UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial,
Aeroespacial i Audiovisual de Terrassa

Study of efficient numerical tools for Machine Learning

BACHELOR FINAL THESIS

Document:
Report

Author:
Antonio Darder Bennassar

Director /Co-director:
Àlex Ferrer Ferre

Degree:
Bachelor's degree in Aerospace Vehicle Engineering

Examination session:
Spring

Acknowledgements

Abstract

Artificial intelligence has been a field of interest in the scientific community since the 20th century, inside this field is found deep learning. This discipline of Machine learning focuses on the creation of artificial intelligence capable of reproducing human tasks by learning from raw data. In fact, the reason why deep learning has experienced a great increase in attention and scientific articles in the last decades is the settlement of the digital era, which has meant an exponential increase in stored data, and therefore has allowed algorithms to achieve exceptional results.

Based on the growing need for trained professionals in this field, this final thesis is a study of the fundamentals of deep learning and machine learning that aims to serve as a bridge to more cutting-edge knowledge. The central pillar of the project are artificial neural networks applied to classification, though other general Machine Learning concepts such as gradient descent optimization methods and some regularization methods are also studied at the same time.

The main objective of the project is to analyze the effectiveness of neural networks in some databases, as well as to compare the impact of implementing different features in the network. The programming of all the necessary codes from scratch without the use of support libraries has also been an important part of the project.

To address these objectives, extensive research has been carried out in the field, not only on the theoretical knowledge but also on the mathematics behind it. In addition, a methodology based on object-oriented programming has been followed to create a clean, readable, and easily extendable code.

The results obtained demonstrate the effectiveness of neural networks applied to the classification of both data and images. They also highlight the effectiveness of some of the above features over others. The next logical step after this project would be to further explore the use of neural networks as autoencoders, and study the impact of the implementation of the convolution operation for image classification.

Resumen

La inteligencia artificial ha sido un campo de interés en el colectivo científico desde el siglo XX, se trata de un campo muy extenso dentro del cual se halla el aprendizaje profundo. Este se enfoca en la creación de inteligencias artificiales capaces de reproducir tareas humanas aprendiendo a base de datos. De hecho, el motivo por el cual el aprendizaje profundo ha experimentado un gran aumento de atención y de artículos científicos en las últimas décadas es el asentamiento de la era digital, la cual ha supuesto un incremento exponencial en los datos almacenados, y por tanto ha permitido a los algoritmos conseguir resultados excepcionales.

Basándose en la creciente necesidad de profesionales formados en este ámbito, este trabajo de final de grado es un estudio de los fundamentos del aprendizaje profundo y "Machine Learning" que pretende servir de puente hacia conocimientos más punteros. El pilar central del proyecto son las redes neuronales artificiales aplicadas a la clasificación, aunque paralelamente también se estudian otros conceptos generales de "Machine Learning" como métodos de optimización basados en el descenso del gradiente y algunos métodos de regularización.

El objetivo principal del proyecto es analizar la efectividad de las redes neuronales en algunas bases de datos, así como comparar el impacto de implementar diferentes características en la red. Así mismo la programación de todos los códigos necesarios desde cero sin utilizar librerías de apoyo ha tenido un peso importante en el proyecto.

Para abordar estos objetivos se ha llevado a cabo una investigación extensa del campo, no solo del conocimiento teórico sinó también de las matemáticas que hay detrás. Además, se ha seguido una metodología basada en la programación orientada a objeto para crear un código limpio, legible y fácilmente extensible desde el principio.

Los resultados obtenidos demuestran la efectividad de las redes neuronales aplicadas a la clasificación tanto de datos como imágenes. Además, resaltan la eficacia de algunas de las anteriores características por encima de otras. El siguiente paso lógico tras este proyecto consistiría en profundizar más en el uso de las redes neuronales como "autoencoders" y estudiar el impacto de la implementación de la convolución para la clasificación de imágenes.

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

AF Activation Functions. 24–27, 45, 46

AI Artificial Intelligence. 2, 7, 35

CNN Convolutional Neural Network. 55, 62

DCT Discrete Cosine Transformation. 57, 58

LR Learning Rate. 10, 38, 39

ML Machine Learning. 6, 7

MLP Multilayer Perceptrons. 3, 6, 8

MSE Mean Squared Error. 13

NN Neural Network. VI, 3, 13, 18, 20–22, 24–27, 29–32, 36, 38, 45, 46, 50, 51, 55, 56, 62

OOP Object Oriented Programming. 1, 3

PCA Principal Component Analysis. 10

ReLU Rectified Linear Unit. 25, 26, 58

SGD Stochastic Grdient Descent. 3, 10, 39, 40

SVD Singular Value Decomposition. 11

1 Introduction

1.1 Aims

The goal of this project is to create an in-house code containing the basic Deep Learning tools currently used for classification and regression tasks. This code will be built on the foundations of neural network algorithms, and it will be constantly improved by adding features such as autodifferentiation, stochastic gradient or the kernel trick. The clearance of the code will also be a vital part of the project as it will become increasingly complicated and it will be a need to navigate easily through it. For this reason, another pillar of this study will be object-oriented programming (OOP).

1.2 Requirements

The object of this study is to present in a clear way the most common algorithms in the field of deep learning. Under this pretext it is also planned to implement a simple code from scratch. The software used will be MATLAB and for the purpose of the project, the basic functions and some more complex ones such as fmincon or dct will be used without internal implementation.

Another requirement established for the project is that the code should be able to be executed and inspected on any computer, easy to read and extendable. Databases will also play an important role, thus it will be also required free external sources.

The requirements can be summarized into five main points:

- Developed in MATLAB software
- Open Access databases
- Code almost completely implemented from the standard MATLAB functionalities
- For any user and computer
- Easy to read and extendable

1.3 Justification

Deep learning methods have drastically increased in interest in the recent years. Although the mathematical concepts used in this technology were already known back in the 1950s, it was not until 2005 when the internet became widely used and the computational power and size of the data grew dramatically (see Figure 1.1), that this technology began to look promising. During these years the name of the field was shifted from "conectionism" to "deep learning". [6]

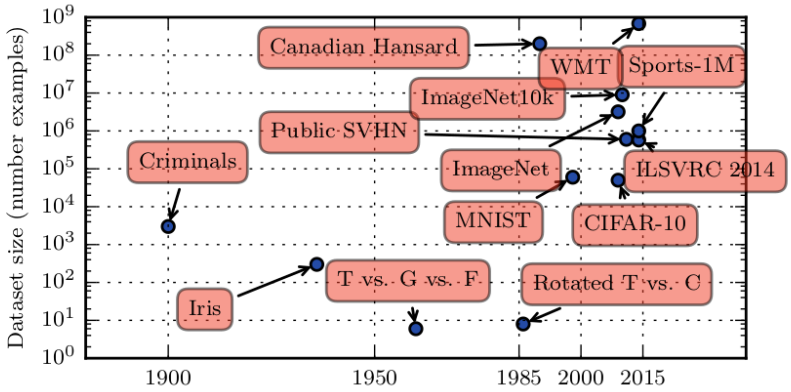


Figure 1.1 Increase in the size of the datasets for training deep learning models from 1950 to 2015. *Source [6]*

In the last years several Artificial Intelligence (AI) have appeared demonstrating better performance than humans at certain tasks, solving quotidian problems with ease. Some examples of these are:

- Alphazero (from deep mind 2017) is an AI trained to play chess, learned from scratch and eventually defeated the strongest computer (a non deep learning programm).
- GPT-3 (from openAI 2020) is an autoregressive language model capable of reproducing human writing.
- Codex (from openAI 2021) it is capable of parsing natural language and generating code in response.

Nowadays we live in the automation era, where AIs are starting to merge, and getting into society making human life easier. It is a general believe that artificial intelligence will become a big part of the picture in the coming years, contributing to some of the major scientific challenges of our age [11]. Developing a project about its foundations is a great way to start building knowledge towards much more complex and innovative algorithms that are currently in use.

1.4 Scope

The development of this project will include the study and implementation in an in-house code of the following concepts:

- Feedforward Neural Networks (NN) or Multilayer Perceptrons (MLPs)
- Automatic Differentiation
- Stochastic Gradient Descent (SGD)
- Autoencoders
- Kernel Trick

All these will be enclosed and performed in:

- MATLAB language/environment
- OOP paradigm

The project will not include:

- Creation of databases
- Development of a constrained solver
- Discrete Cosine Transformation

High level deliverables:

- Project report
- Budget
- Code files

1.5 Planning

The first step in every project is to settle a planning. But before doing so, in order to get the best possible organization, a work breakdown structure has been done. This scheme is helpful to identify the tasks that will be developed, and break them into smaller ones (see next page).

The main tasks are the five ones described in the scope of the project, jointly with the project management. These six tasks have been subdivided into different subtasks. This last level is the one that will be represented in the Gantt diagram.

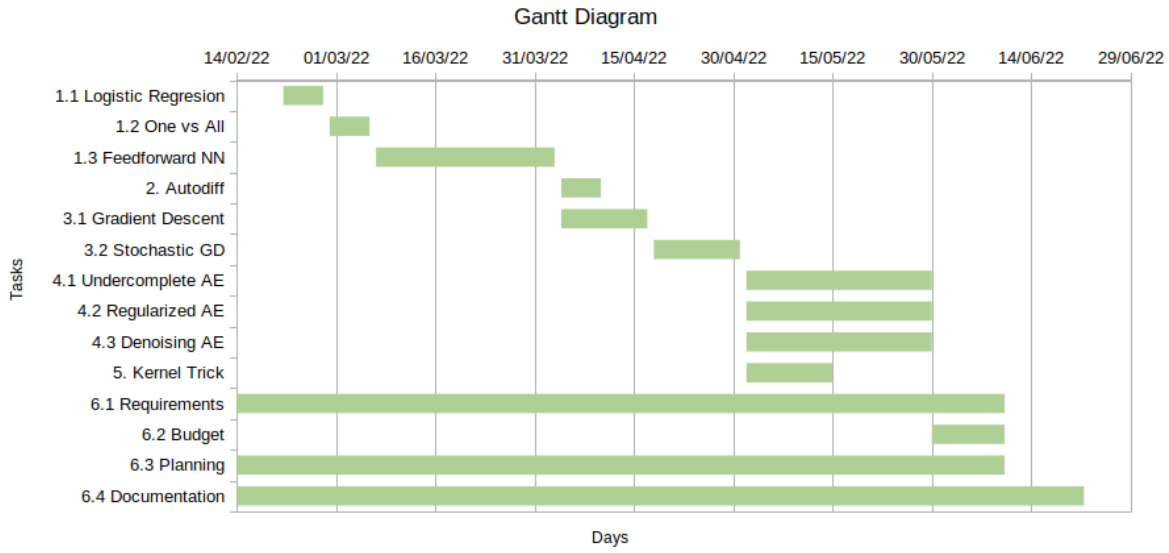
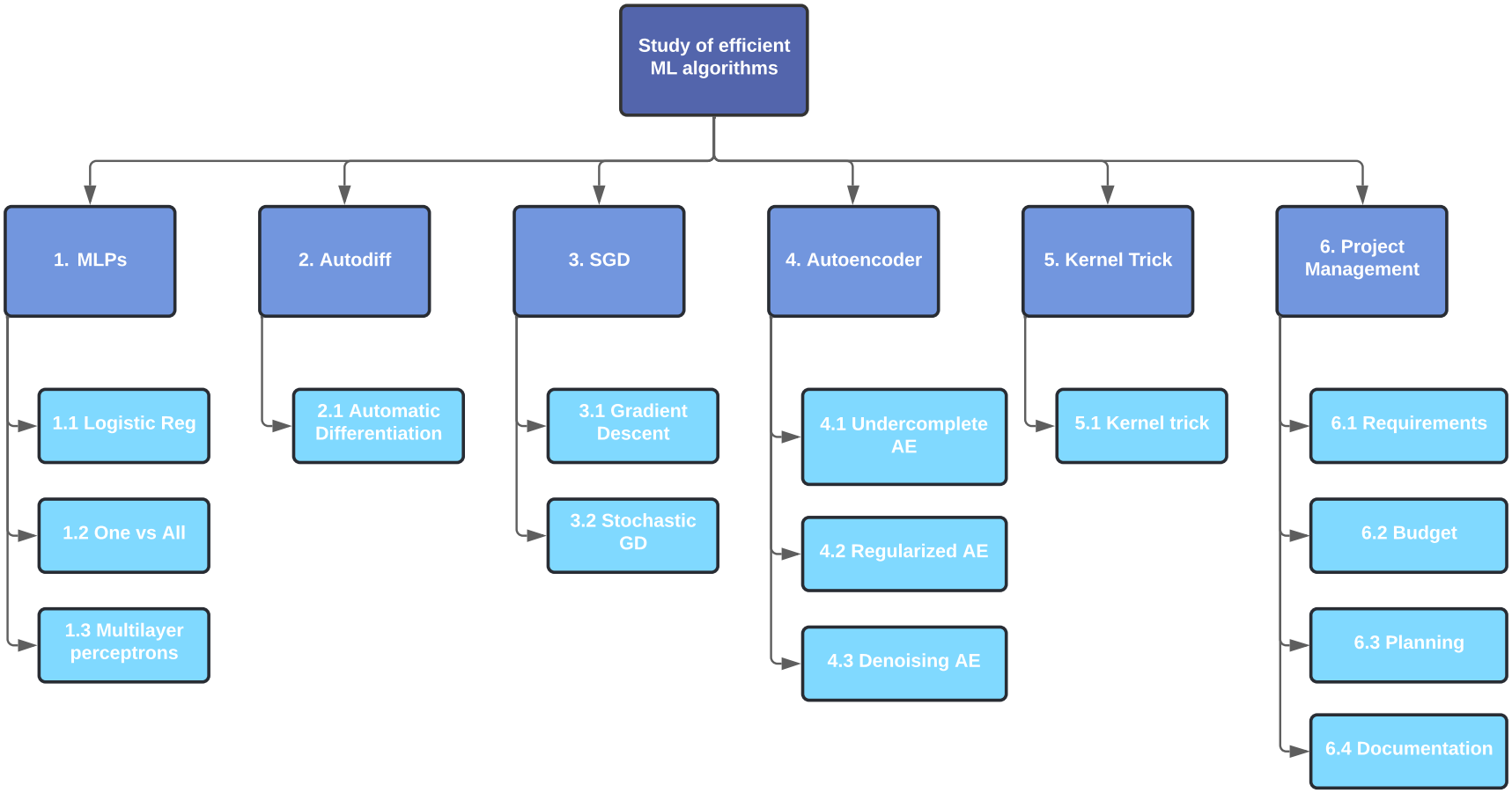


Figure 1.2 Gantt Diagram for this project. *Source own*

It is worth mentioning this Gantt diagram is planned for a regular TFE semester. In spite, the project was started at the beginning of the previous semester and some progress has been made. To set the context, tasks 1 and 2 have been developed almost in their totality, and tasks 3 and 6 have already been started. This will leave more room for maneuver in the last parts of the project.

5.1 Work Breakdown Structure (WBS)



2 State of the art

In this section, the methodology process of this thesis will be briefly explained. It includes a state of the art of the different parts of the project, starting from the outside layers to more complex concepts. A more in depth study of the mathematics behind each algorithm is shown at the beginning of its respective section.

2.1 Deep Learning

Deep Learning is a field in computation science which started almost in the 1950s but it was not until 2005 when it really started to emerge thanks to globalization and the start of the "data era". With higher amounts of data and higher computational power, deep learning started to make its own way outstanding over other Machine Learning algorithms.

Machine Learning refers to the study of computer algorithms which use mathematical models to represent a sample of data, expecting that new inputs will be correctly predicted by the model. Inside Machine Learning is found representation learning, this field differs from traditional ML as the features are not hand-designed but learned by the computer itself. Finally inside representation learning is found deep learning, and as its name suggests it is based on the addition of more layers to the model, this enhancing the creation of more abstract features. Deep Learning is most of the times based on deep neural network architectures or multilayer perceptrons (MLPs). (In Figure 2.1 a Venn diagram of deep learning is shown)

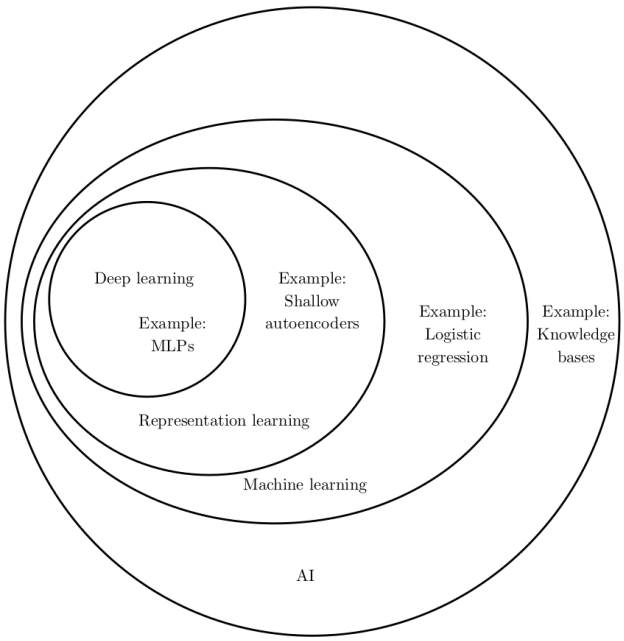


Figure 2.1 Venn Diagram of the layers of Deep Learning. *Source [6]*

Deep learning is one of the most used techniques in AI and it has a good outlook for the coming years. It is expected to grow from $ 4,065.0 million in 2016 to $ 169,411.8 million in 2025. This means a compound annual growth rate of 55.6% in just 7 years (2018-2025) [7]. According to another study the computational annual growth rate from 2019 to 2026 will be around 35.4% [9]. These values represent how strong the market related to AI is becoming and how these new technologies are getting into people's daily lives. In figure 2.2 the growth of different artificial intelligence fields including ML is shown for the next years.

On the other hand, figure 2.3 shows a survey done to Italian firms in 2020. Where only 9% of them answered that they had no plan for AI implementation. However, a stunning 71% of them answered that AI is already being used or in an experimental phase in some of their business areas.

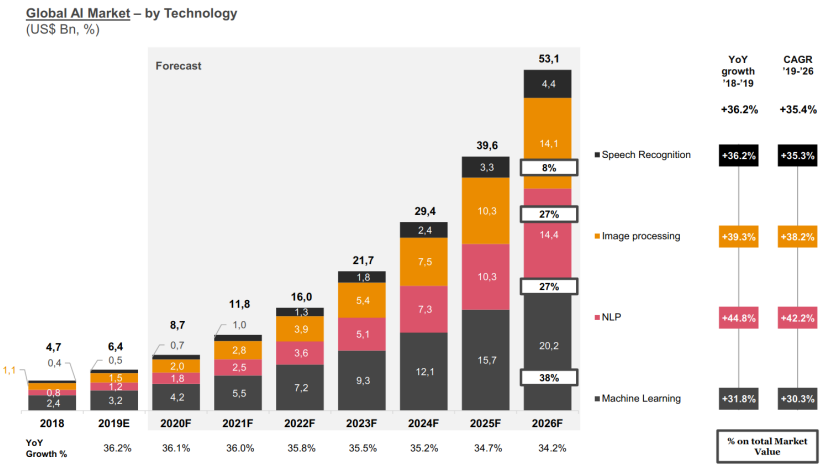


Figure 2.2 Compound annual growth rate from 2018 to 2026 for different AIs. *Source [9]*

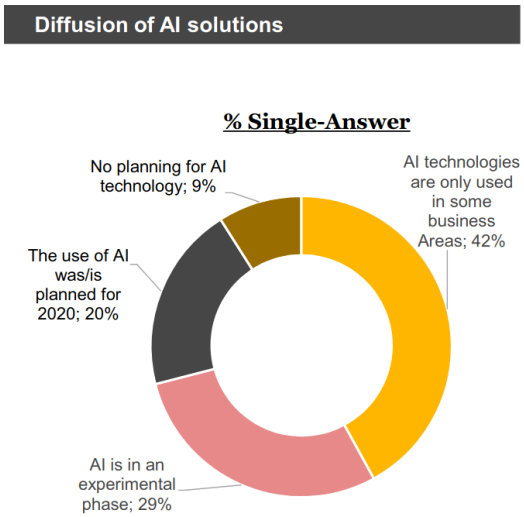


Figure 2.3 Italian firms intention in the implementation of AI techniques. *Source [9]*

2.2 Feedforward Neural Networks

As Ian Goodfellow says in his book, neural networks are the basic and most spread models currently in use. "Deep feedforward networks, also often called feedworward neural networks or multilayer perceptrons (MLPs), are the quintessential deep learning models." (Ian Goodfellow, et al. 2016 [6])

Neural networks hide a loosely connection to neuroscience. The name was firstly inspired by the biological neural networks that constitute animal brains, which base their functioning in electrical signals. Analogously MLPs are formed by multiple layers of individual units (neurons) that receive inputs and compute an output in response (see Figure 2.4). They are called feedforward as there are no backward connections, if they are extended to incorporate feedbacks they are called recurrent neural networks.

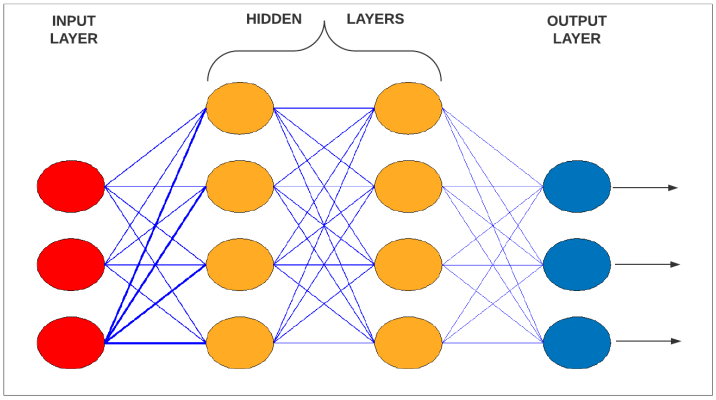


Figure 2.4 The general architecture of a neural network. With its inputs, hidden layers and outputs. *Source own*

Convolutional networks, autoencoders, and recurrent networks all use neural networks as their foundation. These are the most often utilized machine learning algorithms in several fields like object/facial recognition, parsing human language or even modeling key dynamic simulations in aerospace engineering.

2.3 Automatic differentiation

Optimization is one of the pillars that support deep learning. It is often the case that the model uses a scalar function $f : \mathbb{R}^n \to \mathbb{R}$ to represents the model error. In order to learn, the model needs to minimize f and to do so the derivatives are needed.

Calculating derivatives can be a computer demanding task for several reasons. In order to save as much computation time as possible lots of optimization algorithms have been developed

throughout the years. Gradient descent algorithm are usually chosen over the others. As their name suggest they use the gradient of the function to minimize tehir parameters. In this context automatic differentiation proposes a heuristic and elegant way to compute the derivatives of such functions.

In computer science automatic differentiation (abbreviated autodiff) is a way to evaluate the derivatives of functions. Besides autodiff, there are two other traditional ways of computing derivatives, these three are:

- Symbolic differentiation
- Numerical differentiation
- Automatic differentiation

Symbolic differentiation is the "human" way of calculating derivatives, this approach is difficult to apply for computers and frequently results in inefficient code. On the other hand, numerical differentiation is the most basic and widely used methodology which is performed using the finite difference method, this method is usually prone to round-off errors.

Both of these methods have problems addressing higher derivatives, where mistakes grow, and are sluggish at computing partial derivatives, which is a characteristic feature of machine learning.

Autodiff, a chain rule-based method, overcomes the majority of these issues by breaking the problem into smaller pieces over and over until it reaches the simplest derivative possible. The computational process follows a flow through the different simplest steps broken before. In figure 2.5, this process is represented via a flow computational diagram.

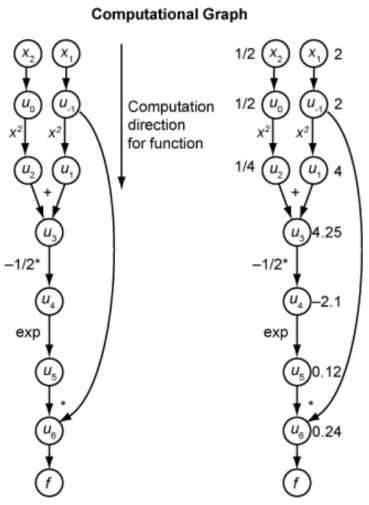


Figure 2.5 Autodiff flow diagram for a per-example function ($f(x) = x_1 \cdot exp0.5(x_1^2 + x_2^2)$). *Source [18]*

2.4 Gradient Descent Optimization

The automatic differentiation process was justified on the premise that the solvers for the optimization process usually need at least first derivatives. This is the case of gradient descent methods.

Optimization algorithms are often classified by different characteristics:

- First-order || Second-order
- Batch || Minibatch Stochastic
- With adaptive LR || Without adaptive LR

Stochastic Gradient Descent or SGD is the "standard" method when talking of gradient descent methods. It is a first-order minibatch stochastic method that might or might not have an adaptive learning rate. This means that: it only uses first derivatives, and it uses a small batch of data to estimate the gradient.

It is difficult to give a proper introduction to gradient descent methods without getting stuck in the equations which govern it. Later in section 6, the explanation of these optimization processes will be extended.

In conclusion, these properties leave SGD optimization as a robust and easy to implement method, and this makes it to be among the top solvers in deep learning techniques.

2.5 Autoencoders

Autoencoders can be seen as a particular case of feedforward neural networks. The idea of an autoencoder is to learn to copy its input to its output. This may be understand as a composition of two functions: the encoder $h = f(x)$ and the decoder $r = g(h)$. The expectation is that the autoencoder does not learn to perfectly copy x, but it is forced to learn to prioritize which information is useful.

There are several types of autoencoders but one of the simplest is the undercomplete autoencoder which reduces the input dimension in the encoder and reconstructs it in the decoder (see figure 2.6). Then in the last layer of the encoder, all the information has been compressed to a much more dense state.

If the decoder is linear and the loss function is represented by the L2 norm the undercomplete autoencoder learns to span the same subspace as PCA. One goal of this project is to try to assemble this behavior and compare these results.

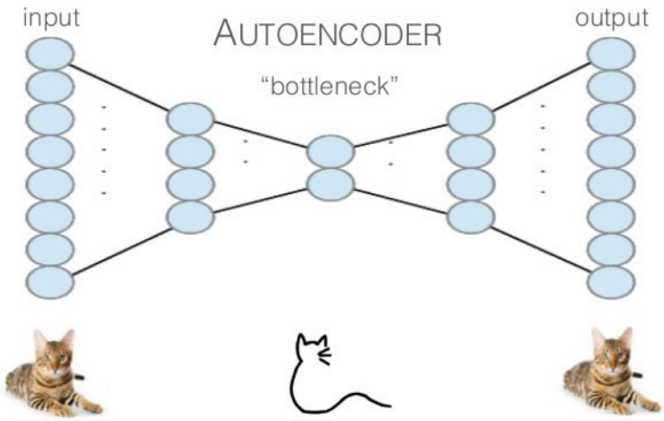


Figure 2.6 Example of undercomplete autoencoder and its latent space. *Source [16]*

Figure 2.6 shows how the inner layers of an autoencoder can be used to compress images. But more surprisingly this architecture allows to introduce "random" inputs at the bottleneck in order to generate images.

2.6 The kernel method

The kernel method is a family of algorithms used for pattern analysis which is a machine learning modality that tries to recognize similarities or relations in the datasets.

Kernel techniques get their name from the fact that they employ kernel functions to work. These allow them to work in a high-dimensional implicit feature space without ever computing the coordinates of the data in that space, instead computing the inner products between all pairs of data in the feature space. This procedure is frequently less computationally expensive than explicitly computing the coordinates. This approach is called the "kernel trick".

This method can be applied wherever a dot product in the data is computed. It is a really helpful tool that saves a good amount of computation time, but a critical thing to keep in mind is that when converting data to higher dimensions there are chances of overfitting the model.

It is common practice to use the kernel trick in support-vector machines (SVD) but it is also used in many other algorithms such as the kernel perceptron, linear adaptive filters, and principal component analysis.

3 The classification problem

Classification has been a problem of interest in lots of fields for a long time, and the possibility of creating a model capable of differentiating between several groups has been studied since the 19th century. Few decades after the first linear regressions were studied in 1805 by Legendre and Gauss, the logistic regression was born as a model of population growth by Pierre François Verhulst.

Before entering the main part of this study which are neural networks for classification. It is worth understanding the history and foundations in this field by studying the concepts of logistic regression, and one vs all. These were the predecessors of the modern algorithms that govern classification tasks nowadays.

3.1 Logistic Regression

The logistic regression is the name given to the model that first was used to attack the classification problem for a dataset with only 2 classes. This model is founded on a linear transformation and on a logistic unit (equations 1 and 2). The latter is introduced into the model to convert the outputs of the linear transformation to probabilities. A (0,1) interval where 0 represents the first group and 1 represents the second one. Then a 0.6 output would represent a 60% probability of belonging to group 1 and a 40% of probability of belonging to group 0.

$$z = \beta \cdot x + \beta_0 \tag{1}$$

$$y = \frac{1}{1 + e^{-z}} \tag{2}$$

Equation 2 is the logistic unit (logit) which is usually called sigmoid due to the s shape of the function. Sigmoid is a very important function in Machine Learning and it will appear a lot in this study, in section 4.3 it will be studied along with other activation functions. In order to complete the model one last piece is missing, which is the error measurement. In the next section error functions are detailed, but for now, it is possible to advance that due to the non-linearity of the system, it will be necessary to implement a numerical optimization.

Bearing in mind all this information, it is easy to implement a logistic regression algorithm using one of the optimizer functions that matlab offers. To test the logistic regression the dataset selected is a Male/Female dataset with only 2 features: weight and height. All the databases used in this project are gathered in appendix B where basic information like number of features,

groups or points is presented. In Figure 3.1 the boundary line between the 2 groups found with the logistic regression is shown. Although the dataset is kind of mixed up (for obvious reasons) there is a clear separation with higher heights and weights for men which is correctly represented by the boundary line. This line represents the 50-50 boundary. Regarding to the model, the points exactly on the line have 50% probability of being man and 50% of being woman.

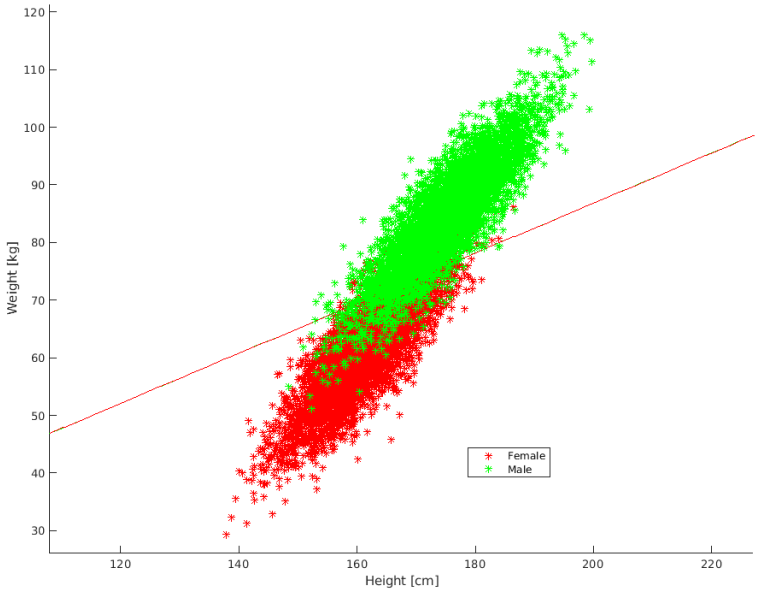


Figure 3.1 Logistic Regression for a Male/Female dataset. *Source Own*

3.2 Cost functions

Cost function is the name given to the objective function in machine learning, that is the function to minimize. How this function is calculated can vary upon different implementations and tasks. The importance of choosing the adequate one for the involved task is enormous as the minimization process will be strongly correlated with it. While in linear regression tasks the cost usually chosen is the mean squared error or MSE, in logistic regression is usually prefered to use the negative log-likelihood also called cross-entropy.

The cost function, is the name of the function which is minimized, and it is primarily composed of a loss function. This is the function with which the error is strictly measured, however, a regularization term can sometimes be added to introduce a penalty to the weights in order to prevent **overfitting**. In section 5 different regularization techniques are debated.

Equation 3 shows the cross entropy loss function, shich is the most used in NN as well. However, other functions such as L^2 (equation: 4) or L^∞ (equation: 5) norms could be used in some cases. The reason behind negative log-likelihood is that it works really well with exponen-

tials. Moreover the logarithm squishes the subspace to numbers with smaller computation time for machines ($ln(1 \cdot 10^{-10}) \approx -23$).

$$L(\theta) = -\frac{1}{m}\sum_{i}^{m} y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i) \tag{3}$$

$$L(\theta) = \sqrt{\sum_{i}^{m} \left[(y_i - \hat{y}_i)^2\right]} \tag{4}$$

$$L(\theta) = max\Big\{|y_1 - \hat{y_1}|, |y_2 - \hat{y_2}|, ..., |y_m - \hat{y_m}|\Big\} \tag{5}$$

3.3 Data augmentation

The model of logistic regression uses a linear model, therefore the boundaries obtained between the classes can only be straight lines. This is not useful for most real scenarios were the data will not be smoothly separated. For example, in Figure 3.2 the *microchip* dataset is shown. These microchips before entering the market have to pass two tests (variables x1 and x2) and then is is the decided if the microchip is accepted or rejected (groups 0 and 1). Figure 3.2 shows that for this dataset a quadratic boundary would fit way better than a straight line.

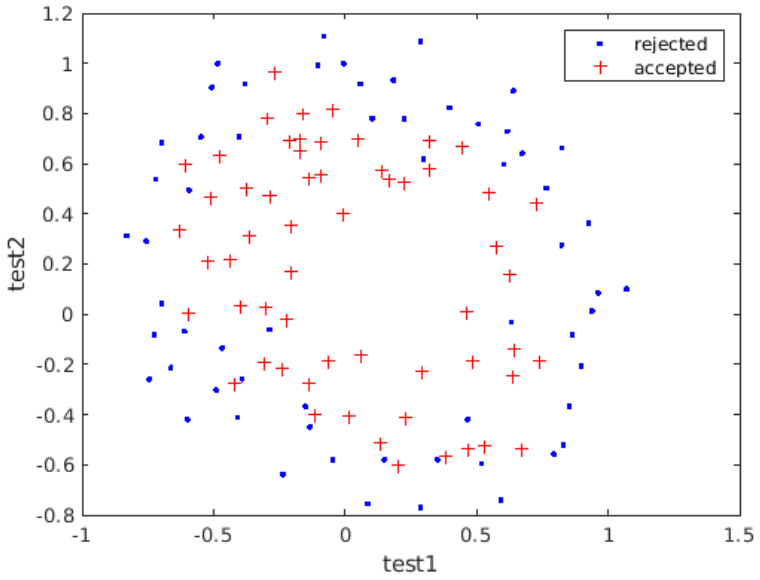
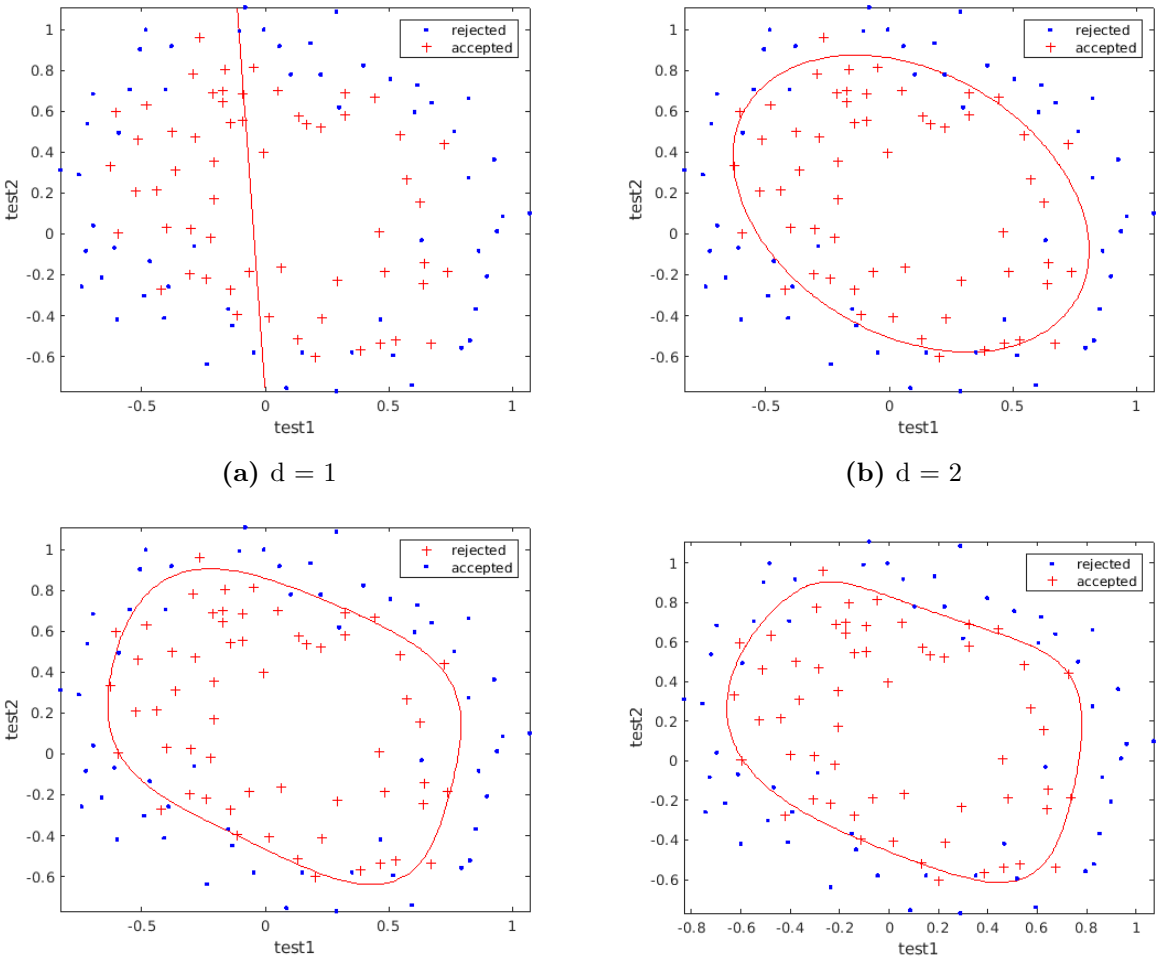


Figure 3.2 Microchip dataset. *Source Own*

There exist some methods to bypass this problem. One of them is data augmentation which is the method of adding non existing features using combinations of the ones that are real. For example if there are features x1 and x2, it might be possible to add more fictional

features such as $x1^2$, $x2^2$, $x1 \cdot x2$, $x1^3$ ... This is similar to what is done in traditional statistics with linear regression.

This technique offers the possibility to insert the first **hyperparameter**, the degree of augmentation d. This parameter represents to which degree the data has been augmented, $d = 1$ means no augmentation, $d = 2$ means to elevate to the power of 2 the data and so on. Hyperparameters will appear often in the study, they are parameters which can not be changed in the model but have a big impact on its performance. In Figure 3.3 the boundaries for the previous dataset are shown with different values of d.



(a) d = 1 **(b)** d = 2

(c) d = 4 **(d)** d = 6

Figure 3.3 Logistic regression with different powers of d for the microchip dataset. *Source own*

In these figures it is obvious that a polynomial degree of 2 is way better than the linear version. In fact d = 6 still looks really good, however if d is raised more, the model will eventually start to overfit.

Data augmentation is a a useful technique, in this subsection this technique has been related to the augmentation of columns in X (features), however in image processing is common to talk about data augmentation referring to the increment of rows in X (data points). This is done by applying transformation to the images like rotation, symmetris etc. giving more data to the model.

3.4 One vs All

The one vs all is the extension of the binary classification problem to n classes. The idea is to extend the output of the logistic regression to a vector of n binary states. In the end we are reducing the problem to n standard logistic regressions were in each one we are trying to classify between the respective class and the "others", this concept is better understood in Figure 3.4

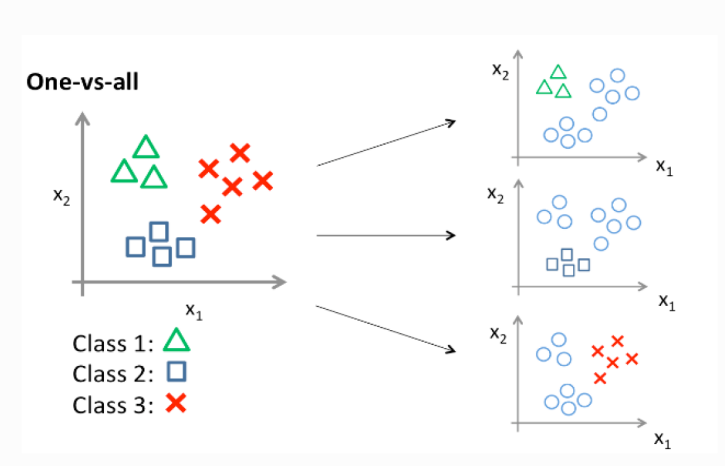


Figure 3.4 The one vs all approach. *Source[12]*

Now the target y will be a matrix instead of a vector, and the loss function will take the sum over all groups as well (shown in equation 6).

$$L(\theta) = \frac{1}{m}\sum_{i}^{m}\sum_{j}^{c} y_i^j \cdot log(y_i^j) + (1 - y_i^j) \cdot log(1 - y_i^j) \tag{6}$$

$$y = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{7}$$

To show the implementation of the one vs all task the famous Iris dataset has been selected. *Iris* is a dataset of 3 different flowers with 150 points and 4 features: sepal width, sepal length, petal width, and petal length. In Figure 3.5 the boundary plot is shown for this task. Again the three different classes seem to be well separated. As it can be seen in the green boundary, data augmentation has been introduced (d = 2).

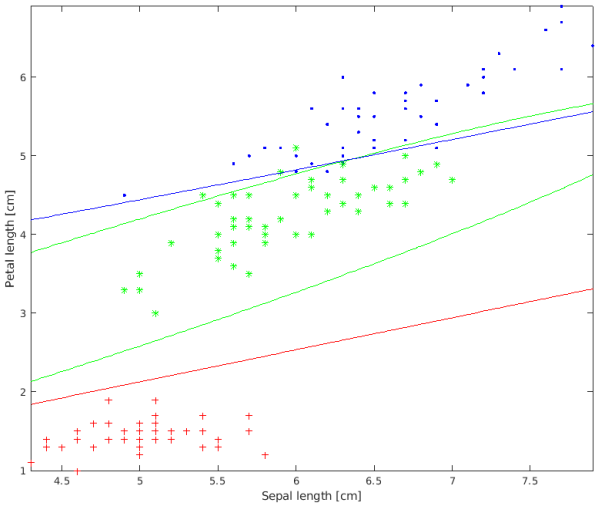


Figure 3.5 One vs all for the Iris dataset. *Source own*

4 Feedforward Neural Networks

As it was mentioned in section 2.2 neural networks are one of the pillars sustaining deep learning. These structures are used in different ways and intentions but for the sake of this study, they will be thought of as a tool for classification learning. For example, if there are two types of dogs (Doberman and Border terrier) that have different body dimensions, the objective of a NN could be to predict the dog's breed only using their dimensions.

In the last example, some hints of the basic structure of a NN can be seen. The "body dimensions" are the inputs for that specific neural network and are usually called **features**. The "prediction" is the output of the NN and it can be understood as a binary state (0 for one breed and 1 for the other) but it is rather preferred to use a continuous interval $I \epsilon (0, 1)$ that symbolizes probability. Finally the operations that the network does in order to get a prediction are called **hidden layers**. As it can be seen some of these characteristics are the same as the ones in the logistic regression, thus it was helpful to understand those first.

Machine learning is usually also classified in supervised, unsupervised, or reinforcement learning. The traditional way to go and the one that will be used in this project is supervised learning. In supervised learning, the model learns the parameters needed to correctly classify the data by minimizing the error between its predictions and the correct answers. In deep learning, the correct answers are called **labels** or **targets** and the error is called **cost**. The choice of the cost function $J(\theta)$ is also important as it will be strongly correlated to the minimization process.

4.1 Architecture

Neural networks are composed by **layers** which in turn are made of **neurons**. The first layer is called the **input layer**, the middle ones are called hidden layers and the last one is the **output layer**.

4.1.1 The neuron

The neuron is the simplest unit of a NN. In Figure 4.1 its functioning is represented. It receives an input and performs a linear transformation to produce the output.

$$y = W^T \cdot x + b \tag{8}$$

The terms y and b are scalars (the output and the free term respectively), x is the vector of inputs and W is the "weights" vector. Then the parameters of the neurons are W and b and are usually enclosed in the term $\boldsymbol{\theta}$. Therefore the neuron performs the transformation $y = f(x; \theta)$. The optimization must find the θ that minimizes the error of the predictions.

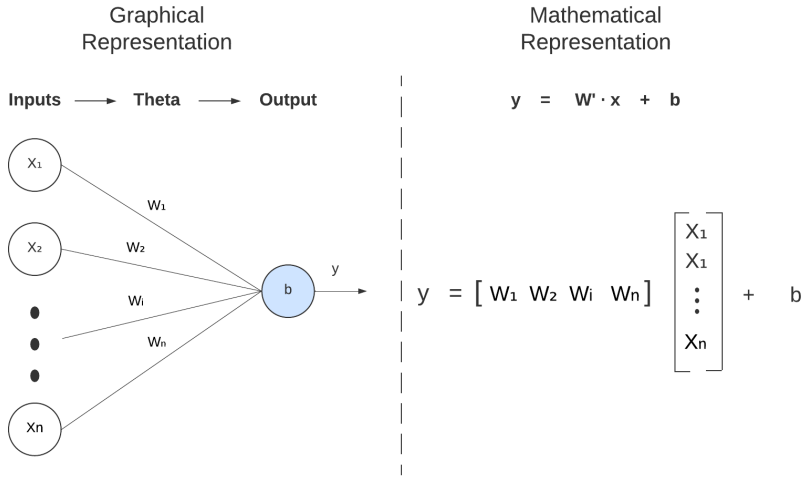


Figure 4.1 Representation of a neuron. *Source Own*

4.1.2 The layer

A layer is a collection of neurons that apply their transformation at the same level of the model, they are acting in parallel (see Figure 4.2). For this reason a layer can also be explained by itself as a linear transformation $f : \mathbb{R}^n \to \mathbb{R}^m$ with n given inputs and m outputs (where m is also the number of neurons in the layer).

Equation 8 can be rescued where now y and b are the outputs of the neurons and their free terms respectively (vectors with length m), x is the input vector (with length n) and W is the "weighted" matrix (dimensions $n \times m$).

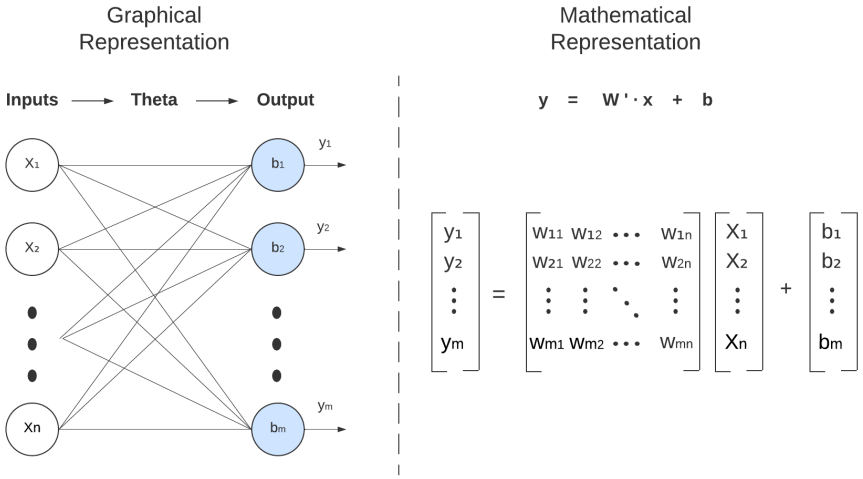


Figure 4.2 Representation of a layer. *Source Own*

4.1.3 Deepening the model

Once the structure of a layer is defined a multilayer NN is obtained by the concatenation of several layers. In the last examples, the final output was omitted, but it is an important characteristic of a NN. The final purpose of the network is to classify data, therefore the commonly used strategy is to have a final layer with m outputs where m is equal to the number of classes. Then a binary state defines whether the data entry belongs to that particular group or not. Figure 4.3 shows the representation of a neural network with 2 hidden layers.

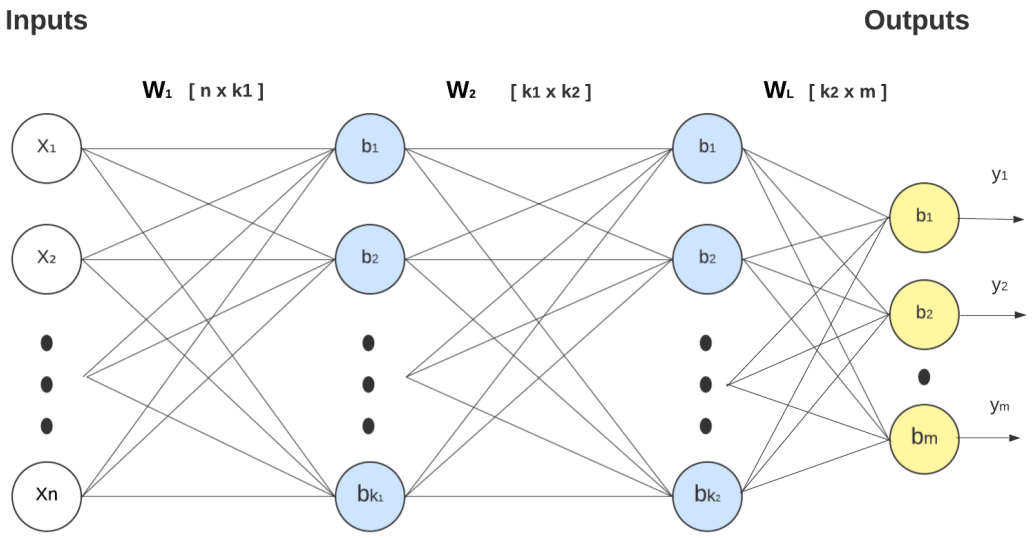


Figure 4.3 Representation of a neural network with depth L. *Source Own*

In Figure 4.3, the mathematical representation has not been included on purpose. In the following section 4.2 the mathematical process behind this structure will be explained. It is known as **forward propagation**.

Finally before advancing in to the optimization, which will be covered in detail later in the study, and before implementing the code, one last key feature is missing. Until this moment everything that the NN does is to concatenate linear transformations, but this is far from optimal for two reasons.

(a) Concatenating linear transformations is a linear transformation as well.

(b) The output can be any real number, but as a classification tool it is expected a value between 0 and 1.

For this reason, one extra function is needed. The **activation function** introduces non linearities to the system and transform the values to the (0,1) interval. Then if a layer performs the linear transformation $z = f(x;\theta)$ the activation function performs the transformation $y = g(z) = g(f(x;\theta))$.

It is interesting to mention that using sigmoid as output function, and without adding any hidden layers the one vs all model is recovered.

4.2 Propagation

In machine learning propagation is the word used to describe the computational process of a NN. It can be either a forward propagation or a backward propagation.

The forward propagation is when the input x traverses the network to the output to compute the predictions $\hat{y}$ and the cost $J(\theta)$. On the other hand the backward propagation, goes from the cost of the output layer and flows backwards through the network, in order to compute the gradient. From now on the notation used to represent the different characteristics of a NN will be present everywhere, the appendix A is a resume of this notation.

4.2.1 Forward propagation

The forward propagation is the act of concatenating the transformations to the input features until reaching the last layer output. From section 4.1.2, it is known that each layer performs a linear transformation. It was mentioned one last transformation (the activation function) was required. Without entering in detail, this transformation can be seen as a function $g(h) : \mathbb{R}^n \to \mathbb{R}^n$, therefore a layer performs the transformation of equation 9.

$$f^{(k)} = g(W_k^T \cdot x + b_k) \tag{9}$$

Then if the network has depth l, the forward propagation is defined in equation 10. Notice that this composition has to be resolved from the inside to the outside which means from the first layer to the last.

$$y = f^{(l)}(f^{(l-1)}(f^{(l-2)}(...))) \tag{10}$$

Using this idea a pseudo-code of how a forward propagation algorithm looks has been written. Algorithm 1 shows this, the function *activation_function()* is a non linear transformation to the output h at every layer. In the following sections different types of activation functions will be presented. It also appears another function called *cost_function()*, which was explained in section 3.2, it is enough to remember that it is a function which given the predictions of the

NN and the real targets, it returns the estimated error. This value will be of interest for the optimization process as the objective will be to reduce this error.

Note[1]: in pseudo-codes the $\times$ symbol is used to represent matrix multiplication and $\cdot$ symbol is used to represent element-wise multiplication.

Note[2]: to understand the notation of the variables and the sizes of the vectors/matrices refer to appendix A

Algorithm 1 Forward propagation

Require:
nL number of layers
θ parameters
X matrix of data inputs
y matrix of labels related to X
Ensure:
J training error
o output at every layer

1: $W, B \leftarrow extractWB(\theta)$
2: $o_1 \leftarrow X$
3: $i \leftarrow 2$
4: **while** $i \leq nL$ **do**
5: $\quad h \leftarrow o_{i-1} \times W_{i-1} + B_{i-1}$
6: $\quad o_i \leftarrow activation\_function(h)$
7: $\quad i \leftarrow i + 1$
8: $J \leftarrow cost\_function(o, y)$

4.2.2 Backward propagation

Backward propagation or usually called "backprop" is a strategy to compute the derivatives of a NN. This method is based on the application of the chain rule of calculus (equation 11).

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}f^{(l)}}{\mathrm{d}f^{(l-1)}} \frac{\mathrm{d}f^{(l-1)}}{\mathrm{d}f^{(l-2)}} \cdots \frac{\mathrm{d}f^{(1)}}{\mathrm{d}x} \tag{11}$$

A neural network with depth l computes forwardprop saving the linear transformations $h^{(k)}$ and activation $o^{(k)}$ at each layer. Afterwards the gradient of the cost is obtained in a reverse process applying equation 11. Starting in the last layer and applying the chain rule, the gradient of the cost respect to θ is the following:

$$\nabla_\theta J = \nabla_{\hat{y}} L(\hat{y}, y) \cdot \frac{\mathrm{d}\hat{y}}{\mathrm{d}h^{(l)}} \frac{\mathrm{d}h^{(l)}}{\mathrm{d}\theta} \tag{12}$$

Notice that $\hat{y}$ is indeed the activation function of the last layer $o^{(l)}$ and its derivative respect to θ is the derivative of the activation function times the derivative of the linear function. The derivative of the linear function is one respect to the biases and $h^{(l-1)}$ respect to the weights.

$$\frac{\mathrm{d}h^{(l)}}{\mathrm{d}\theta} = (\frac{\partial h^{(l)}}{\partial b}, \frac{\partial h^{(l)}}{\partial W}) = (1, h^{(l-1)}) \tag{13}$$

Equations 12 and 13 obtain the gradient of the last layer thetas. In order to propagate the error backwards the gradient of the cost respect to the activation function has to be multiplied by $W^{(k)}$ then another version of equation 12 (see eq 14) can be summoned along with equation 13.

$$\nabla_{\theta^{(k)}} J = \nabla_{o^{(k)}} J \cdot \frac{\mathrm{d}o^{(k)}}{\mathrm{d}h^{(k)}} \frac{\mathrm{d}h^{(k)}}{\mathrm{d}\theta^{(k)}} \qquad where \qquad \nabla_{o^{(k)}} J = W^{(k)T} \cdot \nabla_{o^{(k+1)}} J \tag{14}$$

This is the theory behind the backprop, the pseudo code of this method will help to make a deeper understanding around this concept. One last thing to add is that in deep learning these terms tend to be compressed into just two. One for the last derivative of the linear transformation (θ), and one for derivative of the cost respect the activation function (δ).

$$\delta^{(k)} = \nabla_{o^{(k)}} J = W^{(k)T} \cdot \delta^{(k+1)} \cdot \frac{\mathrm{d}o^{(k)}}{\mathrm{d}h^{(k)}} \tag{15}$$

Below is shown the algorithm 2 for backward propagation. The function $extractWB()$, decomposes all the parameters that are enclosed in θ to the weights and biases. On the other hand the function $reconstruct\_grad()$ does the opposite, builds a gradient with the same distribution as θ with W and B data structures given as inputs.

The back propagation algorithm is a really useful tool to compute the gradient of the cost function. Summarizing all the concepts developed through this section it is seen that back propagation is indeed a specific case of **autodifferentiation**.

Algorithm 2 Backward propagation

Require:
nL number of layers
θ parameters
X matrix of data inputs
y matrix of labels related to X
o outputs at each layer

Ensure:
$grad$ gradient of the cost function

$W, B \leftarrow extractWB(\theta)$
$i \leftarrow nL$
while $i \geq 2$ **do**
  $o' \leftarrow activation\_function\_derivative(o_i)$
  if i = nL **then**
    $aux \leftarrow negLogLikelihood\_derivative(y, o_{end}$
    $\delta_i \leftarrow aux \cdot o'$
  else
    $\delta_i \leftarrow (W_{i-1} \times \delta_{i-1}^T)^T \cdot o'$
  $gradW_{i-1} \leftarrow \frac{1}{nD} o_{i-1}^T \cdot \delta_i$
  $gradB_{i-1} \leftarrow \frac{1}{nD} \sum \delta_i$
  $i \leftarrow i - 1$
$grad \leftarrow reconstruct\_grad(gradW, gradB)$

4.3 Activation functions

The activation function (sometimes abbreviated as AF) is used to introduce non-linearity to the model. When this functions are applied to the main corpse of the network they are called **hidden units**, and when they are used in the output layer they are called **output units**. The latter are designed in a way their range is $R = \{[0, 1]\}$.

The NN usuallly uses the same AF for all of its neurons. This is not necessary always the case, sometimes different layers could be assigned different functions. There are dozens of different functions implemented in the literature. In spite of this, in this study only the most remarkable will be taken in to consideration.

4.3.1 The Sigmod function

The sigmoid function was introduced in the logistic regression and it is the activation function used par excellence in feedforward neural networks [10]. The sigmoid is a non-linear and bounded differentiable AF, defined for any real input, and with positive derivatives everywhere.

It is given by the relationship:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{16}$$

Sigmoid AF are really good for the output units of NN and in general for shallow networks. Its major drawbacks are sharp damp gradients, gradient saturation, slow convergence and bad behavior with small weights.

4.3.2 The hyperbolic tangent function

The hyperbolic tangent AF has a similar response to he sigmoid function (both of them are exponential functions). In Figure 4.4 this AF is represent along with the sigmoid function. The function is defined as:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{17}$$

The implementation of tanh function improves the training speed respect to sigmoid. Nevertheless, it has a big disadvantage as it only attains gradient of 1 when the input is 0. This produces some dead neurons as a result of zero gradient. This limitation pushed further the research of AF and it birthed the rectified linear unit or ReLU. [10]

Tanh overrides others in recurrent neural networks and natural language processing.

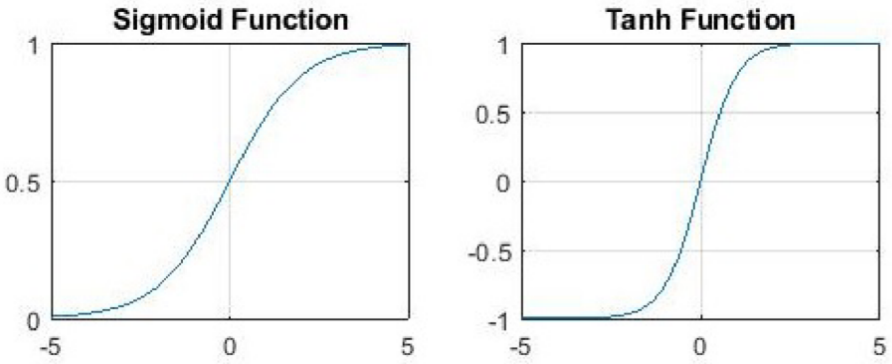


Figure 4.4 Representation of Sigmoid and Tanh response. *Source [10]*

4.3.3 The ReLU function

The rectified linear unit has been the most widely used AF since it was first proposed by Nair and Hinton in 2010 [2]. It offers better performance than sigmoid and tanh activation functions. The ReLU is almost a linear function and therefore preserves the property of easy optimization from linear models. It is defined as:

$$f(x) = \begin{cases} 0 & if \quad x < 0 \\ x & if \quad x \geq 0 \end{cases} \tag{18}$$

Equation 18 grants high computation speed avoiding exponentials and divisions. Moreover it introduces sparsity as it squishes the values between zero to its maximum. The downside of ReLU as AF is that it tends to overfit the model much more than sigmoid or tanh. Nonetheless the "dropout" technique works really well to reduce overfitting (Read [5] for more information on dropout regularization), but other regularization methods that will be discussed in section 5 can be used as well.

This activation function has been used in a handful of NN architectures. It is used in the hidden units with another AF used in the output layers. Typical examples of its application are classification and speech recognition NN.

In Figure 4.5 the three activation functions explored till this point are showed and compared.

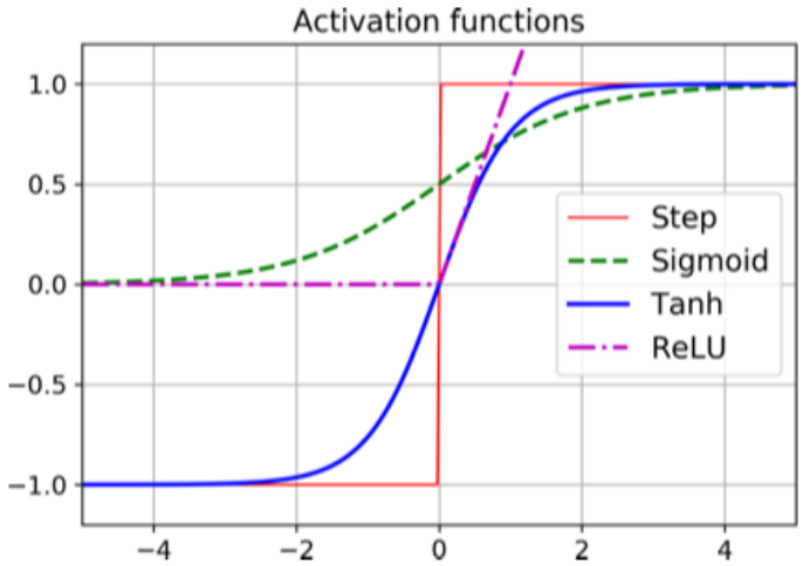


Figure 4.5 Comparison of ReLU, sigmoid and tanh AF. *Source [15]*

4.3.4 The softmax function

The softmax function is a normalized exponential function. For this reason it is normally only used in the output layer. It can be said that softmax is similar to both tanh and sigmoid, with the first one shares a similar exponential response, and with sigmoid shares the monopoly of being the 2 king output activation functions.

$$f(x) = \frac{e^{x_i}}{\sum_j e^{-x_j}} \tag{19}$$

The sofmatx outputs a vector of real numbers belonging to the interval (0,1) where the sum of all of them is equal to 1. This is useful for classification tasks where the data trying to be classified has clearly separated groups. With sigmoid it could be possible to obtain a result where the NN gives a certain input more than 50% probability for 2 different classes.

4.3.5 Other activation functions

In the previous sections the most known and used AF have been resumed. This does not mean that there are not other adequate activation functions for different tasks, but that those are the most reliable in general terms and most used for some reasons.

Other well known AF are the *Softsign*, *Softplus* or ELU functions [10]. This study will limit the implementation to the four stated in previous sections.

4.3.6 Output function selection

Regarding the selection of output AF there are two options, sigmoid and softmax. To identify in which case is preferable to use one over the other it is necessary to understand the difference between multi-class and multi-label classification.

Multi-class classification occurs when a dataset can be classified in different groups. For example, if there are different images of cats, dogs and horses and the task is to classify between them. In this scenario a cat image can be classified only as a cat, not as dog neither as a horse.

On the other hand, multi-label classification occurs when the dataset contains different groups but they are not clearly separated. Following the example of the cat/dog/horse classification, now a picture could contain a cat and a dog, or a cat and a horse, or any combination. In this case the image has to be classified with more than one group because it contains both a cat and a dog.

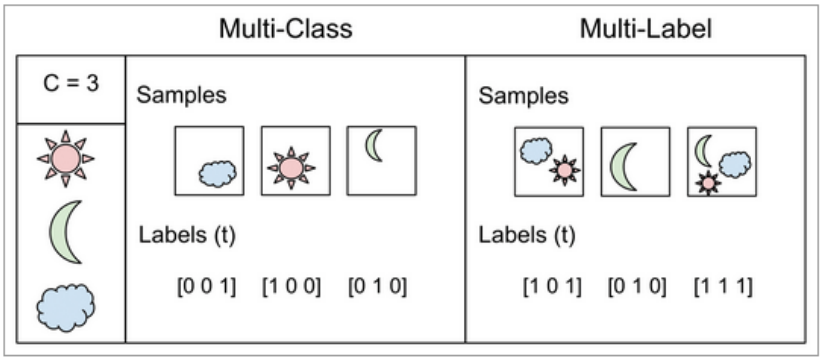


Figure 4.6 Multi-class and Multi-label classification *Source [8]*

Figure 4.6 shows an easy example of multi-class and multi-label classification. Understanding this difference it is seen that softmax is more suitable for multi-class and that sigmoid is more suitable for multi-label. Although sigmoid is better in multi-label, it can be used in multi-class as well, not like softmax which would be inadequate for multi-label.

4.3.7 Hidden function selection

To choose the hidden units, there are three critical characteristics of activation functions to consider:

- Saturation
- Zero-centering
- Computational expense

Saturation refers to the fact that for large values the function gets a 0 gradient. Both sigmoid and tanh saturate for $x \to \pm\infty$. However ReLU does not saturate in the positive region.

Zero-centered functions are those which are defined with both positive and negative gradients. Tanh is zero-centered, but sigmoid and ReLU are not. This means that sigmoid and ReLU will always have a completely positive or completely negative $\nabla\theta$, nevertheless tanh could have both positive and negative derivatives in different directions.

An activation function is said to be computationally expensive if it contains exponentials, divisions or other complex functions. Tanh and sigmoid can be considered computationally expensive and ReLU is not [14, 13]. The table below shows a compact resume of these three properties for the activation functions in study.

	Saturation	Zero-centered	Comp. expense
sigmoid	✗	✗	✗
tanh	✗	✓	✗
ReLU	~	✗	✓

Table 1 General properties of the mentioned activation functions. *Source Own*

With this in mind there seems to be no reason in favour of using sigmoid. And although ReLU is the one used nowadays, tanh can be casted in some specific cases.

4.4 The learning process

Once everything in the network is settled, is time to obtain the optimum parameters. The learning process of a NN is based on the idea of the minimization of a cost function $J(\theta)$. This process is usually made in an iterative way; at first some random parameters of θ are initialized, with those the cost is computed via forwardprop and the gradient via backprop. With this information the θ is updated expecting a lower value of J. This process is repeated till a local minimum is found.

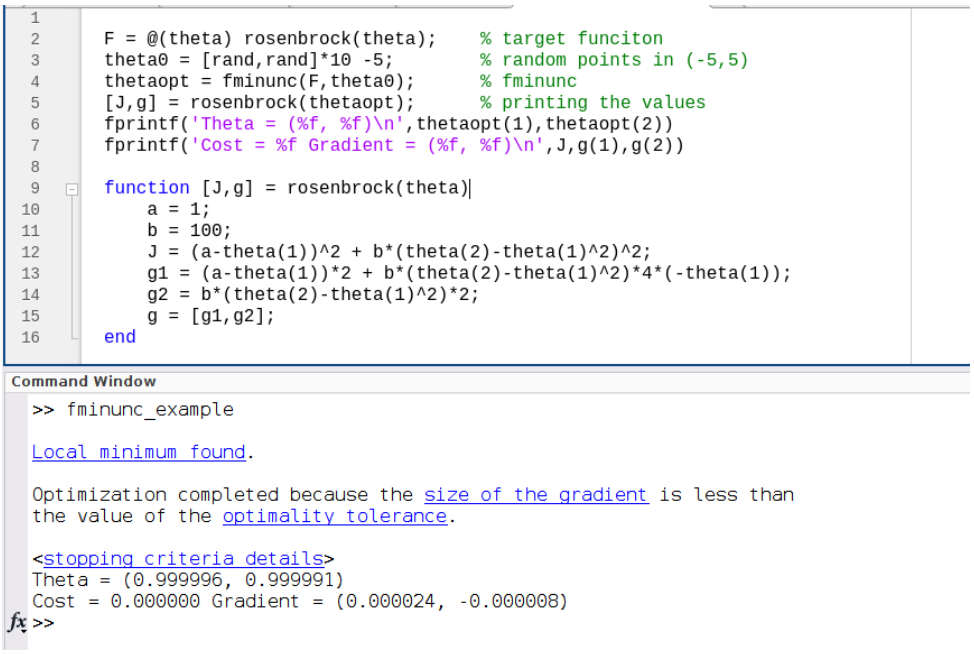
There are several approaches in the literature on how to tackle the learning process, differing ones from others in how the θ is updated. They are classified in:

- **First-order optimization algorithms**
- **Second-order optimization algorithms**

Although the second-order optimization algorithms use the hessian matrix to update θ and are usually faster converging to a local minimum, they are far from ideal due to their tendency on getting stuck near saddle points. This disadvantage added to the higher computation cost make the first-order algorithms or just **gradient based** methods more suitable for deep learning.

Later in this study this gradient based methods will be further explained and implemented to the NN. But at first, in order to make the implementation easier, the function *fminunc()* from matlab will be used.

Given a handle function $F(\theta)$ and an initial point θ_0, *fminunc(F,*θ_0*)* will return the optimum set of parameters θ_{opt}. In Figure 4.7 the fminunc has been used to minimize the rosenbrock function. From the theory it is known this function has a global minimum at (a, a^2).



```
F = @(theta) rosenbrock(theta);    % target funciton
theta0 = [rand,rand]*10 -5;        % random points in (-5,5)
thetaopt = fminunc(F,theta0);      % fminunc
[J,g] = rosenbrock(thetaopt);      % printing the values
fprintf('Theta = (%f, %f)\n',thetaopt(1),thetaopt(2))
fprintf('Cost = %f Gradient = (%f, %f)\n',J,g(1),g(2))

function [J,g] = rosenbrock(theta)
    a = 1;
    b = 100;
    J = (a-theta(1))^2 + b*(theta(2)-theta(1)^2)^2;
    g1 = (a-theta(1))*2 + b*(theta(2)-theta(1)^2)*4*(-theta(1));
    g2 = b*(theta(2)-theta(1)^2)*2;
    g = [g1,g2];
end
```

```
Command Window
>> fminunc_example

Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.

<stopping criteria details>
Theta = (0.999996, 0.999991)
Cost = 0.000000 Gradient = (0.000024, -0.000008)
fx >>
```

Figure 4.7 Minimization of rosenbrock function using fminunc. *Source Own*

4.5 Implementation of feedforward neural networks

The aim of this section is to help the comprehension of how a feedforward neural network can be implemented. A pseudo-code of the implementation along with some graphs showing the results for simple datasets are presented for this purpose.

A NN is composed by all the algorithms explained throughout section 4, these are forward propagation, backward propagation, and minimization. Unless otherwise specified the cost function will be -loglikelihood (equation 3), and the activation function can be anyone from the ones in section 4.3.

Algorithm 3, shows how a general NN works; it uses the $forwardprop()$ and the $backprop()$ functions (algorithms 1, 2). The minimization used is the *fminunc* function from matlab for now. This will make the implementation of the NN will look a little different in the future when the minimization is self-coded. To see how *fminunc* is used relate to figure 4.7.

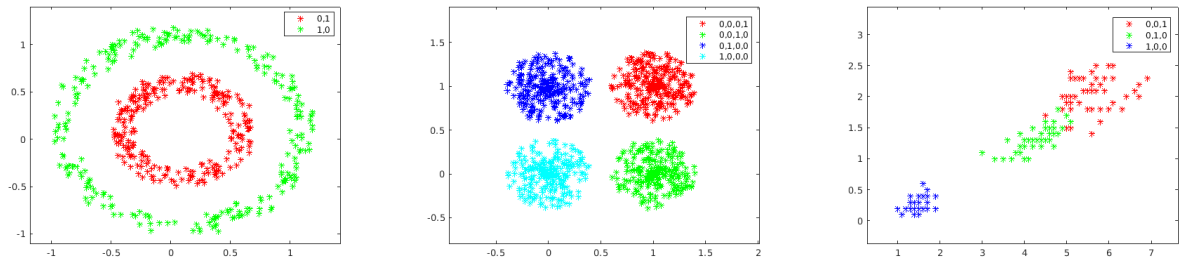
Algorithm 3 Feedforward Neural Network (fmin)

Require:
nL number of layers
θ_0 initial parameters
X matrix of data inputs
y matrix of labels related to X

Ensure:
θ_{OPT} optimum parameters

1: $F(\theta) \leftarrow Cost\_Gradient(\theta)$
2: $J, o \leftarrow forwardprop(nL, \theta, X, y)$
3: $grad \leftarrow backprop(nL, \theta, X, y, o)$
4: $\theta_{OPT} \leftarrow fminunc(F, \theta_0)$

In Figure 4.8 three simple datasets that will be used through this project to show the performance of the computer algorithms are depicted. Their principal characteristics are shown in table 2, more information can be found in appendix B. Then using algorithm 3, this procedure has been implemented (for the whole development of the codes relate to ANNEX I). Using **sigmoid** in the output and hidden units and a **network structure** of [4, 8, x] neurons the boundaries found by the NN are presented in Figure 4.9.

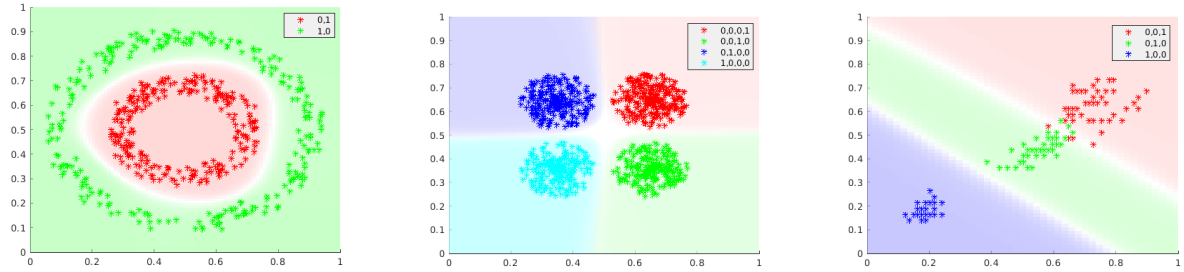


(a) Self-made daset *"ConCricles"* **(b)** Self-made dataset *"4circles"* **(c)** Iris datsets *"Iris"*

Figure 4.8 Three simple datasets used to train the feedforward neural network. *Source Own*

	ConCircles	4circles	Iris
Points	500	1000	150
Groups	2	4	3
Features	2	2	4

Table 2 Characteristics of *ConCircles, 4circels* and *Iris*. *Source Own*

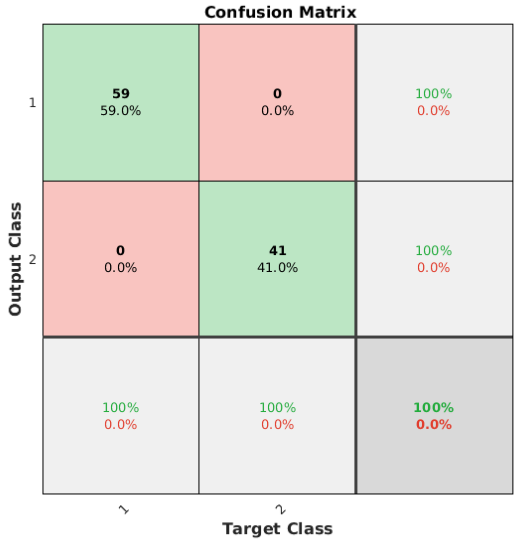


(a) ConCircels boundary plot **(b)** 4circles boundary plot **(c)** Iris boundary plot

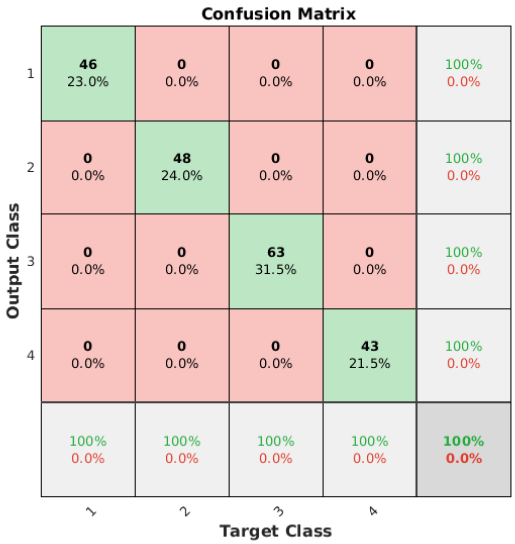
Figure 4.9 Boundaries found by the NN when trained with the data. *Source Own*

Different colours are used to represent different classes in the dataset. The task of the NN is to separate them correctly and in order to see what the algorithm decides the boundaries are plotted. All the 2D region is filled with the same but more transparent colours, representing the regions that the model believes belong to the different groups. The whiter sections are the boundaries between regions, where the model "thinks" the point could belong to either group.

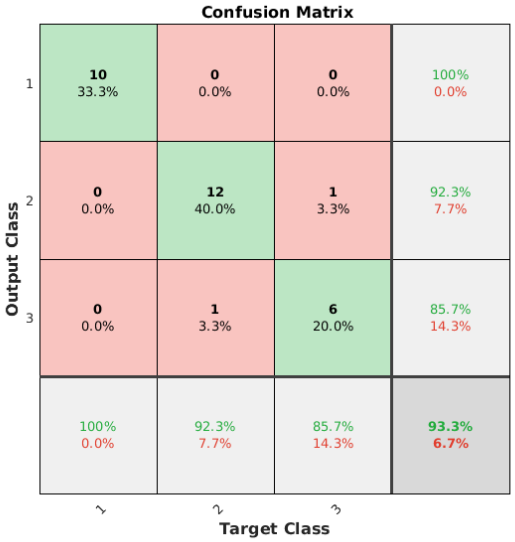
With this simple examples it does not make sense yet to study the different performance of the activation functions. Nevertheless, in section 7 more complex datasets will be used and the different activation functions will be put to test. All the training sessions are performed with a 20% test ratio, which means that 20% of the data is stored separately to test the error of the NN afterwards. This procedure is commonly used in machine learning to ensure the model does not overfit to the points it used for training. Another spread tool used is the confusion matrix which shows the test error in a matrix with columns and rows representing real and predicted values respectively. The confusion matrices for the three examples above are shown in Figure 4.10.



(a) ConCircels



(b) 4circles



(c) Iris

Figure 4.10 Confusion matrix for each of the previous examples. *Source Own*

Figures 4.9 and 4.10 show good results, the classification regions seem to be correctly placed and the test error is 0% for the two custom datasets and only 6% for the iris one. These results are satisfying for this section.

5 Regularization

Regularization techniques are introduced in machine learning to prevent overfitting. When there is few data or the model is deep enough, it is often the case that the model is capable of memorizing all the training data and hypothetically reaching a loss function of 0. This can be far from ideal as the model will not predict new outputs outside the training set really well, in this case we say the model overfits. There exist a wide range of regularization techniques, but the most used are:

- Parameter penalties
- Early Stopping
- Dropout

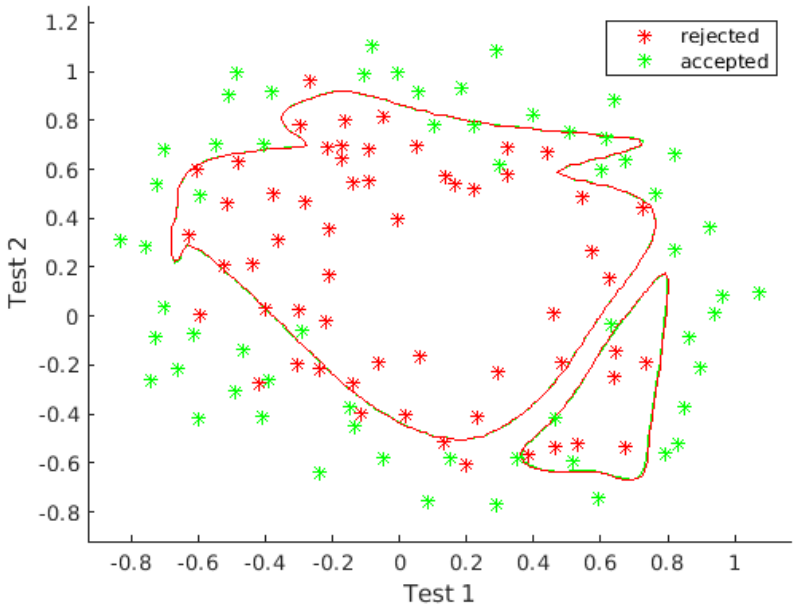


Figure 5.1 Example of overfitting in the *microchip* dataset. *Source Own*

5.1 Parameter penalties

The norm penalty regularization is based on limiting the capacity of the model by adding to the loss function a regularization term. This term is multiplied by a constant hyperparameter λ which weights the contribution of the term.

$$J(\theta) = L(\theta) + \lambda \cdot R(\theta) \tag{20}$$

The two most common penalties are L^2 and L^1:

$$R(\theta) = \frac{\lambda}{2} w^T w \tag{21}$$

$$R(\theta) = \lambda ||w||_1 \tag{22}$$

The first one is known as weight decay, it causes the weights to move closer to the origin, this property scales inversely with the curvature of the loss function. In the directions of minimum curvature the weights are pushed closer to the origin than in other directions. The figure below tries to show how L^2 regularization works in a function with 2 weights.

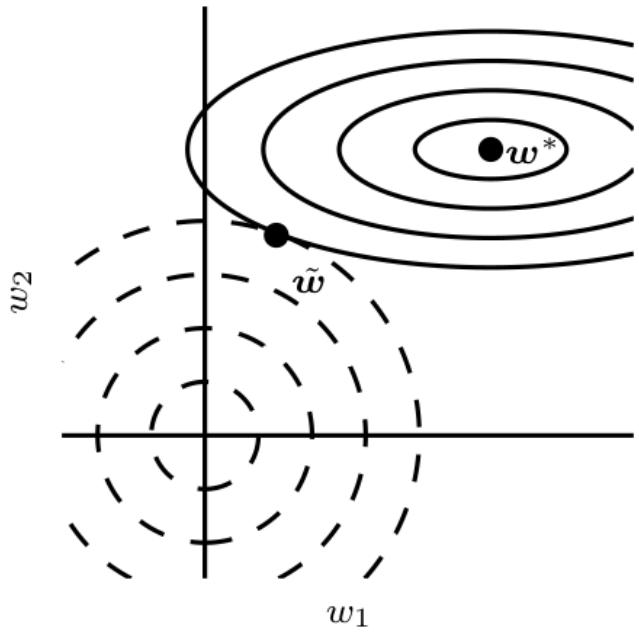


Figure 5.2 The effect of L^2 regularization. The dashed lines represent contours of L^2 while the continuous ones represent contour lines of the loss function. w^* is the minimum of the loss function but due to the influence of the L^2 regularization $\tilde{w}$ is the final optimal. *Source [6]*

On the other hand the L^1 regularization adds the same constant penalty to all the weights in the model which surpass a certain boundary value while the rest are pushed to 0. This regularization results in a more sparse solution as some of the weights are turned to 0. This sparsity has proved to be a useful property as a feature selection mechanism in different AIs.

5.2 Early Stopping

Early stopping is a different regularization method which does not rely on the modification of the cost function. In fact, what it does is to check the test loss every epoch. Although the training loss could get smaller, if after a certain number of epochs the test loss has not been reduced the minimization is stopped. With this it is ensured that the model does not continue minimizing the cost function with the training set at expense of augmenting the test loss.

In this method the analogue hyperparameter to λ in the penalty strategy is the optimum number of epochs/iterations before overfitting.

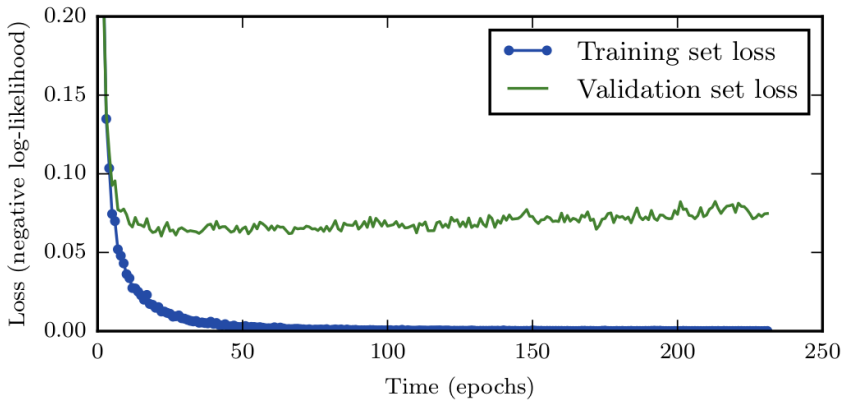


Figure 5.3 Learning curves showing the optimum point to stop before overfitting. *Source [6]*

From the figure above is clearly seen that is much better to stop the minimization before the validation error starts to grow. This is done by storing the value of the parameters in the lowest validation error point, and after few iterations of non improvement in validation error rescue this value as the optimum found.

5.3 Dropout

The last technique is the dropout. This is relatively new technique (Srivastava et al., 2014) compared to the others. It consists of training n different models with different combinations of a "base" model. Suppressing neurons to see the impact on the performance of the NN, in Figure 5.4 a scheme of all sub-networks that are constructed by removing non-output units from a base network is shown. The implementation of this technique is more complex and out of the scope of this study.

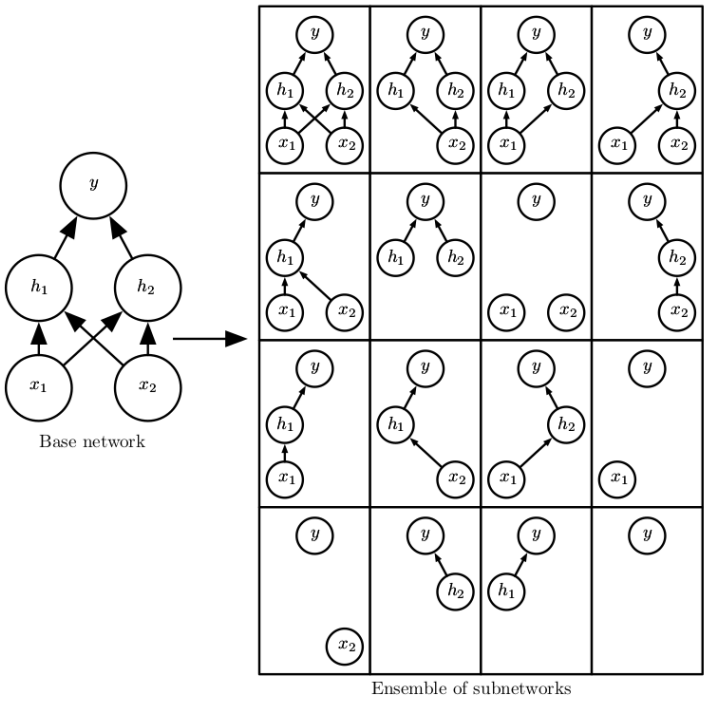


Figure 5.4 Sub-networks formed from a base network. *Source [6]*

6 Optimization methods

Optimization is a branch of mathematics which focuses in the labour of finding the best possible parameters in a function or model. In the simplest case, optimization consists of maximizing or minimizing a mathematical function. The task of the NN is to learn to classify between different groups using a set of inputs, this process is done by modelling the error of the model using a cost function (for more information relate to 3.2) and then trying to minimize this error respect to the parameters θ.

The cost function $J(\theta)$ is a scalar multi-variable function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, it depends on a set of parameters $\theta_1, \theta_2 ... \theta_n$ and returns a real number that represents the error of the predictions. This functions have one partial derivative for each parameter and these are enclosed in the term named gradient $\nabla J(\theta) = [\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, ...]$, therefore the gradient is a vector that represents the ascending line in the maximum steep direction. First-order optimization methods use this property to update the parameters in the optimal direction from iteration to iteration. On the other hand, second-order methods apart from the gradient use the hessian matrix (second order derivatives) to update the parameters bearing in mind the curvature of the function.

Although it is possible to use second-order methods, in deep learning first-order methods are more predominant, probably because the benefits of the second derivatives are simply not worth the computation expense. First-order algorithms give faster results using rough but good enough approximations.

6.1 Gradient descent

In section 2.4 descent methods were briefly introduced. The gradient descent is a first-order algorithm really easy to implement. The idea behind it is to slowly update the parameters of the function in the direction of the gradient (equation 23).

$$x' = x - \epsilon \cdot \nabla f(x) \tag{23}$$

The negative sign comes from the intention to minimize $f(x)$ (contrary to maximize which would be the positive direction of the gradient). The other term in the equation is the step size ϵ. In machine learning this is a hyper-parameter called **learning rate**. The selection of an appropriate learning rate is crucial and will be strongly correlated to the performance of the optimization process. There are two approaches when selecting the LR, it is possible to either leave it static (usually small constant) or to change it over time. The latter is called **adaptive learning rate**. It is also possible to perform a **line search**, evaluate few possible ϵ and choose the one that results in the smallest objective function value.

Algorithm 4 shows how a general gradient descent looks like. It has a main *While* loop with an arbitrary optimization criteria and it calls the $lineSearch()$ function which finds the best LR between a small sample. It could be possible to make the line-search return always a small constant, therefore simplifying it to a static method. A typical optimization criteria is to use the norm of the gradient, if it is close to 0 it means the function at least reached a local minimum. Again the notation used is the same as the one exposed in appendix A.

Algorithm 4 Gradient Descent applied to NN

Require:
ϵ_0 initial learning rate
θ_0 initial parameters
Ensure:
θ_{OPT} optimum parameters

1: $\theta \leftarrow \theta_0$
2: **while** stopping criteria not met **do**
3: $J, o \leftarrow forwardprop(nL, \theta, X, y)$
4: $grad \leftarrow backprop(nL, \theta, X, y, o)$
5: $\epsilon_k \leftarrow lineSearch(args, \epsilon_k)$
6: $\theta \leftarrow \theta - \epsilon \cdot grad$
7: $\theta_{OPT} \leftarrow \theta$

6.2 Stochastic Gradient Descent

Stochastic Gradient Descent (abbreviated SGD) is a variation of the gradient descent. The need to modify algorithm 4 comes from the fact that gradient descent algorithms need to compute the derivative of the function every iteration. This might not be problematic for a standard mathematical function, however, in deep learning the gradient needs to take the sum over all the training points (in big models this can be thousands or millions). This is a hindrance to obtain a practical and fast optimization.

The concept is simple, if you need to compute the gradient over millions of points just take a few hundred as an estimation. According to Goodfellow, et al. [6] optimization methods that update the parameters using only the gradient are robust and can handle smaller batch sizes like 100, second-order methods typically require much larger batch sizes like 10000.

Algorithm 5 is a variation of algorithm 4, where the difference is in line 3. A function called minibatch selects a small sample of points from the entire dataset. For the algorithm to make sense, the function minibatch should not repeat the points from the dataset until all of them are used, one pass through all the points in the dataset is called epoch.

Algorithm 5 Stochastic Gradient Descent applied to NN

Require:
ϵ_0 initial learning rate
θ_0 initial parameters
I batch size
Ensure:
θ_{OPT} optimum parameters

1: $\theta \leftarrow \theta_0$
2: **while** stopping criteria not met **do**
3: $y_b, X_b \leftarrow minibatch(y, X, I)$
4: $J, o \leftarrow forwardprop(nL, \theta, X_b, y_b)$
5: $grad \leftarrow backprop(nL, \theta, X_b, y_b, o)$
6: $\epsilon_k \leftarrow lineSearch(args, \epsilon_k)$
7: $\theta \leftarrow \theta - \epsilon \cdot grad$
8: $\theta_{OPT} \leftarrow \theta$

6.3 Stochastic Gradient Descent with momentum

Stochastic Gradient Descent with momentum is a variation of the traditional SGD trying to improve its performance. The novelty is the incorporation of a new hyperparameter α with the purpose to consider previous gradients. If after few iterations the gradient keeps pointing in the same direction, the step size is reinforced and therefore the optimization accelerated. In contrast if the gradient is different at each iteration the step size stays the same or even decays.

It is called "momentum" because it has a simple analogy to the Newtonian momentum theory, more precisely the Stokes law. It resembles a particle moving inside a fluid with viscous drag. Applying the second law of Newton (equation 24) it is known that the variation of the velocity of a particle depends on the external forces to the system. In our analogy there would be 2 forces; the first one is proportional to the negative gradient of the cost function ($-\nabla J(\theta)$) and accelerates the particle, and the second one is the viscous drag proportional to the velocity ($-v$), decelerating the particle. From the Stokes law it is known that a system with this properties will accelerate until reaching a terminal velocity.

$$\sum f = \frac{\partial^2 \theta}{\partial t^2} = \frac{\partial v}{\partial t} \tag{24}$$

Updating the parameters would look like the following:

$$\begin{aligned} v &= \alpha \cdot v - \epsilon \cdot \nabla J(\theta) \\ \theta &= \theta + v \end{aligned} \tag{25}$$

The size of the step depends on the variation of the velocity which is related to the gradients pointing in the same direction successively. It is helpful to think of the momentum hyperparameter as $\frac{1}{1-\alpha}$, because if the gradient is always the same, the step size will accelearte until reaching a terminal velocity of.

$$\frac{\epsilon||grad||}{1-\alpha} \tag{26}$$

6.3.1 Nesterov Momentum

In 2013 Sutskever et al. (2013)[4] introduced a modification to the momentum algorithm. The update rules are the same as the momentum algorithm, the difference lies in where the gradient is evaluated. In Nesterov momentum the update is evaluated after the current velocity is applied. In the convex batch gradient case, Nesterov modification reduces the rate of convergence to $O(1/k^2)$. Unfortunately, in the stochastic gradient case it does not improve the rate of convergence.

Algorithm 6 shows the differences that the momentum algorithm introduces to the typical SGD. Line 4, highlighted in blue, is the Nesterov variation which comptues an interim update of θ before computing the gradient. Without this line the algorithm would become a normal SGD with momentum.

Algorithm 6 SGD with momentum applied to NN

Require:
- ϵ learning rate
- α momentum parameter
- v initial velocity
- θ_0 initial parameters
- I batch size

Ensure:
- θ_{OPT} optimum parameters

1: $\theta \leftarrow \theta_0$
2: **while** stopping criteria not met **do**
3: $\quad y_b, X_b \leftarrow minibatch(y, X, I)$
4: $\quad \tilde{\theta} \leftarrow \theta + \alpha \cdot v$
5: $\quad J, o \leftarrow forwardprop(nL, \theta, X_b, y_b)$
6: $\quad grad \leftarrow backprop(nL, \theta, X_b, y_b, o)$
7: $\quad v \leftarrow \alpha \cdot v - \epsilon \cdot grad$
8: $\quad \theta \leftarrow \theta + v$
9: $\theta_{OPT} \leftarrow \theta$

6.4 AdaGrad

All algorithms previously studied in this section had the same learning rate in all of their directions. The family of algorithms with adaptive learning rates were born with the idea to overcome the problem of setting the same learning rate in 2 directions with very different steepness.

AdaGrad is yet one of the simplest but most used methods with adaptive learning rate. This algorithm individually adapts the learning rate in every direction by scaling them inversely proportional to the square root of the sum of the previous squared values. Hence, the parameters with steep gradients have smaller learning rates while the parameters with flat gradients have bigger learning rates.

$$r = r + \nabla J(\theta)^2 \tag{27}$$

$$\theta = \theta - \frac{\epsilon}{\delta + \sqrt{r}} \cdot \nabla J(\theta) \tag{28}$$

Where δ is a small constant (usually 10^{-7}) to prevent a division by 0, and r is the cumulative sum of the squared gradients. All the operations are applied element-wise.

Algorithm 7 AdaGrad NN

Require:
ϵ learning rate
θ_0 initial parameters
I batch size

Ensure:
θ_{OPT} optimum parameters

$\theta \leftarrow \theta_0$
while stopping criteria not met **do**
  $y_b, X_b \leftarrow minibatch(y, X, I)$
  $J, o \leftarrow forwardprop(nL, \theta, X_b, y_b)$
  $grad \leftarrow backprop(nL, \theta, X_b, y_b, o)$
  $r \leftarrow r + grad^2$
  $\theta \leftarrow \theta + \frac{\epsilon}{\delta + \sqrt{(r)}} \cdot grad$
$\theta_{OPT} \leftarrow \theta$

6.4.1 RMSProp

The AdaGrad algorithm is designed to improve the regular SGD in the convex problem. However, in a non convex problem taking into consideration gradients from past convex bowls that are not at the global minima AdaGrad could make the learning rate too small before arriving to the desired function value. RMSProp is a variation of AdaGrad which introduces yet another hyperparameter ρ in order to discard previous gradients. It uses an exponentially decaying average of the gradients instead of the original value. Empirically RMSProp has been shown to be an effective algorithm and it is currently one of the go-to methods [6].

Another thing that it has not been mentioned is that both AdaGrad and RMSProp can be implemented with SGD or Momentum methods.

Algorithm 8 RMSProp NN

Require:
ϵ learning rate
θ_0 initial parameters
I batch size

Ensure:
θ_{OPT} optimum parameters

1: $\theta \leftarrow \theta_0$
2: **while** stopping criteria not met **do**
3: $y_b, X_b \leftarrow minibatch(y, X, I)$
4: $J, o \leftarrow forwardprop(nL, \theta, X_b, y_b)$
5: $grad \leftarrow backprop(nL, \theta, X_b, y_b, o)$
6: $r \leftarrow \rho r + (1 - \rho) grad^2$
7: $\theta \leftarrow \theta + \frac{\epsilon}{\delta + \sqrt{(r)}} \cdot grad$
8: $\theta_{OPT} \leftarrow \theta$

7 Results

The final phase of this study is to analyze the performance of the different techniques developed throughout this thesis. Four topics will be debated: the influence of different activation functions, the performance of two regularization techniques, a batch size analysis for different optimization algorithms, and the hyperparameters from the optimization algorithms.

Up to this point, the datasets used have been relatively "small" with few features. To make this part of the study more realistic a larger dataset has been selected. *MNIST* is a world-known dataset containing 10,000 gray-scale images of 28x28 pixels (784 features) representing hand-written digits (see figure 7.1). The advantage of choosing a "classic" dataset is that lots of information can be found on the internet. The original MNIST dataset is already separated in training-test and it consists of 60,000 images in the training set and 10,000 in the test set, however, in this project the test set has been used as a whole and has been divided 80/20. This has been done to reduce possible problems of memory and time.

Lastly, after these 4 analysis are done, and using the best characteristics based on the previous results the code developed will be put to test against the MNIST and the Flower databases.

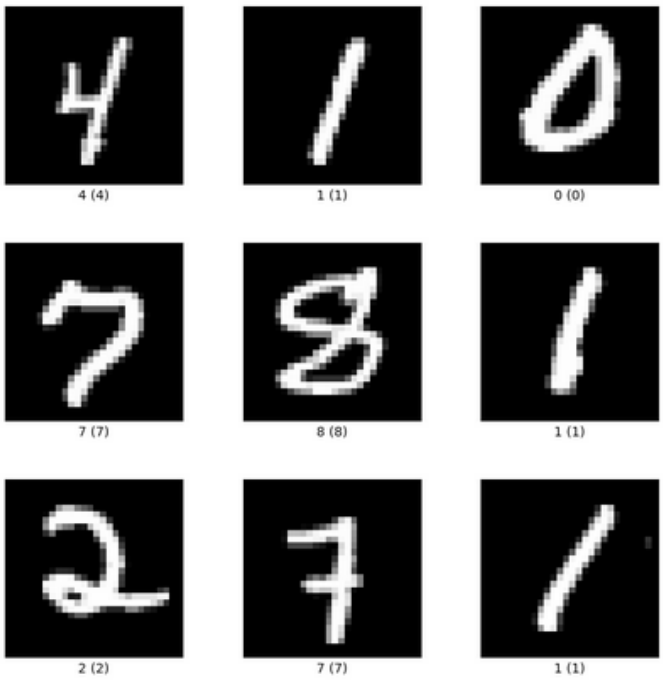


Figure 7.1 *MNIST* dataset. *Source [17]*

7.1 Activation Function comparison

Although it is possible to use different activation functions for different layers, and even for different neurons, the NN studied will only use 1 type of activation function for all the hidden units and 1 for the output layer.

Hidden units

Using softmax output unit, which is the common activation function for multi-class classification, and using the same general architecture and minimization method, three networks will be trained using: sigmoid, tanh and ReLU AFs. The parameters that will be tracked through the minimization are speed, training loss and test loss.

The general scheme of the network trained is shown in table 3:

Topology		**Optimization**		**Regularization**	
ANN		SGD		Early stopping	
Architecture:	784:500:150:10	Learning Rate:	0.01	L2 (lambda)	None
Output units:	Softmax	Momentum param:	None	Early Stopping:	10 epoch
Hidden units:	analysis	Mini-batch:	200	Stopping criteria:	↑ ↑ ↑ ↑

Table 3 Network composition for the activation function analysis. *Source Own*

The architecture has been selected like this in order to be able to compare the results with LeCun et al. 1998 [1]. In this article a NN of 500:150 hidden units is trained and it eventually reached an accuracy of 97.05% in the test set.

		time [s]	**funciton value**	**Test error [%]**
Sigmoid	mean	242	0.5342	15.1
	std	87.5	0.0771	0.0771
tanh	mean	113	0.1965	8.3
	std	28.7	0.0511	0.0040
ReLU	mean	115	0.1412	7.2
	std	31	0.0448	0.0056

Table 4 Hidden unit analysis (mean and standard deviation for 10 samples). *Source Own*

The results shown in table 4 are similar to the expected. The sigmoid activation function performed clearly worse than the other 2, both in computation time and accuracy. Regarding tanh and ReLU the results are not conclusive as both have similar times and accuracy, though ReLU performed slightly better in accuracy. These results are in line with the historical tendency of NN, where sigmoid was the used function at first, then it was changed for the tanh and finally ReLU appeared more or less in 2013. Since then ReLU is the most common choice in the models built nowadays.

Output units

On the other hand, using the previous AF ReLU, two networks with sigmoid and softmaxt output functions will be compared. This change should not have the same impact as the hidden activation function selection, though it could make the boundaries look quite different. The networks have the same characteristics as the ones described in the table 3.

		time [s]	**funciton value**	**Test error [%]**
Sigmoid	mean	123	0.2811	6.7
	std	23.5	0.0759	0.005
Softmax	mean	115	0.1412	7.2
	std	31	0.0448	0.0056

Table 5 Output unit analysis (mean and standard deviation for 10 samples). *Source Own*

Again, really similar results between the 2 networks. The sigmoid obtained 0.5 % less test error at the expense of some extra computation time. The results are not conclusive, however, as it has been explained before, sigmoid does not make much sense in multi-class classification. For this reason the softmax will be preferred in multi-class classification.

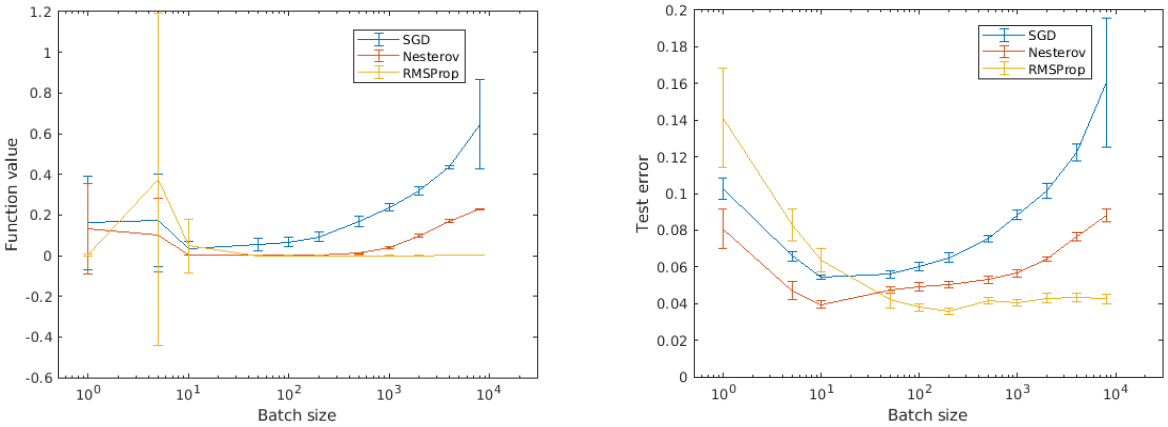
7.2 Batch Size analysis

The next step is to put to test the performance of different NN using fminunc, gradient descent, stochastic gradient descent, Nesterov momentum and RMSProp. Two comparisons will be made, in one a fixed time of 120s will be given and the accuracy reached will be the indicator, on the other all the algorithms will aim for a function value of 0.01 and the time of computation will be tracked . Other factors that will be tracked are: epochs, iterations, and test accuracy. It is important to understand the difference between epochs and iterations; the first represents the number of passes through all the dataset before stopping, and the second one represents the number of parameter updates ($iterations \approx nData/batchSize \cdot epochs$).

The general scheme of the networks trained is shown in table 6:

Topology		**Optimization**		**Regularization**	
ANN		SGD/Nesterov/RMSProp		None	
Architecture:	784:500:150:10	Learning Rate:	0.01	L2 (lambda)	None
Output units:	Softmax	Momentum param:	0.9	Early Stopping:	None
Hidden units:	ReLU	Mini-batch:	analysis	Stopping criteria:	120s/0.01fv

Table 6 Network composition for the batch size analysis. *Source Own*



(a) Function value vs batch size

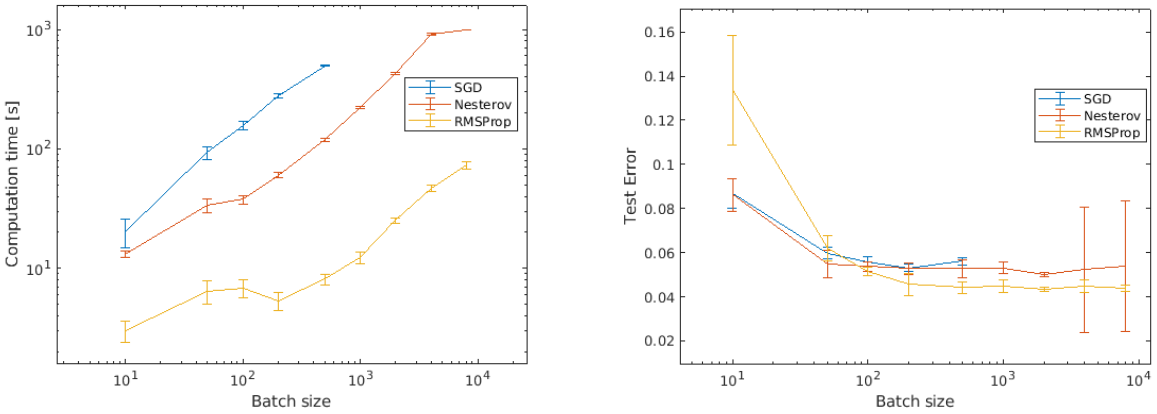
(b) Test error

Figure 7.2 Cost function and its test error for different batch sizes with a fixed minimization time of 120s (mean and standard deviation with sample of 10 cases). *Source Own*

First of all, it is remarkable that *fminunc* crashed due to insufficient memory. The algorithm demanded 2.2 TB of RAM which is obviously unrealistic and strengthens the idea of using gradient descent algorithms.

Figure 7.2 shows the results with a fixed minimization time. Although the time was fixed RMSProp did exit the minimization before the imposed time several occasions, this was due to passing the optimalty tolerance (norm of the gradient) which was set to 10^{-10}, it is understood that past this point it does not make sense to continue minimizing the cost function.

The results shown in Figure 7.2b point in the direction that Nesterov momentum worked similarly to SGD but with greater performance, both had a peak in their results between 10 to 100 points in the mini-batch. Differently, the RMSProp had worse results for these batch sizes but surpassed the previous for larger sets. Interestingly RMSProp did not drop its performance with big batch sizes, the comfort working zone is from 100 to the whole 8,000 points. All in all, as the general literature recommends the best possible configuration might be few hundreds of points to fortify the gradient against possible outliers, as shown in Figure 7.2a the function value had really high standard deviations for less than 10 points. Two hundred points is a reasonable amount without compromising too much computation time that worked well for the 3 algorithms.



(a) Time vs batch size

(b) Test error

Figure 7.3 Computation time and its test error for different batch sizes with an objective function value of 0.01 (mean and standard deviation with sample of 10 cases). *Source Own*

The results in Figure 7.3 show the same tendency as the ones obtained in Figure 7.2. Looking at Figure 7.3a RMSProp performed the best by far with the least computation time, Nesterov also performed better than SGD which only reached the 500 mini-batch size before exiting by the limit time which was set to 1000s (approx 16 minutes). The figure of test error does not give any important information as the minimization criteria was based on a function value which is strongly correlated to the test accuracy, therefore all of them have similar results except for the batch size of 10 which gives really unstable exits when using a function value criteria. For this same reason the 1 and 5 sizes were excluded from these graphs.

7.3 Optimization parameters analysis

The last analisys to the MNIST dataset will be the effect of the yhperparameters from the optimization algorithms. The learning rate in the SGD, the momentum parameter in Nesterov, and the decay parameter in RMSProp.

Learning rate

The learning rate is a really important parameter that might give headaches when choosing it. A too small learning rate can slow down the optimization and be a hindrance to overcome local minimums. A too large learning rate might become unstable and never reach a minimum at all.

Moreover the selection of learning rate is specific to every minimization scenario, meaning this that the optimum learning rate for a certain network might be not the best for other networks. For the case of the MNIST dataset and with the network described in table 7, the minimization of the cost function has been plotted against the passes through the dataset.

Topology		Optimization		Regularization	
ANN		SGD/Nesterov/RMSProp		None	
Architecture:	784:500:150:10	Learning Rate:	analysis	L2 (lambda)	None
Output units:	Softmax	Momentum param:	analysis	Early Stopping:	None
Hidden units:	ReLU	Mini-batch:	200	Stopping criteria:	30 epochs

Table 7 Network composition for the optimization parameters analysis. *Source Own*

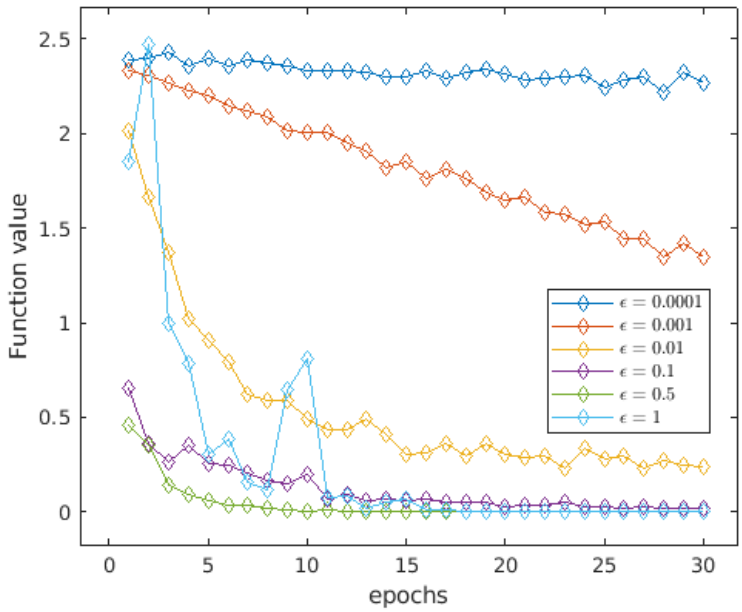


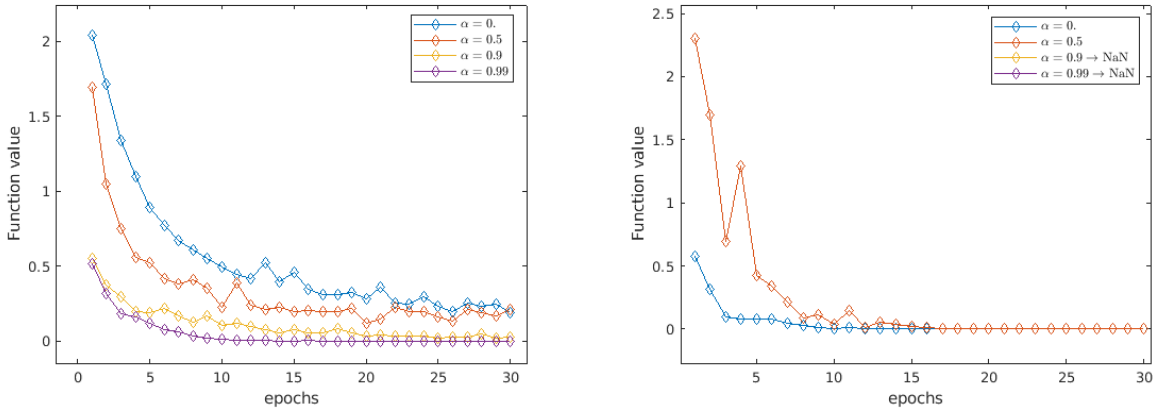
Figure 7.4 Different minimization curves of the cost function for different learning rates. *Source Own*

In Figure 7.4 it is seen that the learning rates of 10^{-4} and 10^{-3} are too small in the convergence. From 10^{-2} until $5 \cdot 10^{-1}$ the algorithm showed stability and fast minimization. Although the $\epsilon = 1$ is shown in the graph being more or less stable, when the process was repeated the algorithm broke most of the times. For the case presented the recommended learning rate would be 0.1, as 0.5 was also a bit unstable.

Momentum parameter

The function of the momentum parameter is to accelerate the step size within consecutive gradients pointing in the same direction. This parameter allows to set a lower learning rate giving more stability for the beginning of the minimization without compromising computation time due to the acceleration.

The choice of α is not as important as ϵ. The usual values are 0.5, 0.9 and 0.99, using the the same dataset and network as the previous ones and with a learning rate of 0.01 and 0.5 the minimization of the function value is shown in the graph below.



(a) $\epsilon = 0.01$ **(b)** $\epsilon = 0.5$

Figure 7.5 Different minimization curves of the cost function for different momentum parameters. *Source Own*

From Figure 7.5a it seems that the minimization just keeps getting better with higher α. On the other hand from Figure 7.5b a higher learning rate can cause the minimization to fail with lower α.

Decay parameter

The parameter in the RMSProp algorithm is also not as important to choose as learning rate. The most used value is 0.9, but other typical are 0.5 or 0.99. In the figure below the inlfuence of ρ is depicted for the same NN and dataset with a learning rate of 0.01.

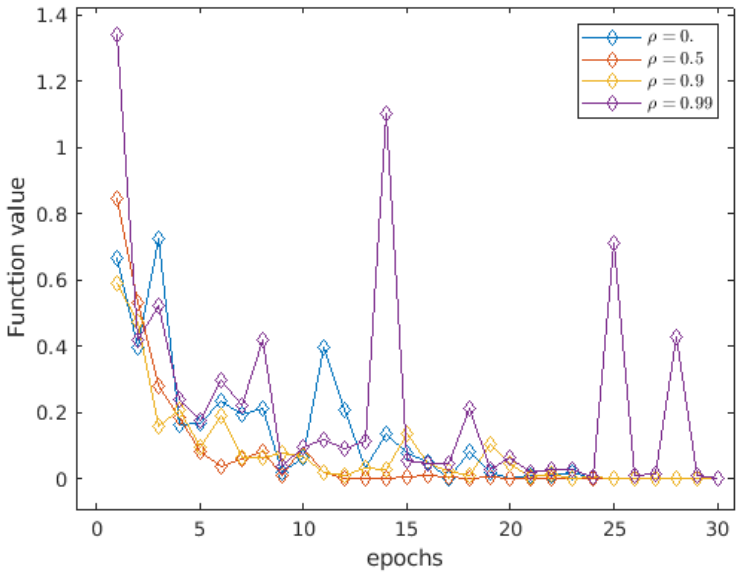


Figure 7.6 Different minimization curves of the cost function for different decay parameters. *Source Own*

The results suggest that 0.99 is too high giving unstability. Consequently 0.5 and 0.9 are the best possible parameters for the case, both showing good minimization curves.

7.4 Regularization performance

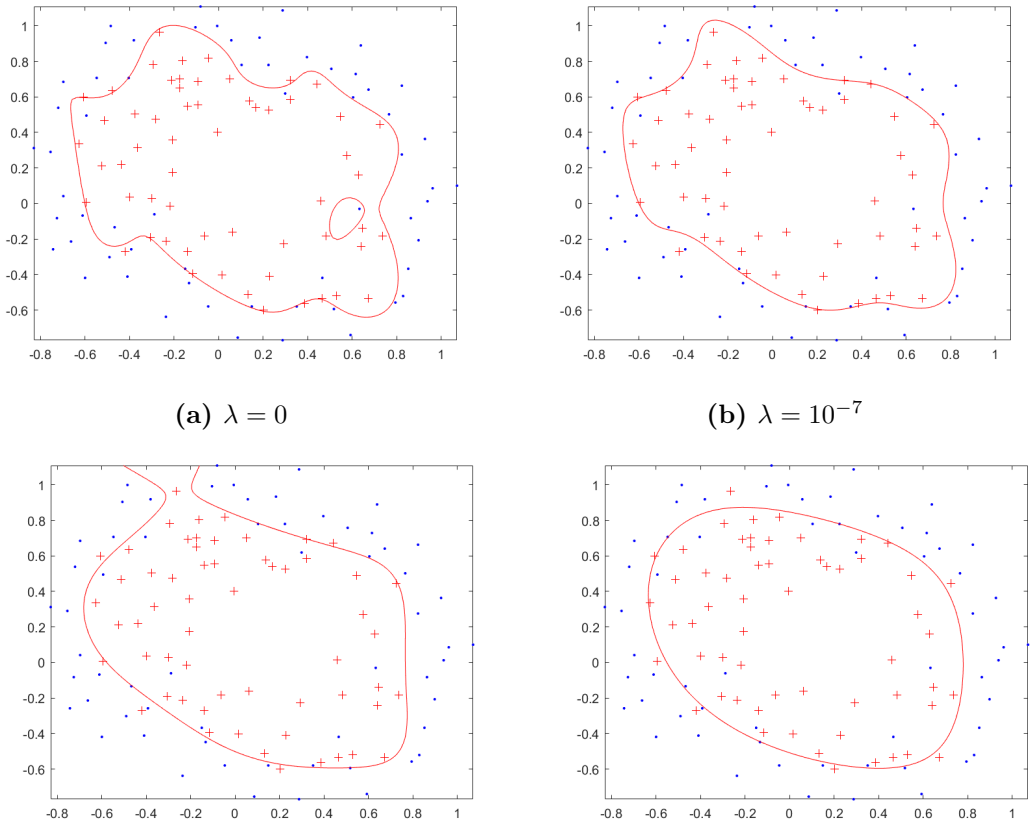
The regularization will be studied on a different dataset, because the NN in $MNIST$ did not overfit due to the immense amount of points. Therefore the *microchip* dataset is used to show in a more academic way overfitting.

The effect of both parameter penalties and early stopping is shown in some examples in this section. The idea is to show the importance of the λ hyperparameter for L^2, and the number of epochs for early stopping. Regarding parameter norm penalties, in this section L^2 will be studied but not L^1 because the latter is typically used as a feature selection technique rather than as a regularization method.

Parameter norm penalties

In figure 7.7 the boundary for the *microchip* dataset with a polynomial degree of 10 is shown for different values of λ. As it is clearly seen the introduction of lambda tends to smooth this boundaries shrinking the values of θ in the less influential directions. The value of $\lambda = 10^{-3}$ looks like a good generalized approximation, it looks similar to what it would look with a polynomial degree of 2.

In order to see if the introduction of the hyperparameter is really improving the performance, in figure 7.8 the test error with different values of lambda has been tracked for different test ratios. As it can be seen the hyperparameter is introducing a regularization that penalizes the optimum solutions in the flattest directions the most, therefore generalizing better to the test set. Confirming the previous assumption, the optimum value of λ lies between 10^{-4} to 10^{-2} for this case.



(a) $\lambda = 0$ **(b)** $\lambda = 10^{-7}$

(c) $\lambda = 10^{-5}$ **(d)** $\lambda = 10^{-3}$

Figure 7.7 Boundaries for different values of λ on the microchip dataset. *Source Own*

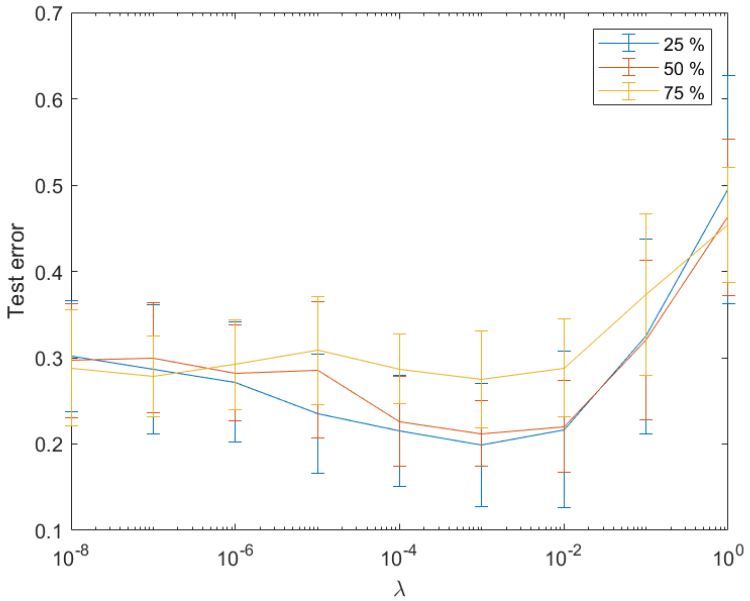
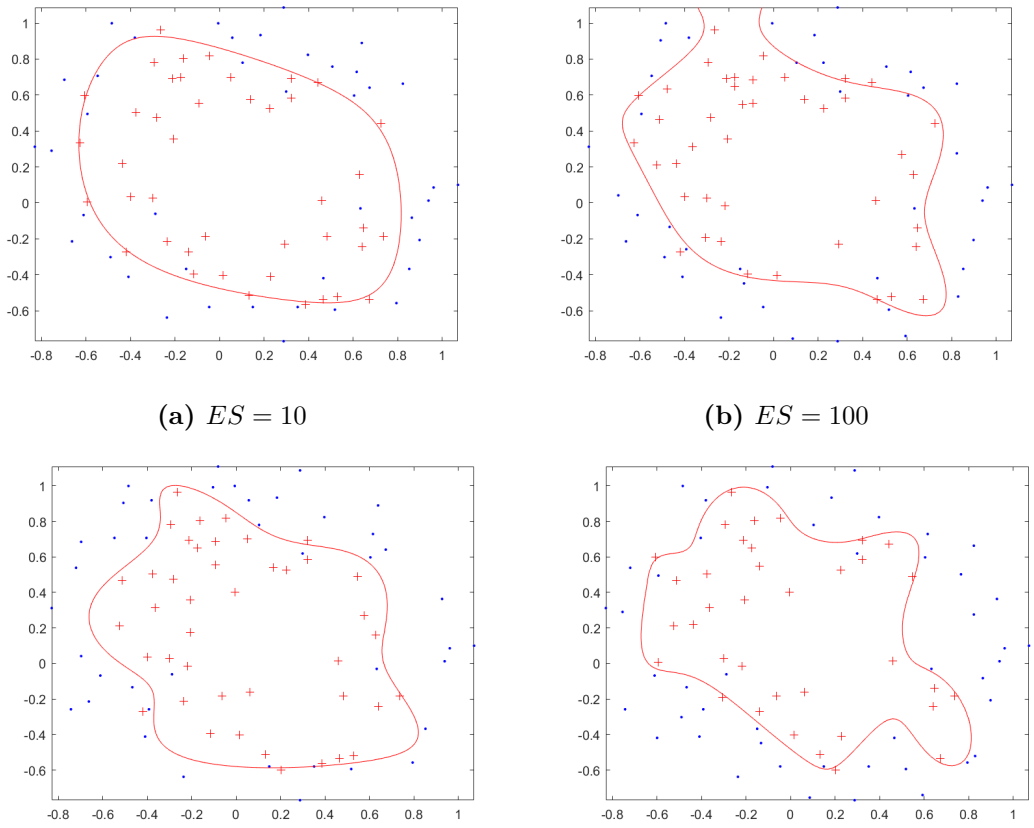


Figure 7.8 Influence of hyperparameter λ on the performance in MNIST dataset with different test ratios. *Source Own*

Early stopping

The controllable parameter in early stopping is the number of epochs that the minimization will continue if there is not improvement in test accuracy before stopping. In figure 7.9 the final boundary for different values of this parameter is shown. The early stopping technique requires of a test set since the minimization criteria depends on the test accuracy. In consequence the test ratio can be influential in the analysis. The ratio used for this analysis is 30%.



(a) $ES = 10$ **(b)** $ES = 100$

(c) $ES = 150$ **(d)** $ES = 200$

Figure 7.9 Boundaries for different values of the early stopping parameter on the microchip dataset. *Source Own*

As it can be seen in figure 7.10 around 10 epochs is the optimum parameter. The test error reached with this regularization is around 20% which was the same obtained with L^2 regularization. When the parameter tends towards positive infinite epochs, the test error should be the same to the obtained when there is no regularization, that is to say the same for $\lambda = 0$ which is around 30%.

Both regularization methods showed the phenomena of overfitting for the microchip dataset, improving the test accuracy respect to the results when regularization is not applied.

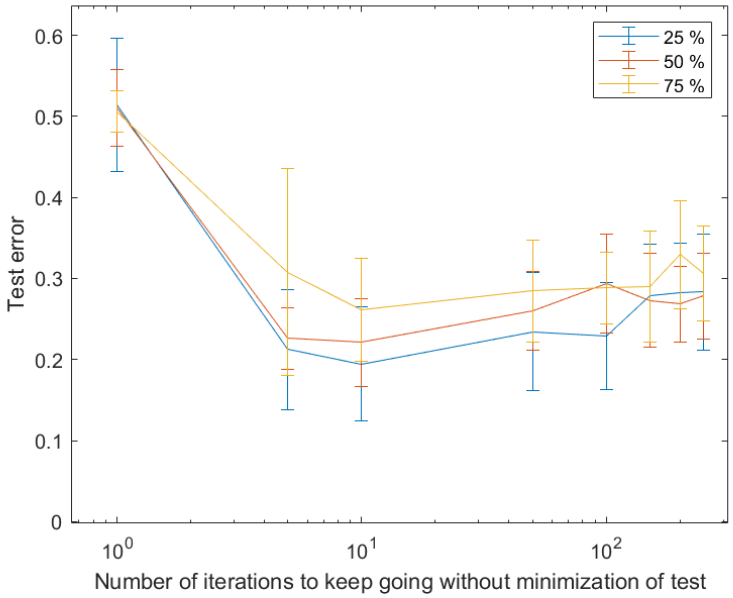


Figure 7.10 Influence of the early stopping hyperparameter on the performance in MNIST dataset with different test ratios. *Source Own*

7.5 Final Results

7.5.1 MNIST

The final results for MNIST are shown in this section. After analyzing the most important hyperparameters the network that performed the best is the following:

Topology		**Optimization**		**Regularization**	
ANN		RMSProp		Early Sopping	
Architecture:	784:500:150:10	Learning Rate:	0.01	L2 (lambda)	None
Output units:	Softmax	Momentum param:	0.9	Early Stopping:	10 epochs
Hidden units:	ReLU	Mini-batch:	200	Stopping criteria:	↑ ↑ ↑ ↑

Table 8 Network composition for the *MNIST* dataset. *Source Own*

Lots of simulations have been carried on but the best result obtained in the training set is an accuracy of **97.4%**. Figure 7.11 shows the confusion matrix on the test set for this scenario. This result is very similar but slightly higher than the one exposed by LeCun in his article [1] where the network reached an accuracy of 97.05%.

This is a really good score for this dataset, though using other deep learning techniques like convolutional neural networks (CNN) it could be raised a little more. One of the best results for this data is found in Dan C. Ciresan [3], where it reached a precision of 99.65% using a long CNN.

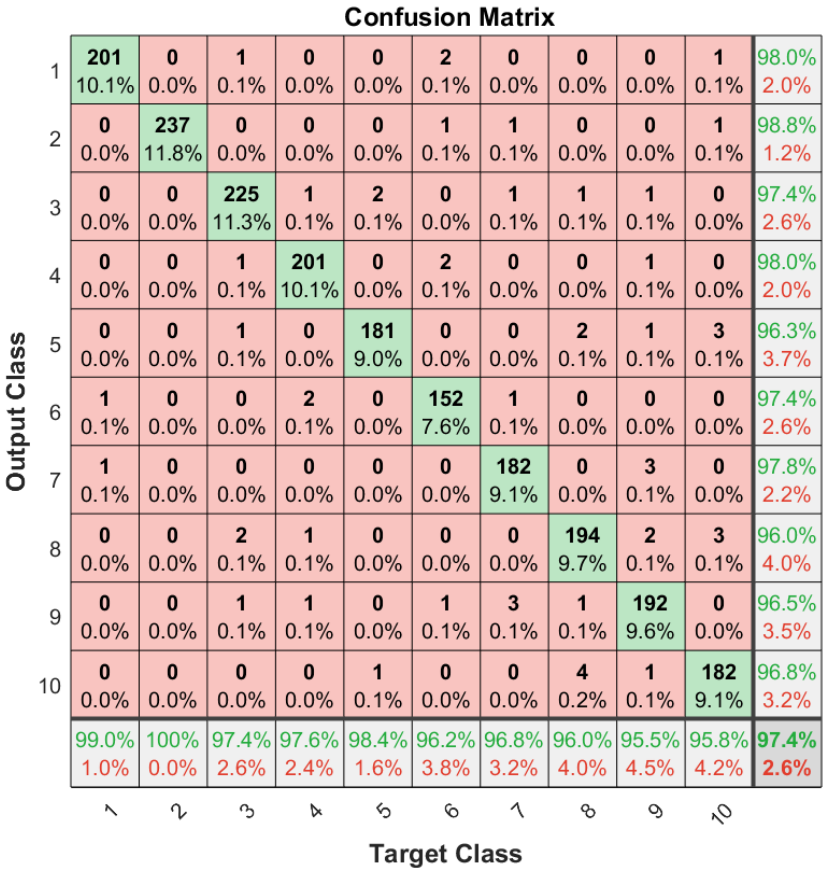
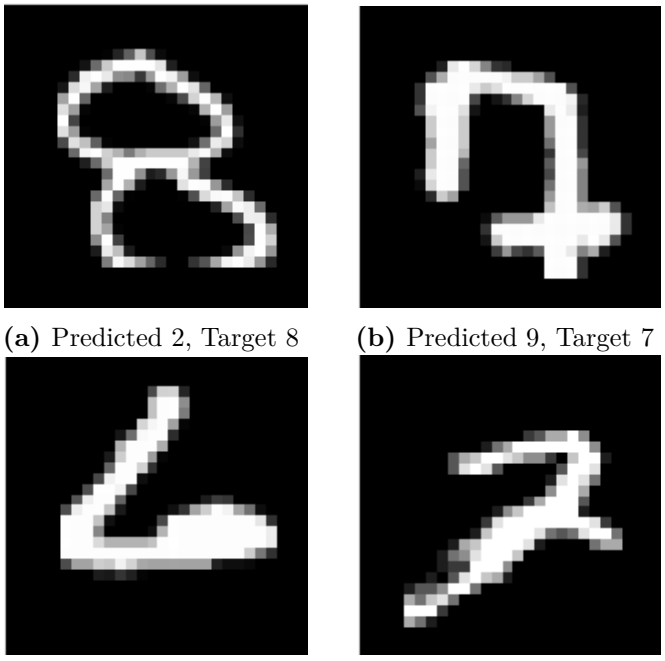


Figure 7.11 Confusion matrix for the MNIST dataset. *Source Own*

Lastly it might be interesting to look up some of the images the network did not classify correctly. Figures 7.12a, 7.12b, 7.12c and 7.12d show some of the digits that have been wrongly classified. The first picture might clearly be an eight for any human being but it has a little cut in the bottom that seems to mess up the NN. The mismatch in figures 7.12b and 7.12c can be somewhat understood as the seven almost looks like a nine with some pixels missing in the bottom left of the circle and the six is flattened to the extreme that can be seen as the bottom part of a two. Lastly, figure 7.12d is a reasonable error as most humans would choose a seven as well. In the end this digits are written by humans and can be a bad example of good calligraphy.



(a) Predicted 2, Target 8 **(b)** Predicted 9, Target 7

(c) Predicted 2, Target 6 **(d)** Predicted 7, Target 2

Figure 7.12 Mismatched digits using a ANN for the MNIST dataset. *Source Own*

7.5.2 Flowers

To push one step further the algorithms developed in this study one final NN will be put to test against another image database. The flower database contains more than 3,000 real photos of 5 differents species: daisy, dandelion, roses, sunflowers and tulips (see figure 7.13). All these images are in color (rgb) and have a broad variety of sizes (from 180 pixels in the shortest dimension to 300 in the largest), this equals to almost 100,000 features for the smallest image, much more than the 784 from the 28x28 gray scale images from MNIST.

(a) 1 Daisy **(b)** 2 Dandelion

(c) 3 Roses **(d)** 4 Sunflowers **(e)** 5 Tulips

Figure 7.13 The 5 groups from the flower dataset. *Source Own*

This amount of features is unreasonable for the network under inspection. For this reason some kind of preprocessing to the data is needed. The first idea is to use *imresize*() function from Matlab to rescale the images to 32x32 rgb images, this means 3,072 features. As it will be seen the loss in information from this reduction led to poor results which brought a second idea. This idea is to use the discrete cosine transformation (DCT) of the images to select the most important information choosing the top left triangle of the array in the frequency domain. In order to compare both results this triangle is set to have around the same number of features as 32x32 image.

The preprocess of the second method is not trivial and it is explained in the following steps:

i Resize

ii Discrete Cosine Transformation

iii Top left triangle selection

iv Scale every group by its maximum and minimum to 0-1 interval

v Scale all with the absolute maximum and minimum to 0-1 interval

Step 1: In order to apply the DCT all images must have the same size, otherwise the modes of the transformation will not represent the same asset for two different images. For this reason the first step is to resize all images to 180x180 pixels.

Step 2: Matlab offers a DCT for 2D arrays which serves perfectly for the duty commended.

Step 3: In natural image processing the general procedure when applying Fourier transformations is to pick the top left corner of the array after the transformation as the most important numbers fall there. In figure 7.14 the new bases after applying the transformation are shown, the idea is to select the red triangle (the top left corner of the matrix) which for natural images usually has the highest weights.

Step 4: Scaling the inputs is important to avoid saturation for the ReLU units. For this dataset, it has proved to be useful to first scale every group with its maximums.

Step 5: Finally scaling again to the absolute maximum of all groups which is the commonly used data scaling.

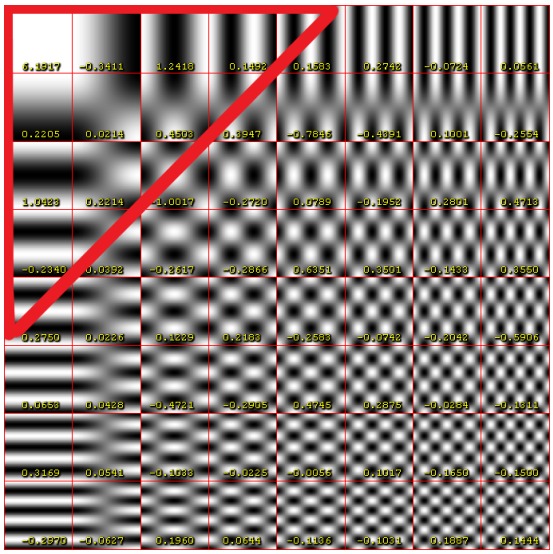
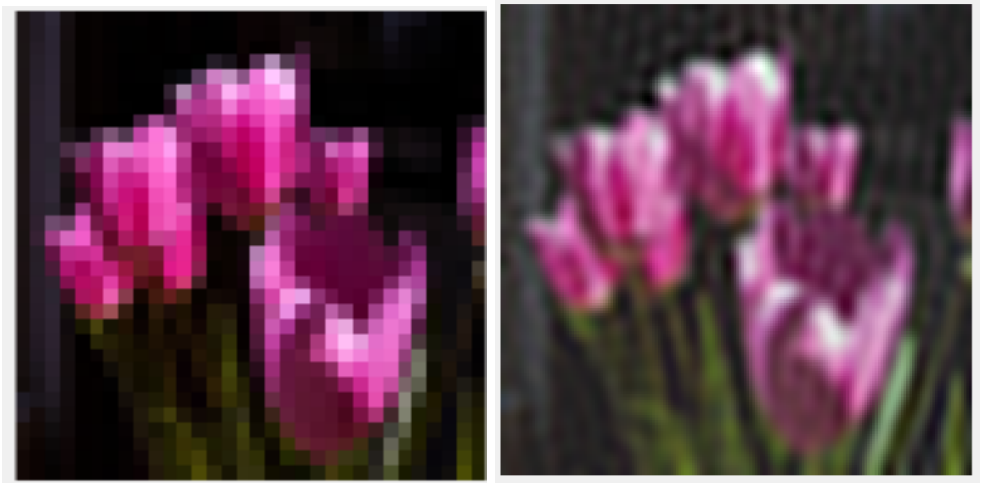


Figure 7.14 Top left triangle selection for natural image processing. *Source [19]*

In the figure below a random image from the database can be seen in its original form and in its two reduced versions. The loss of resolution is big for both methods but the technique that uses DCT is clearly better, the final number of features after applying the proccess explained to the three channels rgb is 3,105.

(a) Original



(b) 32x32 **(c)** dct 3105

Figure 7.15 Sample of the flower database, on the top the original, on the left resized to 32x32, and on the right with dct transformation 32x32 bits. *Source Own*

Once the data is ready the next step is to define the network. The architecture will be bigger to the one used in *MNIST* as the number of inputs is bigger, it has been arbitrarily selected x:1000:500:150:10. The activation function will be softmax for the ouput units and ReLU for the hidden ones. The optimization algorithm selected is Nesterov as RMSProp was tried and performed weaker. There is no parameter penalty applied and the network will minimize for 500 epochs.This data is resumed in the following list:

Topology		**Optimization**		**Regularization**	
ANN		Nesterov		None	
Architecture:	3072:1000:500:150:5	Learning Rate:	0.01	L2 (lambda)	None
Output units:	Softmax	Mom param:	0.5	Early Stopping:	None
Hidden units:	ReLU	Mini-batch:	200	Stopping criteria:	500 epochs

Table 9 Network composition for the *Flower* dataset. *Source Own*

The training with the 32x32 rescaled images did not succeed. Although it achieved an accuracy of 54% which is way better than a random prediction, this result is far from reliable. The figure below shows the confusion matrix for this case. This result motivated the idea to use the other preprocess technique for the data.

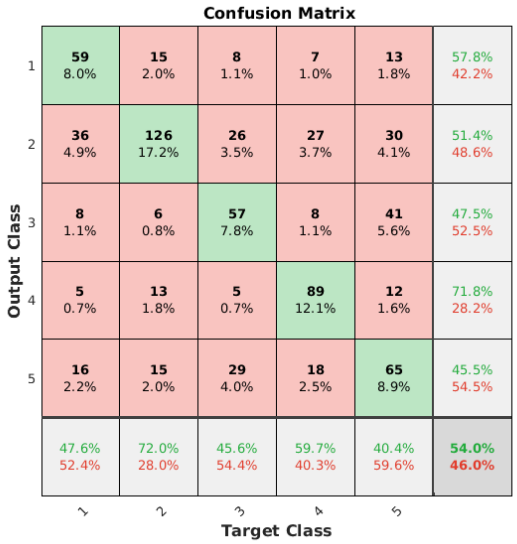


Figure 7.16 Confusion matrix for the NN trained with the 32x32 rescaled images. *Source Own*

After the first failure, the network was trained with the data in the frequency domain after applying a discrete cosine transformation. The result for this case was surprisingly higher, with the network almost perfectly differentiating 3 groups. The accuracy reached 83.7%.

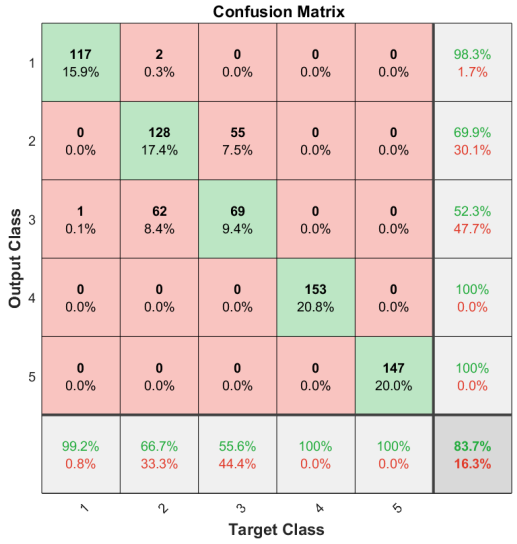


Figure 7.17 Confusion matrix for the NN trained with the dct most important coefficients. *Source Own*

The confusion matrix shows how well the network learned to classify daisy, sunflowers and tulips. But it was between dandelion and roses where it really failed. It is curious as roses are very distinctive flowers for humans, in the figures below some of the images that were wrongly identified are shown.

(a) Predicted Daisy, Target Dandelion

(b) Predicted Dandelion, Target Rose

(c) Predicted Rose, Target Dandelion

(d) Predicted Rose, Target Dandelion

Figure 7.18 Mismatched flowers using an ANN for the flower dataset. *Source Own*

The pictures in figure 7.18 do not clarify why the network does not classify between Dandelion and Rose. For humans the difference is obvious as roses are mostly pink, but it seems that the network did not find its way through this issue. Data augmentation could be introduced to try to increase the network's performance.

8 Conclusions

The aim of this thesis was to navigate and develop the foundations of deep learning. Artificial neural networks have been the central pillar of the project, and the mathematics behind them have been deeply analyzed. Alternatively, optimization methods have also been detailed and developed. All of these have been developed while trying to present the results obtained throughout the sections in the most rigorous way possible, using statistics (mean and standard deviation) to diminish uneven results.

By analyzing the performance of NN in some of the most famous databases such as *iris* and *MNIST*, this thesis has proved the utility of these architectures for classification tasks. The influence of some key parameters like learning rate, momentum parameter or test ratio has also been shown. Not only that, but for the *MNIST* dataset a set of parameters have been proposed for a certain architecture as the best, with those, a higher accuracy than the same network in LeCun [1] was obtained.

The test ratio has to be set as high as possible without compromising the training performance, typical values are between 15% to 40%. Data augmentation enables the model to be enriched with more information, but can also trigger overfitting dangerously. There is no general procedure recommended, so trial and error is the way to go. Data scaling has stood up as a crucial preprocess step to avoid saturation in the network. Finally, regarding the optimization, the learning rate has proved to be, if not the most important, one of the most important parameters in gradient descent algorithms. The momentum parameter and the decay parameter for Nesterov and RMSProp have to be set in accordance with learning rate.

The final results for *MNIST* and *Flowers* datasets were satisfactory. *MNIST* eventually reached a higher performance than in a renowned article. On the other hand, the *Flower* dataset added an extra interesting preprocess step, which is feature selection. The result obtained (83.7% accuracy) is good but not as high as desired.

All these objectives were successfully met. Nevertheless, during the development of this thesis, the planning shifted directions, leaving the kernel method and autoencoders out of the scope. Thanks to this, it has been possible to deepen into optimization algorithms, letting in the momentum and adaptive learning rate variations.

There are other basics in deep learning that have been left behind from the beginning like support-vector machines, principal component analysis and many others. However, the most useful and logical followings steps would be convolutional neural networks and autoencoders. Consequently, as a continuation to this project it would be absorbing to explore CNN, applying them to image classification in order to compare the results obtained with the ones obtained in this project using ANN. It would also be interesting to study autoencoders which are specific constructions of traditional NN that try to learn useful properties of the data.

Appendices

Appendix A. Neural Network notation

Notation	Explanation	Dimension
B_i	vector of biases at layer i	$\mathbb{R}^{nS_i}$
grad	gradient vector (analogous to θ)	$\mathbb{R}^k$
h_i	outputs after the linear transformation at layer i	$\mathcal{M}(nD, nS_i)$
k	number of parameters to minimize in the NN	$\mathbb{N}$
nD	number of data points	$\mathbb{N}$
nF	number of features	$\mathbb{N}$
nG	number of groups	$\mathbb{N}$
nL	number of layers	$\mathbb{N}$
nS	vector with the architecture of the NN	$\mathbb{N}^{nL}$
o_i	outputs after the activation function at layer i	$\mathcal{M}(nD, nS_i)$
W_i	matrix of weights between layer i-1 and i	$\mathcal{M}(nS_{i-1}, nS_i)$
x	matrix with different inputs in rows and features in col	$\mathcal{M}(nD, nF)$
y	matrix with different inputs in rows and classes in col	$\mathcal{M}(nD, nG)$
δ_i	reverse propagation of the gradient at layer i	$\mathcal{M}(nD, nS_i)$
θ	vector with all the parameters W,B	$\mathbb{R}^k$

Appendix B. Databases used

Name	Data points	Features	Classes
Gender	10,000	2	2
https://www.kaggle.com/code/pratiksatriani/gender-classification			
Microchip	118	2	2
https://www.kaggle.com/datasets/johnjyjy/microchip-quality-control			
ConCircles	500	2	2
Artificially Created			
4circles	100	2	4
Artificially Created			
Iris	150	4	3
https://www.kaggle.com/datasets/uciml/iris			
MNIST	10,000	748	10
https://ch.mathworks.com/help/deeplearning/ug/data-sets-for-deep-learning.html			
Flowers	3451	$\sim 10^5$	5
https://ch.mathworks.com/help/deeplearning/ug/data-sets-for-deep-learning.html			

References

[1] Yann LeCun et al. "Gradient-Based Learning Applied to Document Recognition". In: 1998.

[2] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *ICML 2010*. 2010, pp. 807–814.

[3] Dan Ciresan et al. "Flexible, High Performance Convolutional Neural Networks for Image Classification." In: July 2011, pp. 1237–1242. DOI: 10.5591/978-1-57735-516-8/IJCAI11-210.

[4] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning*. Vol. 28. Proceedings of Machine Learning Research 3. 2013, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html.

[5] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[6] Ian Goodfellow Yoshua Bengio and Aaron Courville. *Deep Learning*. Massachusetts Institute of Technology, 2016.

[7] Inc Future Industry Insight. *Artificial Intelligence (AI) Market Research Report 2018*. 2018. URL: https://www.academia.edu/37211873/Artificial_Intelligence_AI_Market_Research_Report_2018.

[8] Raúl Gómez. *Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names*. 2018. URL: https://gombru.github.io/2018/05/23/cross_entropy_loss/ (visited on 04/2022).

[9] Dario Saracino Nicola Anzivino Paolo Anfossi. *Artificial Intelligence Evolution – main trends*. 2018. URL: https://www.pwc.com/it/it/publications/assets/docs/pwc-ai-evolution-financial-services.pdf.

[10] Chigozie Nwankpa et al. "Activation Functions: Comparison of trends in Practice and Research for Deep Learning". In: *CoRR* abs/1811.03378 (2018). URL: http://arxiv.org/abs/1811.03378.

[11] John D. Kelleher. *Deep Learning*. Massachusetts Institute of Technology, 2019.

[12] Jatin Nanda. *Scene Recognition with Bag of Words*. 2019. URL: https://www.cc.gatech.edu/classes/AY2016/cs4476_fall/results/proj4/html/jnanda3/index.html (visited on 04/2022).

[13] Leonid Datta. "A Survey on Activation Functions and their relation with Xavier and He Normal Initialization". In: *CoRR* abs/2004.06632 (2020). URL: https://arxiv.org/abs/2004.06632.

[14] Parveen Khurana. *Activation Functions and Initialization Methods*. 2020. URL: https://prvnk10.medium.com/activation-functions-and-initialization-methods-e2f5f83b7a64 (visited on 03/2022).

[15] Kinder Chen. *Hidden Layer Activation Functions*. 2021. URL: https://kinder-chen.medium.com/hidden-layer-activation-functions-6fd65489ed25 (visited on 03/2022).

[16] Fernando Sancho Caparrini. *Variational Autoencoder*. URL: http://www.cs.us.es/~fsancho/?e=232.

[17] Tensor Flow. *MNIST*. URL: https://www.tensorflow.org/datasets/catalog/mnist (visited on 05/2022).

[18] MathWorks. *Automatic Differentiation Background*. URL: https://es.mathworks.com/help/deeplearning/ug/deep-learning-with-automatic-differentiation-in-matlab.html (visited on 02/2022).

[19] Wikipedia. *Discrete Cosine Transform*. URL: https://en.wikipedia.org/wiki/Discrete_cosine_transform (visited on 06/2022).