 0 & \text{if } g(\mathbf{x}) < \mu \end{cases}$$

where μ is a predefined threshold and $0 < \mu < n\omega$, with n the number of inputs signals. This condition is established because since the input signals are binary and ω is fixed, then if $\mu > n\omega$, the neuron will never be excited and will generate a signal with a value of 0 permanently. A pseudocode for the McCulloch-Pitt algorithm is provided in Alg 2.

Input: The training set $\{\boldsymbol{x_i}, y_i\}_{i=1}^n$ with $\boldsymbol{x_i} \in \mathbb{R}$, $y_i \in \{0, 1\}$, $\forall i \in \{1, 2, \ldots, n\}$, the parameter $\omega \in \mathbb{R}_+$ and the threshold $\mu > 0$.
Process:
for *all* $\boldsymbol{x_j}, y_j \in \{\boldsymbol{x_i}, y_i\}_{i=1}^n$ **do**
    $\hat{y}_j = f_\mu(\boldsymbol{\omega^T x_j})$
end
Output: The predictions $\{\hat{y}_i\}_{i=1}^n$ for all the observations.

Algorithm 2: McCulloch-Pitt algorithm.

Figure 3.2 shows a graphical representation of the McCulloch-Pitt model, in which it can be seen that this model consist of two main parts. The first part computes the value $g(\boldsymbol{x})$ using the hyperparameter ω. The second part, compress $g(\boldsymbol{x})$ with the threashold μ, in order to obtain the output of the neuron $y \in \{0, 1\}$.

This model has many limitations due to the initial assumptions. Therefore, the number of real problems that can be addressed using the McCulloch-Pitts model is reduced. In addition, since this model assumes that all the weights of the input predictors are equal, it does not allow to have inclined decision surfaces, i.e. it is not possible to consider that certain predictors are more important than others for a given output prediction [35].

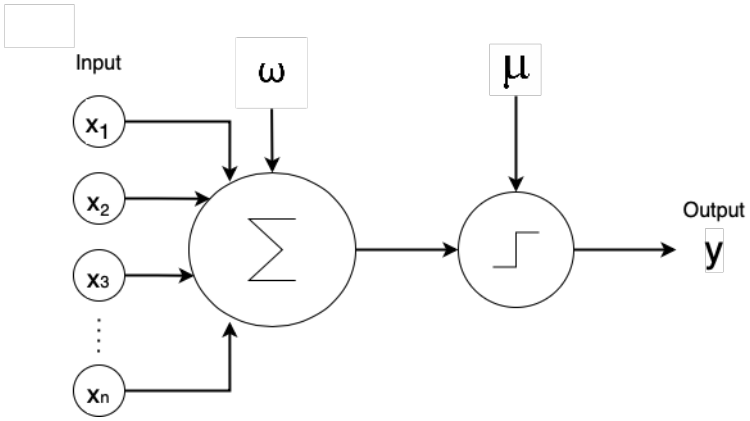

FIGURE 3.2: McCulloch-Pitt model diagram representation.

3.4 The Perceptron

The perceptron is an improvement of the McCulloch-Pitts model and is one of the most popular artificial neural representations. This model is attributed to the psychologist, theoretical computer scientist and neuroscientist Frank Rosenblatt in 1957, and was developed at the Cornell Aeronautical Laboratory. The perceptron was initially thought to be an image recognition machine, and although initially it was theoretically developed, consequently by the use of electronic components it was implemented in hardware. This implementation was known as the "Mark 1 perceptron" [36].

This model in contrast to the MPC assumes that the weights $\boldsymbol{\omega} \in \mathbb{R}$ and the threshold $\mu \in \mathbb{R}_+$ can be different for each input signal. This not only allows certain predictors to have more importance than others in the output $\hat{y}$ of the neuron, but also widens the parameter space of the model, since the weights can be positive or negative. Likewise, the existence of inhibitory inputs is eliminated, which allows the model to operate regardless of the incoming signals.

One of the main characteristics of the perceptron, and the reason why the perceptron was a very significant advance for machine learning, is that it has a learning rule which based on the training data computes an error determined as the difference between the neuron's output, $\hat{y}$, and the expected output, y, and uses it to update the model parameters $\boldsymbol{\omega}$. The perceptron is an artificial neuron that uses supervised learning. Similarly, instead of the output of the neuron $\hat{y}$ belonging to the set $\{0, 1\}$, it is assumed that $\hat{y} \in \{-1, 1\}$. This relaxes the model, allowing a larger range for the update of the parameters $\boldsymbol{\omega}$.

Figure 3.3 shows a graphical representation of the perceptron, where the output of the model is given by $\hat{y} = \varphi(z)$ with $z = \boldsymbol{\omega}^{\mathrm{T}}\mathbf{x}$ and φ defined as the following threshold function:

$$\varphi(z) = \begin{cases} 1 & \text{if } z \geq \mu \\ -1 & \text{if } z < \mu \end{cases},$$

where μ is a predefined threshold which usually is defined as 0.5. The perceptron as well as the MPC uses the step transformation as activation function.

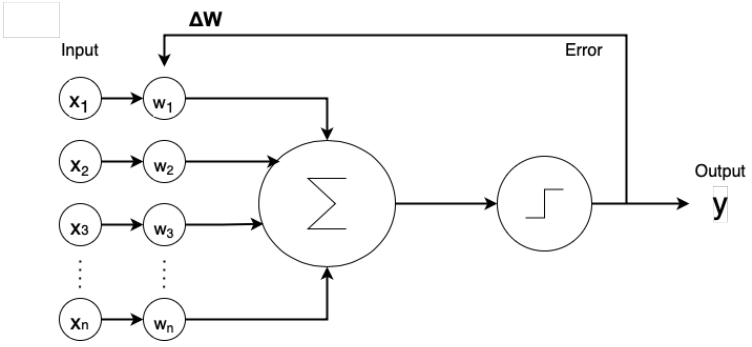

FIGURE 3.3: Perceptron model diagram representation.

Perceptron learning begins by initializing the $\boldsymbol{\omega}$ parameters to $\mathbf{0}$ or to small random numbers. It then updates its $\boldsymbol{\omega}$ parameters iteratively based on the error defined by the difference between the predicted and the expected output value, $e = \mathbf{y} - \hat{\mathbf{y}}$. Intuitively what the perceptron rule aims to do, is to make the parameters $\boldsymbol{\omega}$ as close to the values of the elements with label $y = +1$, and far away from the values of elements with class $y = -1$. This can be represented by the following update rule:

$$\boldsymbol{\omega} := \boldsymbol{\omega} + \Delta\boldsymbol{\omega},$$
$$\Delta\boldsymbol{\omega} = \eta \mathbf{X}^{\mathrm{T}}(\mathbf{y} - \hat{\mathbf{y}}),$$

where $\eta \in \mathbb{R}_+$ is the learning rate, $\mathbf{X}$ is a matrix of dimensions $N \times p$, with N being the number of observations, and p the number of regressors including the vector of ones associated to the bias. Another difference between the perceptron and the MPC is that the perceptron includes an extra parameter ω_0, also known as the bias, which helps to relax the solution, by moving the decision surface along the feature space. A pseudocode for the perceptron learning algorithm is provided in Alg 3.

Input: The training set $\{\boldsymbol{x}_i, y_i\}_{i=1}^n$ with $\boldsymbol{x}_i \in \mathbb{R}$, $y_i \in \{-1, 1\}$,
$\forall i \in \{1, 2, \ldots, n\}$, η, the learning rate, which is usually set to 1.
Process:
`# All vector` $\boldsymbol{x}_j$ `includes an extra 1 in a fixed position.`
Initialize $\boldsymbol{\omega}$, with zeros or small random numbers.
Repeat until convergence
  for *all* $\boldsymbol{x}_j, y_j \in \{\boldsymbol{x}_i, y_i\}_{i=1}^n$ **do**
    $\hat{y}_j = \varphi(\boldsymbol{\omega}^T \boldsymbol{x}_j)$
    error = $y_j - \hat{y}_j$
    `# Update` $\boldsymbol{\omega}$ `if the prediction was wrong.`
    if $error \neq 0$ **then**
      `# Update` $\boldsymbol{\omega}$
      $\boldsymbol{\omega} = \boldsymbol{\omega} + \eta \cdot error \cdot \boldsymbol{x}_j$
    end
  end
Output: The model parameters $\boldsymbol{\omega}$ including ω_0 as the bias term.

Algorithm 3: Perceptron learning algorithm.

Although the perceptron has several advantages over the McCulloch-Pitt model and marked an important milestone for machine learning, the truth is that this model has many limitations. One of it's most significant limitations is the fact that it can only handle classification tasks for linearly separable classes. The decision boundary

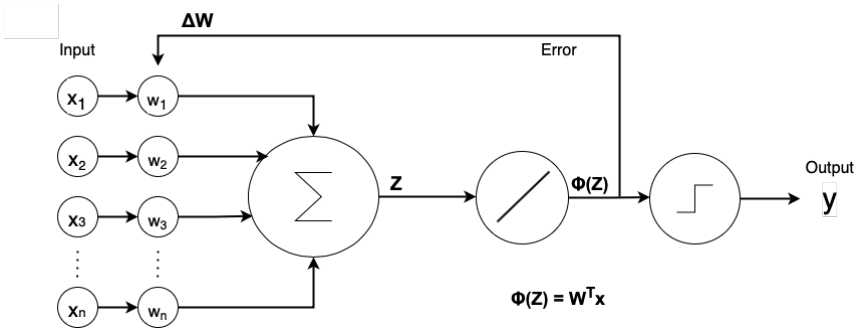

FIGURE 3.4: Adaline model diagram representation.

for the perceptron is given by:

$$\boldsymbol{\omega}^T \boldsymbol{x} = w_0 + x_1 w_1 + \ldots + x_n w_n = 0$$

This means that the decision boundary is given by a hyperplane, oriented by the vector $\boldsymbol{\omega}$. Likewise, another limitation is that the model can only produce a binary output, and in some cases it is desired to have the value of the probability for an observation to belong to a given class [35].

3.5 The Adaline (ADAptive LInear NEuron)

The adaline an Adaptive Linear Neuron seeks to generate a linear regression between the input data and the output. This model was introduced in 1959 by the electrical engineers Bernard Widrow and Ted Hoff at Standford university. Similarly to the perceptron, the adaline was initially implemented and tested in hardware.

Figure 3.4 shows a graphical representation of the adaline model. As can be seen, although the diagram of this model looks very similar to that of the perceptron and both models use a threshold function, the reality is that the adaline has a much more robust learning algorithm than the other models.

The perceptron learns and updates its parameters $\boldsymbol{\omega}$ by comparing the real classes $y \in \{-1, 1\}$ with the model predicted classes $\hat{y} \in \{-1, 1\}$ via the error $e = y - \hat{y}$. In contrast to the perceptron, Adaline make use of directional derivatives from a cost function based on the error, in order to learn faster.

The adaline calculates the error before entering the threshold function, that is, it calculates the difference between the expected class value $y \in \{-1, 1\}$ and the linear function output value $\phi(z) \in \mathbb{R}$. This allows prediction errors to be continuous rather than discrete, which makes possible the use of gradient descent. In addition, it allows the model to learn even when no error has been made.

Adaline, like many other machine learning models, bases its learning process on the minimization of a cost function. In this case, the squared error of prediction error is used as the model cost function:

$$\mathbf{J}(\boldsymbol{\omega}) = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - \phi(z^{(i)}))^2,$$

where $z = \boldsymbol{\omega}^{\mathrm{T}}\mathbf{x}$, and $\phi(z) = z$.

The adaline learning problem can be viewed as an optimization problem which, as mentioned in previous chapters, can be solved using gradient descent. Thus, the gradient descent formula to update the weights is given by:

$$\boldsymbol{\omega} := \boldsymbol{\omega} - \Delta\boldsymbol{\omega},$$
$$\Delta\boldsymbol{\omega} = \eta\nabla\mathbf{J}(\boldsymbol{\omega}),$$

Differentiating the cost function in terms of the model parameters ω_j, we obtain:

$$\frac{\partial \mathbf{J}}{\partial \omega_j} = -\sum_{i=1}^{m}\left(y^{(i)} - \phi(z^{(i)})\right)x_j^{(i)}$$

Since $\phi(z) = \phi(\boldsymbol{\omega}^{\mathrm{T}}\mathbf{x}) = \boldsymbol{\omega}^{\mathrm{T}}\mathbf{x}$, then:

$$\Delta\omega_j = -\eta\sum_{i=1}^{m}(y^{(i)} - \boldsymbol{\omega}^{\mathrm{T}}\mathbf{x})x_j^{(i)}$$

which can be seen in vector form as:

$$\Delta\boldsymbol{\omega} = \eta\mathbf{X}^{\mathrm{T}}(\mathbf{X}\boldsymbol{\omega} - \mathbf{y})$$

where η is the learning rate and X is a matrix of dimensions $N \times p$, with N being the number of observations, and p the number of regressors including the vector of ones for the bias. A pseudocode for the adaline learning algorithm is provided in Alg 4.

Input: The training set $\{\boldsymbol{x_i}, y_i\}_{i=1}^{n}$ with $\boldsymbol{x_i} \in \mathbb{R}$, $y_i \in \{-1, 1\}$, $\forall i \in \{1, 2, \ldots, n\}$ and the learning rate $\eta > 0$.
Process:
`# All vector` $\boldsymbol{x_j}$ `includes the 1 in a fixed position as in previous algorithm.`
Initialize $\boldsymbol{\omega}$, with zeros or small random numbers.
Construct the matrix $\mathbf{X}$ and vector $\boldsymbol{y}$ from the input data $\{\boldsymbol{x_i}, y_i\}_{i=1}^{n}$.
Repeat until convergence
    $\Delta\boldsymbol{\omega} = \eta\mathbf{X}^{\mathbf{T}}(\mathbf{X}\boldsymbol{\omega} - \boldsymbol{y})$
    `# Update` $\boldsymbol{\omega}$
    $\boldsymbol{\omega} = \boldsymbol{\omega} - \Delta\boldsymbol{\omega}$
Output: The model parameters $\boldsymbol{\omega}$ which includes the bias.

Algorithm 4: Adaline learning algorithm.

3.6 The Logistic Neuron

There is some similarity between the architecture of a logistic regression for binary classification problems 2.3 and the architecture of an Adaline model 4. Actually the only difference between these two models is the activation function ϕ and the cost function.

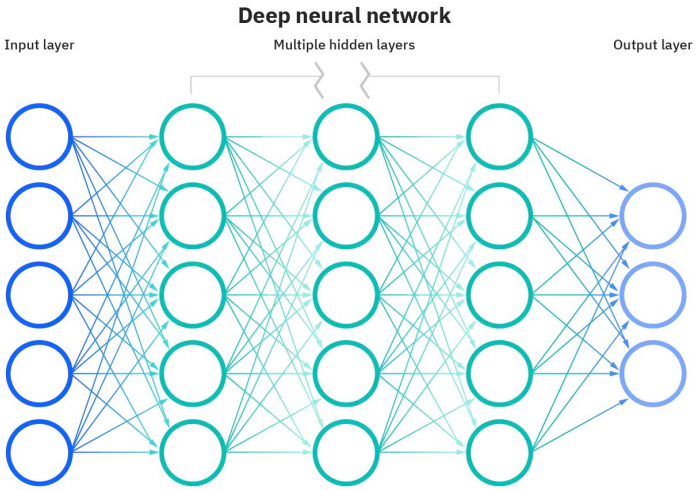

FIGURE 3.5: Neural Network diagram representation. *Credit: ibm*

While logistic regression uses the sigmoid function $\phi_{rl}(\boldsymbol{\omega}^T\boldsymbol{x}) = \frac{1}{1+e^{-\omega^T x}}$ as activation function,the Adeline model uses the identity function $\phi_a(\boldsymbol{\omega}^T\boldsymbol{x}) = \boldsymbol{\omega}^T\boldsymbol{x}$. However, this can be generalized to any other activation function.

The cost function in logistic regression is the binary cross entropy, while Adaline uses the mean square error. Both functions can be used in a binary classification problem although binary cross entropy is naturally used. This is due to the fact that classification problems should not be solved using a regression model.

When using a logistic function, and a cost function such as cross-entropy, the model is called a logistic neuron. Logistic neurons are the fundamental blocks to build the classical architecture of the artificial neural networks.

3.7 Classic Neural Networks: Feed-forward architecture

In the previous sections we introduced the concept of artificial neuron and explained in detail three of the most influential models in the history of machine learning. Likewise, we are encouraged to view logistic regression for binary classification problems as a special case of the generalized Adaline model, the logistic neuron. In this section we are going to build the concept of an artificial neural network from the basic units *the artificial neurons*, and due to the purpose of the thesis we will focus specifically from the *logistic neurons* which are neural representations of the Logistic regression.

Suppose that, as with biological neurons, we want to group artificial neurons so that they work together. To achieve this, we define a layered architecture, where each layer is a set of neurons that are connected to all the elements of the previous layer, but are not connected to elements from the same layer. This architecture is called a fully connected network or a feed-forward neural network. Figure 3.5 shows the structure of a classical neural network, in which each of the nodes represents a logistic neuron. In this figure, we can see that the layers of neurons are colored

with three different colors where each color represents a type of layer. There exist 3 different types of layers:

- **Input Layer:** the layer that connects the input elements to the network. This layer does not have an activation function, it only represents the connection interface with the input data.
- **Output Layer:** the layer that provides the output of the network and in the classification problems it usually has the same amount of neurons as the number of classes.
- **Hidden layer:** the layers between the input layer and the output layer. These layers seeks to expand and/or reduce the dimensionality of the data space.

Classical Neural networks are also called feed-forward neural networks because the input data is processed in a forward direction in order to produce the output.

3.7.1 Forward propagation process

In the forward propagation process, the input data is fed in the forward direction (left to right) through the network. Each hidden layer receive the input signal, processes it by passing through the activation function and forward it to the successive layers. The processing performed by each of the hidden layers can be seen as an embedding which depends of the parameters of the layer to a d-dimensional space, where d is the number of neurons in the layer.

The forward propagation process in a more mathematical approach can be seen as following: Consider a neural network with $I \in \mathbb{N}_+$ hidden layers where the jth layer has $n_j \in \mathbb{N}_+$ neurons, for all $j\ in\{1,\dots,I\}$. Suppose that the matrix $\mathbf{X}$ of size $N \times p$ is the input data of the neural network, where N is the number of observations and p the number of predictors per observation. Let's define $\mathbf{W^{(j)}}$ the parameter matrix of the j-th hidden layer. Note that since the output of the (j-1)th layer is the input of the j-th layer, then

$$|\mathbf{W^{(j)}}| = \begin{cases} n_{j-1} \times n_j & \text{if } j \in \{2,3,\dots,I\}, \\ p \times n_1 & \text{if } j = 1 \end{cases}$$

where $|\cdot|$ represents the matrix size. The jth layer activation function can be defined by $f^{(j)}$ and the input to the jth layer is defined by $\mathbf{a^{(j)}}$. Therefore the input to the activation functions in the j-layer are given by:

$$\mathbf{z^{(j)}} = \begin{cases} \mathbf{a^{(j)}W^{(j)}} & \text{if } j \in \{2,3,\dots,I\} \\ \mathbf{XW^{(1)}} & \text{if } j = 1 \end{cases}$$

where $\mathbf{a^{(j)}} \in \mathbb{R}^{N \times n_{j-1}}$ and $\mathbf{W^{(j)}} \in \mathbb{R}^{n_{j-1} \times n_j}$.

Therefore, the forward propagation process consists on the propagation of each layer information to the right direction in order to calculate a output which has the following form:

$$g(\mathbf{x}) = f^{(I)}\left(f^{(I-1)}\left(f^{(I-2)}\left(\left(f^{(1)}(\mathbf{XW}^{(1)})\right)\dots\mathbf{W}^{(I-2)}\right)\mathbf{W}^{(I-1)}\right)\mathbf{W}^{(I)}\right) \tag{3.1}$$

The output of a neural network is a composite function. The output for one of these functions depends not only on the input values but also on the weights of the neural network. It is important to mention that the parameters $\mathbf{W^{(j)}}$ for each of the hidden layers are initialized with small random numbers. A pseudocode for the forward propagation algorithm is provided in Alg 5.

Input: The training data $\mathbf{X_{N\times p}}$, $I \in \mathbb{N}_+$ the number of hidden layers in the network, and $\{f^{(i)}\}_{i=1}^{I}$ the activation functions of each hidden layer.
Process:
`# The matrix` $\mathbf{X}$ `includes the column of 1 representing the bias.`
Initialize $\mathbf{W^{(j)}}$ with zeros or small random numbers, for all $j \in \{1, 2, \ldots, I\}$.
for *all* $j \in \{1, 2, \ldots, I\}$ **do**
  if $j = 1$ **then**
    $\mathbf{Z^{(j)}} = f^{(j)}(\mathbf{XW^{(j)}})$
  end
  else
    $\mathbf{Z^{(j)}} = f^{(j)}(\mathbf{Z^{(j-1)}W^{(j)}})$
  end
end
Output: $Z^{(I)}$

Algorithm 5: Forward propagation algorithm.

3.7.2 Backward propagation process

To train neural networks, i.e. to find the parameters $\mathbf{W^j}$ that better fit the data, we use gradient descent. However, gradient descent need the computations of $\frac{\partial J}{\partial \mathbf{W^j}}$. The derivatives of the cost function in terms of the parameters $\mathbf{W^j}$ are computed using the *back-propagation algorithm*.

Backpropagation propagates the error from the output to the internal layers of the network. The Backpropagation algorithm is probably the most fundamental piece for the training of classical neural networks. This algorithm was first introduced in 1960 but was only popularized and used to train NN in 1989 by psychologist David Rumelhart and computer scientists Geoffrey Hinton and Ronald J. Williams [37].

Backpropagation seeks to take advantage of the fact that the output of the neural network is a composite function of the form 3.1, to calculates the gradient in each of the layers using the chain rule for derivatives. In addition, this algorithm computes the derivatives by propagating the error from right to left, or in a back-propagation direction.

To calculate the gradient in the j-th layer, the algorithm instead of recomputing the derivatives of all subsequent layers, uses the previously calculated information from layers $\{j+1, , I\}$ in order to optimize the process.

In general, we have the following cost function:

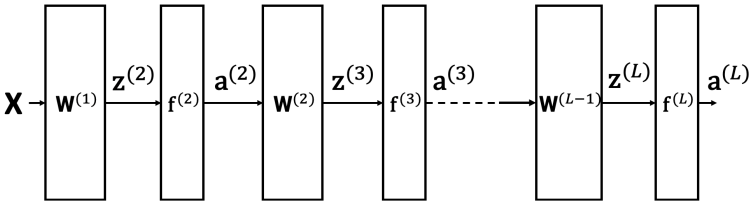

FIGURE 3.6: Backpropagation algorithm representation.

$$J\left(\mathbf{y}, g(\mathbf{X})\right) = J\left(\mathbf{y}, f^{(I)}\left(f^{(I-1)}\left(f^{(I-2)}\left(\left(f^{(1)}(\mathbf{X}\mathbf{W}^{(1)})\right)\ldots\mathbf{W}^{(I-2)}\right)\mathbf{W}^{(I-1)}\right)\mathbf{W}^{(I)}\right)\right).$$

To calculate the derivatives in each of the layers and propagate them backwards recursively, let's take the figure 3.6 as a reference. Since the cost function depends on the output $g(\mathbf{X})$, it can be evaluated as if we were calculating the cost at the output.

From figure 3.6 we can see that the output of each block is the matrix product between the output of a previous block and the parameters of the block. In case the block is a function, the output is the evaluation of that function. Now, let's calculate the derivative of the cost as a function of the input data X:

$$\frac{\partial \mathbf{J}}{\partial \mathbf{X}} = \frac{\partial \mathbf{J}}{\partial \mathbf{a}^{(L)}}.\frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L)}}.\frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{a}^{(L-1)}}.\frac{\partial \mathbf{a}^{(L-1)}}{\partial \mathbf{z}^{(L-1)}}.\frac{\partial \mathbf{z}^{(L-1)}}{\partial \mathbf{a}^{(L-2)}}\ldots\frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{z}^{(2)}}.\frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{X}} \tag{3.2}$$

From here we can see that the partial derivatives of the outputs of the activation functions with respect to the inputs are the same derivatives of the activation functions, so we have that:

$$\frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L)}} = \left(f^{(L)}\right)'.$$

In addition $\mathbf{z}^{(L)} = \mathbf{a}^{(\mathbf{L-1})}\mathbf{W}^{(L-1)}$ then

$$\frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{a}^{(L-1)}} = \mathbf{W}^{(L-1)},$$

Replacing recursively $\frac{\partial \mathbf{a}^{(j)}}{\partial \mathbf{z}^{(j)}}$ and $\frac{\partial \mathbf{z}^{(j)}}{\partial \mathbf{a}^{(j-1)}}$ on the equation 3.2 we get that

$$\begin{aligned} \frac{\partial \mathbf{J}}{\partial \mathbf{x}} &= \frac{\partial \mathbf{J}}{\partial \mathbf{a}^{(L)}}\left(f^{(L)}\right)'\mathbf{W}^{(L-1)}\left(f^{(L-1)}\right)'\mathbf{W}^{(L-2)}\ldots\left(f^{(2)}\right)'\mathbf{W}^{(1)} \iff \\ \nabla_{\mathbf{x}}\mathbf{J} &= \left(\mathbf{W}^{(1)}\right)^{\mathrm{T}}\left(f^{(2)}\right)'\ldots\left(\mathbf{W}^{(L-2)}\right)^{\mathrm{T}}\left(f^{(L-1)}\right)'\left(\mathbf{W}^{(L-1)}\right)^{\mathrm{T}}\left(f^{(L)}\right)'\nabla_{\mathbf{a}^{(L)}}\mathbf{J} \end{aligned} \tag{3.3}$$

Let's define the error in the l-th layer as:

$$\delta^{(l)} = \left(f^{(l)}\right)'\left(\mathbf{W}^{(l)}\right)^{\mathrm{T}}\ldots\left(\mathbf{W}^{(L-2)}\right)^{\mathrm{T}}\left(f^{(L-1)}\right)'\left(\mathbf{W}^{(L-1)}\right)^{\mathrm{T}}\left(f^{(L)}\right)'\nabla_{\mathbf{a}^{(L)}}\mathbf{J}$$

Observe that $\delta^{(l)}$ has the same size as the number of neurons in the l-th layer. Thus, this term express the error propagation from the output layer until the l-th

layer. Now, the derivative of the cost function as a function of the parameters $\mathbf{W}^{(l-1)}$ is given by:

$$\nabla_{\mathbf{W}^{(l-1)}}\mathbf{J} = \nabla_{\mathbf{z}^{(l)}}\mathbf{J}\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l-1)}}$$

Since $\nabla_{\mathbf{z}^{(l)}}\mathbf{J} = \delta^{(l)}$ and $\mathbf{z}^{(l)} = \mathbf{a}^{(l-1)}\mathbf{W}^{(l-1)}$ then:

$$\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l-1)}} = \left(\mathbf{a}^{(l-1)}\right)^{\mathrm{T}} \iff \nabla_{\mathbf{W}^{(l-1)}}\mathbf{J} = \delta^{(l)}\left(\mathbf{a}^{(l-1)}\right)^{\mathrm{T}}$$

The factor $\mathbf{a}^{(l-1)}$ indicates that the influence of the weights $\mathbf{W}^{(l-1)}$ on the output is being affected by the magnitude of the activation functions in layer $l-1$. Therefore, the error propagation for layers closer to the input layer can be obtained recursively by means of the equation:

$$\delta^{(l-1)} = \left(f^{(l-1)}\right)'\left(\mathbf{W}^{(l-1)}\right)^{\mathrm{T}}\delta^{(l)}$$

In this way the gradient of the cost function with respect to changes in the weights can be calculated with matrix multiplications at each layer. Finally coupling this with gradient descent, the update weights at layer (j) are given by:

$$\mathbf{W}^{(j)} := \mathbf{W}^{(j)} - \eta\nabla_{\mathbf{W}^{(j)}}\mathbf{J}$$

where $\nabla_{\mathbf{W}^{(j)}}\mathbf{J} = \delta^{(j+1)}\left(\mathbf{a}^{(j)}\right)^{\mathrm{T}}$ and η is the learning rate.

If j is the output layer, then $\delta^{(j+1)} = \nabla_{\mathbf{a}^{(j)}}\mathbf{J}$. Since $a^{(j)}$ is the output of the network, if we define the cost function as $\mathbf{J} = \frac{1}{2}||\mathbf{y} - \mathbf{a}^{(j)}||^2$, then the weights of the immediately preceding layer are:

$$\nabla_{\mathbf{W}^{(j-1)}}\mathbf{J} = \delta^{(j)}\left(\mathbf{a}^{(j-1)}\right)^{\mathrm{T}} = \left(\mathbf{a}^{(j)} - \mathbf{y}\right)\left(\mathbf{a}^{(j-1)}\right)^{\mathrm{T}}$$

Thus, we can express the calculation of the error as:

$$\delta^{(j-1)} = \left(f^{(j-1)}\right)'\left(\mathbf{W}^{(j-1)}\right)^{\mathrm{T}}\left(\mathbf{a}^{(j)} - \mathbf{y}\right)$$

and then we can update the parameters

$$\mathbf{W}^{(j-1)} := \mathbf{W}^{(j-1)} - \eta\nabla_{\mathbf{W}^{(j-1)}}\mathbf{J}$$

this process is repeated until all the parameters has been updated.

In summary, in NN the forward propagation process is performed to transform the incoming signal until reaching the last layer. For training we calculate the error at the output and, through the backward propagation process, propagates the error to each layer in order to update its parameters. This process is repeated until the parameters $\mathbf{W}^{(j)}$ for each layer converge to the optimal ones, those which minimize the cost function $\mathbb{J}$.

3.8 Convergence problems of Classical Neural Networks

Neural networks are one of the most powerful models of machine learning, which currently have applications in different areas. However, these models are computationally expensive, and if not trained correctly the models will not converge.

Neural networks have a large number of hyperparameters that need to be defined for the network architecture. This makes it necessary to perform a search to determine the hyperparameters that optimize the network performance. This could take a long time depending on how complex is the architecture of the model. Some of these hyperparameters are the number of hidden layers, the number of neurons per layer, the activation functions for each layer, the learning rate, etc.

In addition, neural networks are sensitive to over-fit, when a neural network is trained, it needs to be able to generalize, that is, extract the most relevant information from the training data without memorizing it. This is very important because when new data is process through the network, we expect similar results than during training. For this purpose, several factors must be taken into account, such as the complexity of the architecture, the surface formed by the cost function and the initialization of the parameters.

This leads to another important point to highlight, the parameters initialization of the classical neuronal networks is one of the most fundamental factors in the training phase. This due the fact that the training of a neural network can generally be expressed as a non-convex optimization problem, which, due to the large-scale training data and the complexity of the network, is composed of millions of parameters and does not have an analytical solution [38]. Generally, when using gradient descent, the parameters of the NN are initialized randomly, which could set the training process in problematic regions of the cost function. It will be better if prior information of the training data can be use to properly initialize the model.

3.9 Initialization proposal

There exist several ways to initialize NN, however, the most popular ones nowadays, initialize the NN randomly from a predefined distribution. The Xavier method uses the Uniform distribution while the He method uses the Gaussian distribution. Those methods vary the values of the statistics (μ, σ) for the distribution depending on the training data size, in order to mitigate the vanishing gradient problem [38]. More recent research has focused on initializing NN based on the training data characteristics. Jeff Donahue proposed in 2016 to normalize the initial random parameters according to the magnitude of the training data [39]. However, due to the randomness, these methods are still stochastic.

Here we proposed a deterministic initialization algorithm that depends on the training data structure. We defined this method as an extension of the work presented in Chap 2. This initialization algorithm is also dependent on the architecture of the NN, and it can provide insights about the optimal architecture for a given

problem. We will first explain the initialization proposal for hidden layers of a neural network without an specific architecture. Then, we will present the generalization for any other architecture.

3.9.1 Hidden Layer initialization

Let's remember that neural networks are made up of several layers of neurons. Where, each of the layers performs an embedding of the data it receives into a d-dimensional space, where d is the number of neurons of the layer. This models, unlike logistic regression, are characterized by being a nonlinear classifier, i.e. they have the ability to classify data whose decision boundaries are nonlinear.

The main idea for the initialization for neural networks lies in expressing each of the nonlinear boundaries, which separate the data of the different classes, as the union of hyper-planes which divide the data set of a class into subsets that are linearly separable from the other classes.

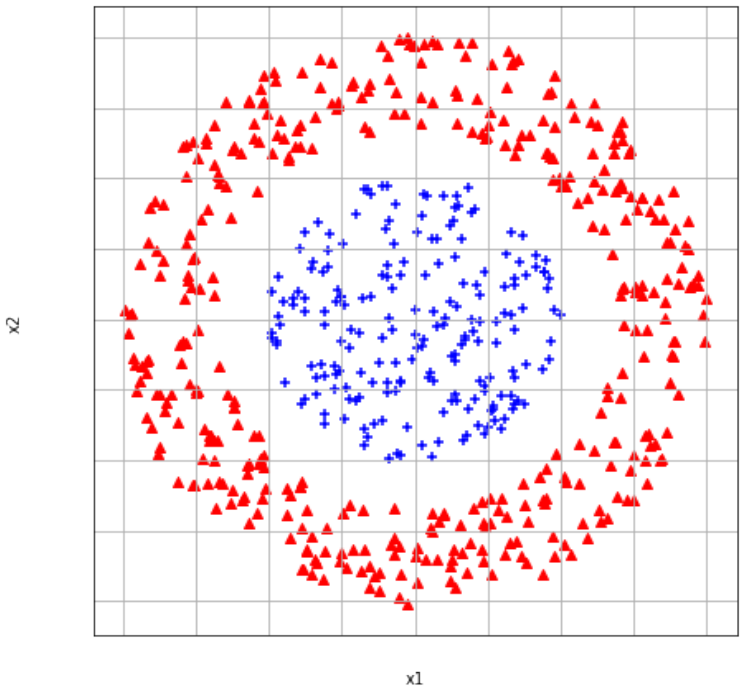

FIGURE 3.7: Illustration of a binary classification problem with only one non-linear decision boundary.

Figure 3.7 shows an example of a binary classification problem (red and blue classes) in which we have a data set $\mathbf{X}$ in two-dimensional space and only one circular decision boundary (non-linear). To estimate the initial parameters $\boldsymbol{\theta}$ of the first hidden layer of a neural network, we are going to linearize the circular boundary, i.e. divide it in linearly separable regions in a way that when we join those regions, the decision surface is adequately mapped.

In order to do that, we select the elements from a given class (could be the red or the blue class) that are placed close to the decision boundary and that map it in

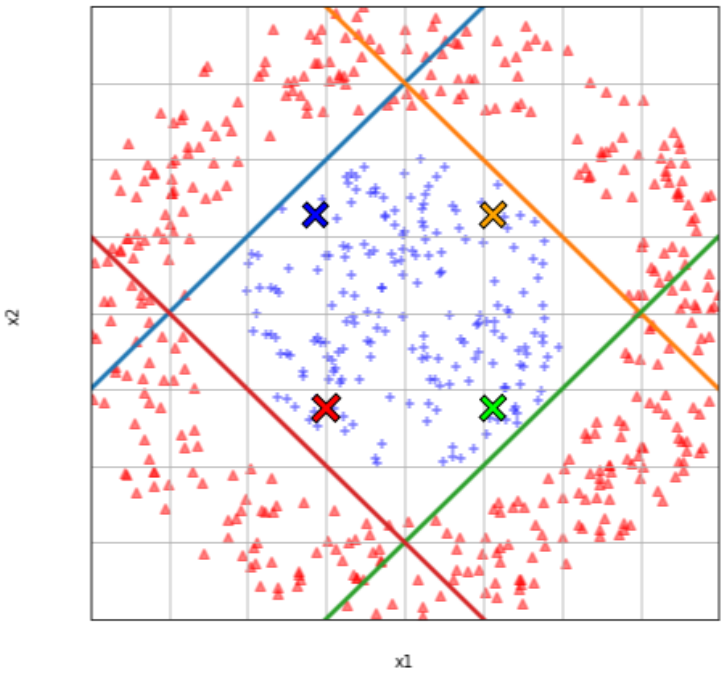

FIGURE 3.8: Illustration of the circular boundary (non-linear) linearization using 4 elements of the blue class.

the adequate shape. We select as many observations from the class as neurons in the first hidden layer of the network. Figure 3.8 shows an example of the circular boundary linearization. There are several strategies to map the decision frontier optimally, such as using distance criteria, entropy maximization [40], self-organizing maps [41], etc. However, this is a problem that goes beyond the main focus of this thesis.

Once we have the set of characteristic vectors of a class, in this example the blue class, that uniformly map the decision surface. From these vectors we partition the mapped data using proximity criteria in regions so that each vector is assigned a neighborhood that has data from the adjacent classes in the dataset. Figure 3.9 shows an example of the characteristic vectors neighborhood. As in the case of mapping, there are several techniques to assign such neighborhoods such as Voroni diagrams [42], Clustering [43], Locally sensitive hashing [44], etc.

In each neighborhood we compute the bias as we explain in Chap 2 using the associated characteristic vector as the initial parameters $\boldsymbol{\theta}$. The characteristic vectors, together with the calculated biases initialize the first hidden layer of the neural network. Figure 3.10 shows a visualization of the complete process to initialize the first hidden layer of a network.

Now, for the initialization of subsequent layers, we use the proposed initial parameters of the previous layers in order to make a forward propagation of the data until the layer we want to initialize, and we repeat the process with the output of the layer immediately behind. Figure 3.11 illustrates the initialization of an entire neural network. This proposal causes that the initialization of a hidden layer is based on the prior knowledge of the embedded data structure collected by the previous layers and the target output.

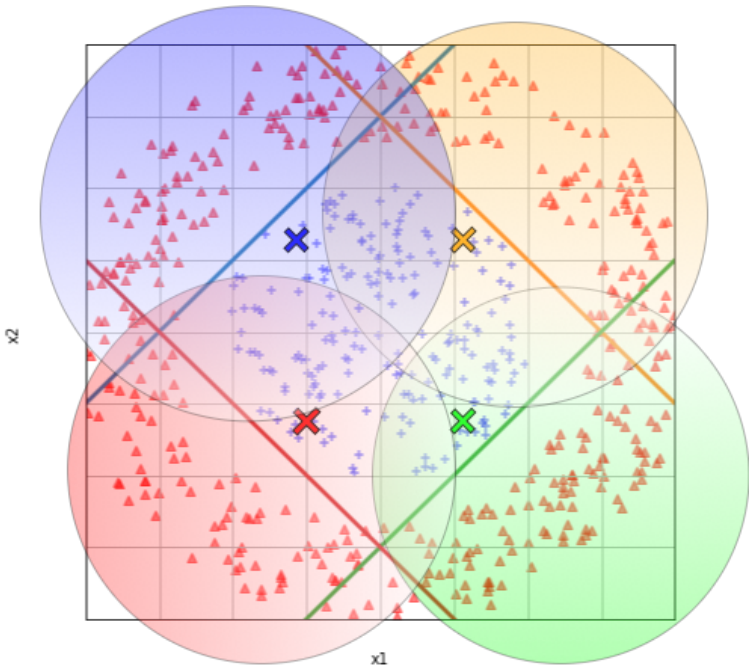

FIGURE 3.9: Illustration of the neighborhoods assigned to the 4 characteristic vectors of the blue class that uniformly map the decision boundary.

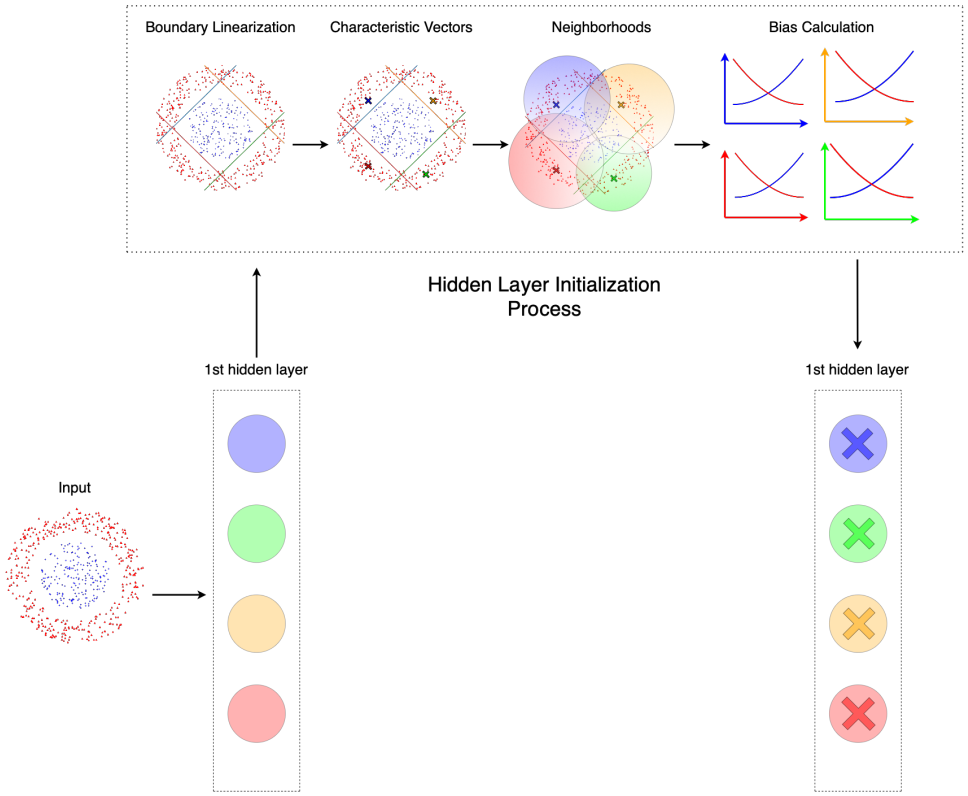

FIGURE 3.10: Illustration of the first hidden layer initialization. The $\boldsymbol{X}$ in the neurons of the first hidden layer at the end of the process represents that the neurons are already initialized.

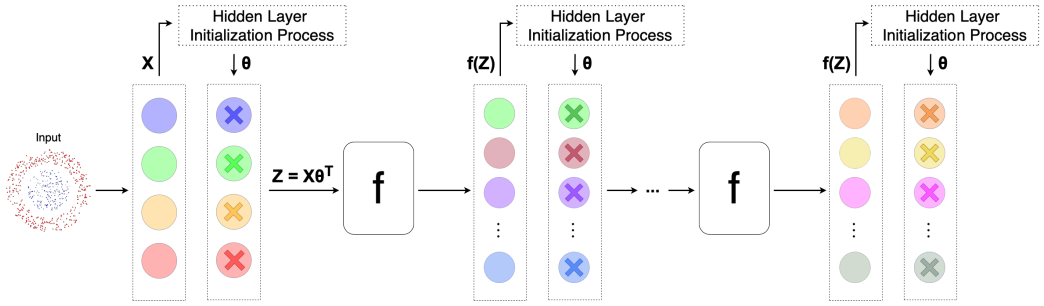

FIGURE 3.11: Illustration of the process to initialize a neural network. The X in the neurons of the hidden layers at the end of the processes represents that the neurons are already initialized.

In summary, the general idea for finding the initialization for the neurons in the jth layer is to use the output of the layer $j-1$. The output $\boldsymbol{a}^{(j-1)}$ is obtained by forward propagation of the input. Therefore, each observation in $\boldsymbol{a}^{(j-1)}$ has a class associated. We define a number of points from $\boldsymbol{a}^{(j-1)}$ equal to the number of neurons in the jth layer. These points are selected such that they are close to the decision boundary and map it adequately. From these points we partition the mapped data $\boldsymbol{a}^{(j-1)}$ using a proximity criteria. From each partition we compute the bias as indicated in Chap 2, here we use the points as initial parameters. The initialization of the jth layer is given by the mapped points and the calculated biases.

In the previous case, only the elements of the blue class were used for the initialization of the layers. However, it is possible to consider characteristic vectors for both classes, i.e. have certain neurons that represent the blue class and others that represent the red class. In this case, the same initialization process is performed for both the blue class and the red class in order to initialize the hidden layer. This allows the initialization to consider information of all the classes and of the boundaries of separation.

Figure 3.12 shows a visualization of a hidden layer initialization process when all classes are taken into account. The number of neurons that are dedicated to each of the classes is a hyperparameter of the proposed initialization method. When there are multiple classes, the same concept is applied for each class, following a similar pattern to the one proposed by the one vs rest strategy.

This method aims to linearize the decision boundary of nonlinear problems. We can see that the representative data that allow us to develop said linearization are prototype data of each one of the classes. These prototype data are not the most representative data of the entire class distribution, but in this case, they are locally representative data, i.e. data that map the separation boundary with the other classes. It is interesting to highlight that when there are non-linear separation regions, as is the case of neural networks, the representative vectors of the distribution are not of interest, but instead, vectors that allow mapping the non-linearity of the problem are sought. This has some similarity with the philosophy behind support vector machines.

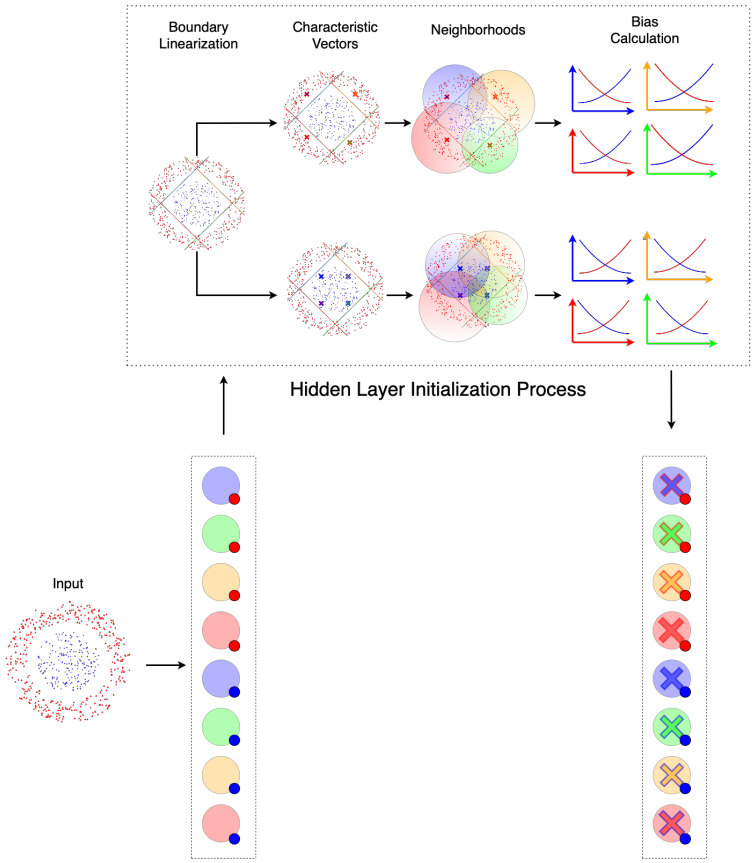

FIGURE 3.12: Illustration of the initialization of a hidden layer when the two classes are considered. The $\boldsymbol{X}$ in the neurons of the hidden layer at the end of the process represents that the neurons are already initialized. The color filled circle represents the class to which the weights that initialize the neuron belong to.

3.9.2 Open Questions

In this algorithm an initialization of the parameters of the neural network based on the distribution of the training data is proposed. One of the major problems in the proposed initialization is how to select the prototype vectors and the corresponding neighborhoods. We propose to select the prototype vectors through the entropy maximization criteria and the neighborhoods of each obtained vector through a distance criteria. This can be computationally expensive, so another option is that, depending on the problem, the prototype vectors are selected manually. It is important to mention that this strategy differs from k-means, because this method takes into account the entire data distribution, but we are only interested in the data that maps the separation region.

Another problem is how to select the number of layers and neurons per layer. Although this issue is not part of this thesis, we observe a relationship between the number of neurons in a layer and the number of hyper-planes necessary to linearize the decision boundary of the input data to the layer. Therefore, the number of neurons in a layer should be associated with how complex is the non-linear separation regions of the layer input data. The idea is that as the forward propagation process is performed, the complexity of boundary should decrease. Consequently, the number of layers should be defined based on the number of transformations necessary for

the decision boundary of the embedded data to be linearly separable.

On other hand, the proposed initialization method uses prototype vectors for each class to establish the initialization parameters of the network. However, the strategy will present problems when more neurons than observations are required. In these cases we can use strategies from data augmentation prior to the initialization, in order to increase the amount of data preserving their initial distributions.

We explained the initialization proposal for logistic regression in the chapter 2. In this chapter, we explained the extension of the initialization to classical feed-forward neural networks. The next chapter will be devoted to evaluate these strategies with toy examples and with real data sets in order to generate a benchmark of the proposed methods.

Chapter 4

Results

The algorithms developed in this thesis were implemented using `Python 3.7.6`. Specifically, the following libraries were used:

- `numpy 1.19.5` [45]
- `matplotlib 3.1.3` [46]
- `pandas 1.1.4` [47]
- `seaborn 0.11.1` [48]
- `scikit-learn 0.22.1` [49]

In this chapter we will present the results of applying the initialization strategy in different datasets for logistic and multinomial regression models presented earlier (Chapter 2), and also for neural networks models (Chapter 3). A comparison with other initialization strategies will be also presented.

4.1 Datasets Description

A benchmark for the initialization proposal was made using various real datasets for classification tasks from the UCI and Kaggle repositories. A description for each of these is presented below, also we illustrate the pre-processing we applied for each dataset.

- **Wine** [50]: This dataset is the results of a chemical analysis of wines grown in a region located in Italy but derived from three different cultivars. The analysis determined the quantities of 13 continuous constituents found in each of the three types of wines. The main purpose of this dataset is to classify the wine quality according to a 1-5 scale.

  We decide to reduce the classes to only 3 classes since there was a considerable imbalance between the 5 classes originally proposed by the dataset. The 3 classes where assigned in the following way:

  - The **low** class corresponds to the very low and low quality.
  - The **medium** class takes the same medium quality proposed by the dataset.
  - The **high** class correspond to the high and very high quality.

  Likewise, the data were centered and standardized.

- **Iris** [50]: The iris dataset, contains three balanced classes of 50 instances each, where each class refers to a type of iris plant. The features of this dataset are the sepal length (cm), the sepal width (cm), the petal length (cm) and the petal width (cm). The main objective of this dataset is to classify an iris plant between Setosa, Versicolour or Virginica. In this case, the data was just normalized and standardized.

- **Madelon** [51]: The Madelon dataset is an artificial dataset which contains data points grouped in 32 clusters placed on the vertices of a five dimensional hyper-cube and randomly labeled by 1 or -1. The five dimensions constitute 5 informative features and 15 linear combinations of those features were added to form a set of 20 (redundant) informative features where the order of the features and patterns were randomized. The main objective of this dataset is to classify the constructed labels; however, the structure of the artificial generated data made this dataset one of the most challenging.

- **Ozone**: This dataset was generated based on real measurements of ozone that were collected from 1998 to 2004 in Houston, Galveston and Brazoria areas. It contains two ground ozone level data sets; the eight hour peak set, and the one hour peak set (onehr.data), which represent the frequency of the measurements. The main features for this dataset are the local ozone peak prediction, the ipwind ozone background level, the precursor emissions related factor, the maximum temperature in degrees F, the base temperature where net ozone production begins (50 F), the total solar radiation for the day, the wind speed near sunrise (using 09-12 UTC forecast mode) and the wind speed mid-day (using 15-21 UTC forecast mode). The main purpose is to classify if the ozone levels are higher or lower than normal ones. This dataset has null values, so we perform a mean interpolation to fill those values according to the ozone measurements of the 30 previous observations. We also normalize and standardize the data.

- **Breast Cancer** [50]: This dataset was published by the Oncology Institute and it has repeatedly appeared within the machine learning literature. The breast cancer dataset has two classes that indicate whether a patient suffers from the disease or not. One class includes 201 instances while the other one has 85 instances. Each observation is described by 9 attributes, from which 7 of those are categorical variables. The data quantitative predictors were normalized and standardized while the categorical variables were encode using one-hot encoding.

- **Image Segmentation** [50]: This dataset was manually generated based on outdoor images, where each image was hand-segmented to create a classification for every pixel, in order to recollect 19 important features of the image; the column of the center pixel of the region, the row of the center pixel of the region, the contrast of horizontally adjacent pixels in the region and the average over the region of the *green* value. The main purpose of this dataset is to classify the image in one of the following classes: grass, foliage, cement, window, path, sky or brickface. In this case, the data was just normalized and standardized.

- **Mnist**: The MNIST is one of the most famous and important datasets in supervised learning. It contains $60,000$ small square 28×28 pixel gray-scale images of handwritten single digits between 0 and 9. The main objective is to classify

a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9. In this case the dataset were standardized in order to re-scale the intensities into 0-1.

- **KDD Cup 1999 Data**: This dataset is about network traffic and was used for The Third International Knowledge Discovery and Data Mining Tools Competition, which was held in conjunction with KDD-99 The Fifth International Conference on Knowledge Discovery and Data Miningand. The data set is approximately 4 gigabytes of compressed raw TCP (binary) dump data from 7 weeks of network traffic, which can be processed in approximately 5 million connection records, each with approximately 100 bytes. There exist 5 classes that corresponds to the simulated attacks; denial of service (DoS), user to root (U2R), remote to local (R2L) and probing attack. This dataset contains 150 features but only the 22 most relevant were used. Likewise, the data were normalized and standardized.

4.2 Logistic Regression Results

To test the proposed initialization strategy for logistic regression, in addition to using the datasets mentioned in 4.1 as a benchmark of the strategy, we created toy examples in order to illustrate and compare the behavior of logistic regression with random initialization and with the proposed initialization. For the results shown below, we use mini-batch gradient descent with a learning rate of 0.001.

For the toy datasets, the following pipeline was used:

1. Train a randomly initialized logistic regression model for 500 epochs using 10-fold cross validation.
2. Calculate the 95% confidence intervals for the loss and accuracy for both the training and the validation set.
3. Set as model parameters, those that, on average, minimized the loss of the validation set.
4. Evaluate on the test set.
5. Calculate the proposed initialization parameters ω from the training and validation sets.
6. Repeat steps 2-5 with a logistic regression model initialized with the proposed parameters.

4.2.1 Binary Toy Datasets

The first toy dataset which represents a binary classification problem consists of two balanced Gaussian distributions (one for each class), linearly separable and overlapping distributions, this data is shown in figure 4.1. Figure 4.2 shows the performance of the random and the proposed initialization for the validation set, after applying 10-fold cross validation.

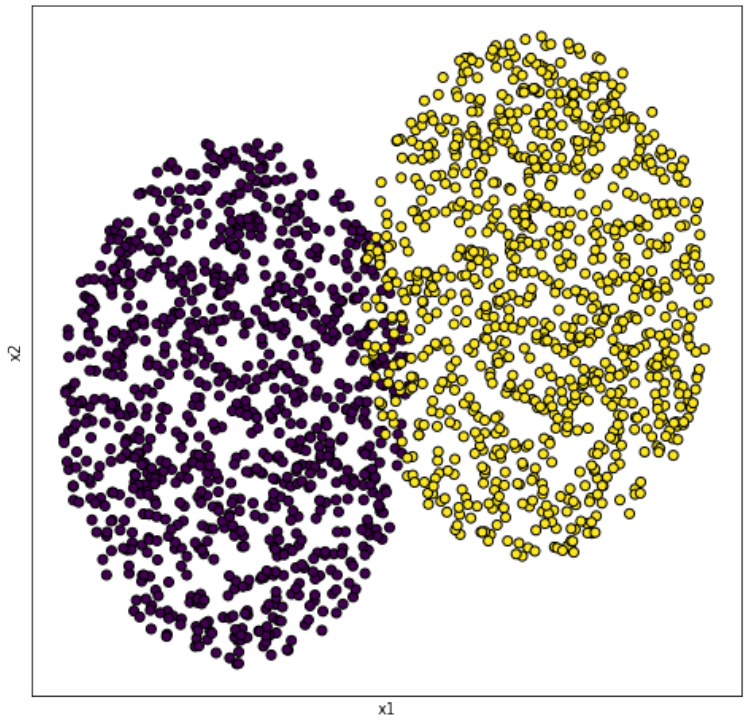

FIGURE 4.1: Generated binary dataset using two balanced Gaussian distributions.

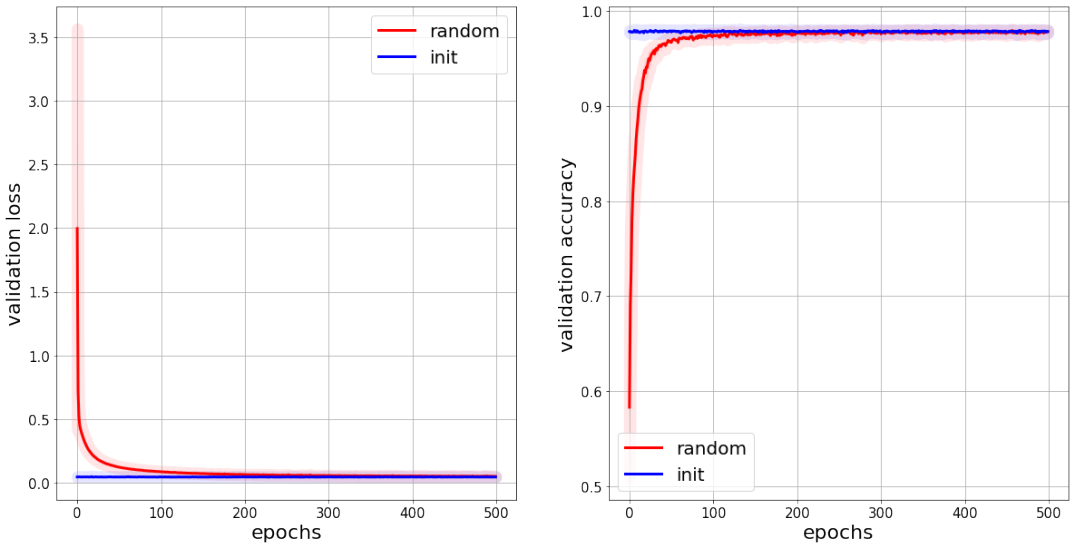

FIGURE 4.2: Random and proposed initialization performance (loss on the left and accuracy on the right) for the validation set with 95% confidence intervals, presented as the shadow regions. The dataset used for this test is the one presented in figure 4.1

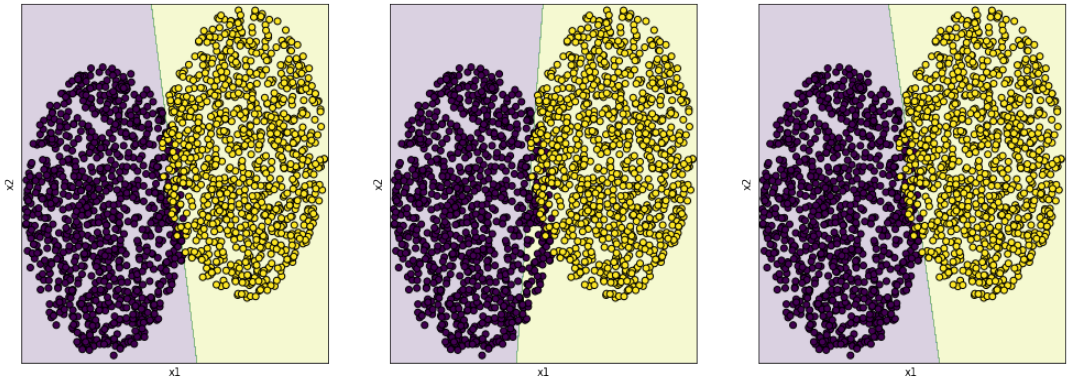

FIGURE 4.3: Illustration of the classification regions with a class-balanced dataset for the proposed initialization (without training), the convergence after 25 epochs (0.67s) training with random initialization and the convergence after 25 epochs (0.65s) training with proposed initialization respectively (the initialization time is already included).

The model with random initialization converged to the $\boldsymbol{\theta}$ parameters that produced the smallest validation loss value at epoch 246 with a training time of approximately 4.17 seconds, while the model with the proposed initialization took only 68 epochs with a training time of approximately 1.13 seconds. Since the initialization took approximately 4.38e$-$3 seconds, then the total training time for the model that was initialized with our proposal was approximately 1.1348 seconds.

	Logistic regression with random initialization (246 epochs - 4.17 seconds)		**Proposed method without training (0.00438 seconds)**		**Logistic regression with proposed initialization (68 epochs - 1.13 seconds)**	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Purple	1	0.98	0.99	0.98	0.99	0.99
Yellow	0.98	1	0.98	0.99	0.99	0.99
Total	0.99	0.99	0.99	0.99	0.99	0.99

TABLE 4.1: Results for the test set for the proposed initialization parameters for a test set without training (center); after convergence of the logistic with both, the random (left) and the proposed (right) initialization. Logistic regression models were trained using a learning rate of 0.001 and a maximum of 500 epochs. The model chosen was the one that produced the lowest loss value in the validation set. The dataset used for this test is the one presented in figure 4.1

Table 4.1 shows the performance for the models using the proposed initialization (without training), the model with random initialization after training, and the model with the proposed initialization (with training) for the test set. It is worth mentioning that these models were selected based on the performance on a validation set obtained after 10-fold cross validation. The classification regions for each of the models were also calculated using all the available data, these are found in the figure 4.3.

A second test was carried out with the same toy dataset, but this time with an imbalance of 90% in favor of the yellow class, in order to evaluate the robustness of the initialization strategy against imbalance. The performance of the models with

	Logistic regression with random initialization		Proposed method without training		Logistic regression with proposed initialization	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Purple	0.9	1	0.97	0.70	0.94	1
Yellow	1	0.99	0.95	0.99	1	0.99
Total	0.99	1	0.95	0.96	0.99	0.99

TABLE 4.2: Results for the test set with class-unbalanced dataset of the proposed initialization parameters without training in the center and after convergence of the logistic with both, the random (in the left) and the proposed (in the right) initialization. Logistic regression models were trained using a learning rate of 0.001 and for 500 epochs. The model that produced the least loss in the validation set was chosen.

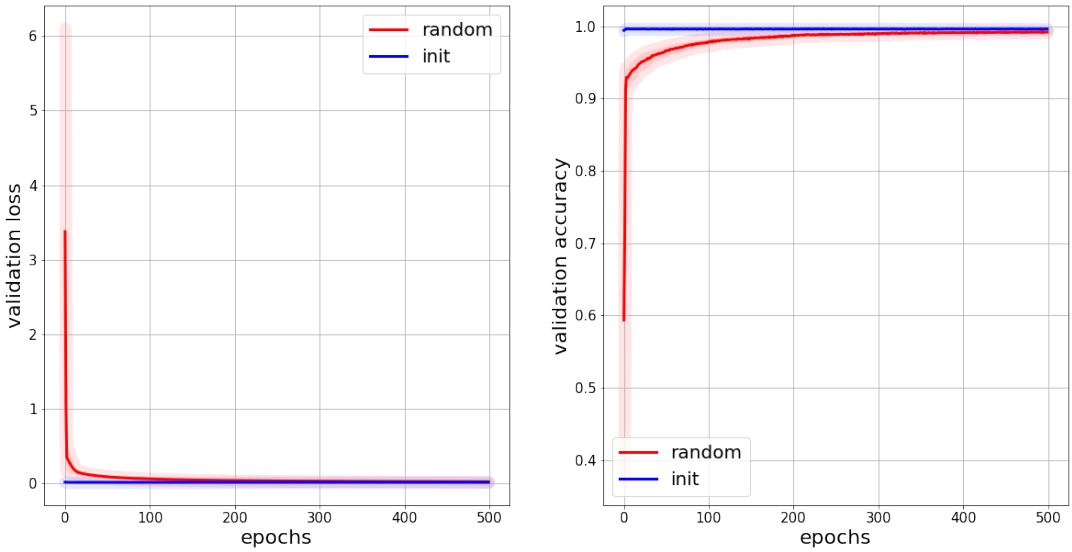

FIGURE 4.4: Random and proposed initialization performance (loss and accuracy respectively) for the validation set with 95% confidence intervals with a class-unbalanced dataset.

random and proposed initialization with the validation set are shown in figure 4.4 and the results for the test set are found in table 4.2, the classification regions are shown in the figure 4.5. For this experiment, the model with random initialization converged in 318 training epochs (2.92s), while the model with the proposed initialization took only 2 training epochs (0.023s). The total training time for the model that was initialized with our proposal was approximately 0.0247 seconds.

4.2.2 Multi-Class Toy Datasets

The second toy dataset represents a multi-class classification problem that consists of three balanced Gaussian distributions (one for each class), linearly separable and overlapped. These distributions were located with equidistant centers, they can be seen in figure 4.6. Figure 4.7 shows the average performance of the random and the proposed initialization for the validation set, after applying 10-fold cross validation.

The model with random initialization converged in 137 training epochs (4.33s), while the model with the proposed initialization took only 77 epochs (2.48s). Since

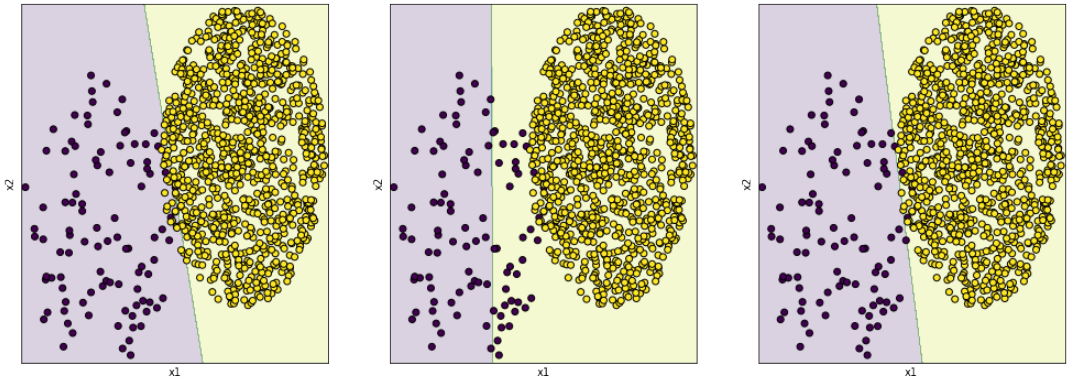

FIGURE 4.5: Illustration of the classification regions with a class-unbalanced dataset for the proposed initialization (without training), the convergence after 25 epochs (0.37s) training with random initialization, and the convergence after 25 epochs (0.29s) training with proposed initialization respectively (the initialization time is already included).

the initialization required 0.0029 seconds, the total time of the model that was initialized from the proposed strategy was 2.4829 seconds. Table 4.3 shows the performance of the three models for the test set. The classification regions are shown in figure 4.8.

	Logistic regression with random initialization		**Proposed method without training**		**Logistic regression with proposed initialization**	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Purple	0.96	0.96	0.95	0.94	0.96	0.96
Green	0.95	0.96	0.95	0.95	0.95	0.97
Yellow	0.93	0.91	0.90	0.89	0.93	0.91
Total	0.95	0.95	0.93	0.93	0.95	0.95

TABLE 4.3: Results for the test set with multi-class balanced dataset for the proposed initialization without training (center) and after convergence of training with both, the random (left) and the proposed (right) initialization. Logistic regression models were trained using a learning rate of 0.001 and 500 epochs. The model that produced the lowest loss value in the validation set was chosen.

Two other tests where performed with this multi-class toy dataset. The first one consider an imbalance of 90% in favor of the yellow and green classes, and the second one takes 95% less data points for the purple class and 98% less data points for the green class. This, in order to evaluate the robustness of the initialization strategy against imbalance affecting more than two classes. The results of the test that considers only one unbalanced class for the validation set are shown in the figure 4.9.

For this first test, the performance for the test set is found in table 4.4 and the classification regions are shown in the figure 4.10. This time, the model with random initialization converged in 177 training epochs (3.22s), while the model with the proposed initialization took only 80 epochs (1.61s). The total training time for the model that was initialized with our proposal was approximately 1.62s seconds, including initialization.

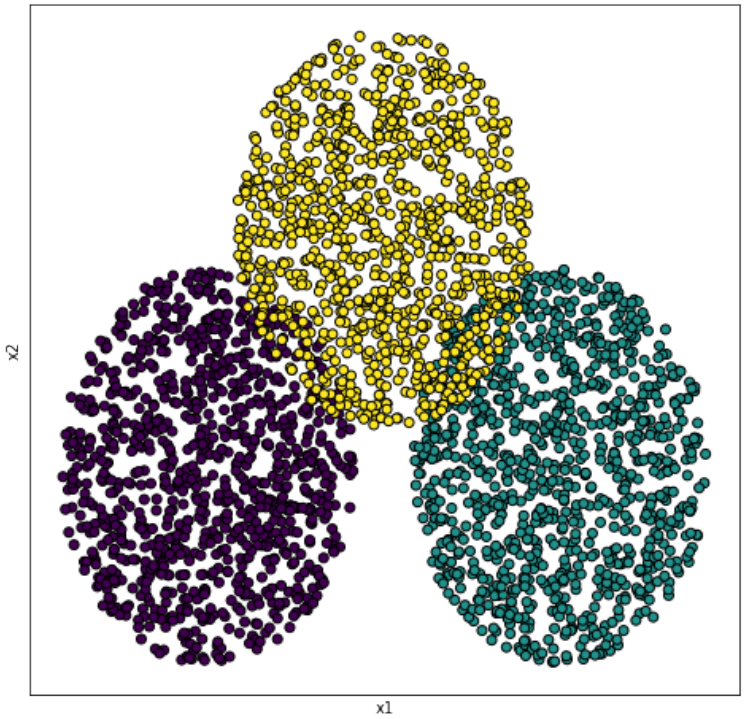

FIGURE 4.6: Generated data for the multi-class classification dataset using three balanced Gaussian distributions.

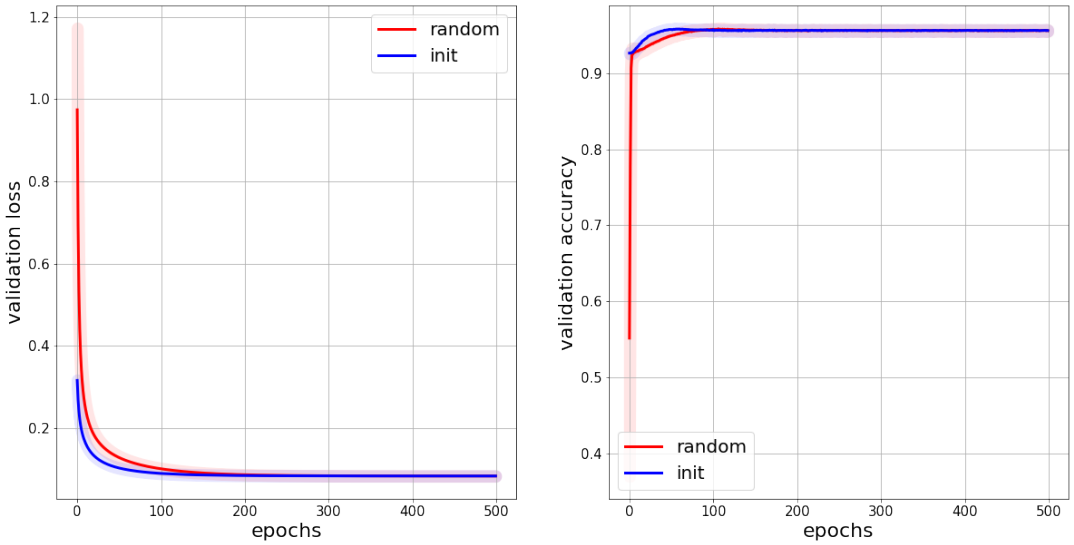

FIGURE 4.7: Random and proposed initialization performance (loss and accuracy respectively) for the validation set with 95% confidence intervals with a multi-class balanced dataset.

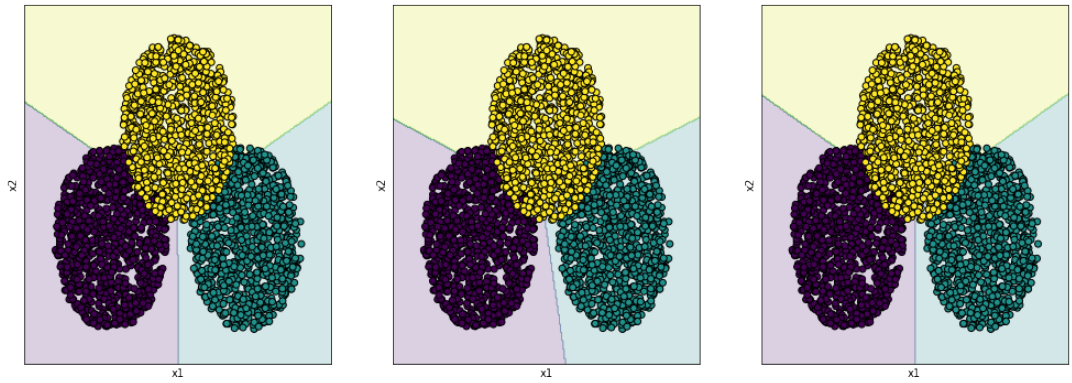

FIGURE 4.8: Illustration of the classification regions with a multi-class balanced dataset for the proposed initialization (without training), on the left axis, the convergence after 30 epochs (1.52s) training with random initialization in the middle axis, and the convergence after 30 epochs (1.47s) training starting with the proposed initialization on the right axis. The initialization time is already included.

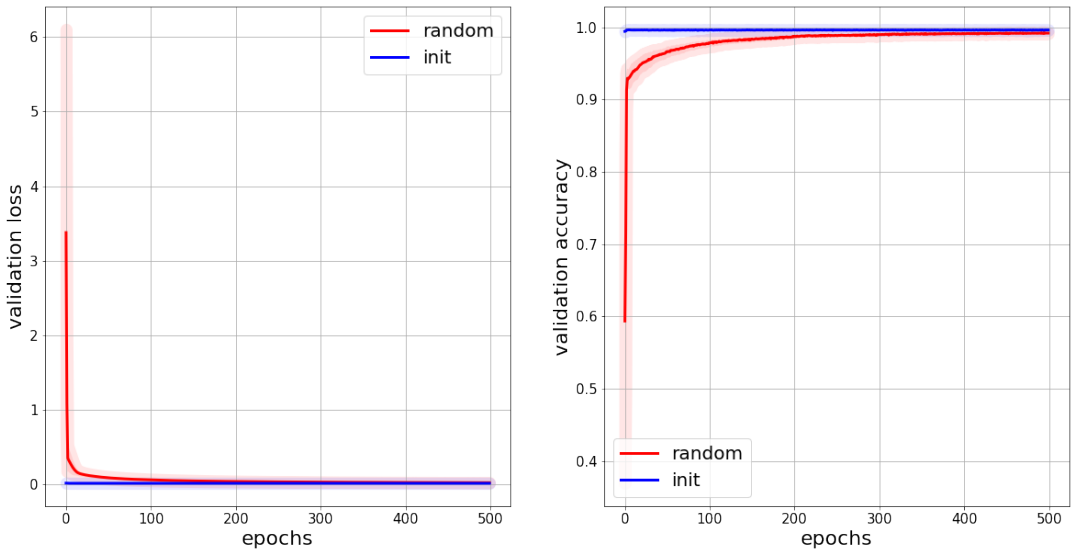

FIGURE 4.9: Random and proposed initialization performance for unbalanced in one class, loss on the left and accuracy on the right axis. The figures show the 95% confidence intervals.

	Logistic regression with random initialization		Proposed method without training		Logistic regression with proposed initialization	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Purple	0.87	1	0.90	0.71	0.87	1
Green	0.96	0.96	1	0.87	0.96	0.96
Yellow	0.96	0.95	0.81	0.99	0.96	0.95
Total	0.96	0.96	0.90	0.92	0.96	0.96

TABLE 4.4: Results with multi-class dataset with an unbalance class. Test set performance for the proposed initialization parameters without training and after convergence for the random and the proposed initialization. Logistic regression models were trained using a learning rate of 0.001 and 500 epochs. The model that produced the least loss in the validation set was chosen.

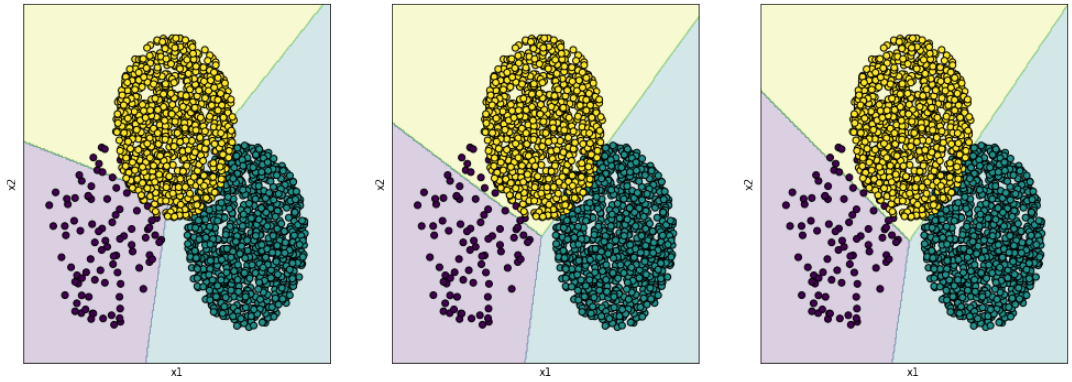

FIGURE 4.10: Illustration of the classification regions with an unbalance class for the proposed initialization without training, the convergence after 40 epochs (1.38s) training with random initialization and the convergence after 40 epochs (1.31s) training with proposed initialization respectively. The initialization time is already included.

For the second experimental setup we have that the results of the validation set with 95% confidence intervals are found in figure 4.11.

	Logistic regression with random initialization		Proposed method without training		Logistic regression with proposed initialization	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Purple	0.87	1	1	0.6	0.8	1
Green	0.33	1	0.83	0.5	0.66	1
Yellow	1	0.98	0.85	0.99	1	0.98
Total	0.98	0.98	0.86	0.96	0.98	0.98

TABLE 4.5: Results with multi-class dataset and an unbalance of 95% in the purple class and 98% in the green class. Test set performance for the proposed initialization parameters without training (center), and after convergence for the random (left) and the proposed (right) initialization. Logistic regression models were trained using a learning rate of 0.001 and 500 epochs. The model that produced the least loss in the validation set was chosen.

In addition, the results for the test set after selecting the best models (random

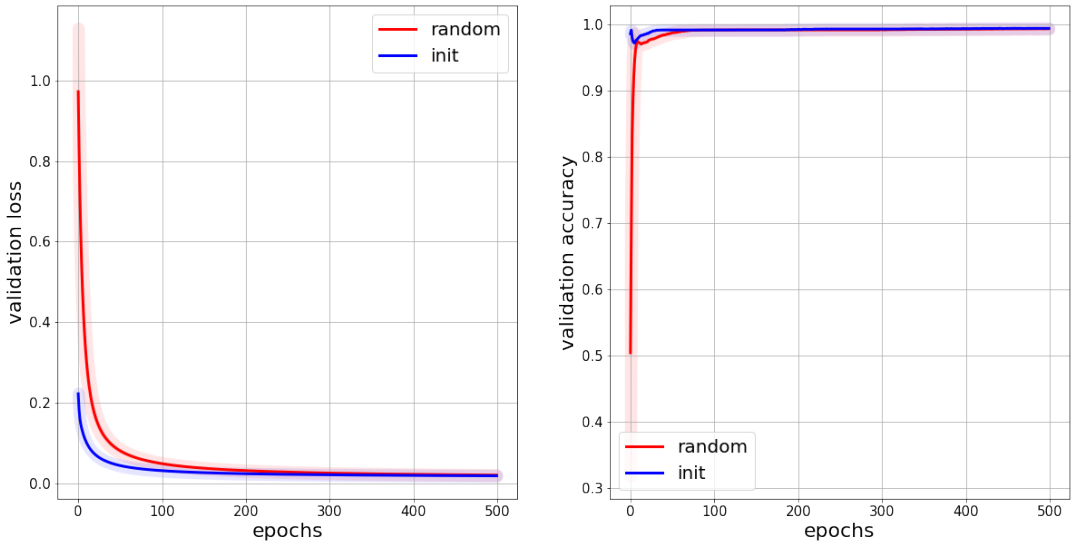

FIGURE 4.11: Random and proposed initialization performance, loss on the left and accuracy on the right, for the validation set with 95% confidence intervals with an unbalance of 95% in the purple class and 98% in the green class.

and proposed initialization) according to the validation set results are found in the table 4.5. The training of these models took 266 epochs (2.39s) and 137 epochs (1.22s) for the random and proposed initialization respectively. The initialization time is already considered. The classification regions for this unbalanced test are found in figure 4.12.

4.2.3 Benchmark with Real Datasets

This part aims to show the results of the logistic regression initialized both with the random strategy and with the one proposed in this thesis, for the datasets described in the section 4.1. These evaluations are from 10 repetitions of the test set to generate 95% confidence intervals. This results are found in table 4.6. To generate the data for each repetition we randomly sample the available data using bootstrap with replacement; and generated each time different training, validation and test sets.

4.3 Feed-forward Neural Networks Results

We created toy examples to test the initialization strategy for feed-forward neural networks proposed in this thesis, similar to the case of the logistic regression model we illustrate and compare the behavior of the neural networks with the state of the art initializations for these models, Xavier and He initialization. For the results shown below, the *RMSprop* optimizer was used, with a learning rate of 0.0001.

For the toy datasets, the following pipeline was used:

1. Train a neural network model initialized using He and Xavier strategies during 500 epochs and using cross validation. Select the model that minimizes the loss on the validation set with a patience of 50.

	Logistic Regression with random initialization				Proposed method without training			Logistic Regression with proposed initialization			
Datasets	**Accuracy**	**F1-score**	**Training time** (s)	**Training epochs**	**Accuracy**	**F1-score**	Training time (s)	Accuracy	F1-score	Total training time (s)	Training epochs
Wine	0.78 [0.77 - 0.78]	0.73 [0.72 - 0.74]	125.54. [77.10 - 174.10]	3509 [2151 - 4866]	0.48 [0.47 - 0.49]	0.53 [0.52 - 0.54]	4.58e-3 [3.55e-3 - 5.61e-3]	0.78 [0.77 - 0.78]	0.74 [0.73 - 0.74]	88.66 [67.81 - 90.12]	2981 [2455 - 3010]
Iris	0.95 [0.94 - 0.96]	0.95 [0.95 - 0.96]	3.70 [1.42 - 5.99]	1558 [592 - 2523]	0.82 [0.80 - 0.84]	0.82 [0.79 - 0.83]	.97e-4 [6.61e-4 - 7.32e-4]	0.95 [0.94 - 0.97]	0.95 [0.94 - 0.97]	2.90 [1.10 - 3.46]	1182 [409 - 1400]
Madelon	0.53 [0.52 - 0.54]	0.53 [0.52 - 0.54]	68.48 [63.60 - 73.40]	5000	0.57 [0.56 - 0.58]	0.57 [0.56 - 0.58]	3.50e-3 [3.22e-3 - 3.78e-3]	0.54 [0.53 - 0.54]	0.54 [0.53 - 0.54]	57.12 [56.32 - 57.89]	4500
Ozone	0.95 [0.95 - 0.96]	0.94 [0.94 - 0.96]	42.76 [30.62 - 54.91]	4123 [3226 - 5020]	0.76 [0.74 - 0.78]	0.84 [0.83 - 0.85]	1.33e-3 [1.27e-3 - 1.39e-3]	0.95 [0.95 - 0.96]	0.95 [0.95 - 0.96]	37.89 [34.55 - 40.12]	3765 [2965 - 3978]
Breast Cancer	0.93 [0.91 - 0.94]	0.93 [0.91 - 0.94]	11.43 [5.55 - 17.31]	1874 [967 - 2781]	0.91 [0.90 - 0.92]	0.91 [0.90 - 0.92]	9.56e-4 [8.54e-4 - 1.10e-3]	0.93 [0.92 - 0.94]	0.93 [0.92 - 0.94]	9.45 [9.23 - 11.12]	1350 [825 - 1423]
Image Segmentation	0.94 [0.94 - 0.95]	0.94 [0.94 - 0.95]	37.28 [27.13 - 47.43]	3971 [2895 - 5046]	0.63 [0.60 - 0.65]	0.60 [0.57 - 0.63]	2.97e-3 [2.60e-3 - 3.35e-3]	0.94 [0.94 - 0.95]	0.94 [0.94 - 0.95]	28.72 [18.77 - 38.68]	3089 [2008 - 4169]
MNIST	0.90 [0.89 - 0.91]	0.90 [0.89 - 0.91]	251.24 [249.32 - 253.77]	500	0.75 [0.73 - 0.77]	0.74 [0.72 - 0.77]	0.83 [0.82 - 0.84]	0.90 [0.89 - 0.91]	0.90 [0.89 - 0.91]	160.34 [160.20 - 160.49]	500
KDD cup 1999 Data	0.97 [0.96 - 0.98]	0.97 [0.96 - 0.97]	855.01 [781.80 - 928.22]	499 [411 - 488]	0.81 [0.78 - 0.84]	0.86 [0.84 - 0.88]	0.14 [0.13 - 0.15]	0.97 [0.96 - 0.98]	0.97 [0.96 - 0.98]	542.13 [537.89 - 555.93]	500

TABLE 4.6: Benchmark results of the proposed initialization parameters without training and after convergence of a logistic regression model with the Xavier, He and the proposed initialization for the testing dataset. This models where trained for 500 epochs with a learning rate of 0.01 and a validation-loss stopping criteria with a patience of 50. The results are shown with a 95% confidence interval which was obtained using bootstrap for 10 iterations.

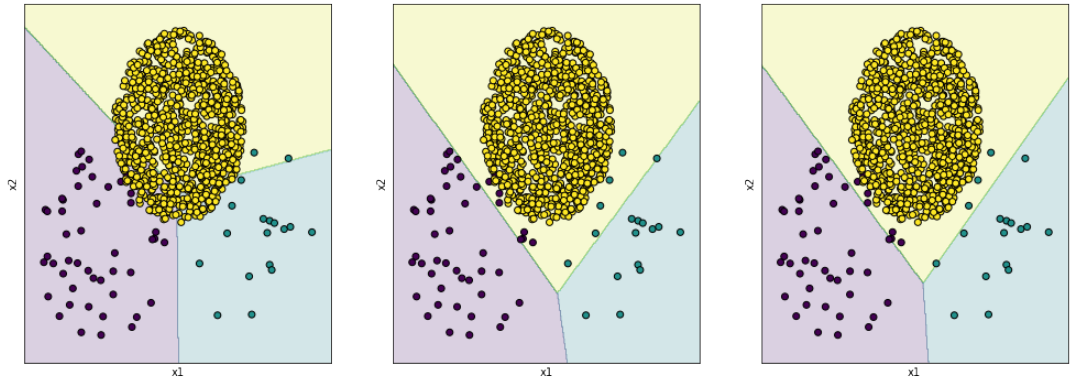

FIGURE 4.12: Illustration of the classification regions with a multi-class dataset for an unbalance of 95% in the purple class and 98% in the green class. The proposed initialization without training (left), the convergence after 100 epochs (1.78s) training with random initialization (center) and the convergence after 100 epochs (1.73s) training with proposed initialization (right). The initialization time is already included.

2. Evaluate on the test set.
3. Calculate the proposed initialization parameters $\boldsymbol{\theta_0}$ using the training and validation sets.
4. Repeat steps 2 and 3 of the process with a neural network model initialized using the proposed algorithm.

To train the models until the epoch that minimizes the validation loss with a given patience, we use a stopping criterion already implemented in *Keras*. As in the case of logistic regression, we create a binary and a multi-class datasets. These datasets are non-linearly separable. We test the balanced and unbalanced cases in order to demonstrate the robustness of the proposed initialization strategy.

4.3.1 Binary Toy Dataset

The first toy dataset which represents a non-linear binary classification problem consists of a circular and a disc overlapped distributions. These distributions form a donut, and they are shown in figure 4.13. In this case, we are going to divide the test into two parts. The first test uses a model with a single hidden layer neural networks that have 10 neurons, 5 for each class. The second test will consider two hidden layers in its architecture, each one defined in the same way as the first test, 5 neurons for each class.

One Hidden Layer Architecture

The performance on the validation set of the model with one hidden layer architecture for the different initializations is shown in figure 4.14. The model with Xavier initialization stop its training at epoch 393 with a training time of approximately 23.32 seconds using a patience of 50. While the models that use He and the proposed initialization took only 291 epochs (17.72s) and 155 epochs (10.12s) respectively. Since the proposed initialization takes approximately 0.014 seconds, the total

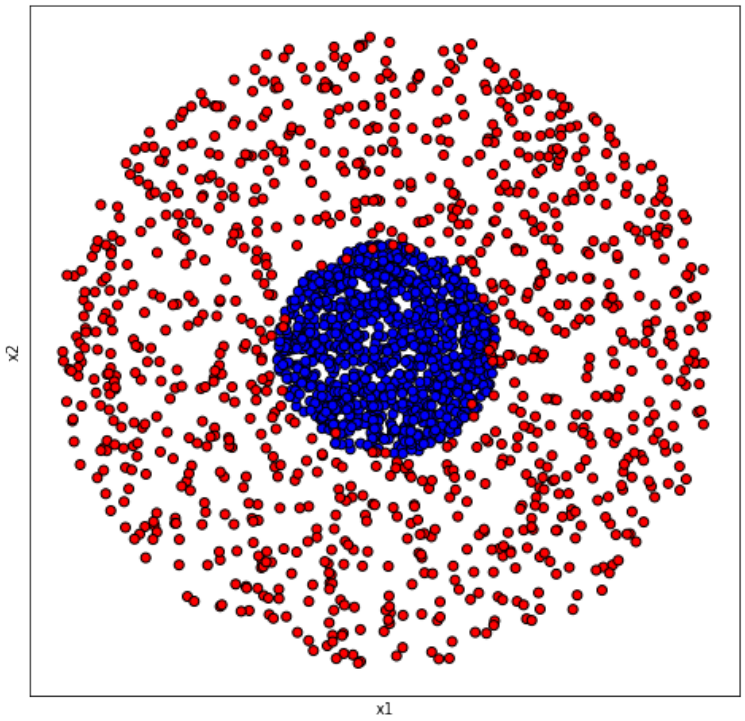

FIGURE 4.13: Generated samples for the binary classification dataset using a circular and a disc distribution.

training time for this model was 10.13s. The results of all the tested models for the test set is presented on table 4.7. The classification regions, using the complete dataset for training, are shown in figure 4.15.

We also created an unbalance dataset using the same distribution shape, but considering an imbalance of 95% in the blue class. The performance of these models for the validation set are shown in figure 4.16 and the results for the test set are found in table 4.8. This time, the model using Xavier initialization stopped its training at epoch 369 with an approximately training time of 21.7 seconds, the one using He initialization took 279 epochs (17.6s) and the model using the proposed initialization took 231 trained epochs (15.2s). The classification regions using the complete dataset for training are found in figure 4.17.

Two Hidden Layer Architecture

We tested an architecture containing two hidden layers, each one conformed by 10 neurons distributed in 5 per each class. The validation set performance for the three initializations, Xavier, He and proposed is presented in figure 4.18. The test set results are shown in table 4.9. In this experiment, the model using Xavier initialization converges after 291 training epochs (19.01s), while the ones that were initialized using the He and the proposed strategies took 151 epochs (10.19s) and 145 epochs (10.02s). The classification regions for these models (with two hidden layer architectures) are found in figure 4.19.

We also performed tests for the unbalanced case, 95% unbalance in the blue class. The results for the validation set are found in figure 4.20, and the performance for

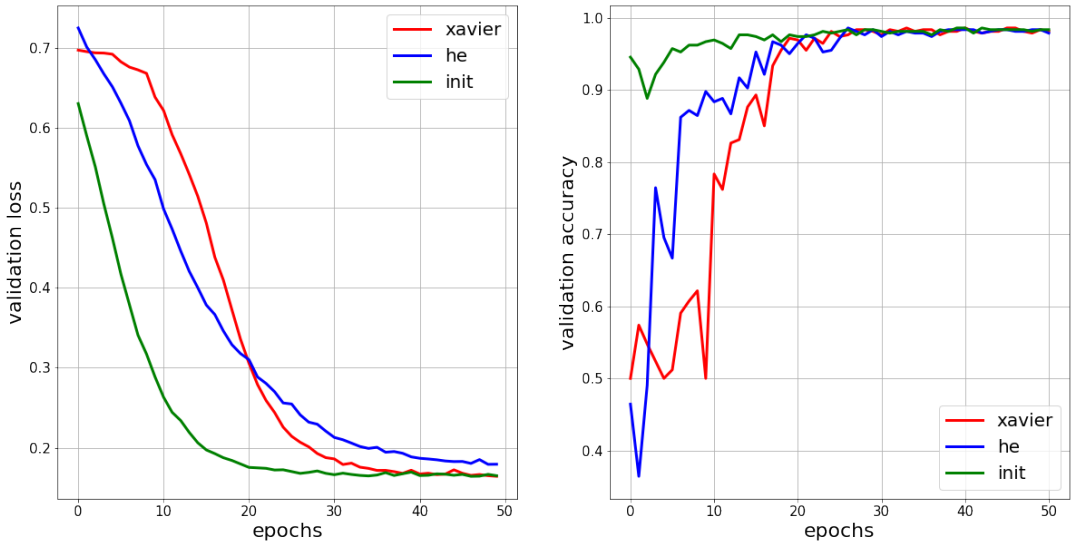

FIGURE 4.14: Xavier, He and proposed initialization performance, loss on the left and accuracy on the right, for the validation set using a class balanced dataset. These models were trained using a learning rate of 0.01.

	Neural network with Normal Xavier initialization		**Neural network with Normal He initialization**	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Blue	1	0.95	1	0.95
Red	0.95	1	0.96	1
Total	0.97	0.98	0.98	0.98
	Proposed initialization without training		**Neural network with proposed initialization**	
Blue	0.94	0.98	1	0.98
Red	0.98	0.94	0.95	0.96
Total	0.96	0.96	0.98	0.97

TABLE 4.7: Results with a class balanced test set performance for the proposed initialization without training, after convergence of a neural network model with Xavier, He and the proposed initialization. These models were trained using a learning rate of 0.001 and a patience of 50.

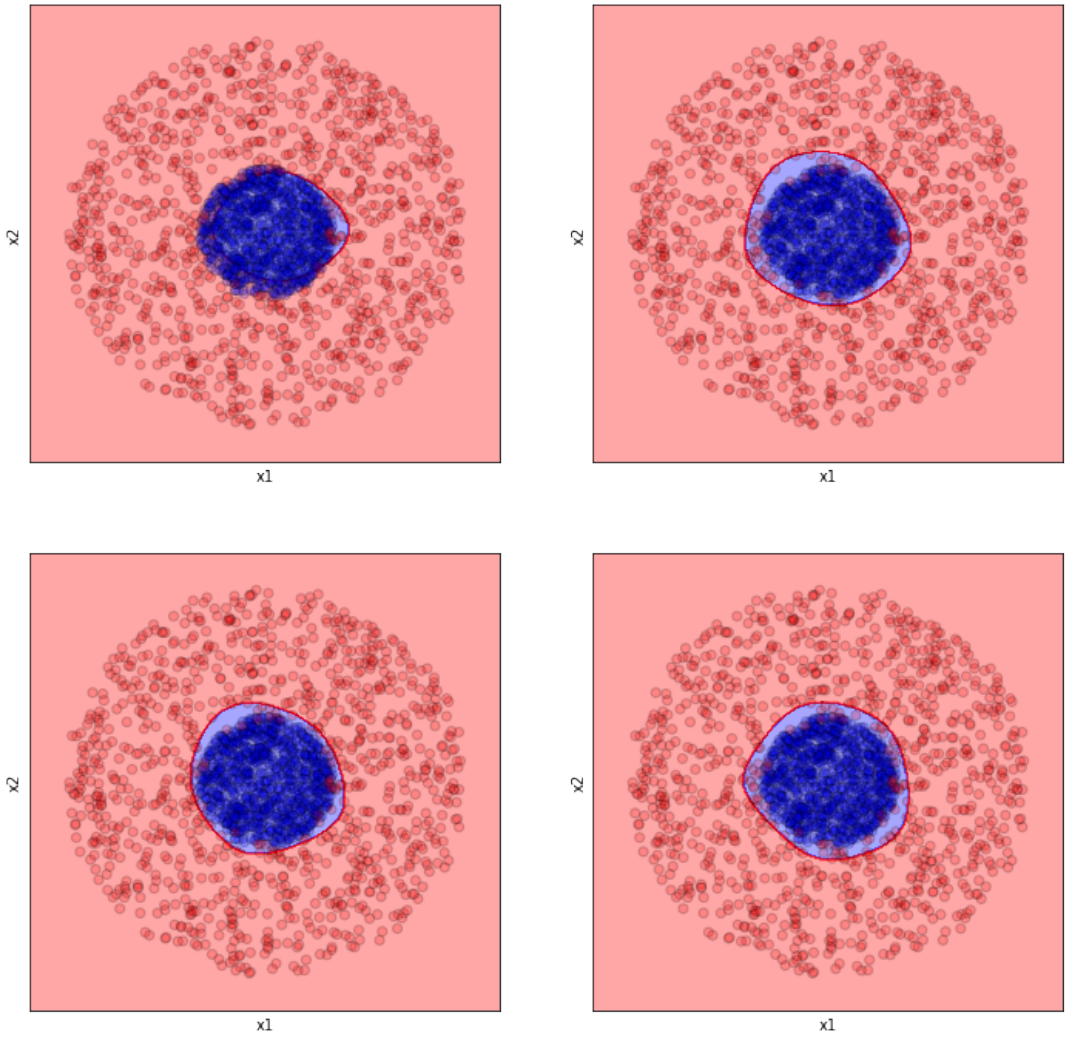

FIGURE 4.15: Illustration of the classification regions with a class balanced dataset for the proposed initialization (without training) at the top left; the convergence after 15 epochs (2.22s) training with Xavier initialization at top right; the convergence after 15 epochs (1.19s) training with He initialization at bottom left, and the convergence after 15 epochs (1.8s) training with proposed initialization at the bottom right. The initialization time is already included.

	Neural network with Normal Xavier initialization		Neural network with Normal He initialization	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Blue	0	0	0	0
Red	1	0.95	1	0.95
Total	0.95	0.91	0.95	0.91
	Proposed initialization without training		**Neural network with proposed initialization**	
Blue	0.96	0.63	0.25	1
Red	0.97	0.99	1	0.96
Total	0.97	0.97	0.97	0.97

TABLE 4.8: Results with a class unbalanced dataset. Test performance for the proposed initialization without training and after convergence of a neural network model using Xavier, He and the proposed initialization. These models were trained using a learning rate of 0.001 and a patience of 50.

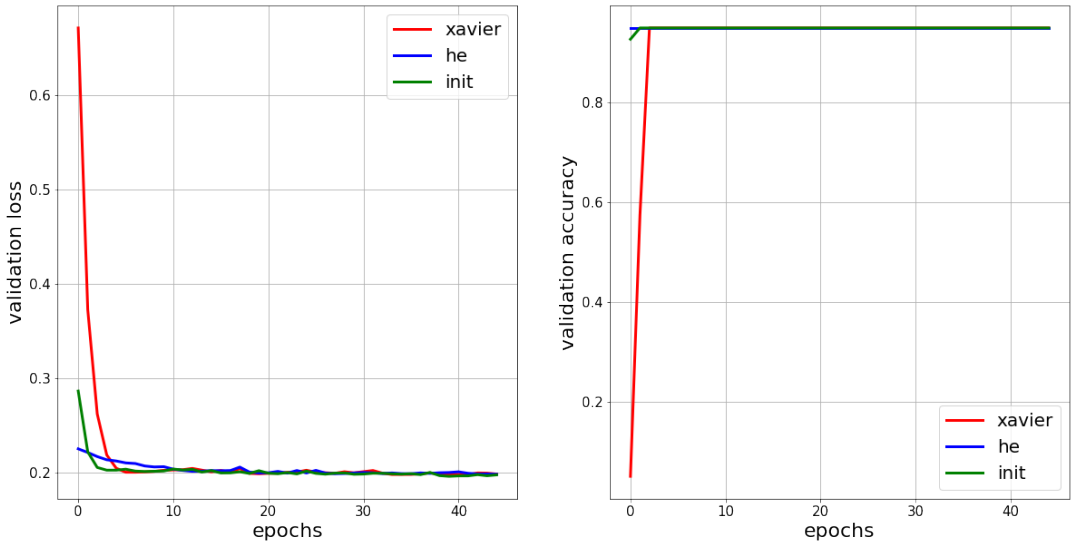

FIGURE 4.16: Xavier, He and proposed initialization performance, loss on the left and accuracy on the right, for the validation set with a 95% unbalance in the blue class. This models where trained using a learning rate of 0.01.

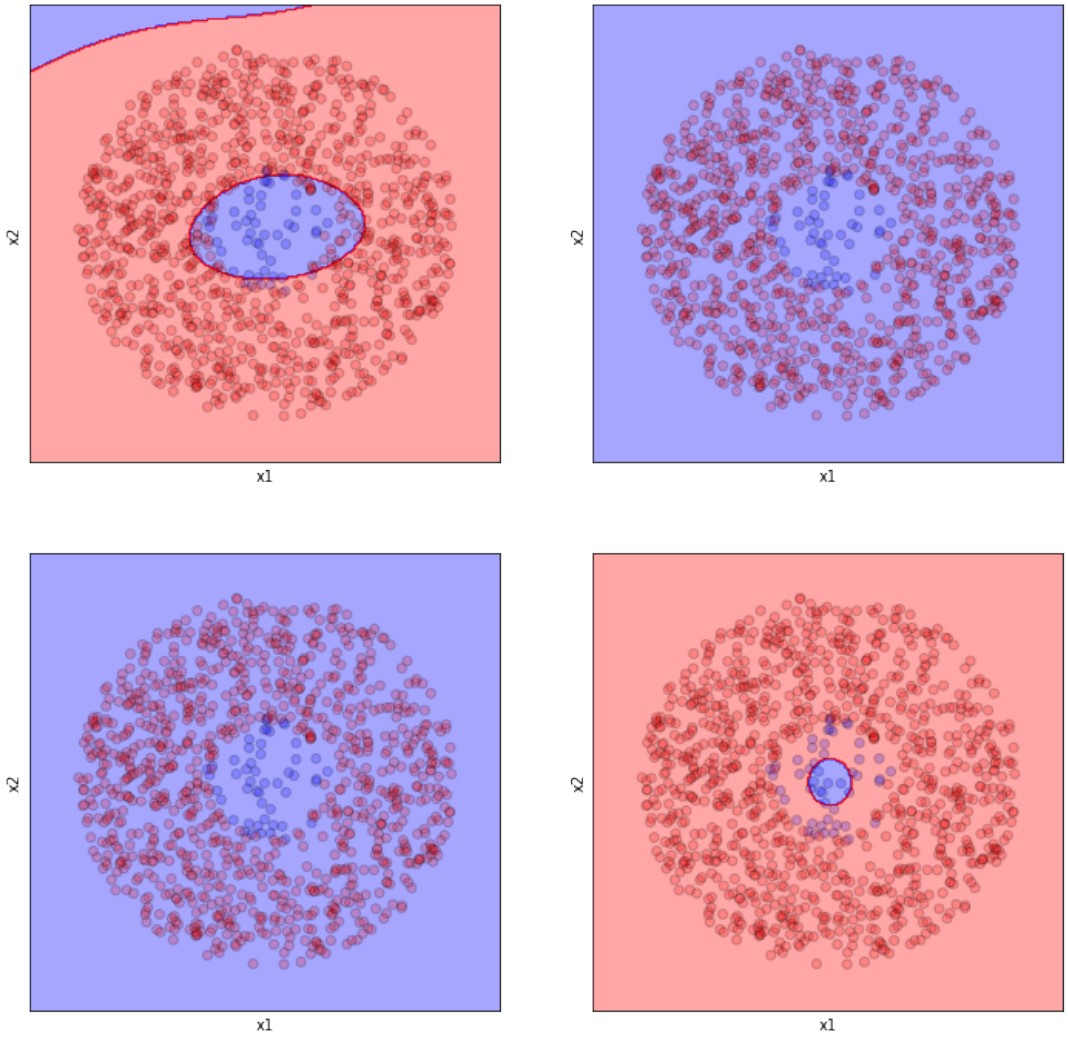

FIGURE 4.17: Illustration of the classification regions with 95% unbalance in the blue class region for the proposed initialization (without training) at the top left, the convergence after 15 epochs (2.22s) training with Xavier initialization at top right, the convergence after 15 epochs (1.19s) training with He initialization at bottom left, and the convergence after 15 epochs (1.8s) training using the proposed initialization at the bottom right. The initialization time is already included.

Note that no balance strategy was used to train the networks.

	Neural network with Normal Xavier initialization		Neural network with Normal He initialization	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Blue	1	0.95	1	0.95
Red	0.95	1	0.96	1
Total	0.97	0.98	0.98	0.98
	Proposed initialization without training		**Neural network with proposed initialization**	
Blue	0.97	0.96	1	0.98
Red	0.95	0.97	0.95	0.96
Total	0.96	0.96	0.98	0.97

TABLE 4.9: Results with a balanced dataset. Test performances for the proposed initialization parameters without training and after convergence of a neural network model using Xavier, He and the proposed initialization. These models have two hidden layers in its architecture and were trained using a learning rate of 0.001 and a patience of 50.

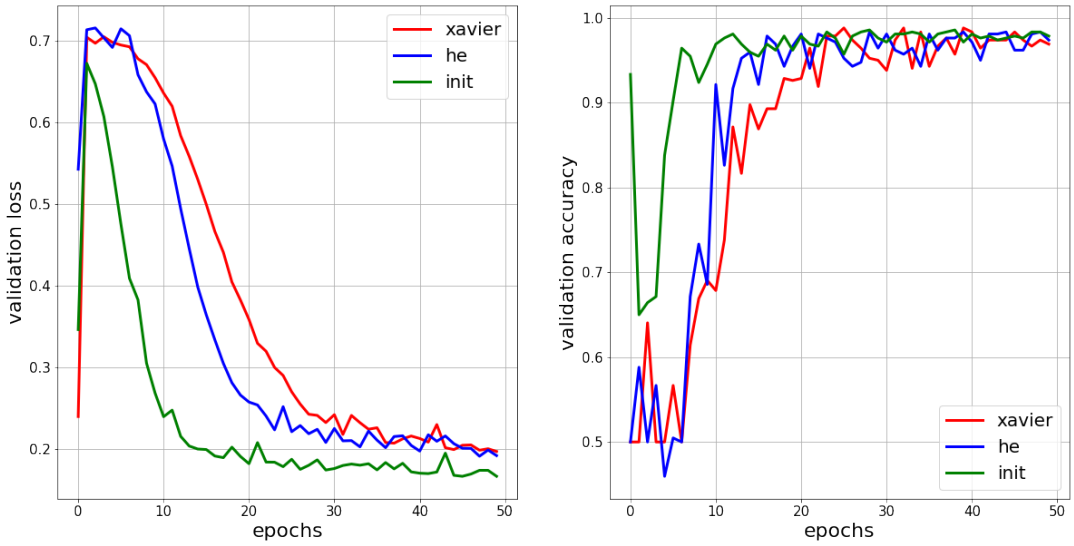

FIGURE 4.18: Xavier, He and proposed initialization performance, loss on the left and accuracy on the right. These models have two hidden layer architecture and were trained using a learning rate of 0.01.

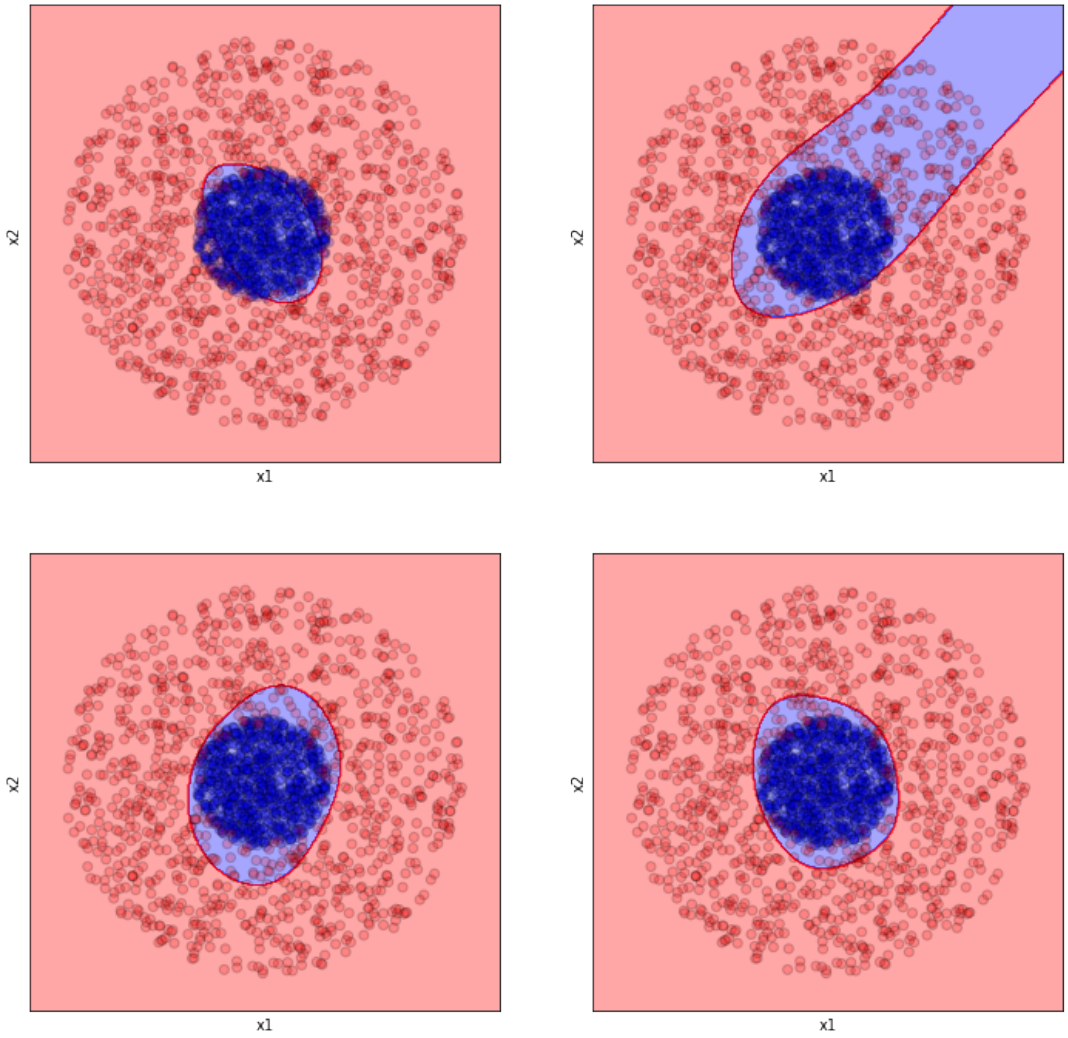

FIGURE 4.19: Illustration of the classification regions for the two hidden layer architecture using a balanced dataset regions for the proposed initialization (without training) at the top left; the convergence after 60 epochs (5.88s) training using Xavier's initialization at top right; the convergence after 15 epochs (5.77s) training using He's initialization at bottom left and the convergence after 15 epochs (5.69s) training using the proposed initialization at the bottom right. The initialization time is already included.

	Neural network with Normal Xavier initialization		Neural network with Normal He initialization	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Blue	0	0	0	0
Red	1	0.95	1	0.95
Total	0.95	0.90	0.95	0.91
	Proposed initialization without training		**Neural network with proposed initialization**	
Blue	0.87	0.77	0	0
Red	0.99	0.99	1	0.95
Total	0.98	0.98	0.95	0.91

TABLE 4.10: Results with an unbalance of 95% in the blue class for the two layer architecture. Test performance for the proposed initialization without training and after convergence of a neural network model using Xavier, He and the proposed initialization. These models have two hidden layers in its architecture and were trained for 500 epochs using a learning rate of 0.001 and with a patience of 50.

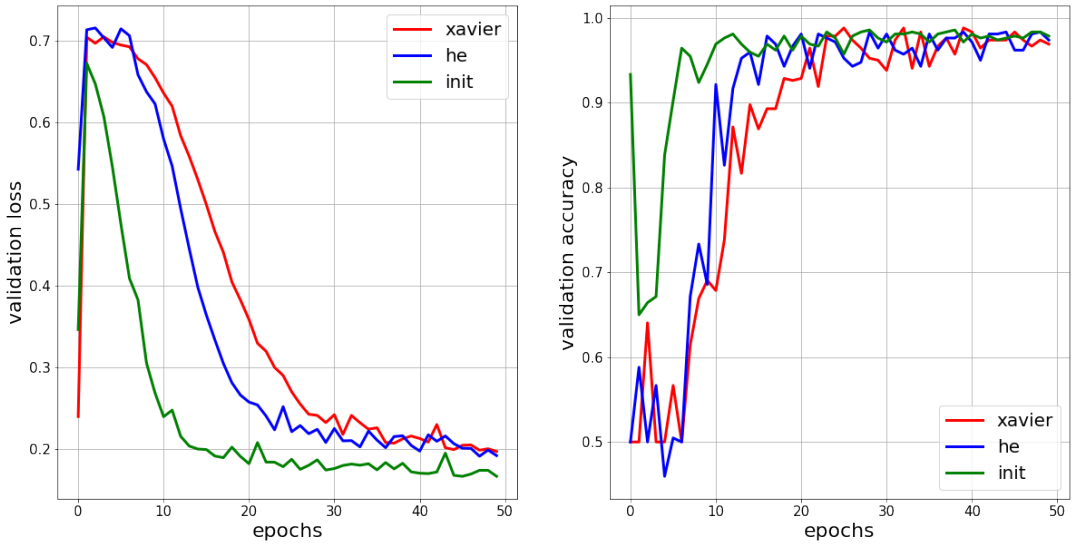

FIGURE 4.20: Xavier, He and proposed initialization performance for the validation set, loss on the left and accuracy on the right, when there is an unbalance of 95% in the blue class. These models have two hidden layer architecture and where trained using a learning rate of 0.001.

	Neural network with Normal Xavier initialization (178 epochs - 13.51 seconds)		Neural network with Normal He initialization (304 epochs - 23.17 seconds)	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Blue	0.98	0.96	1	0.87
Red	0.85	0.97	0.85	0.90
Green	0.99	0.90	0.9	1
Total	0.94	0.95	0.92	0.92
	Proposed initialization without training (0.041 seconds)		**Neural network with proposed initialization (315 epochs - 23.59 seconds)**	
Blue	0.94	0.95	1	0.93
Red	0.90	0.89	0.92	0.95
Green	0.94	0.95	0.95	1
Total	0.93	0.93	0.96	0.96

TABLE 4.11: Results for the test set in a multi-class balance dataset using the proposed initialization without training, and after convergence of a neural network model using the Xavier, He and the proposed initialization. These models have two hidden layers in its architecture and were trained for 500 epochs using a learning rate of 0.01 and a patience of 50.

the test set is observed in the table 4.10.

This time, the model using Xavier's initialization trained during the maximum number of epochs (500) which represents approximately 26.7 seconds. The model initialized using He's strategy trained for 409 (22.0s) and the model initialized with the proposed strategy took 156 epochs (9.23s). The classification regions generated by these models are shown in the figure 4.21.

4.3.2 Multi-class Toy Datasets

The second toy dataset represents a multi-class non-linear classification problem. It consists of a circular distribution and 2 overlapped discs. This dataset can be seen in figure 4.22. In this case, we are going to test the dataset with a model using two hidden layers, and initialized by Xavier, He and the proposed strategies.

Figure 4.23 shows the performance in the validation set for a neural network composed by two hidden layers, 15 neurons each layer divided in 5 neurons per class, using as initialization Xavier, He and the proposed strategies. Results for the test set are displayed in table 4.11.

The model using Xavier's initialization converged in 178 training epochs (13.51s), while the model using the He and the proposed initialization took 304 epochs (23.17s) and 315 epochs (23.59s) respectively. Since the initialization required 0.041 seconds, the total time for the proposed strategy was 23.63 seconds. The classification regions are shown in figure 4.24.

Two other simulations were performed using the same architecture and the same multi-class toy dataset. The first simulation considers an unbalance of 90% in the blue class and the second one considers 98% in the blue class and 95% in the red class. This, in order to evaluate the robustness of the initialization strategy when

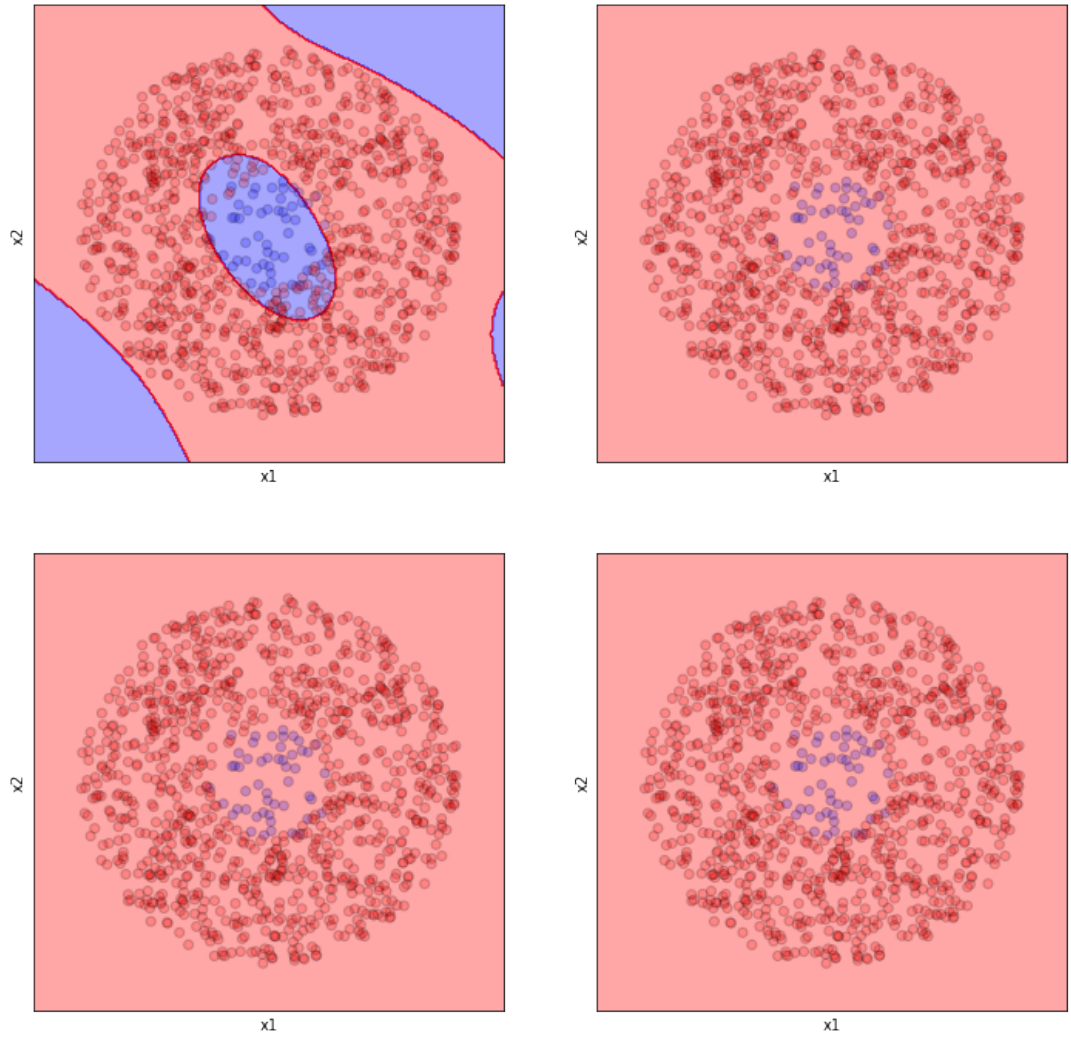

FIGURE 4.21: Illustration of the classification regions for the two layer architecture when there is a 95% unbalance in the blue class for the proposed initialization (without training) at the top left; the convergence after 50 epochs (3.97s) training with Xavier initialization at top right; the convergence after 50 epochs (3.82s) training with He initialization at bottom left, and the convergence after 50 epochs (3.77s) training with proposed initialization at the bottom right, the initialization time is already included. Note that no strategy for training unbalanced datasets was used.

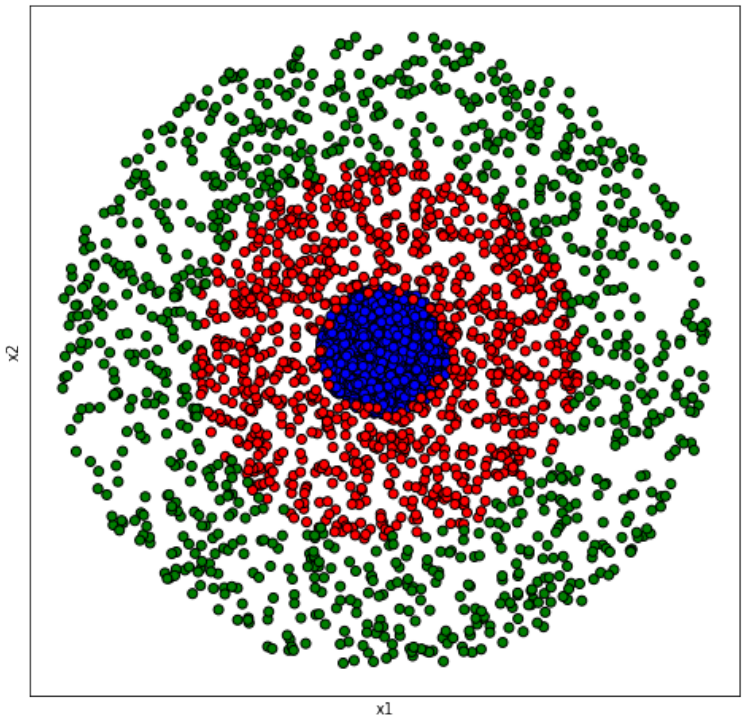

FIGURE 4.22: Generated dataset for multi class classification using a circular distribution and 2 discs distributions.

more classes are unbalanced.

For this first simulation, the performance for the validation set are found in figure 4.25. Results for the test set are shown in table 4.12. The classification regions are found in figure 4.26.

For the second simulation we considered unbalance in two (blue and red) of the three classes. Results for the validation set are found in figure 4.27. Results for the test set are shown in table 4.13 and figure 4.28 shows the classification regions obtained after training.

4.3.3 Benchmark with Real Datasets

Here, we show the results of the neural network initialized using Xavier, He and proposed strategy, for the datasets described in the section 4.1. This evaluation computes confidence intervals from the test set after applying bootstrap, 10 iterations, to generate 95% confidence intervals. Results are displayed in table 4.14.

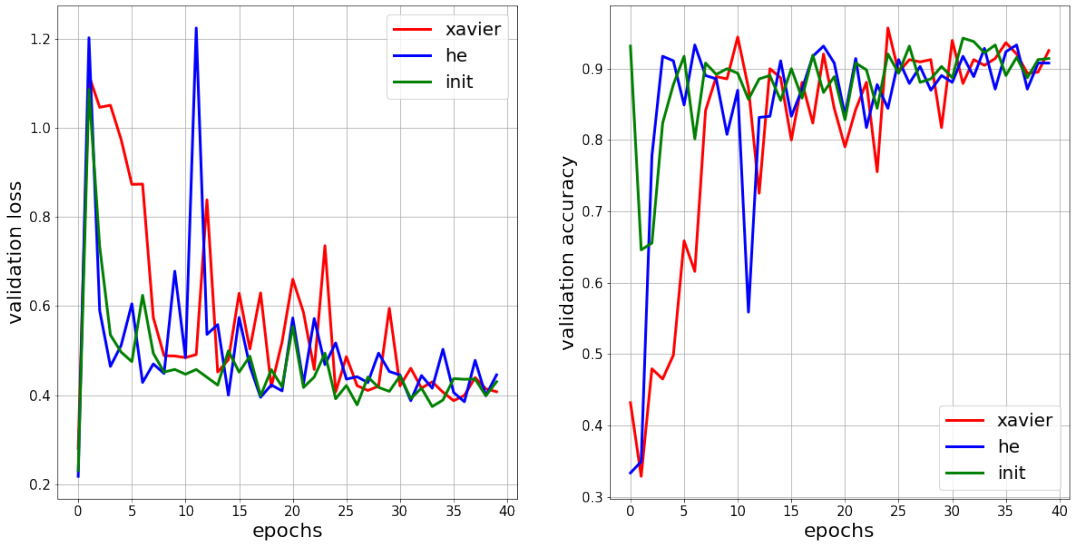

FIGURE 4.23: Xavier, He and proposed initialization performance for the validation set, loss on the left and accuracy on the right. These models have two hidden layers and were trained using a learning rate of 0.01.

	Neural network with Normal Xavier initialization (487 epochs - 29.83 seconds)		**Neural network with Normal He initialization (434 epochs - 26.75 seconds)**	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Blue	0	0	0	0
Red	1	0.90	0.92	0.95
Green	0.94	1	1	0.92
Total	0.94	0.93	0.93	0.91
	Proposed initialization without training (0.038 seconds)		**Neural network with proposed initialization (429 epochs - 27.27 seconds)**	
Blue	0.66	0.64	0	0
Red	0.86	0.82	1	0.88
Green	1	0.95	0.92	1
Total	0.94	0.90	0.94	0.92

TABLE 4.12: Results for the test set using an unbalance of 90% in the blue class. Performance for the proposed initialization parameters without training and after convergence of a neural network model using the Xavier, He and the proposed initialization. This models have two hidden layers in its architecture and where trained for 500 epochs using a learning rate of 0.01 and a patience of 50.

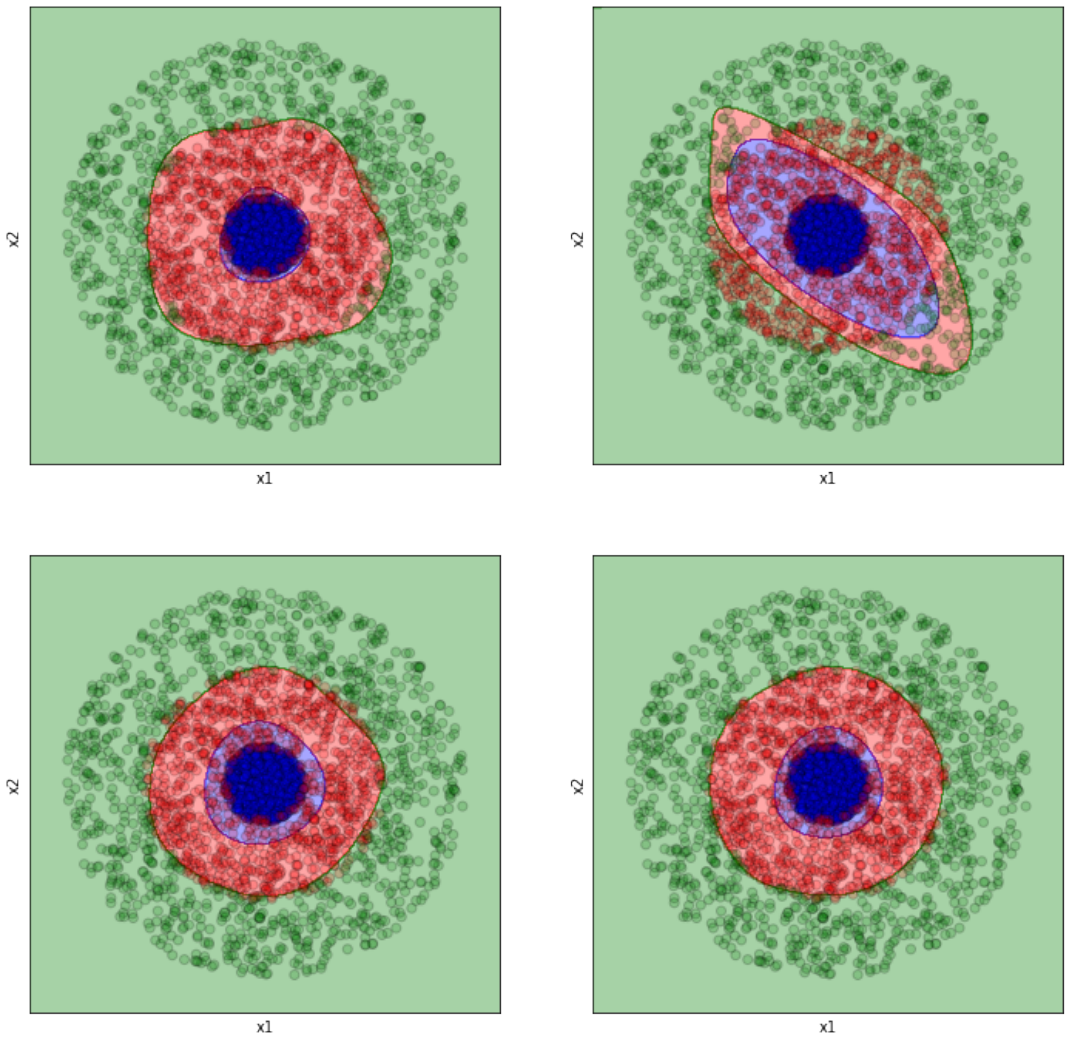

FIGURE 4.24: Illustration of the classification regions with a multi-class balanced dataset for the proposed initialization, without training, at the top left; the convergence after 60 epochs (6.85s) training using Xavier initialization at top right; the convergence after 60 epochs (6.61s) training using He initialization at bottom left and the convergence after 60 epochs (6.63s) training using the proposed initialization at the bottom right. The initialization time is already included. These models have two hidden layer architecture each one of 15 neurons, 5 neurons per class, and were trained using a learning rate of 0.01.

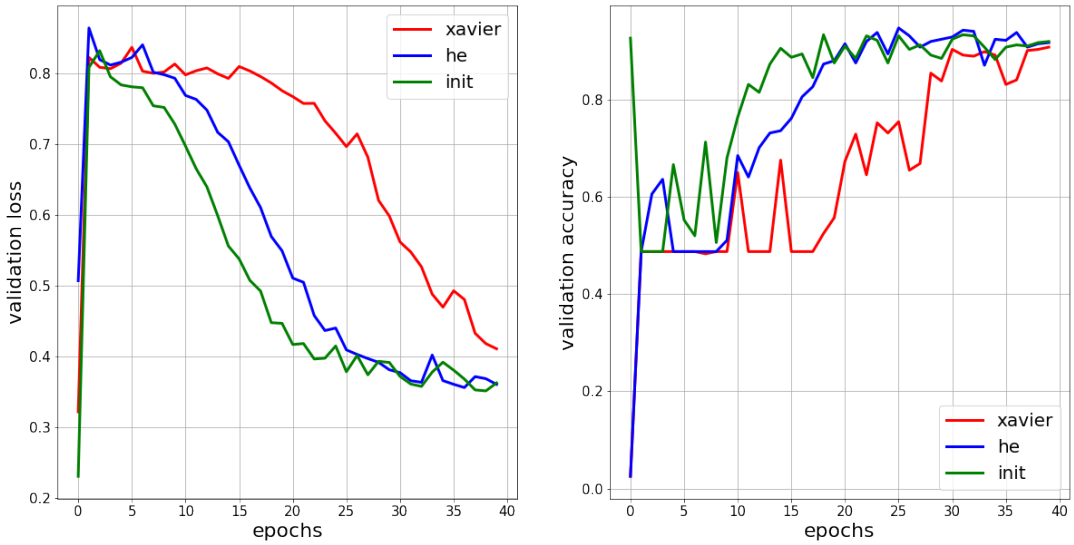

FIGURE 4.25: Xavier, He and proposed initialization performance, loss on the left and accuracy on the right. The dataset has a unbalance of the 90% in the blue class. The models have two hidden layer architecture and were trained using a learning rate of 0.01.

	Neural network with Normal Xavier initialization (487 epochs - 29.83 seconds)		**Neural network with Normal He initialization (434 epochs - 26.75 seconds)**	
Class	**Accuracy**	**Precision**	**Accuracy**	**Precision**
Blue	0	0	0	0
Red	0	0	0	0
Green	0.94	1	1	0.92
Total	0.93	0.92	0.93	0.90
	Proposed initialization without training (0.038 seconds)		**Neural network with proposed initialization (429 epochs - 27.27 seconds)**	
Blue	0.66	0.64	0	0
Red	0.90	0.88	0	0
Green	1	0.95	0.94	1
Total	0.95	0.91	0.93	0.92

TABLE 4.13: Results for the test set using an unbalance of 98% in the blue class and 95% in the red class performance for the proposed initialization without training and after convergence of a neural network model using Xavier, He and the proposed initialization. These models have two hidden layers in its architecture and were trained for 500 epochs using a learning rate of 0.01 and a patience of 50.

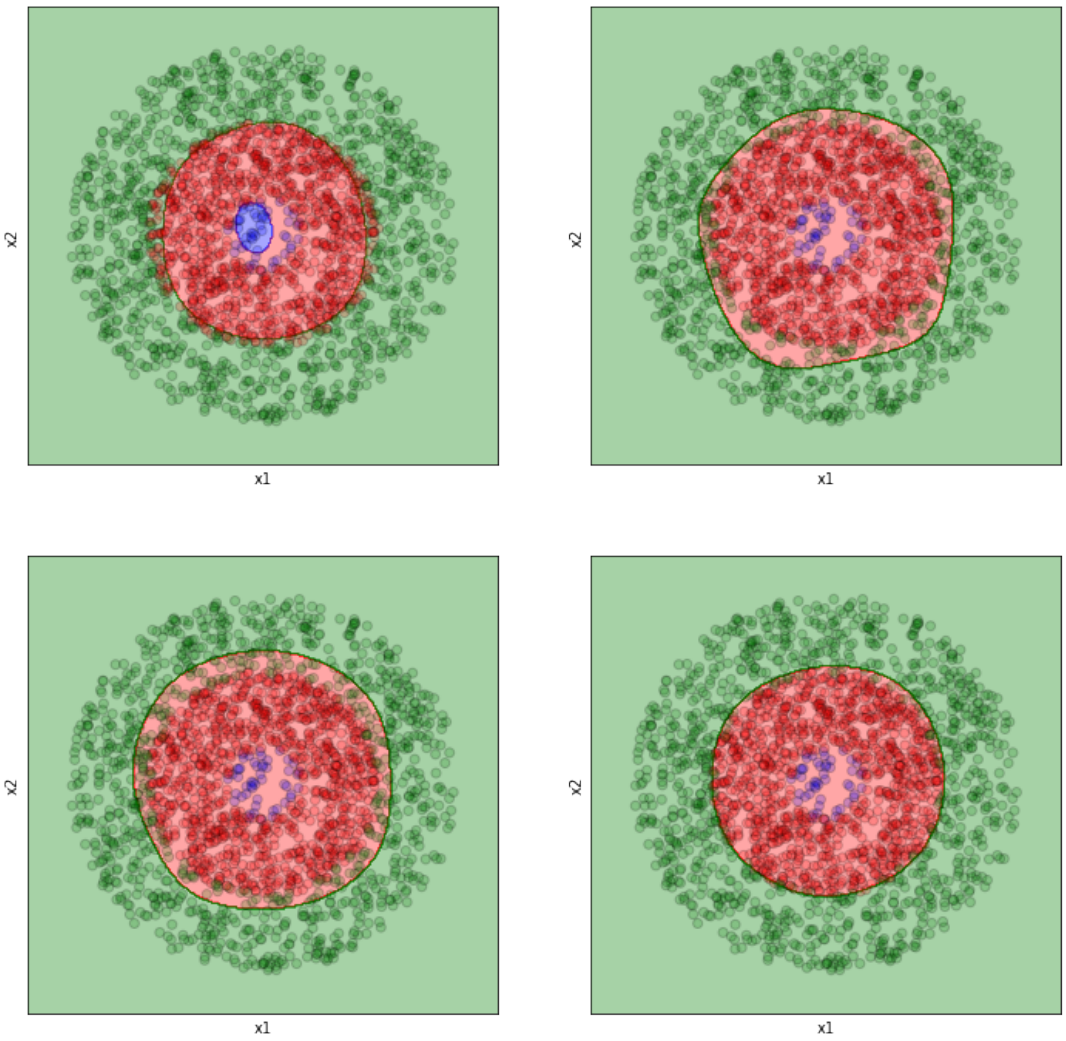

FIGURE 4.26: Illustration of the classification regions with a multi-class dataset with an unbalance of 90% in the blue class for the proposed initialization (without training) at the top left; the convergence after 200 epochs (16.79s) training using Xavier initialization at top right; the convergence after 200 epochs (16.92s) training using He's initialization at bottom left and the convergence after 200 epochs (16.92s) training using the proposed initialization at the bottom right. The initialization time is already included. These models have two hidden layer architecture each one of 15 neurons, 5 neurons per class, and were trained using a learning rate of 0.01.

	Neural Network with Xavier initialization			Neural Network with He initialization			Proposed method for neural networks without training			Neural Network with Proposed initialization		
Datasets	**Accuracy**	**F1-score**	**Training time (s)**	**Accuracy**	**F1-score**	**Training time (s)**	**Accuracy**	**F1-score**	**Training time (s)**	**Accuracy**	**F1-score**	**Total training time (s)**
Wine	0.78 [0.77 - 0.79]	0.74 [0.73 - 0.75]	28.31 [28.15 - 28.45]	0.78 [0.77 - 0.79]	0.74 [0.73 - 0.75]	28.44 [28.27 - 28.60]	0.55 [0.52 - 0.57]	0.60 [0.57 - 0.62]	3.53e-2 [2.92e-2 - 4.13e-2]	0.79 [0.78 - 0.79]	0.75 [0.75 - 0.76]	28.47 [28.25 - 28.70]
Iris	0.62 [0.51 - 0.73]	0.56 [0.43 - 0.69]	1.09 [1.06 - 1.13]	0.56 [0.46 - 0.66]	0.47 [0.34 - 0.59]	1.09 [1.05 - 1.13]	0.94 [0.93 - 0.97]	0.95 [0.93 - 0.97]	3.53e-2 [2.92e-2 - 4.13e-2]	0.84 [0.75 - 0.90]	0.80 [0.72 - 0.90]	1.06 [1.02 - 1.09]
Madelon	0.54 [0.53 - 0.55]	0.54 [0.53 - 0.55]	11.95 [11.87 - 12.02]	0.54 [0.53 - 0.55]	0.54 [0.53 - 0.55]	12.05 [11.93 - 12.16]	0.53 [0.51 - 0.53]	0.52 [0.51 - 0.53]	2.43e-2 [2.26e-2 - 2.59e-2]	0.57 [0.55 - 0.57]	0.57 [0.55 - 0.57]	12.84 [11.98 - 12.19]
Ozone	0.96 [0.96 - 0.97]	0.95 [0.94 - 0.96	8.50 [8.42 - 8.58]	0.96 [0.96 - 0.97]	0.95 [0.94 - 0.96	8.44 [8.38 - 8.51]	0.81 [0.75 - 0.87]	0.87 [0.83 - 0.91]	1.44e-2 [9.14e-3 - 1.97e-2]	0.96 [0.96 - 0.97]	0.95 [0.94 - 0.96	8.50 [8.44 - 8.56]
Breast Cancer	0.92 [0.91 - 0.93]	0.92 [0.91 - 0.93]	2.89 [2.85 - 2.93]	0.90 [0.89 - 0.92]	0.90 [0.89 - 0.92]	2.89 [2.84 - 2.95]	0.89 [0.87 - 0.91]	0.89 [0.87 - 0.91]	9.40e-3 [4.32e-3 - 1.44e-2]	0.92 [0.91 - 0.93]	0.92 [0.91 - 0.93]	2.90 [2.86 - 2.95]
Image Segmentation	0.89 [0.87 - 0.89]	0.89 [0.87 - 0.89]	10.75 [10.69 - 10.81]	0.89 [0.87 - 0.89]	0.89 [0.87 - 0.89]	10.81 [10.73 - 10.88]	0.82 [0.80 - 0.84]	0.82 [0.80 - 0.84]	6.76e-2 [6.25e-2 - 7.27e-2]	0.94 [0.94 - 0.95]	0.94 [0.94 - 0.95]	10.84 [10.79 - 10.89]
Mnist	0.95 [0.94 - 0.95]	0.95 [0.94 - 0.95]	412.05 [408.75 - 415.35]	0.95 [0.94 - 0.95]	0.95 [0.94 - 0.95]	407.58 [405.16 - 410.01]	0.73 [0.72 - 0.73]	0.71 [0.71 - 0.72]	7.89 [7.73 - 8.03]	0.97 [0.96 - 0.97]	0.97 [0.96 - 0.97]	410.51 [407.17 - 413.85]
KDD Cup 1999 Data	0.97 [0.96 - 0.98]	0.97 [0.96 - 0.97]	1118.83 [1111.17 - 1126.49]	0.97 [0.96 - 0.98]	0.97 [0.96 - 0.98]	1115.67 [1109.64 - 1121.70]	0.91 [0.89 - 0.92]	0.92 [0.92 - 0.94]	1.58 [1.57 - 1.59]	0.97 [0.96 - 0.98]	0.97 [0.96 - 0.98]	1120.19 [1114.67 - 1125.72]

TABLE 4.14: Benchmark test set results for the proposed initialization parameters without training, and after convergence of a neural network model with the Xavier, He and the proposed initialization. These models have only one hidden layer in its architecture, and were trained for 100 fix epochs with a learning rate of 0.01. The results are shown with a 95% confidence interval which was obtained using bootstrap for 10 iterations.

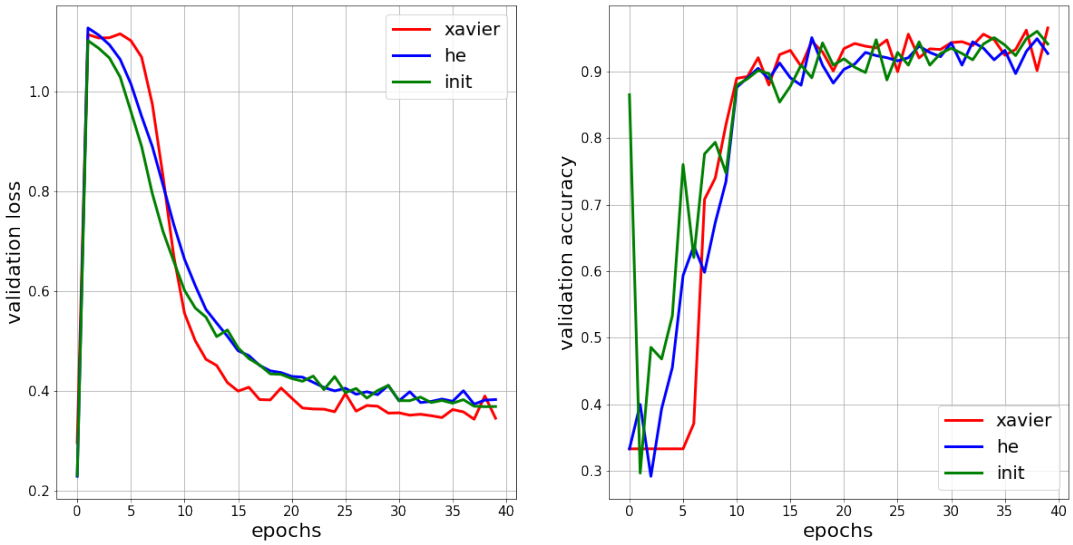

FIGURE 4.27: Xavier, He and proposed initialization performance for validation set, loss on the left and accuracy on the right. The dataset has an unbalance of 98% in the blue class and 95% in the red class. The models have two hidden layer architecture and were trained using a learning rate of 0.01. Note that no strategy to compensate for unbalance classes was used in the training.

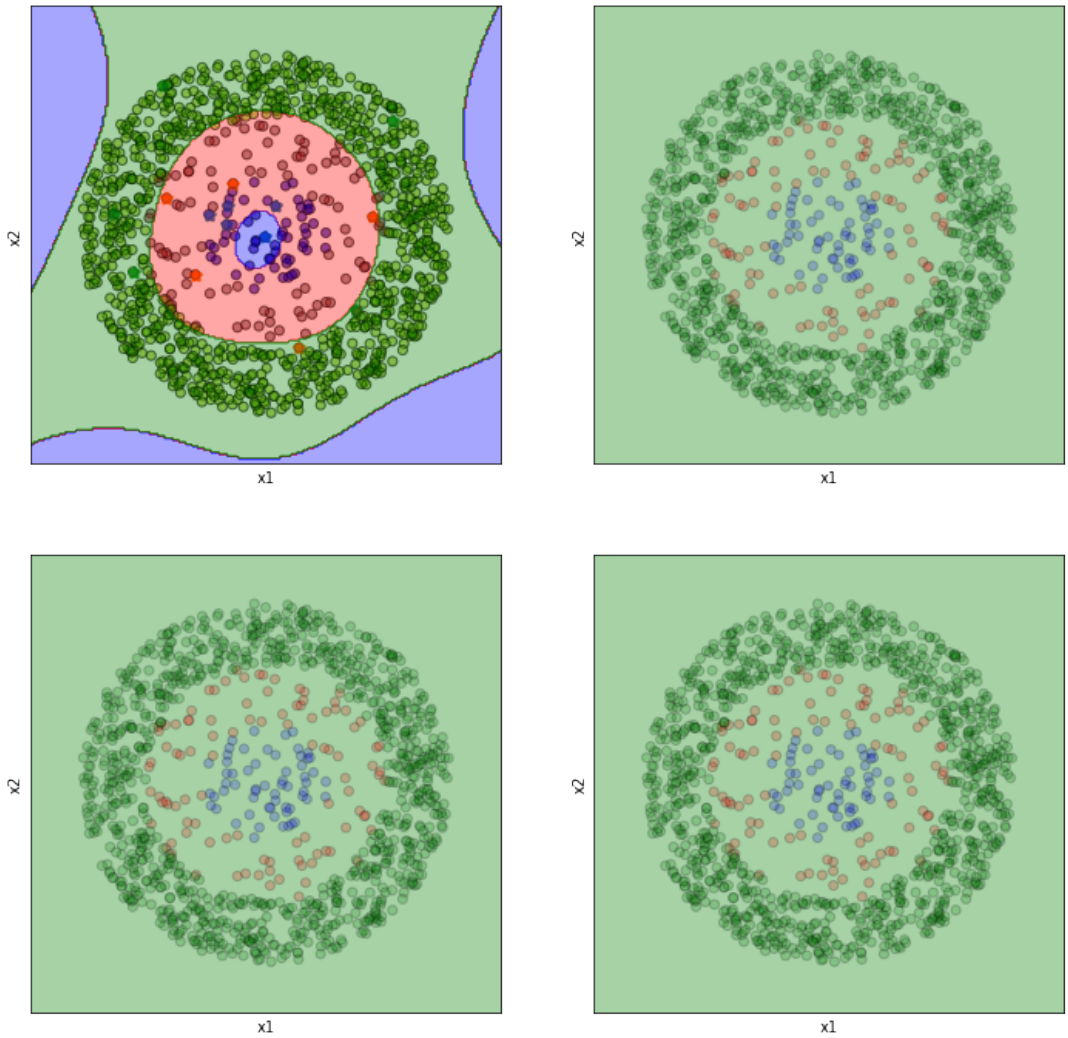

FIGURE 4.28: Illustration of the classification regions with a multi-class dataset with an unbalance of 98% in the blue class and 95% in the red class. Regions for the proposed initialization without training at the top left; the convergence after 200 epochs (16.79s) training using Xavier's initialization at top right; the convergence after 200 epochs (16.92s) training using He's initialization at bottom left, and the convergence after 200 epochs (16.92s) training using the proposed initialization at the bottom right. The initialization time is already included. These models have two hidden layer architecture each one of 15 neurons, 5 neurons per class, and were trained using a learning rate of 0.01. Note that no strategy to compensate for unbalance classes was used in the training.

Chapter 5

Discussion

In this chapter we discuss the results presented in chapter 4, and we contrast our strategy with other initialization algorithms found in the literature.

5.1 Logistic and Multinomial Regression Results

The first analysis will be based on the results form the first dataset shown in figure 4.1. In this case, the confidence intervals for the validation set of the model initialized with the proposed strategy are narrow, which illustrates, for this example, the robustness of the initialization. As can be seen in figure 4.2, for the validation set, this model starts, without training any epoch, with a loss and an accuracy very close to the optimal one, almost equal to that obtained after the learning algorithm converges.

Table 4.1 shows that a similar behavior is evident for the test set. In this case, the difference between the convergence for the randomly initialized model and the one using the proposed strategy is approximately 200 epochs (3 seconds difference in training).

The classification regions in figure 4.3 show that the decision boundary obtained from the untrained initialization strategy is very close to the optimal. With only 25 training epochs, the model initialized by our proposed algorithm achieves a border that manages to separate the classes considerably well. This does not happen with the randomly initialized model,in which the border, in this simulation, sacrifices the purple class considerably.

Regarding the second test, which considers a 90% imbalance for the purple class, if we analyze the results presented in table 4.2, we realize that as the models are trained, they tend to penalize the minority class. This happens because, in this case, the *binary cross entropy* is being used as a cost function, which is sensitive to imbalance, and no balancing strategy is being considered.

What is interesting is that the proposed initialization is not affected by such imbalance, obtaining, without training, an accuracy for the purple class even higher than that from the logistic regression after convergence. This happens because the strategy bases the initialization on the average of the class distribution, thereby it is a robust statistical estimator given the number of observations.

If we take a look at the classification regions shown in figure 4.5, we can see that the untrained initialization provides a fairly good separation boundary.

Analyzing on the multi-class dataset shown in figure 4.1, we notice that for the balanced case, the loss for the validation set is not initialized as close to the minimum as in the binary case. However, table 4.3 shows that the model initialized with the proposed strategy converges almost 60 epochs (2 seconds) faster than the one initialized randomly.

In the case in which there is a 90% imbalance in the purple class, it can be seen in figure 4.9 that the proposed strategy puts both the loss and the accuracy, for the validation set, very close to the optimal values. This produces convergence in about 120 epochs (2.5 seconds) faster, compared to the random initialization.

In the case of an imbalance of 95% in the purple class and 98% in the green class, we can see in the table 4.5 that, for the test set, the accuracy for the green class is greatly affected when training, producing an accuracy of 0.33 in the case of the random initialization and 0.66 in the case of the proposed initialization for the green class. The proposed initialization without training achieves an accuracy of 0.83 for the same class. This is interesting because in certain cases it may be better to use the parameters proposed by our strategy as the optimal ones and avoid training. However, it is important to mention that in this simulation we did not use any strategy that considers unbalance dataset.

If we analyze the classification regions shown in figure 4.12, it is observed that when trained, the decision boundary tends to penalize minority classes (purple and green). On the other hand, the proposed method without training establishes the borders in such a way that the separation between the classes is preserved regardless of the imbalance.

In the results shown in the table 4.6, which refer to the benchmark of real datasets, the initialization strategy proposed for the multinomial regression models works fairly good. We realize that the results provided by the proposed method without training, in most datasets, are very close to those obtained by logistic regression models after training. The confidence intervals are considerably narrow, which indicates that the initialization strategy is stable regarding the training data. In addition, for almost all the datasets, the convergence time for the learning algorithm presents a considerable reduction when it is initialized with the proposed strategy, compared to the random initialization.

Many of the datasets tested in the benchmark had an imbalance in their classes. When the randomly initialized logistic regression is trained and no unbalance strategy during training is considered, minority classes are severely penalized by the cost function. However, since the proposed method is robust against class imbalance, and the epochs necessary for the convergence of the initialized model with the proposed strategy are considerably reduced, the penalty suffered by minority classes is also reduced. This allows the logistic regression initialized with our method to be more robust to imbalance than random initialization.

Although the results presented in table 4.6 show that the reduction in the number of epochs required for the convergence of the model, and the total training time including initialization, is inversely proportional to the size of the dataset, as is the case of the *Mnist* and *KDD Cup 1999 Data* datasets, in which the training time is reduced by approximately 100 and 300 seconds when using our initialization. One of

the disadvantages of the proposed initialization is that its time depends on the number of observations. To determine the initial parameters certain statistical estimators are calculated, they depend on the size of the population. This can become problematic in presence of very large datasets, like the ones usually used in deeplearning.

5.2 Feed-forward Neural Network Results

In the case of the feed-forward neural networks, tests were performed using Xavier, He and the proposed initialization algorithm, we used models that had one and two hidden layers in their architecture, each of these was composed by 5*K neurons, where K is the number of the dataset classes, i.e. 5 neurons were assigned to each class. The number K can vary for more complex problems, here we just wanted to provide a proof of concept. Analyzing the results of the models that only had one hidden layer, and used the dataset shown in figure 4.13, we realized from figure 4.14 that in the validation set, the proposed method starts with a lower loss value, approximately 10%, and with a higher accuracy, approximately 50%, compared to the other initializations. This produces in the network a convergence in approximately 140 epochs (8 seconds) earlier than the network that uses He's initialization, and 250 epochs (14 seconds) earlier than the one that uses Xavier's initialization. The classification regions displayed in figure 4.15 indicate that only the proposed strategy without training manages to separate the different classes.

Studying the case in which an imbalance of 90% in the blue class is considered, it is observed in figure 4.16 that for the validation set, both the loss and accuracy of our initialization starts very close to, and even slightly worse than the He initialization. However, in just 4 training epochs, the loss of the network initialized with the proposed strategy converges, while the other initializations require more time. This tells us that although He's method starts closer to a local minimum of the cost function, our strategy is located in a better region of the cost function, not too rough and not too flat, which accelerates the convergence of gradient descent.

If we take a look at the performance of the test set shown in table 4.9, and the classification regions shown in figure 4.17, we realize that the training of the networks is strongly penalizing the minority class, producing an accuracy of 0.0 for the blue class when using models with Xavier and He initialization. In contrast, the proposed initialization without training achieves a total accuracy of 0.97 and maintains a blue class accuracy of 0.96.This indicates that the initialization strategy for neural networks is also robust against imbalance, which makes a lot of sense since it is a direct extension of the proposal for logistic regression. Also, although the accuracy for the blue class in the network that is initialized with the proposed strategy after training is better than the other models (0.25 accuracy), this class is still penalized by training. This is because we are not using any training strategy in presence of unbalanced datasets.

Regarding the proposed initialization and the neural networks that consist of two hidden layers, it can be seen in figure 4.18 that both the loss and the accuracy, for the validation set, of the method proposed in the balanced case, presents a strange behavior. The learning algorithm with the proposed initialization starts close to a local minimum of the cost function, but when training begins it quickly jumps to another

region of the cost function, and from it starts to converge. This may be due to the fact that the parameters proposed by the initialization algorithm are located very close to the decision boundary, which guarantees a correct separation of the atypical data of the classes, i.e. the parameters for initialization of the model are observations that have a high probability of belonging to more than one class, since they are close to the border. But at the same time, these initial parameters can ignore the structure of the representative data of the class; thereby, causing a penalization in the cost function. Despite this, the test set results and classification regions shown in table 4.10 and figure 4.19 respectively, indicate that the proposed method without training manages to separate the classes very well, and has a higher performance than the other initializations.

When considering the imbalanced case of 90% in the blue class and the model with two hidden layers, the accuracy for the blue class when the network is trained is 0.0, independently of what initialization strategy is used. This happens because the network is becoming more complex and this makes the cost function penalize the minority class even more. However, the proposed method without training behaves considerably well for both classes. This can be seen in the classification regions of figure 4.21. Once again, here we are not considering any strategy for the training in presence of unbalanced datasets.

Studying on the multi-class dataset shown in figure 4.22, which was only tested with the two hidden layers model, we notice from the classification regions shown in figure 4.24, that for the balanced case, the untrained initialization strategy manages to separate the classes fairly good. In the same way, although table 4.11 shows that the network initialized with Xavier takes only 178 epochs (13.51 seconds) to converge, we see a higher accuracy for the test set in networks initialized with He and with the proposed method, which exceed 300 epochs (23 seconds) of training. This indicates that, in this case, Xavier's initialization got stuck in a non-optimal local minimum, while the other two methods initialized the network in a more suitable region of the cost function.

Analyzing the case that considers a 90% imbalance in the blue class, as in the binary case, the network training strongly penalizes the minority class in the test set, obtaining an accuracy of 0.0 for this class (table 4.12). In contrast, the proposed method without training achieves an accuracy of 0.66 for the blue class, which in turn reduces network convergence time by almost 90 epochs (2 seconds). The classification regions shown in figure 4.26 reflect the difference in performance.

When tests are performed considering an imbalance of 98% in the blue class and 95% in the red one, as shown in figure 4.27 for the validation set, we observe the same behavior that was presented in the imbalanced binary case with networks constructed with two hidden layers. The initial parameters give more importance to the observations close to the boundaries than to the bulk of their distribution per class, producing for the proposed initialization parameters to be penalized by the cost function in the earliest epochs of training. Despite this, we observed in the classification regions shown in figure 4.28 and in the test set results found in table 4.13, that the performance of the trained networks, regardless of the initialization, is not satisfactory. All the data is being classified in the majority class, thus having an accuracy of 0.0 for the other classes. This indicates that although the proposed initialization without training achieves considerably good results for the three classes,

the cost function used during training is penalizing the minority classes.

Results shown in table 4.14, which refer to the real datasets, indicate that our initialization strategy for neural network models, as in the case of logistic regression, works fairly good. We realized that the results provided by the proposed method without training, in most of the datasets, are very close to those obtained by the networks after convergence of the training.

It is important to notice that in this case all networks were trained for the same number of epochs (100 epochs), which is why the times for training are very close. However, it is observed that the models that were initialized with the proposed strategy, in some datasets, achieve higher accuracy than that obtained with other initializations. An example of this is the *Iris* dataset, in which the networks initialized with Xavier's and He's strategy achieve an accuracy of 0.62 and 0.56, respectively; while the one initialized with the proposed method achieved 0.84.

On the other hand, if we look at the performance of the model by class in some datasets, we realize that the networks initialized with Xavier's and He's strategy have a higher total accuracy, because they are strongly penalizing minority classes; while the model initialized with the proposed method with and without training present a better behavior in all classes, regardless of their imbalance. Specifically, one of the examples in which this occurs is the *Wine* dataset. Here, the class with label 0 receives an accuracy of 0.0 from the networks initialized using Xavier's and He's, while with the proposed method achieves an accuracy of 0.52.

The initialization method proposed for the case of neural networks has several disadvantages compared to other initializations. First of all, as in the case of logistic regression, the initialization time of the proposed strategy depends on the size of the dataset, which can cause problems when dealing with large volumes of data. Second, the proposal consists of finding characteristic points of the classes that map the decision boundary. However, although there are various strategies to find these datapoints, such as selecting point that maximize the entropy [40], use self-organizing maps [41]; adequately finding these points specifically in high dimensional can become complicated and costly in terms of computational resources. Finally, once the characteristic points have been obtained, i.e. the initial parameters for the network, it is necessary to assign a data region to each of these in order to calculate the bias. Depending on the distribution of the data, finding such regions can become computationally expensive, however, there are currently several strategies to perform this task properly and optimally such as Voroni diagrams [42], Clustering techniques [43], Locally sensitive hashing [44], etc.

Chapter 6

Conclusions and Future Work

In this thesis we proposed new initialization strategies for multinomial logistic regression and the neural network models for classification problems. The logistic regression initialization is based on statistical estimators of the data distribution and distance metrics, particularly the mean and the euclidean distance between different scalar products.

First of all, in Chapter 1, we present an introduction to machine learning. Here we described the importance that these models have had in recent years, the learning algorithms they use - gradient descent and its variations -. Likewise, we discussed the initialization problem and how this affects their learning. Also, we presented some initialization strategies that are currently considered the state of the art, He initialization and Xavier initialization.

Chapter 2 presented a detailed description of the logistic regression model, the cost function used for its training, as well as its relationship with neural networks. This chapter describes the initialization strategy proposed for logistic regression, which, in essence, is the basis for the initialization of the other models. We also describe how to extend this initialization algorithm for multinomial logistic regression.

Chapter 3 began with an introduction to neural models and fully connected feed-forward neural networks, its learning algorithm - Backpropagation and gradient descent -. We also discussed the main challenges that these models face during their training, specifically, time and computational resources. The chapter ended with the extension of the initialization strategy explained in chapter 2 to neural networks.

In Chapter 4 we presented and discussed the results of the initialization strategy for logistic regression, multinomial regression, and neural networks models. We used some artificial datasets in order to illustrate the behavior of the models when using the proposed initialization. In addition, 8 real datasets from the *UCI repository* and *Kaggle* were used to generate a benchmark study to compare its performance with the classical training of these models. In the case of logistic and multinomial regression, the randomly initialized model is compared with the proposed strategy and in the case of neural networks, the initialized model is compared with He and Xavier strategies. Codes and tables are available at `https://bitbucket.org/davidsantiago1011/thesis-ml/`.

From the main findings already presented in this work, we want to highlight the following ideas:

1. Through this study we have seen that it is possible to initialize the logistic regression and the neural networks models based on the statistical information of the training data distribution. These strategies considerably reduce the computational resources required for the training of these models and increase their performance.

2. We have seen that these strategies work specially well for class-unbalanced data such as *Wine* and *Breast Cancer* datasets.

3. In the results presented throughout this thesis, only the *sigmoid* activation function was used. However, the initialization strategy works similarly for other activation functions.

4. Neural networks are considered black box models, because it is difficult to interpret and justify their architecture. However, this thesis presents a way to define the neural network architecture based on the complexity - non-linearity - of the class separation boundary, and produces another way to interpret these models.

In the future, we aim to extend this research and publish a paper summarizing the most important findings. There are some important points that we have not explored yet in this work, but that are a potential opportunity to not only improve the results presented in this thesis, but also to open new research fields. Some of these points are:

1. Throughout this thesis, only the classification problem was analyzed. However, the proposed initialization strategy can be extended to regression problems with some adaptations.

2. In the case of neural networks, algorithms to efficiently map the decision frontier of the data are needed. The use of more efficient and robust strategies could lead to a considerable improvement in the initialization results.

3. Deep learning networks are used in most real-life problems. Therefore, the extension of this initialization strategy to convolutional networks, recurrent networks, GANs and transformers should be considered.

4. It is required to adapt the strategy and perform more tests for the cases in which the model architecture contains more neurons than the number of observations. These cases could be addressed using strategies from data augmentation prior to the initialization, in order to increase the amount of data preserving their initial distributions.

Bibliography

[1] F. Luis and G. Moncayo, *No Title*, Third, T. Dietterich, C. Bishop, D. Heckerman, M. Jordan, and M. Kearns, Eds., ISBN: 9788490225370.

[2] R. S. M. Carbonell J.G. and T. M. Mitchell, "Machine Learning: A Historical and Methodological Analysis," *AI Mag.*, vol. 4, no. 3, 1983.

[3] R. Sathya and A. Abraham, "Comparison of Supervised and Unsupervised Learning Algorithms for Pattern Classification," *International Journal of Advanced Research in Artificial Intelligence*, vol. 2, no. 2, 2013, ISSN: 21654050. DOI: 10.14569/ijarai.2013.020206.

[4] L. M. Castro Heredia, Y. Carvajal Escobar, and Á. J. Ávila Díaz, "Análisis Clúster como Técnica de Análisis Exploratorio de Registros Múltiples en Datos Meteorológicos," *Ingeniería de Recursos Naturales y del Ambiente*, vol. enero-dici, no. 11, pp. 11–20, 2012.

[5] H. Alashwal, M. El Halaby, J. J. Crouse, A. Abdalla, and A. A. Moustafa, "The application of unsupervised clustering methods to Alzheimer's disease," *Frontiers in Computational Neuroscience*, vol. 13, no. May, pp. 1–9, 2019, ISSN: 16625188. DOI: 10.3389/fncom.2019.00031.

[6] D. N. Wagner, "Economic patterns in a world with artificial intelligence," *Evolutionary and Institutional Economics Review*, vol. 17, no. 1, pp. 111–131, 2020, ISSN: 1349-4961. DOI: 10.1007/s40844-019-00157-x. [Online]. Available: https://doi.org/10.1007/s40844-019-00157-x.

[7] Z. Wang, Q. Wang, and D. W. Wang, "Bayesian network based business information retrieval model," *Knowledge and Information Systems*, vol. 20, no. 1, pp. 63–79, 2009, ISSN: 02193116. DOI: 10.1007/s10115-008-0151-5.

[8] C. Lemaréchal, "Cauchy and the Gradient Method," *Documenta Mathematica*, vol. ISMP, pp. 251–254, 2012. [Online]. Available: https://www.math.uni-bielefeld.de/documenta/vol-ismp/40_lemarechal-claude.pdf.

[9] S. Ruder, "An overview of gradient descent optimization algorithms," pp. 1–14, 2016. arXiv: 1609.04747. [Online]. Available: http://arxiv.org/abs/1609.04747.

[10] A. Lydia and S. Francis, "Adagrad - an optimizer for stochastic gradient descent," vol. Volume 6, pp. 566–568, May 2019.

[11] H. Shaziya, "A study of the optimization algorithms in deep learning," Mar. 2020. DOI: 10.1109/ICISC44355.2019.9036442.

[12] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. DOI: 10.48550/ARXIV.1412.6980. [Online]. Available: https://arxiv.org/abs/1412.6980.

[13] Z. Liu, H. Wang, L. Weng, and Y. Yang, "Ship Rotated Bounding Box Space for Ship Extraction from High-Resolution Optical Satellite Images with Complex Backgrounds," *IEEE Geoscience and Remote Sensing Letters*, vol. 13, no. 8, pp. 1074–1078, 2016, ISSN: 1545598X. DOI: 10.1109/LGRS.2016.2565705.

[14] S. K. Kumar, "On weight initialization in deep neural networks," pp. 1–9, 2017. arXiv: 1704.08863. [Online]. Available: http://arxiv.org/abs/1704.08863.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2015 Inter, pp. 1026–1034, 2015, ISSN: 15505499. DOI: 10.1109/ICCV.2015.123. arXiv: 1502.01852.

[16] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," *Advances in Neural Information Processing Systems*, vol. 2018-December, no. NeurIPS 2018, pp. 6389–6399, 2018, ISSN: 10495258. arXiv: 1712.09913.

[17] E. Hurtado Dianderas, "Modelo De Regresión Logística," *Gestión en el Tercer Milenio*, vol. 10, no. 20, pp. 25–27, 2007, ISSN: 1560-9081. DOI: 10.15381/gtm.v10i20.9059.

[18] V. F. Dr. Vladimir, *No Title No Title No Title*, 69. 1967, vol. 1, pp. 5–24, ISBN: 9781107057135.

[19] A. Field, "Logistic regression Logistic regression Logistic regression," *Discovering Statistics Using SPSS*, pp. 731–735, 2012.

[20] L. Camarero Rioja, A. Almazán Llorente, and B. Mañas Ramírez, "Regresión Logística : Fundamentos y aplicación a la investigación sociológica," *Análisis Multivariante*, p. 61, 2011. [Online]. Available: https://www2.uned.es/socioestadistica/Multivariante/Odd_Ratio_LogitV2.pdf.

[21] G. Zurek, M. Blach, P. Giedziun, J. Czakon, L. Fulawka, and L. Halo, "Brain tumor classification using logistic regression and linear support vector classifier," no. September 2016, pp. 1–4, 2015.

[22] Ö. Çokluk, "Logistic regression: Concept and application," *Kuram ve Uygulamada Egitim Bilimleri*, vol. 10, no. 3, pp. 1397–1407, 2010, ISSN: 13030485.

[23] M. A. Aljarrah, F. Famoye, and C. Lee, "Generalized logistic distribution and its regression model," *Journal of Statistical Distributions and Applications*, vol. 7, no. 1, 2020, ISSN: 21955832. DOI: 10.1186/s40488-020-00107-8.

[24] M. K. Cain, Z. Zhang, and K. H. Yuan, "Univariate and multivariate skewness and kurtosis for measuring nonnormality: Prevalence, influence and estimation," *Behavior Research Methods*, vol. 49, no. 5, pp. 1716–1735, 2017, ISSN: 15543528. DOI: 10.3758/s13428-016-0814-1.

[25] T. et. all. Hastie, "Springer Series in Statistics The Elements of Statistical Learning," *The Mathematical Intelligencer*, vol. 27, no. 2, pp. 83–85, 2009, ISSN: 03436993. [Online]. Available: http://www.springerlink.com/index/D7X7KX6772HQ2135.pdf.

[26] P. Xu, F. Davoine, T. Denoeux, *et al.*, "Evidential multinomial logistic regression for multiclass classifier calibration To cite this version : HAL Id : hal-01271569 Evidential multinomial logistic regression for multiclass classifier calibration," *Universite de technologie de Compie'gne*, 2016.

[27] R. Rifkin, "In Defense of One-Vs-All Classification In Defense of One-Vs-All Classification," *Journal of Machine Learning Research 5 (2004) 101-141*, no. June 2014, 2004.

[28] N. S. Themudo, " Emanuela Bozzini and Bernard Enjolras (Eds.), Governing Ambiguities: New Forms of Local Governance and Civil Society," *Journal of Comparative Policy Analysis: Research and Practice*, vol. 15, no. 5, pp. 476–477, 2013, ISSN: 1387-6988. DOI: 10.1080/13876988.2013.846961.

[29] N. Psy and C. D. Analysis, "Total Categories of the Outcome, Indexed By the Subscript," *Spring*, pp. 1–5, 2021.

[30] T. M. Mitchell and T. M. Mitchell, "The Need for Biases in Learning Generalizations by Computer Science Department Rutgers University New Brunswick , NJ 08904 published May 1980 as Rutgers CS tech report CBM-TR-117 The Need for Biases in Learning Generalizations," no. May, 1980.

[31] P. T. Pregnancy, T. I. Fertility, and F. Growth, "And Development And Development," *Learning*, vol. 50, no. 2011, pp. 681–730, 2005. DOI: 10.1016/j.cell.2017.06.036.Evolution. [Online]. Available: http://tailieudientu.lrc.tnu.edu.vn/Upload/Collection/brief/brief_49491_54583_TN201500606.pdf.

[32] F Amenta, D Zaccheo, and W. L. Collier, "Neurotransmitters, neuroreceptors and aging," *Mechanisms of Ageing and Development*, vol. 61, no. 3, pp. 249–273, 1991, ISSN: 0047-6374. DOI: https://doi.org/10.1016/0047-6374(91)90059-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0047637491900599.

[33] C. Mavridis and J. Baras, "Towards the One Learning Algorithm Hypothesis: A System-theoretic Approach," 2021. arXiv: 2112.02256. [Online]. Available: http://arxiv.org/abs/2112.02256.

[34] R. Matsumura, K. Harada, Y. Domae, and W. Wan, "Learning based industrial bin-picking trained with approximate physics simulator," *Advances in Intelligent Systems and Computing*, vol. 867, pp. 786–798, 2019, ISSN: 21945357. DOI: 10.1007/978-3-030-01370-7_61. arXiv: 1805.08936.

[35] T. Zamora, Zumbado, "McCulloch-Pitts Artificial Neuron and Rosenblatt ' s Perceptron," pp. 16–29,

[36] S. Haykin, "Rosenblatt ' s Perceptron," *Neural Networks and Learning Machines*, no. 1943, pp. 47–67, 2009.

[37] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[38] B. Cheng, R. Xiao, Y. Guo, Y. Hu, J. Wang, and L. Zhang, "Revisit Multinomial Logistic Regression in Deep Learning: Data Dependent Model Initialization for Image Recognition," 2018. arXiv: 1809.06131. [Online]. Available: http://arxiv.org/abs/1809.06131.

[39] P. Krähenbühl, C. Doersch, J. Donahue, and T. Darrell, "Data-dependent initializations of convolutional neural networks," *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016. arXiv: 1511.06856.

[40] J. W. Grzymala-Busse, Z. S. Hippe, and T. Mroczek, "Reduced data sets and entropy-based discretization," *Entropy*, vol. 21, no. 11, pp. 1–11, 2019, ISSN: 10994300. DOI: 10.3390/e21111051.

[41] S. Parashar, N. Fateh, V. Pediroda, and C. Poloni, "Self organizing maps (SOM) for design selection in multi-objective optimization using modeFRONTIER," *SAE Technical Papers*, no. August 2016, 2008, ISSN: 26883627. DOI: 10.4271/2008-01-0874.

[42] D. Reddy and P. K. Jana, "A new clustering algorithm based on Voronoi diagram," *International Journal of Data Mining, Modelling and Management*, vol. 6, no. 1, pp. 49–64, 2014, ISSN: 17591171. DOI: 10.1504/IJDMMM.2014.059977.

[43] T. H. Sardar and Z. Ansari, "Partition based clustering of large datasets using MapReduce framework: An analysis of recent themes and directions," *Future Computing and Informatics Journal*, vol. 3, no. 2, pp. 247–261, 2018, ISSN: 23147288. DOI: 10.1016/j.fcij.2018.06.002. [Online]. Available: https://doi.org/10.1016/j.fcij.2018.06.002.

[44] O. Jafari, P. Maurya, P. Nagarkar, K. M. Islam, and C. Crushev, "A Survey on Locality Sensitive Hashing Algorithms and their Applications," *ACM Computing Surveys*, no. April, pp. 0–23, 2021. arXiv: 2102.08942. [Online]. Available: http://arxiv.org/abs/2102.08942.

[45] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

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

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

[48] M. L. Waskom, "Seaborn: Statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021. DOI: 10.21105/joss.03021. [Online]. Available: https://doi.org/10.21105/joss.03021.

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[50] D. Dua and C. Graff, *UCI machine learning repository*, 2017. [Online]. Available: http://archive.ics.uci.edu/ml.

[51] B.-H. A. Guyon Isabelle Gunn R. Steve and G. Dror, *Result analysis of the nips 2003 feature selection challenge*, 2004.