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

Study of Variational Autoencoders in Machine Learning

Document:

Report

Author:

Joel Campo Moyà

Director/Co-director:

Àlex Ferrer Ferre

Degree:

Bachelor's degree in Aerospace Vehicle Engineering

Examination session:

Spring

BACHELOR FINAL THESIS

Abstract

Autoencoders are essential in the field of machine learning because of the wide range of applications and distinctive talents they have. The ability of autoencoders to learn condensed and effective representations of complicated input data is one of the main factors in their significance. Autoencoders offer effective data compression by encoding the input data into a lower-dimensional latent space, which is useful in situations with constrained storage or bandwidth. Autoencoders are also frequently employed for unsupervised learning tasks like data generation, dimensionality reduction, and anomaly detection. Without relying on explicit labels or supervision, they enable us to find underlying patterns and structures in the data. Overall, the versatility and utility of autoencoders make them a fundamental tool in the machine learning toolbox, empowering researchers and practitioners to tackle a wide range of problems across diverse domains.

Generative models, such as autoencoders, play a fundamental role in machine learning by enabling the creation of new, synthetic data that closely resembles the original input distribution. These models have revolutionised various domains, including image generation, text synthesis, and music composition, among many others. By capturing the underlying patterns and structures of the training data, generative models provide a powerful framework for creative applications, data augmentation, and simulation studies.

This project aimed to explore the capabilities and applications of autoencoders, a type of neural network architecture, in the field of machine learning. The main focus of the project was to refactor legacy code used for image identification and transform it into an autoencoder capable of generating MNIST images. Through extensive experimentation and analysis, the project demonstrated the effectiveness of autoencoders in learning representations of input data and generating high-quality synthetic images. The findings of this study contribute to our understanding of autoencoders and their potential for various tasks, including image generation. The project also highlighted the importance of clean code practises, code refactoring, and neural network architectural design principles in adapting existing models for new purposes.

Resum

Els Autoencoders són essencials en el camp de l'aprenentatge automàtic a causa de l'ampli ventall d'aplicacions i talents distintius que tenen. La capacitat dels autoencoders per aprendre representacions de dades complicades i que no tenen per què estar perfectes és un dels principals factors de la seva importància. Els autoencoders ofereixen una compressió de dades eficaç mitjançant la codificació de les dades d'entrada en un espai latent de dimensions inferiors, cosa que és útil en situacions amb emmagatzematge o amplada de banda restringida. Els autoencoders també s'utilitzen amb freqüència per a tasques d'aprenentatge no supervisades com la generació de dades, la reducció de la dimensionalitat i la detecció d'anomalies. Sense dependre d'etiquetes o supervisió explícites, ens permeten trobar patrons i estructures subjacents a les dades. En general, la versatilitat i la utilitat dels autoencoders els converteixen en una eina fonamental en l'aprenentatge automàtic, que permet als investigadors i professionals d'abordar una àmplia gamma de problemes en diversos dominis.

Els models generatius, com els autoencoders, tenen un paper fonamental en l'aprenentatge automàtic, ja que permeten la creació de dades noves i sintètiques que s'assemblen molt a la distribució d'entrada original. Aquests models han revolucionat diversos àmbits, com ara la generació d'imatges, la síntesi de text i la composició musical, entre molts d'altres. En capturar els patrons i les estructures de les dades d'entrenament, els models generatius proporcionen un marc potent per a aplicacions creatives, augment de dades i estudis de simulació.

Aquest projecte tenia com a objectiu explorar les capacitats i aplicacions dels autoencoders, definits com un tipus d'arquitectura de xarxes neuronals, en el camp de l'aprenentatge automàtic. L'objectiu principal del projecte era remodelar un codi heretat que funcionava per a la identificació d'imatges i transformar-lo en un autoencoder capaç de generar imatges MNIST. Mitjançant una àmplia experimentació i anàlisi, el projecte va demostrar l'eficàcia dels autoencoders per aprendre representacions de dades d'entrada i generar imatges sintètiques d'alta qualitat. Els resultats d'aquest estudi contribueixen a la nostra comprensió dels autoencoders i el seu potencial per a diverses tasques, inclosa la generació d'imatges. El projecte també destaca la importància de les pràctiques de codi net, la refactorització de codi i els principis de disseny arquitectònic en xarxes neuronals per adaptar els models existents per a nous propòsits.

Agraïments

Primer de tot, vull expressar la meva gratitud i l'afecte màxim al meu tutor del treball, l'Àlex Ferrer, ja que la seva vocació per la docència i la seva passió en l'àmbit que es tracta al treball m'han aconseguit ensenyar un camp totalment nou al que li tenia bastant pudor com és la programació i m'ha fet tornar unes ganes d'aprendre que havia perdut en els últims mesos del grau. També, la naturalitat i la proximitat que m'ha mostrat, que no veia en un professor des que vaig marxar de Lleida. T'admiro molt Àlex.

No em vull deixar tampoc al Toni Darder, qui va desenvolupar la tesi predecessora a la meva i que m'ha ajudat amb diverses reunions amb tot el que he necessitat.

Als meus companys de pis, Adrià, Marc i Jordi. He viscut molt a gust aquests anys i tot ha estat més fàcil. Quina sort haver-nos ajuntat. Moltes gràcies.

Mama, Papa, Pol, i tota la resta de família. Ho hem aconseguit junts. Gràcies per haver-me recolzat en absolutament tot. Des del primer dia m'he treballat el grau per fer-vos sentir orgullosos, i inclús en moments que ni jo mateix n'estava m'heu fet veure que mai marxaríeu del meu costat. Us estimo moltíssim i sempre ho faré tot per vosaltres, perquè us ho mereixeu.

I per acabar a tu Ona. Gràcies per guiar-me, per fer-me riure, per distreure'm, per animar-me a seguir. Vas ser, ets i seràs un dels pilars més importants de la meva vida. Estic molt orgullós del que has assolit i de la manera que ho has fet. T'estic molt agraït per com em dones suport i m'escoltes sempre amb qualsevol cosa. T'estimo.

Contents

1 Introduction **1**
1.1 Object . . . 1
1.2 Scope . . . 1
1.3 Requirements . . . 2
1.4 Justification . . . 2
1.5 Schedule . . . 3

2 State of the art **5**
2.1 Artificial Intelligence . . . 5
2.2 Machine Learning . . . 7
2.3 Generative AI . . . 8
2.4 Autoencoders . . . 9
2.5 Variational Autoencoders . . . 9

3 Theoretical background **11**
3.1 The classification problem . . . 11
3.1.1 Logistic regression . . . 11
3.1.2 Cost Functions . . . 12
3.2 Regularisation . . . 13
3.2.1 Parameter Penalties . . . 13
3.2.2 Early stopping . . . 15
3.2.3 Dropout . . . 15
3.3 Feed forward neural networks . . . 16
3.3.1 Architecture . . . 16
3.3.2 Propagation . . . 19
3.3.3 Activation Functions . . . 21
3.4 Optimization method . . . 24
3.4.1 The learning process . . . 24
3.5 Gradient descent . . . 25
3.6 Stochastic Gradient descent . . . 26

3.7 Autoencoder 27
3.8 Variational autoencoders 29

4 Code refactoring **33**
4.1 Starting point 33
4.2 First steps 34
4.3 Final structure building 36
4.4 Evolving into an AE 37

5 Results **39**
5.1 Refactoring validation 40
5.2 Activation Function Decision 42
5.3 Optimization parameters iteration 42
5.3.1 Learning rate analysis 44
5.3.2 Test Ratio analysis 45
5.3.3 Batch size analysis 45
5.3.4 MaxFunEvals analysis 46
5.3.5 Lambda analysis 47
5.3.6 NN Architecture analysis 47
5.4 Optimization evolution 49
5.5 Final results 50

6 Conclusions **52**

7 Social & Environmental impact **53**
7.1 Social Impact 53
7.2 Environmental Impact 53

8 Budget summary **54**

A Neural Network Notation **57**

B Databases used **58**

List of Figures

1.1 Gantt chart. *Source Own* 4

2.1 The most common way consumers plan to use AI. *Source [8]* 6
2.2 Consumer concerns about AI uses. *Source [8]* 7
2.3 Consumer confidence in businesses using AI. *Source [8]* 7
2.4 Venn diagram of deep learning. *Source [9]* 8
2.5 Representation of an undercomplete AE using the MNIST database. *Source [11]* 9
2.6 Schematic of a VAE showing an example of latent attributes. *Source [12]* 10

3.1 Logistic Regression for a Male/Female data set. *Source [1]* 12
3.2 Example of overfitting in the microchip dataset. *Source [1]* 13
3.3 The effect of L2 regularisation. *Source [9]* 14
3.4 Learning curves showing the optimum point to stop before overfitting. *Source [9]* 15
3.5 Sub-networks formed from a base network. *Source [9]* 16
3.6 Representation of a neuron. *Source [1]* 17
3.7 Representation of a layer. *Source [1]* 18
3.8 Representation of a neural network with depth L. *Source [1]* 18
3.9 Representation of Sigmoid and Tanh response. *Source [1]* 22
3.10 Comparison of ReLU, sigmoid and tanh AF. *Source [1]* 23
3.11 Comparison of max and softmax for the 2D case. Softmax shows its first output, it has the shame shape in each axis. *Source [1]* 24
3.12 Architecture of an AE. *Source [13]* 27
3.13 Difference between AE (deterministic) and VAE (probabilistic). *Source [15]* 29
3.14 Depiction of convolutional neural network. *Source [16]* 31
3.15 Architecture of a VAE. *Source [12]* 32
3.16 Introduction of the random node. *Source [12]* 32

4.2 Confusion Matrix for the understanding run. *Source own* 34
4.1 Previous thesis UML. *Source own* 35
4.3 Intermediate UML. *Source own* 36
4.4 Final UML. *Source own* 38

5.1 MNIST dataset. *Source [17]* 39
5.2 Refactoring validation: Iris Test. *Source own* 40
5.3 Refactoring validation: MNIST. *Source own* 41
5.4 Comparison between target (a) and output (b) in the first run of the code as an AE. *Source own* 42
5.5 Results of test A1_T1. Target (a); Output (b); Cost Function (c). *Source own* 44
5.6 Results of test A1_T9. Target (a); Output (b); Cost Function (c). *Source own* 44
5.7 Results of test A1_T15. Target (a); Output (b); Cost Function (c). *Source own* 46
5.8 Results of test A1_T22. Target (a); Output (b); Cost Function (c). *Source own* 46
5.9 Results of test A1_T27. Target (a); Output (b); Cost Function (c). *Source own* 47
5.10 Results of test A1_T22. Target (a); Output (b); Cost Function (c). *Source own* 47
5.11 Results of test A2_T1. Target (a); Output (b); Cost Function (c). *Source own* 48
5.12 Results of test A4_T1. Target (a); Output (b); Cost Function (c). *Source own* 49
5.13 Results of test A6_T1. Target (a); Output (b); Cost Function (c). *Source own* 49
5.14 Target (a) and Cost function (b) for outputs from figure 5.15. *Source own* 49
5.15 Evolution of the output during the training process. *Source own* 50
5.16 Results of test AF_TF. Target (a); Output (b); Cost Function (c). *Source own* 51
5.17 Results of test AF_Tmax. Target (a); Output (b); Cost Function (c). *Source own* 51

List of Tables

5.1 Initial values before any iteration. Green values are those that will remain constant throughout all iterations. . . . 43
5.2 Learning rate analysis results. . . . 44
5.3 Test Ratio analysis results. . . . 45
5.4 Batch size analysis results. . . . 45
5.5 MaxFunEvals analysis results. . . . 46
5.6 Lambda analysis results. . . . 47
5.7 Architecture analysis results . . . 48
5.8 Final parameters values . . . 50

8.1 Project's budget summarised . . . 54

List of Algorithms

1 Forward propagation . . . 20
2 Backward propagation . . . 21
3 Gradient Descent applied to NN . . . 26
4 Stochastic Gradient Descent applied to NN . . . 27

List of abbreviations / Glossary

AE Autoencoder

AF Activation Function

AI Artificial Intelligence

KL Kulback-Leibler

LR Learning Rate

ML Machine Learning

MLP Multilayer perceptrons

NN Neural Networks

OOP Object Oriented Programming

PCA Principal Component Analysis

SGD Stochastic Gradient Descent

UML Unified Modelling Language

VAE Variational Autoencoder

Chapter 1

Introduction

1.1 Object

The aim of this project is to conduct a comprehensive examination of the various techniques and algorithms related to Variational Autoencoders (VAEs) in Machine Learning. The project will involve researching its theoretical foundations and exploring their practical applications in unsupervised learning, generative models, and dimensionality reduction. The project will also include implementing and evaluating various types of VAEs on different datasets and comparing their performance to other related algorithms. In addition, the project will investigate the use of VAEs in conjunction with other machine learning techniques such as deep learning and reinforcement learning. The overall goal of the project is to provide a thorough understanding of its capabilities and limitations in the field of machine learning.

1.2 Scope

The development of the project will include:

- Refactored Code
- Autoencoders (AE)
- VAEs

Will be developed following:

- MATLAB language
- Object Oriented Programming (OOP) paradigm

The project will not include:

- Creation of databases
- Natural language processing

- Other optimization algorithms
- Unsupervised learning

High level derivable:

- Code files
- Budget
- Final report

1.3 Requirements

In this section, the requirements that need to be fulfilled to achieve the thesis objectives will be outlined.

Firstly, the code based from the previous thesis [1] must implement VAEs, as this is the main objective of the project, being MATLAB the software used to develop it. As commented before, the easier to navigate inside the code the better understanding of it, so OOP will be the way the code is developed.

Apart from pure technical aspects, the databases used should be free to access, at the same time as the code shall be able to be executed in any computer, giving the option to any user to work with it.

Taking all these points in consideration, the requirements can be summarized into:

- Developed in MATLAB software
- VAEs shall be implemented considering the previous code
- Developed with OOP
- Open Access Databases
- For any user and computer

1.4 Justification

Who hasn't heard about Artificial Intelligence (AI) these days? It is a recurring topic in news, universities, companies, etc. that can open the doors to a new way of doing things. But why can it be so revolutionary? Some of the main points, as told by ChatGPT[2] (An AI itself), are the following:

"AI is becoming increasingly prevalent in many areas of our lives: From virtual assistants like Siri and Alexa to self-driving cars, AI is rapidly becoming a part of our daily lives. By learning about AI, you can gain a better understanding of how these technologies work and how they are likely to impact our society in the future.

AI is transforming industries: AI is transforming a variety of industries, including healthcare, finance, manufacturing, and more. By learning about AI, you can gain a competitive edge in your field and better position yourself for career opportunities.

AI has the potential to solve complex problems: AI has the ability to analyse massive amounts of data and identify patterns that humans might miss. This makes it a powerful tool for solving complex problems in areas such as climate change, public health, and more.

AI is shaping the future of work: AI is likely to have a significant impact on the nature of work and the job market in the coming years. By learning about AI, you can better position yourself for success in the rapidly evolving job market.” (ChatGPT, 2023)

Overall, learning about this topic is important for anyone who wants to understand the role that technology is playing in our society and the opportunities and challenges that lie ahead.

As the field of AI is really deep, the thesis will be focused on VAEs as these are a powerful tool for unsupervised learning, and have shown to be effective in a variety of applications such as image and text generation, anomaly detection, and data compression. Moreover, VAEs offer a more flexible approach to generative modelling, allowing for the generation of new data samples that are not present in the original dataset, which is a key advantage compared to traditional machine learning techniques.

By studying this type of unsupervised learning, you can gain a deeper understanding of this powerful tool, and how it can be applied to real-world problems. This can lead to new insights and developments in fields such as AI, data science, and computer vision.

Additionally, as the field of machine learning continues to grow and evolve, it is important to stay up-to-date with the latest techniques and technologies, and studying VAEs can help you stay at the forefront of this rapidly evolving field.

1.5 Schedule

The first step in every project is to settle a planning. But before scheduling the tasks, these have to be determined. To do that, it was decided to separate between management and technical tasks, as they are presented below. Also, the related tasks are grouped under the same number.

- Technical tasks
  - OOP, Machine learning & AE Initiation
  - Analysis & Refactoring of the current code
  - Implementation of VAEs
  - Analysis of the results obtained
- Management tasks
  - Gathering information
  - Project Charter development
  - Documentation

– Derivable preparation

It is important to remark that the beginning of the project is marked in January 23rd, but the initiation tasks started some time before.

Also, there are two general milestones in the project related to deliverables. Project Charter delivery (March 17th) and the final delivery (June 21st).

All this information let to develop a Gantt diagram (figure 1.1), using the "GanttProject Software" [3].

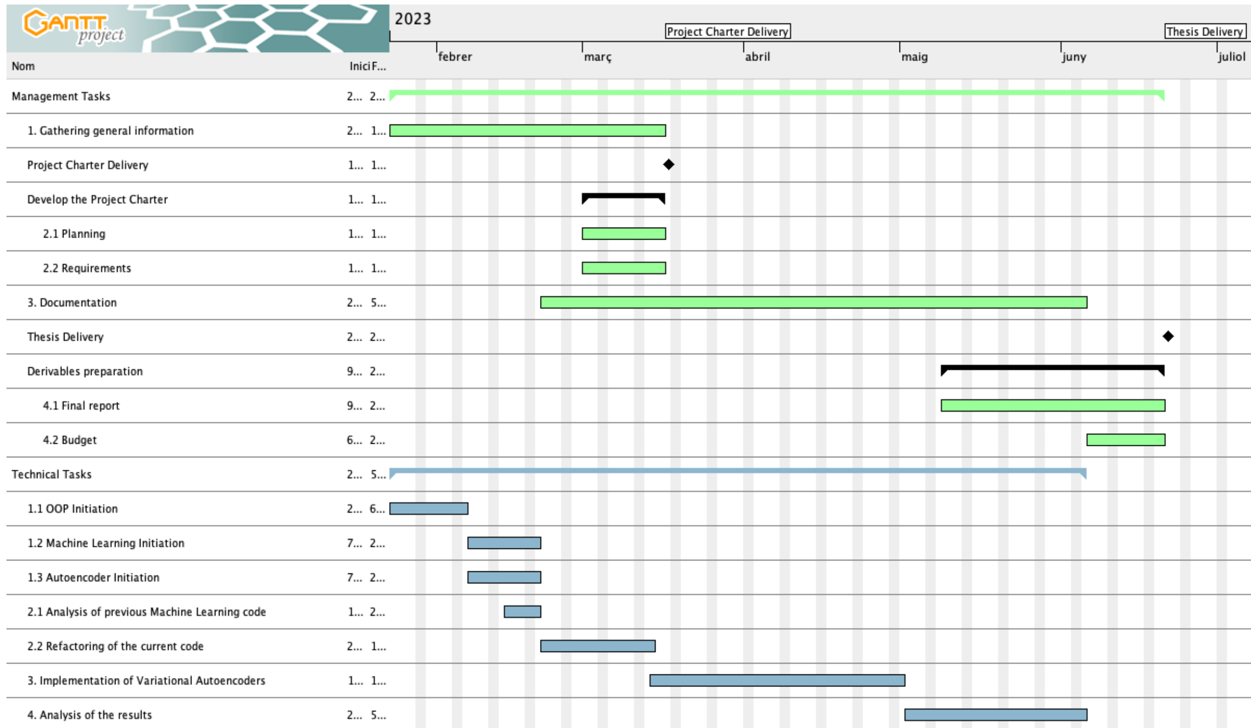


Figure 1.1: Gantt chart. *Source Own*

Chapter 2

State of the art

In this chapter, there is an introduction to the different aspects that this project deals with and the state of the art in each of them. Later chapters will delve deeper into the more specific aspects that are being worked on.

2.1 Artificial Intelligence

AI can be defined as the simulation of human intelligence processes by computer systems that will shape our future more powerfully than any other innovation this century. "Anyone who does not understand it will soon find themselves feeling left behind, waking up in a world full of technology that feels more and more like magic." [6].

In the last few decades, rapid advances in data storage and computer processing power have dramatically changed the game. The history of this discipline originates from the idea of creating machines that can think and learn like humans. Its history dates back to the beginning of the 20th century, with the development of the first theories of computation and mathematical logic.

The term "artificial intelligence" was coined for the first time in 1956, during a conference held at Dartmouth College in the United States, when John McCarthy, Marvin Minsky, Nathaniel Rochester and Claude Shannon proposed that machines could be created to simulate human intelligence as a result of a workshop that lasted 8 weeks[7]. Since then, research in AI has evolved through different periods. During the 1960s and 1970s, programs based on symbolic logic were developed for problem-solving and reasoning. During the 1980s and 1990s, there was an increase in the use of neural networks in order to create mathematical models that imitate the functioning of the human brain, following the initial idea of recreating human intelligence.

Nowadays, AI is one of the areas of technology that is rapidly evolving. The emergence of large amounts of data and the advancement of information and communication technologies have allowed the development of machine learning algorithms that are capable of improving their accuracy as they learn from data.

Forbes magazine published last April an article called *24 Top AI Statistics & Trends in 2023* [8] where it

details some predictions of how AI will impact society and the way to do anything. In terms of economy, the market size is expected to reach $407 billion by 2027, growing more than a 450% from the actual $68,9 billion and being an important contributor to the GDP of a lot of countries around the globe. But how is this impacting the economy so much? Some of the many examples include the following: The rapid growth of chatbots such as ChatGPT from OpenAI, which had more than a million users within the first five days of its release (in figure 2.1 the way consumers plan to use AI can be seen, with chatbots being the main tool to meet these demands); The increase of driverless vehicles, a product that is forecast to account for 10% of the global market by 2030; Voice search is on the rise, with 50% of U.S. mobile users using it daily, according to UpCity. This trend showcases the growing prevalence of AI-powered voice assistants in everyday life. These are some of the topics where AI is the centre of attention nowadays, but 64% of businesses expect AI to increase productivity in all kinds of jobs. This demonstrates the growing confidence in AI's potential to transform business operations.

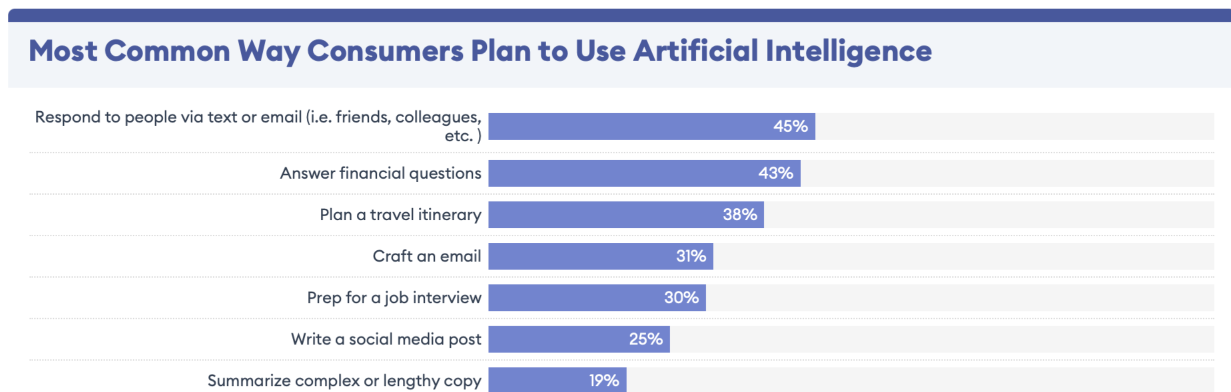


Figure 2.1: The most common way consumers plan to use AI. *Source [8]*

But apart from these good statistics, consumers are concerned about AI usage. As can be seen in figure 2.2, the topics that worry the most are related to product descriptions and reviews, as consumers do not quite trust AI and its training at all. Related to these concerns and the use of this technology in business as explained before, in figure 2.3 Forbes presented a pie chart answering "how likely are the consumers to trust a business that uses AI", revealing that at least 65% of them would not have problems with this change in the industry.

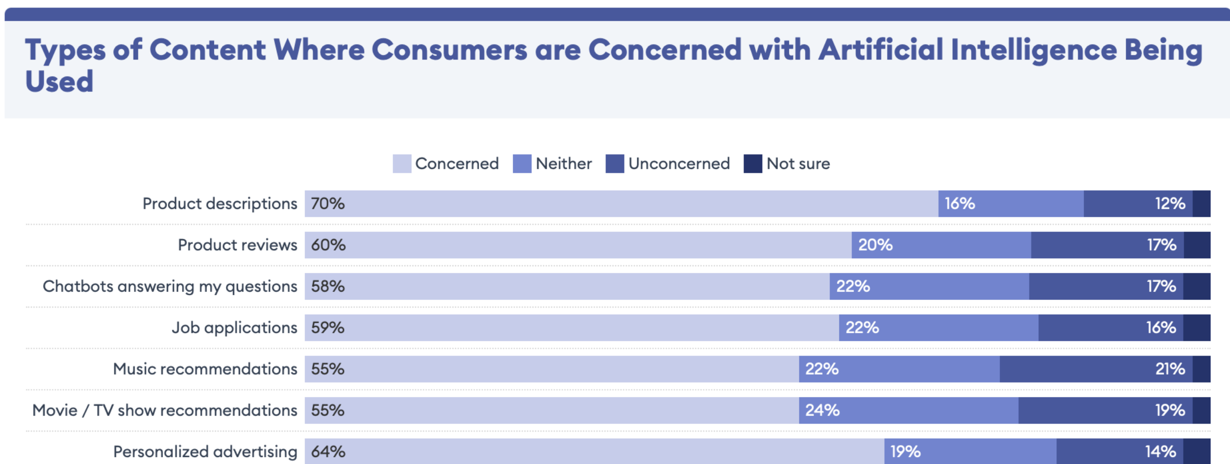


Figure 2.2: Consumer concerns about AI uses. *Source [8]*

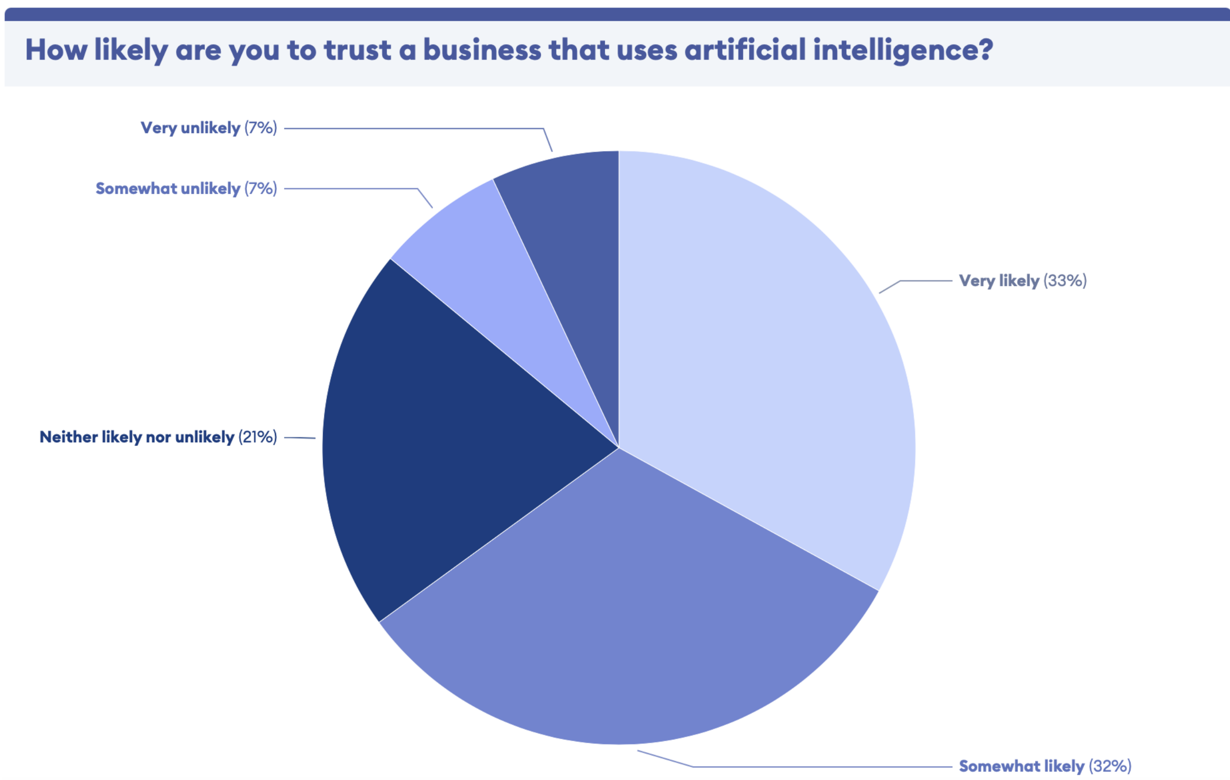


Figure 2.3: Consumer confidence in businesses using AI. *Source [8]*

2.2 Machine Learning

Machine learning (ML) is a branch of AI and computer science that focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy. [5].

It refers to the study of computer algorithms that use mathematical models to represent a sample of data, expecting that new inputs will be correctly predicted by the model. Inside machine learning is 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 deep learning, which, as its name suggests, is based on the addition of more layers to the model, enhancing the creation of more abstract features. Deep learning is most often based on deep neural network architectures or multilayer perceptrons (MLPs). (In figure 2.4 a Venn diagram of deep learning is shown.)

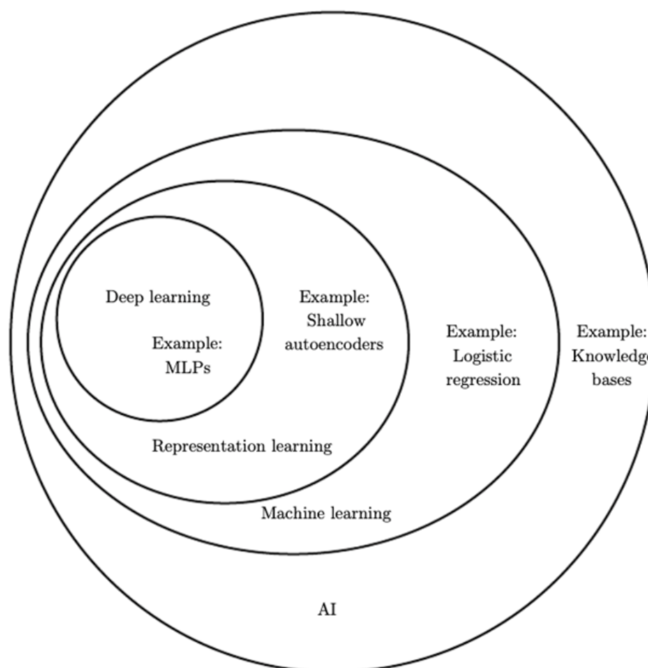


Figure 2.4: Venn diagram of deep learning. *Source [9]*

2.3 Generative AI

[10] Generative artificial intelligence is a type of AI system capable of generating text, images, or other media in response to prompts. Generative AI models learn the patterns and structure of their input training data, and then generate new data that has similar characteristics.

Generative AI has potential applications across a wide range of industries, including art, writing, software development, healthcare, finance, gaming, marketing, and fashion. Investment in generative AI surged during the early 2020s, with large companies such as Microsoft, Google, and Baidu as well as numerous smaller firms developing generative AI models. However, there are also concerns about the potential misuse of generative AI, such as in creating fake news or deep fakes, which can be used to deceive or manipulate people. [10]

A generative AI is constructed by applying unsupervised or self-supervised machine learning to a data set, and the capabilities of the system depends on it. These different modalities are the following: Text, Code, Images, Music, Video, Robot actions & Molecules

2.4 Autoencoders

AE structures can be seen as a particular case of feed forward neural networks, which try to learn when copying their input to their output. This may be understood as a composition of two functions: the encoder h = f(x) and the decoder r = g(h). The expectation is that the AE does not learn to perfectly copy x, but it is forced to learn to prioritise which information is useful.

There are several types of AE, but one of the simplest is the undercomplete AE, which reduces the input dimension in the encoder and reconstructs it in the decoder (see figure 2.5). 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 AE learns to span the same subspace as principal component analysis (PCA). One goal of this project is to try to assemble this behaviour and compare the results. Figure 2.6 shows how the inner layers of an AE can be used to compress images. But more surprisingly, this architecture allows for ”random” inputs at the bottleneck in order to generate images.

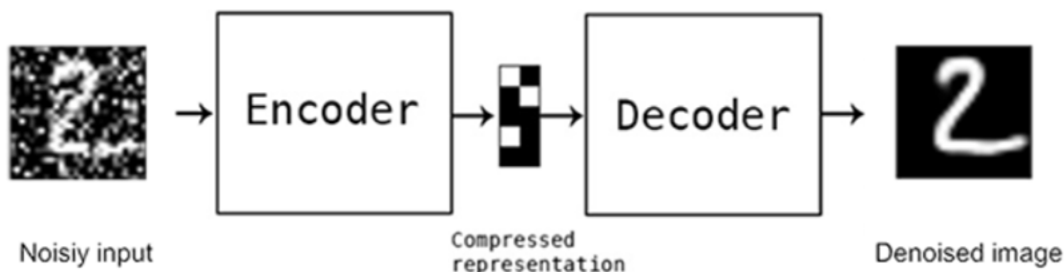


Figure 2.5: Representation of an undercomplete AE using the MNIST database. *Source [11]*

This figure is a perfect example of what AE are used for the most nowadays. Two interesting practical applications of AE are data denoising (which is seen in figure 2.5), and dimensionality reduction for data visualisation. With appropriate dimensionality and sparsity constraints, AE can learn data projections that are more interesting than PCA or other basic techniques.

2.5 Variational Autoencoders

VAE is a type of AE with added constraints on the encoded representations being learned. More precisely, it is an AE that learns a latent variable model for its input data. The training of a VAE is regularised to avoid overfitting and ensure that the latent space has good properties that enable the generative process. So instead of letting the neural network learn an arbitrary function, the program learns the parameters of a probability distribution modelling the data (latent attributes in figure 2.6). If you sample points from this distribution, you can generate new input data samples. A VAE is a ”generative model”.

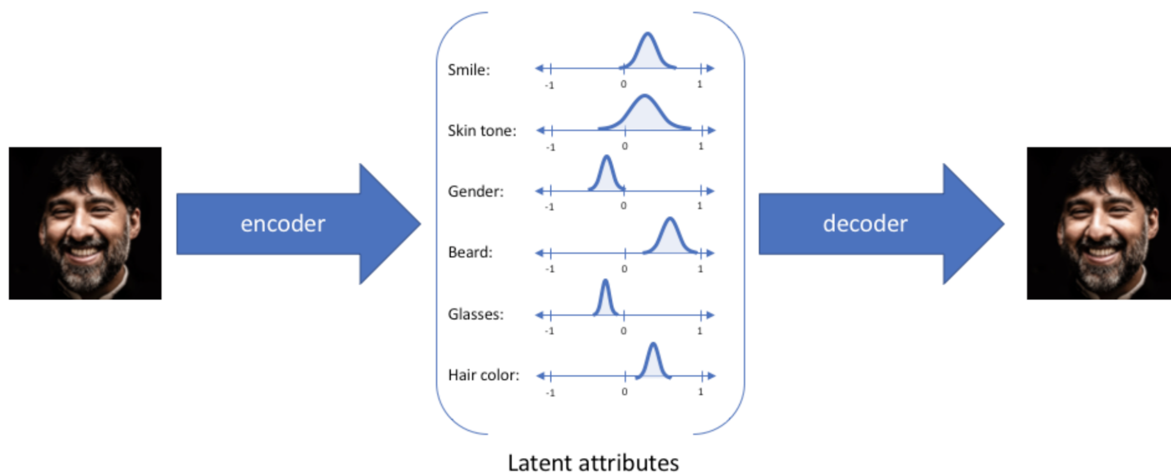


Figure 2.6: Schematic of a VAE showing an example of latent attributes. *Source [12]*

Computers are becoming increasingly creative, and VAEs are playing a big role in furthering that creativity. Some of the examples where VAEs are used the most nowadays are the following:

- Data generation: VAEs can produce fresh samples from the learnt latent space, resulting in the production of fresh data with a distribution like that of the training set. As a result, they are beneficial for projects like music composition, text production, and visual synthesis.
- Anomaly detection: VAEs can be trained on normal data samples and used to detect anomalies or outliers in new data. The normal data distribution is modelled by VAEs in order to find cases that significantly depart from the patterns that have been learned.
- Image and video compression: VAEs can be used for efficient data compression by learning a compact representation of high-dimensional data. The latent space is used by VAEs to encode and decode data, allowing for data compression while maintaining key characteristics.
- Learning representations: VAEs are capable of picking up useful latent representations of data. These representations are helpful for later tasks like classification, grouping, and retrieval because they can capture key aspects and structure in the data.
- Domain adaptation: By converting various domains into a single representation, VAEs can learn a shared latent space. This makes it possible to transfer information between domains, allowing models developed for one domain to excel in another that is related.
- Imputation and denoising: VAEs can be used to denoise noisy observations or to rebuild clean data from partial data by filling in missing values. VAEs are able to infer the most likely complete or denoised data points by learning the underlying data distribution.

It is important to note that VAEs are a powerful tool but may require careful model design, hyperparameter tuning, and training to achieve good results for specific tasks. Their flexibility and generative capabilities make them a popular choice in many machine learning applications.

Chapter 3

Theoretical background

The essential ideas and principles that underpin the subject at hand are thoroughly covered in this chapter. It acts as the basis for understanding the essential theoretical concepts that underlie the discussions and analyses them afterwards.

3.1 The classification problem

The classification problem refers to the task of categorising or assigning predefined labels or classes to input data based on their features or characteristics. The goal is to build a model or algorithm that can learn from labelled training data and then predict the class or category of unseen or future instances.

3.1.1 Logistic regression

Logistic regression is the name given to the model that was first used to attack the classification problem for a data set with only two classes. This model is founded on a linear transformation and on a logistic unit; equations 3.1 and 3.2 show these transformations, where β is the model parameter, x is the input, and z is the output. The logistic unit 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.

$$z = \beta \cdot x + \beta_0 \tag{3.1}$$

$$y = \frac{1}{1 + e^{-z}} \tag{3.2}$$

Equation 3.2 is the logistic unit, which is usually called sigmoid due to the shape of the function. Sigmoid is a very important function in Machine Learning, and it will be studied along with other activation functions

in section 3.3.3. In the previous thesis, logistic regression was studied in much greater depth and was one of the key focal points of the work. From there, some examples have been extracted to explain the theoretical basis of the current thesis. As one of those, in figure 3.1 the boundary line between the two groups found with the logistic regression is shown. Although the data set 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 the model, the points exactly on the line have a 50% probability of being men and 50% of being women.

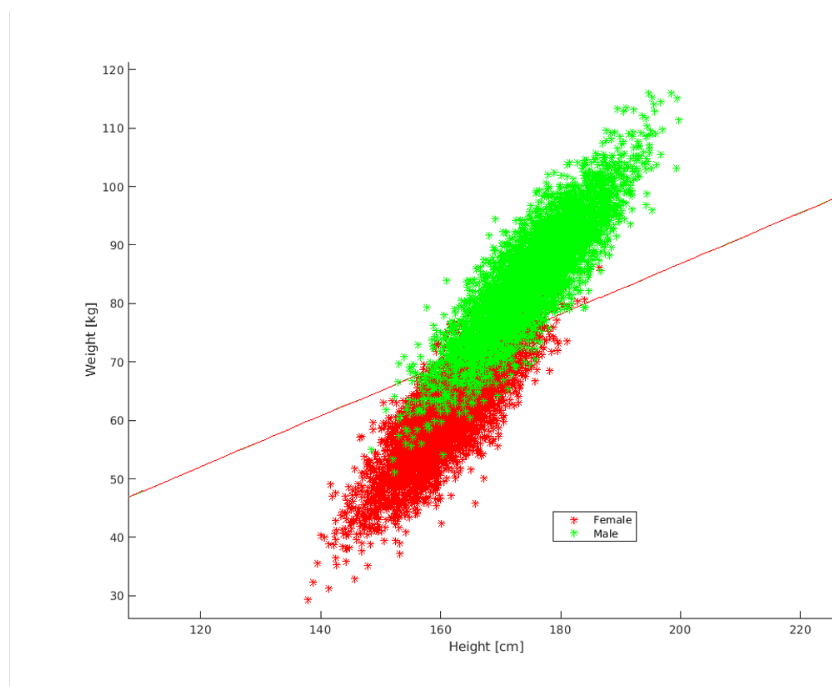


Figure 3.1: Logistic Regression for a Male/Female data set. *Source [1]*

3.1.2 Cost Functions

Cost function is another name for the objective function that appears in the optimisation problem in machine learning. A wide variety of loss functions may be defined depending on different interests. The importance of choosing the appropriate one for the involved task is enormous, as the minimization process will be strongly correlated with it. In particular, for linear regression tasks, the cost usually chosen is the mean squared error, or MSE, whereas in logistic regression, it is usually preferred to use the negative log-likelihood, also called cross-entropy.

The cost function is always composed of a loss function and a regularisation term. The loss is the function with which the error is strictly measured, while the regularisation term is sometimes added to introduce a penalty to the parameters in order to prevent overfitting. For more information, please refer to Section 5 of the thesis [1], where different regularisation techniques are debated. **Equation 3.3** shows the cross entropy loss function, which is the most used in NN; **Equation 3.4** shows the MSE function, which is the most used in linear regressions; **Equation 3.5** shows the norm L^{∞}, which is sometimes used in other models.

Negative log-likelihood has good properties in combination with neural networks. The fact that it uses a logarithm makes it work really well with models using exponentials; in addition, the logarithm squishes the subspace to numbers with smaller computation time for machines ($ln(1 \cdot 10^{-10}) \approx -23$).

$$L(\theta) = -\frac{1}{m}\sum_{i}^{m} y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(\hat{y}_i) \tag{3.3}$$

$$L(\theta) = \sqrt{\sum_{i}^{m} [(y_i - \hat{y}_i)^2]} \tag{3.4}$$

$$L(\theta) = max \{|y_1 - \hat{y}_1|, |y_2 - \hat{y}_2|, ..., |y_m - \hat{y}_m|\} \tag{3.5}$$

3.2 Regularisation

Regularisation techniques are introduced in machine learning to prevent overfitting. When there is little data or the model is deep enough, it is often the case that the model is capable of memorising 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 regularisation techniques, but the most commonly used are: (i) Parameter penalties; (ii) Early stopping; and (iii) dropout.

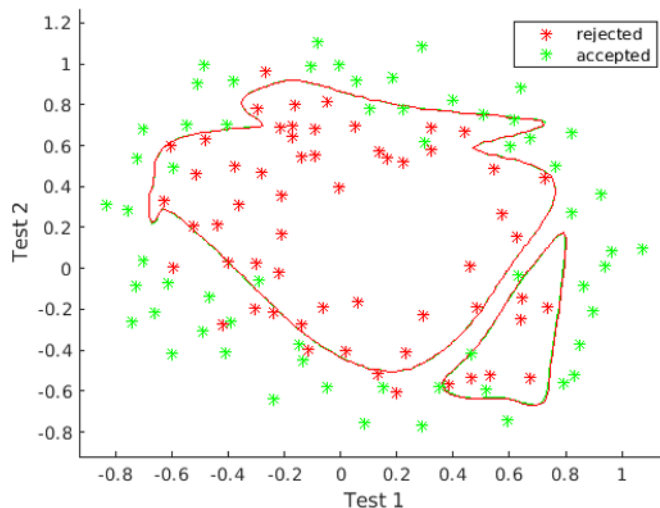


Figure 3.2: Example of overfitting in the microchip dataset. *Source [1]*

3.2.1 Parameter Penalties

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

$$J(\theta) = L(\theta) + \lambda \cdot R(\theta) \tag{3.6}$$

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

$$R(\theta) = \frac{\lambda}{2}\omega^T\omega \tag{3.7}$$

$$R(\theta) = \lambda||\omega||_1 \tag{3.8}$$

The first one is known as weight decay, which 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. Figure 3.3 tries to show how L^2 regularisation works in a function with two weights.

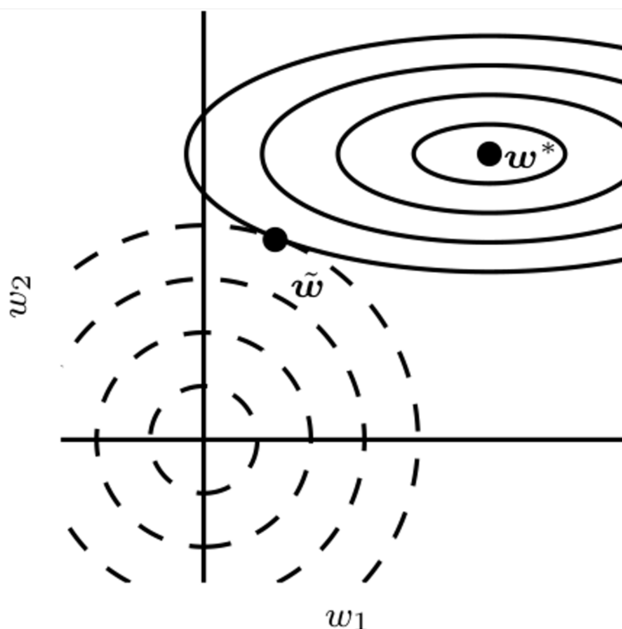


Figure 3.3: The effect of L2 regularisation. *Source [9]*

The dashed lines in figure 3.3 represent contours of L^2 while the continuous ones represent contour lines of the loss function. ω^* is the minimum of the loss function, but due to the influence of the L^2 regularisation, $\tilde{\omega}$ is the final optimal.

On the other hand, the L^1 regularisation adds the same constant penalty to all the weights in the model that surpass a certain boundary value, while the rest are pushed to 0. This regularisation 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.

3.2.2 Early stopping

Early stopping is a different regularisation method that does not rely on the modification of the cost function. In fact, what it does is 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 minimising the cost function with the training set at the expense of augmenting the test loss.

In this method, the analogous hyperparameter to λ in the penalty strategy is the optimum number of epochs or iterations before overfitting.

In figure 3.4 is clearly seen that it is much better to stop the minimization before the validation error starts to grow. This is done by storing the value of the parameters at the lowest validation error point and, after a few iterations of non-improvement in validation error, rescuing this value as the optimum found.

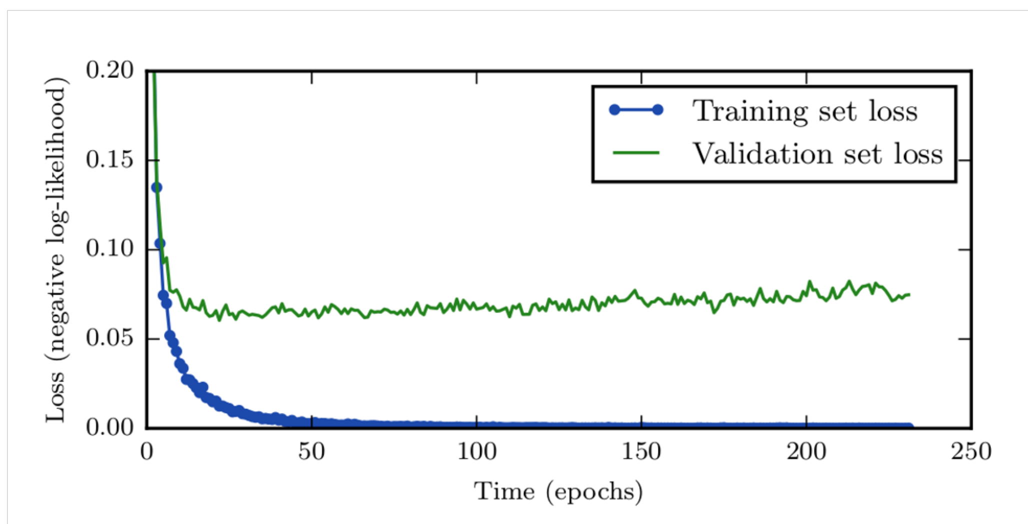


Figure 3.4: Learning curves showing the optimum point to stop before overfitting. *Source [9]*

3.2.3 Dropout

The last technique is the dropout. This is a relatively new technique compared to the others. It consists of training different models with different combinations of a ”base” model. To see the impact of suppressing neurons on the performance of the NN, in Figure 3.5 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 beyond the scope of this study.

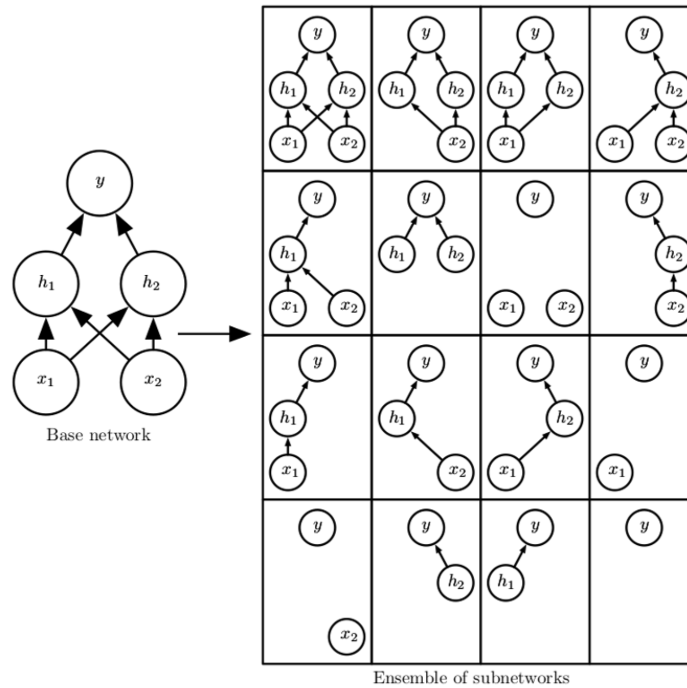


Figure 3.5: Sub-networks formed from a base network. *Source [9]*

3.3 Feed forward neural networks

Neural networks (NN) are one of the pillars sustaining deep learning. These structures are used in different ways and with different intentions, but in our case, 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.

The inputs for a specific neural network 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 $\in$ (0, 1) that symbolises probability. Finally, the operations that the network does in order to get a prediction are called hidden layers. As can be seen, some of these characteristics are the same as those in the logistic regression, so it was helpful to understand those first.

3.3.1 Architecture

Neural networks are composed of **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**.

Neuron

The neuron is the simplest unit of a NN. In Figure 3.6 its functioning is represented. It receives an input and performs a linear transformation to produce the output.

$$y = W^T \cdot x + b \tag{3.9}$$

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 θ = W, b. Therefore, the neuron performs the transformation y = f(x ; θ). The optimisation must find the θ that minimises the error of the predictions.

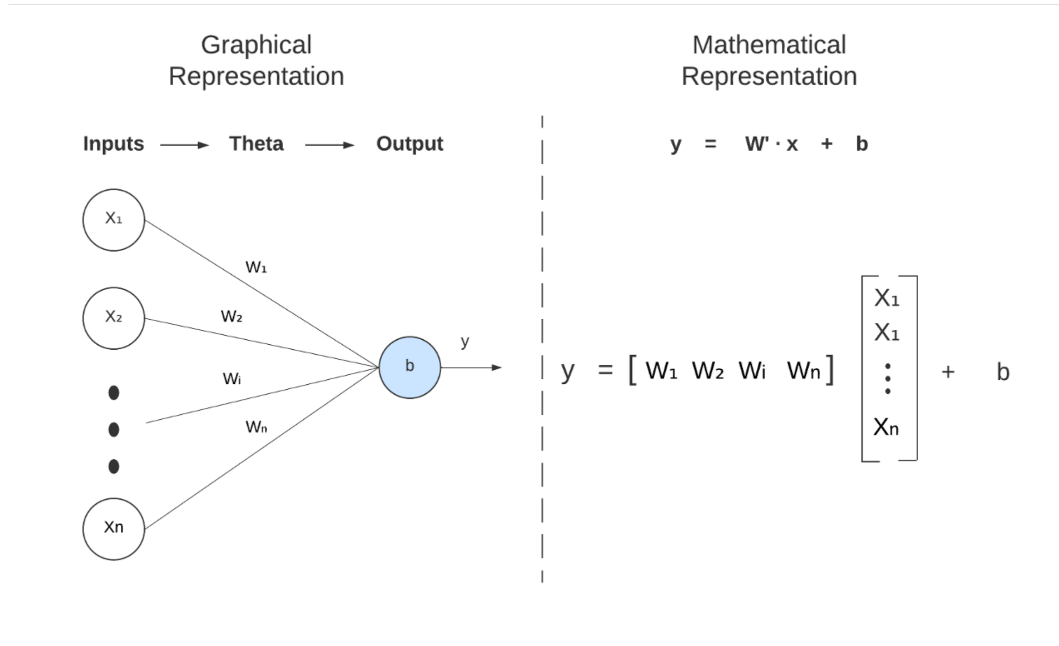


Figure 3.6: Representation of a neuron. *Source [1]*

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 3.7). 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 given outputs (where m is also the number of neurons in the layer). Equation 3.9 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 $n \times m$).

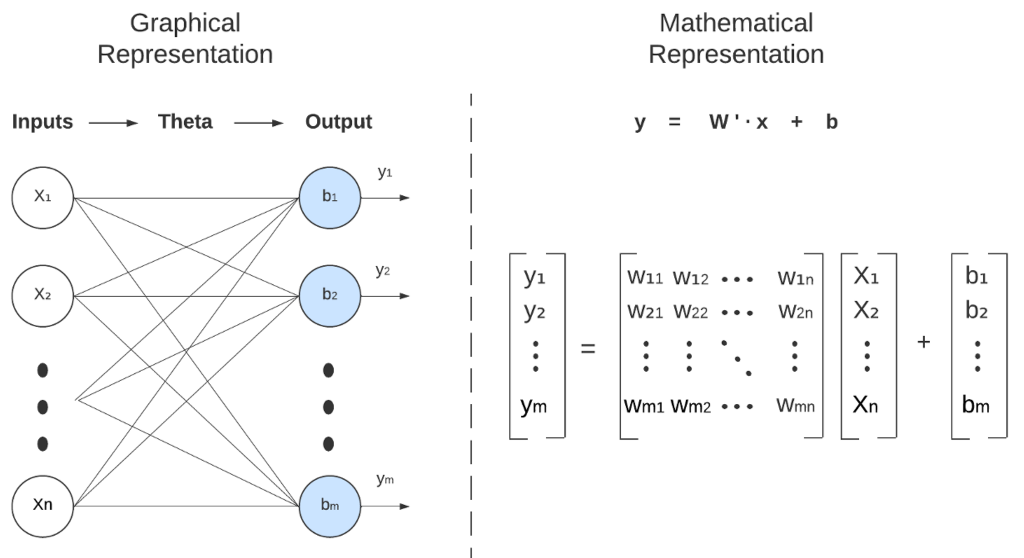


Figure 3.7: Representation of a layer. *Source [1]*

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 3.8 shows the representation of a neural network with two hidden layers.

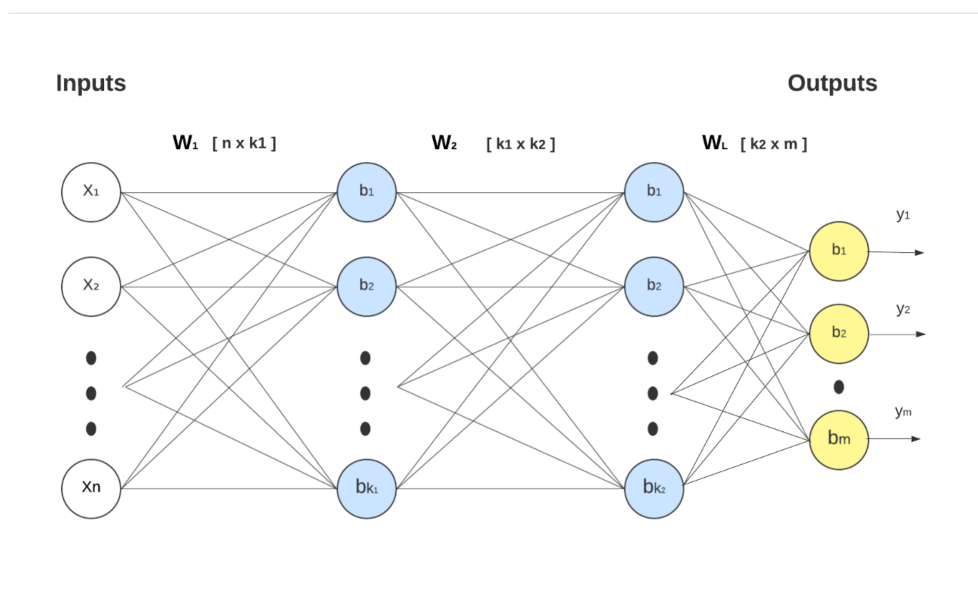


Figure 3.8: Representation of a neural network with depth L. *Source [1]*

3.3.2 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 takes the input x and traverses it through the network to the output to compute the predictions $\hat{y}$ and the cost $J(\theta)$. On the other hand, the backward propagation starts at 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. See Appendix A for a resume of this notation.

Forward propagation

Forward propagation is the act of concatenating the transformations to the input features until reaching the last layer output. From section 3.3.1, it is known that each layer performs a linear transformation; more details and one last transformation (the activation function) were required. Without entering into detail, this transformation can be seen as a function $g(h) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, therefore, a layer performs the transformation of equation 3.10.

$$f^k = g(W_k^T \cdot x + b_k) \tag{3.10}$$

Then, if the network has depth l, the forward propagation is defined as a composition of functions 3.11.

$$y = f^l(f^{l-1}(f^{l-2}(...))) \tag{3.11}$$

Notice that this composition has to be resolved from the inside to the outside, which means from the first layer to the last. In algorithm 1, a pseudo-code is shown of how a forward propagation algorithm looks. Where 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 is important to note that in pseudo-codes, the $\times$ symbol represents matrix multiplication and $\cdot$ represents element-wise multiplication. Also, remember that the notation of the variables refers 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 ← extract W B (θ)
2: $o_1 \leftarrow X$
3: $i \leftarrow 2$
4: **while** $i \leq nL$ **do**
5: $h \leftarrow o_{i-1} \times W_{i-1} + B_{i-1}$
6: $o_i \leftarrow activation_function(h)$
7: $i \leftarrow i + 1$
8: **end while**
9: $J \leftarrow cost_function(o, y)$

Backward propagation

Backward propagation, 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.

$$\frac{\partial y}{\partial x} = \frac{\partial f^{(l)}}{\partial f^{(l-1)}} \frac{\partial f^{(l-1)}}{\partial f^{(l-2)}} \cdots \frac{\partial f^{(1)}}{\partial x} \tag{3.12}$$

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 by applying the equation 3.12. Starting in the last layer and applying the chain rule, the gradient of the cost with respect to θ is the following:

$$\nabla_\theta J = \nabla_{\hat{y}} L(\hat{y}, y) \cdot \frac{\partial \hat{y}}{\partial h^{(l)}} \frac{\partial h^{(l)}}{\partial \theta} \tag{3.13}$$

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

$$\frac{\mathrm{d}h^{(l)}}{\mathrm{d}\theta} = \left(\frac{\partial h^{(l)}}{\partial b}, \frac{\partial h^{(l)}}{\partial W} \right) = \left(1, h^{(l-1)} \right) \tag{3.14}$$

Equations 3.13 and 3.14 compute only the gradient of the last layer. In order to propagate the error backwards, the gradient of the cost with respect to the activation function has to be multiplied by $W^{(k)}$. Then similarly to equation 3.13 there is:

$$\nabla_{\theta^{(k)}} J = \nabla_{o^{(k)}} J \cdot \frac{\mathrm{d}o^{(k)}}{\mathrm{d}h^{(k)}} \frac{\mathrm{d}h^{(k)}}{\mathrm{d}\theta^{(k)}} \quad where \quad \nabla_{o^{(k)}} J = W^{(k)T} \cdot \nabla_{o^{(k+1)}} J \tag{3.15}$$

This is the theory behind the backprop; the pseudocode of this method will help to make a deeper understanding of 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 the derivative of the cost respecting the activation function (δ). We define δ as

$$\delta^{(k)} = \nabla_{o^{(k)}} J = W^{(k)T} \cdot \delta^{(k+1)} \cdot \frac{\mathrm{d}o^{(k)}}{\mathrm{d}h^{(k)}} \tag{3.16}$$

See Algorithm 2 for for full details of backward propagation. The function *extractWB*(), decomposes all the parameters that are enclosed in θ into weights and biases. On the other hand, the function *reconstruct_grad*() does the opposite, building 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. Summarising all the concepts developed through this section, it is seen that back propagation is in contrast to **autodifferentiation**, an explicit way to compute the gradient.

Algorithm 2 Backward propagation

Require:
nL number of layers
θ parameters
X matrix of data inputs
y matrix of labels related to X
o output at every layer
Ensure:
$grad$ gradient of the cost function

```
1: W, B ← extract W B (θ)
2: i ← nL
3: while i ≥ 2 do
4:     o' ← activation_function_derivative(o_i)
5:     if i = nL then
6:         aux ← negLogLikelihood_derivative(y, o_end
7:         δ_i ← aux · o'
8:     else
9:         δ_i ← (W_{i−1} × δ^T_{i−1})^T · o'
10:    end if
11:    gradW_{i−1} ← (1/nD) o^T_{i−1} · δ_i
12:    gradB_{i−1} ← (1/nD) Σ δ_i
13:    i ← i − 1
14: end while
15: grad ← reconstruct_grad(gradW, gradB)
```

3.3.3 Activation Functions

The activation function (AF) is used to introduce non-linearity to the model. When these 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 such a way that their range is (0, 1).

The NN usually uses the same AF for all of its neurons. This is not 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 into consideration.

Sigmoid function

The sigmoid function was introduced in logistic regression, and it is the activation function more commonly used in feed-forward neural networks. 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:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.17}$$

Sigmoid AF is 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 behaviour with small weights. For the sigmoid function, the maximum value of its derivative is 0.25 (see figure 3.10). When this value is passed through the network, it causes the gradient to become smaller and smaller; this is known as the vanishing gradient problem.

Hyperbolic tangent function

The hyperbolic tangent AF has a similar response to the sigmoid function (both of them are exponential functions). In Figure 3.9 this AF is represented along with the sigmoid function. The function is defined as:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.18}$$

The implementation of the tanh function improves the training speed with respect to sigmoid as the gradient achieves higher values. Nevertheless, it has the same disadvantage as it only attains a gradient of 1 when the input is 0 (see figure 3.10). This still produces some dead neurons as a result of the vanishing gradient problem. This limitation pushed further the research of AF, and it birthed the rectified linear unit, or ReLU. Tanh overrides others in recurrent neural networks and natural language processing.

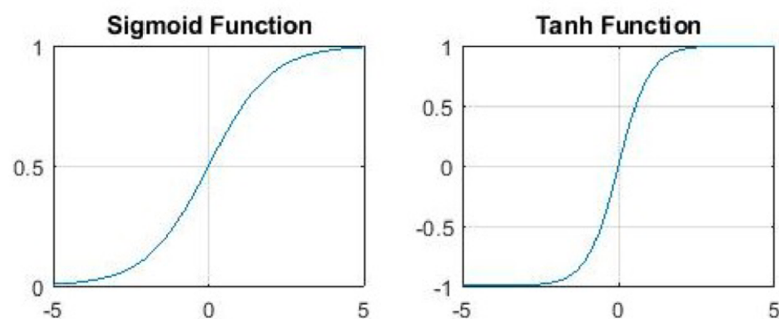


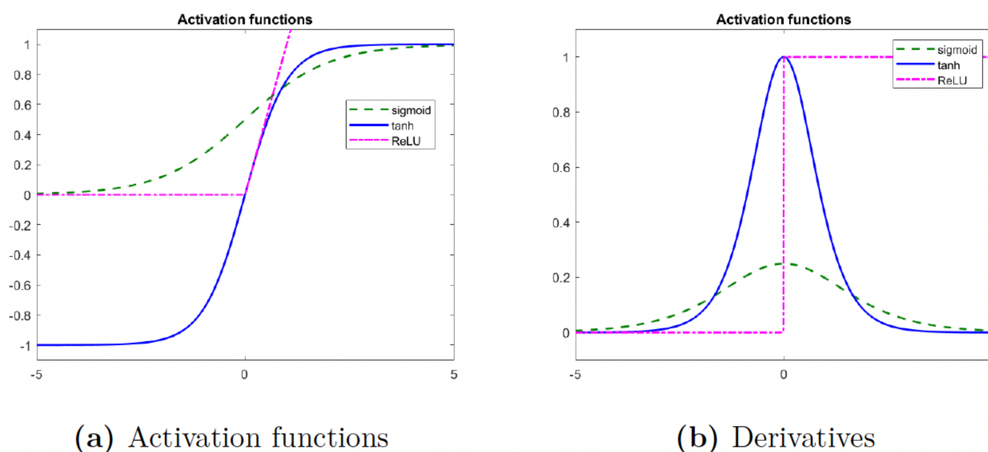
Figure 3.9: Representation of Sigmoid and Tanh response. *Source [1]*

ReLU function

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

$$f(x) = \begin{cases} 0 \quad if \quad x < 0 \\ x \quad if \quad x \geq 0 \end{cases} \tag{3.19}$$

Equation 3.19 grants high computation speed by avoiding exponentials and divisions. Moreover, it introduces sparsity as it squishes the values between zero and their maximum. The downside of ReLU as an 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. 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. In Figure 3.10 the three activation functions explored till this point are shown and compared.



(a) Activation functions **(b)** Derivatives

Figure 3.10: Comparison of ReLU, sigmoid and tanh AF. *Source [1]*

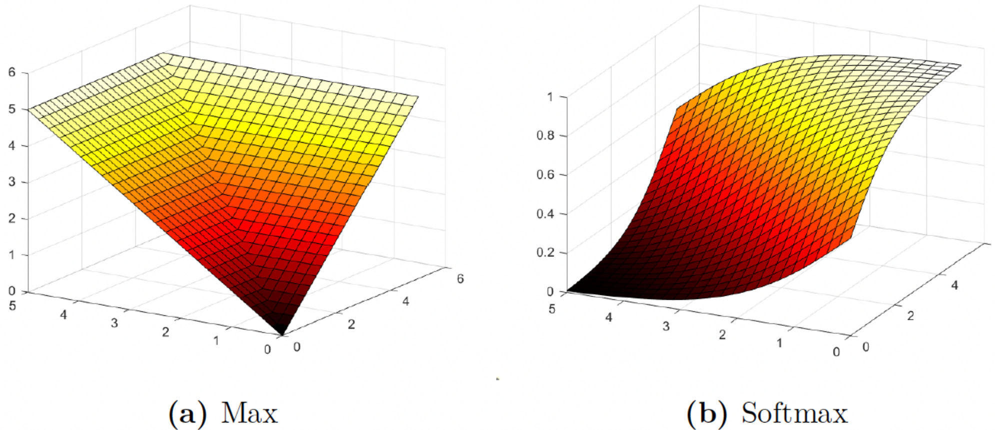
Softmax function

The softmax function is a normalised 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 sharing a similar exponential response, and sigmoid sharing the monopoly of being the two king output activation functions.

$$f(x) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3.20}$$

Softmax is the regularised version of the non-differentiable max function; figure 3.11 shows these two functions.

The softmax 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 two different classes.



(a) Max (b) Softmax

Figure 3.11: Comparison of max and softmax for the 2D case. Softmax shows its first output, it has the shame shape in each axis. *Source [1]*

Other activation functions

In the previous sections, the most known and used AFs 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 functions are the Softsign, Softplus, or ELU functions [10]. This study will limit the implementation to the four stated in previous sections.

3.4 Optimization method

Optimisation is a branch of mathematics that focuses on the labour of finding the best possible parameters in a function or model. In the simplest case, optimisation consists of maximising or minimising 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, refer to 3.1.2) and then trying to minimise this error with respect to the parameters θ.

In this chapter, only the basis of optimisation methods and the one used in the thesis will be explained, as this is something studied in the previous study [1].

3.4.1 The learning process

Once everything in the network is settled, it 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(θ). This process is usually made in an iterative way; at first, some random parameters of θ are initialised, and 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 until a local minimum is found.

The cost function J(θ) is a scalar multi-variable function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, it depends on a set of parameters θ_1, θ_2, ..., θ_n and returns a real number that represents the error of the predictions. This function has one partial derivative for each parameter, and these are enclosed in the term named gradient $\nabla J(\theta) = \left[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, ...\right]$, therefore the gradient is a vector that represents the ascending line in the maximum steep direction. First-order optimisation 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.

There are several approaches in the literature on how to tackle the learning process, differing in how the θ is updated. They are classified into first-order optimisation algorithms and second-order optimisation algorithms.

Although the second-order optimisation algorithms use the hessian matrix to update θ and are usually faster at converging to a local minimum, they are far from ideal due to their tendency to get stuck near saddle points. This disadvantage is added to the need for an inversion of the system of equations, which is very expensive, giving a higher computation cost. These properties make first-order algorithms, or just **gradient based** methods, more suitable for deep learning.

3.5 Gradient descent

The gradient descent is a first-order algorithm that is really easy to implement. The idea behind it is to slowly update the parameters of the function in the direction of the gradient (equation 3.21).

$$x' = x - \epsilon \cdot \nabla f(x) \tag{3.21}$$

The negative sign comes from the intention to minimise $f(x)$ (contrary to maximising, 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 hyperparameter called **learning rate** (LR). The selection of an appropriate LR is crucial and will be strongly correlated to the performance of the optimisation process. There are two approaches when selecting the LR: it is possible to either leave it static (usually a small constant) or change it over time. The latter is called adaptive LR. It is also possible to perform a line search, evaluate a few possible ϵ and choose the one that results in the smallest objective function value.

The algorithm 3 shows what a general gradient descent looks like. It has a main while loop with arbitrary optimisation criteria, and it calls the lineSearch() function, which finds the best LR among 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 optimisation criteria is to use the norm of the gradient; if it is close to 0, it means the function has at least reached a local minimum. Again, the notation used is the same as the one exposed in

Appendix A.

Algorithm 3 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: $\theta = \theta - \epsilon \cdot grad$
7: **end while**
8: $\theta_{OPT} \leftarrow \theta$

3.6 Stochastic Gradient descent

Stochastic Gradient Descent (SGD) is a variation of gradient descent. The need to modify algorithm 3 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 obtaining a practical and fast optimisation.

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. [9] optimisation methods that update the parameters using only the gradient are robust and can handle smaller batch sizes like 100, whereas second-order methods typically require much larger batch sizes like 10000.

Algorithm 4 is a variation of algorithm 3, where the difference is in line 3. A function called **minibatch** selects a small sample of points from the entire dataset. For consistency, 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 an epoch.

Algorithm 4 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: $\theta = \theta - \epsilon \cdot grad$
8: **end while**
9: $\theta_{OPT} \leftarrow \theta$

3.7 Autoencoder

As said in the brief introduction from chapter 2.4, AE are neural networks that aim to copy their inputs to their outputs. They work by compressing the input into a latent-space representation, and then reconstructing the output from this representation. This kind of network is composed of two parts: the encoder and the decoder. The architecture of the autoencoder is explained in figure 3.12.

The encoder compresses the input into a latent-space representation. It can be represented by the encoding function 3.22:

$$h = f\left(x\right) \tag{3.22}$$

And the decoder aims to reconstruct the input from the latent space representation. It can be represented by the decoding function 3.23:

$$r = g\left(h\right) \tag{3.23}$$

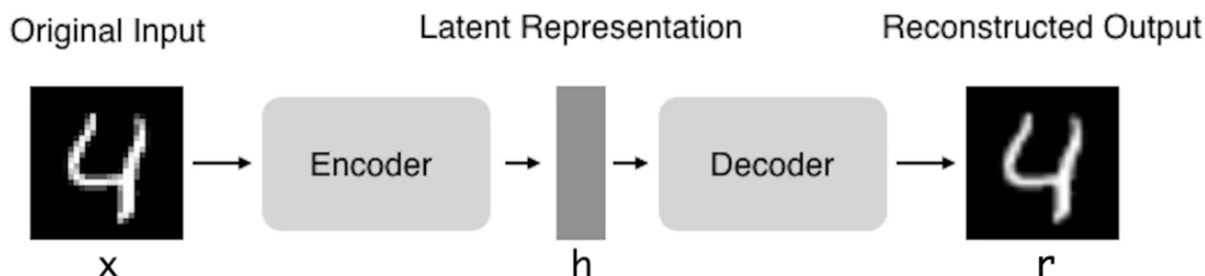


Figure 3.12: Architecture of an AE. *Source [13]*

The AE as a whole can thus be described by the function 3.24, where you want r to be as close as the original

input x.

$$r = g\left(f\left(x\right)\right) \tag{3.24}$$

So the lost function of the AE is expected to be like function 3.25

$$loss = ||x - r||^2 = ||x - g\left(h\right)||^2 = ||x - g\left(f\left(x\right)\right)||^2 \tag{3.25}$$

The overall AE architecture creates a bottleneck for data that ensures only the main structured part of the information can go through and be reconstructed. In the general framework, the family of considered encoders is defined by the architecture of the encoder network, the family of considered decoders is defined by the architecture of the decoder network, and gradient descent is used to find the encoder and decoder that minimise the reconstruction error.

Copying the input to the output could make AEs look useless. Indeed, by training the AE to copy the input to the output, the latent representation **h** will take on useful properties. By placing restrictions on the copying task, this can be accomplished. Constraining h to have smaller dimensions than x is one technique to force the AE to provide usable features; in this case, the AE is called undercomplete. By forcing the AE to learn the most important aspects of the training data, we train an undercomplete representation. If the AE is given excessive capacity, it may learn to replicate data without gaining any insight into how the data is distributed. This can also happen in the overcomplete scenario, where the dimension of the latent representation is larger than the input and the latent representation's dimension is equal to or greater than the input. In these cases, even a linear encoder and decoder can learn to duplicate the input to the output without understanding the data distribution in any useful way. In an ideal world, one might successfully train any AE architecture by selecting the capacity of the encoder and decoder as well as the code dimension based on the complexity of the distribution to be modelled.

There are different types of AE, but the most common are the following:

Denoising AEs create a corrupted copy of the input by introducing some noise. This helps to avoid the AE copying the input to the output without learning features about the data. These AEs take a partially corrupted input while training to recover the original undistorted input. The model learns a vector field for mapping the input data to a lower-dimensional manifold that describes the natural data to cancel out the added noise. [14]

Sparse AEs have hidden nodes greater than input nodes. They can still discover important features from the data. A generic sparse AE is visualised where the obscurity of a node corresponds with the level of activation. A sparsity constraint is introduced on the hidden layer. This is to prevent the output layer from copying input data. Sparsity may be obtained by adding additional terms to the loss function during the training process, either by comparing the probability distribution of the hidden unit activations with some low desired value or by manually zeroing all but the strongest hidden unit activations. Some of the most

powerful AIs in the 2010s involved sparse AE stacked inside deep neural networks. [14]

VAEs make strong assumptions concerning the distribution of latent variables. They use a variational approach for latent representation learning, which results in an additional loss component and a specific estimator for the training algorithm called the Stochastic Gradient Variational Bayes estimator. It assumes that the data is generated by a directed graphical model and that the encoder is learning an approximation to the posterior distribution where ϕ and θ denote the parameters of the encoder (recognition model) and decoder (generative model) respectively. The probability distribution of the latent vector of a variational autoencoder typically matches that of the training data much closer than a standard autoencoder. [14]

This thesis is based on VAEs, as the objective is to evolve the previous code into a functional AE and, in the future, evolve it into a VAE. For this reason, these are explained in more detail in the following section 3.8.

3.8 Variational autoencoders

Just as a standard AE, a VAE is an architecture composed of both an encoder and a decoder that is trained to minimise the reconstruction error between the encoded-decoded data and the initial data. However, in order to introduce some regularisation of the latent space, we proceed with a slight modification of the encoding-decoding process: instead of encoding an input as a single point, we encode it as a distribution over the latent space. The model is then trained as follows (see also figure 3.13):

- First, the input is encoded as distribution over the latent space
- Second, a point from the latent space is sampled from that distribution
- Third, the sampled point is decoded and the reconstruction error can be computed
- Finally, the reconstruction error is backpropagated through the network

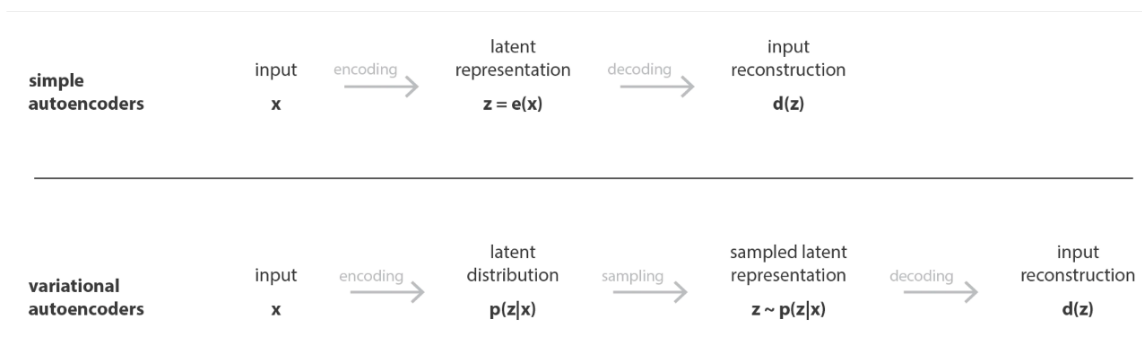


Figure 3.13: Difference between AE (deterministic) and VAE (probabilistic). *Source [15]*

In practise, the encoded distributions are chosen to be normal so that the encoder can be trained to return the mean and the covariance matrix that describe these Gaussians. The reason why an input is encoded as a distribution with some variance instead of a single point is that it makes it possible to express very naturally the latent space regularisation: the distributions returned by the encoder are enforced to be close

to a standard normal distribution. We will see in the next subsection that we ensure in this way both a local and global regularisation of the latent space.

Thus, the loss function that is minimised when training a VAE is composed of a ”reconstruction term” (on the final layer), that tends to make the encoding-decoding scheme as performant as possible, and a ”regularisation term” (on the latent layer), that tends to regularise the organisation of the latent space by making the distributions returned by the encoder close to a standard normal distribution. That regularisation term is expressed as the Kulback-Leibler (KL) divergence (explained after in this section) between the returned distribution and a standard Gaussian and will be further justified in the next section. We can notice that the KL divergence between two Gaussian distributions has a closed form that can be directly expressed in terms of the means and the covariance matrices of the two distributions.

Kulback-Leibler divergence

KL divergence is a non-symmetric metric that measures the relative entropy or difference in information represented by two distributions. It can be thought of as measuring the distance between two data distributions showing how different the two distributions are from each other. There is both a continuous form (equation 3.26) and a discrete form of KL Divergence (equation 3.27).

$$D_{KL}(p(x)||q(x)) = \int_{-\infty}^{\infty} p(x) ln \frac{p(x)}{q(x)} dx \tag{3.26}$$

$$D_{KL}(p(x)||q(x)) = \sum_{x \in X} p(x) ln \frac{p(x)}{q(x)} \tag{3.27}$$

Latent space

Latent space is a representation of compressed data. More often than not, data is compressed in machine learning to learn important information about data points. As a model learns, it is simply learning features at each layer and attributing a combination of features to a specific output. But each time the model learns from a data point, the dimensionality of the image is first reduced before it is ultimately increased. (see figure 3.14). When the dimensionality is reduced, this is considered a form of lossy compression.

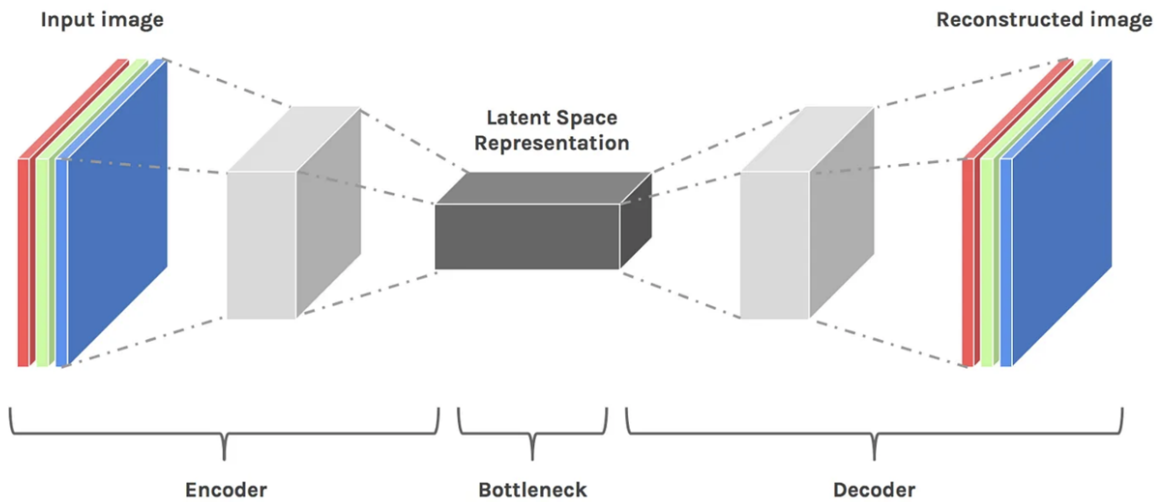


Figure 3.14: Depiction of convolutional neural network. *Source [16]*

Because the model is required to reconstruct the compressed data, it must learn to store all relevant information and disregard the noise. This is the value of compression: it allows us to get rid of any extraneous information and only focus on the most important features. This compressed state is the Latent Space Representation of the data.

But why is Latent Space important? The latent space representation of our data contains all the important information needed to represent our original data point. This representation must then represent the features of the original data. In other words, the model learns the data features and simplifies their representation to make them easier to analyse.

This is at the core of a concept called Representation Learning, defined as a set of techniques that allow a system to discover the representations needed for feature detection or classification from raw data. In this use case, our latent space representations are used to transform more complex forms of raw data (i.e., images, video) into simpler representations that are 'more convenient to process' and analyse.

Implementation

Rather than directly outputting values for the latent state as it would be done in a standard autoencoder, the encoder model of a VAE will output parameters describing a distribution for each dimension in the latent space. Since it is assumed that our prior follows a normal distribution, the output will be two vectors describing the mean and variance of the latent state distributions. If we were to build a true multivariate Gaussian model, we would need to define a covariance matrix describing how each of the dimensions is correlated. However, a simplifying assumption will be made that our covariance matrix only has nonzero values on the diagonal, allowing us to describe this information in a simple vector.

The decoder model will then generate a latent vector by sampling from these defined distributions and proceed to develop a reconstruction of the original input (see figure 3.15).

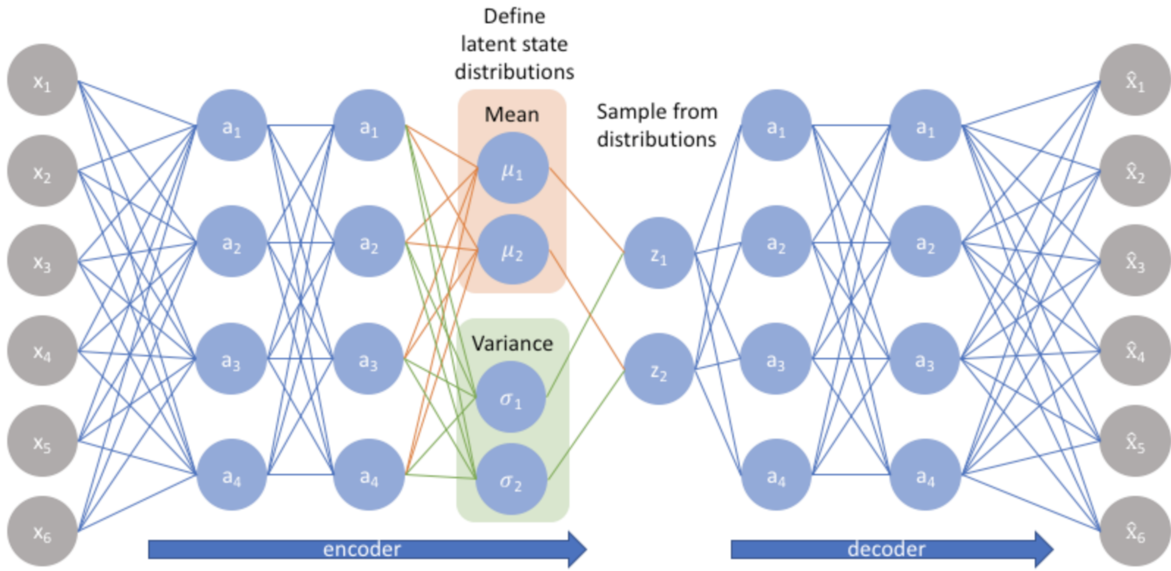


Figure 3.15: Architecture of a VAE. *Source [12]*

With this new parameters the loss function from equation 3.25 evolves to 3.28

$$L_{VAE} = \mathcal{L}(x, \hat{x}) + \sum_{j} KL(q_j(z|x)||p(z)) \tag{3.28}$$

However, this sampling process requires some extra attention. When training the model, we need to be able to calculate the relationship of each parameter in the network with respect to the final output loss using a technique known as backpropagation. However, we simply cannot do this for a random sampling process. Fortunately, we can leverage a clever idea known as the "reparameterization trick," which suggests that we randomly sample ϵ from a unit Gaussian and then shift the randomly sampled ϵ by the latent distribution's mean μ and scale it by the latent distribution's variance σ.

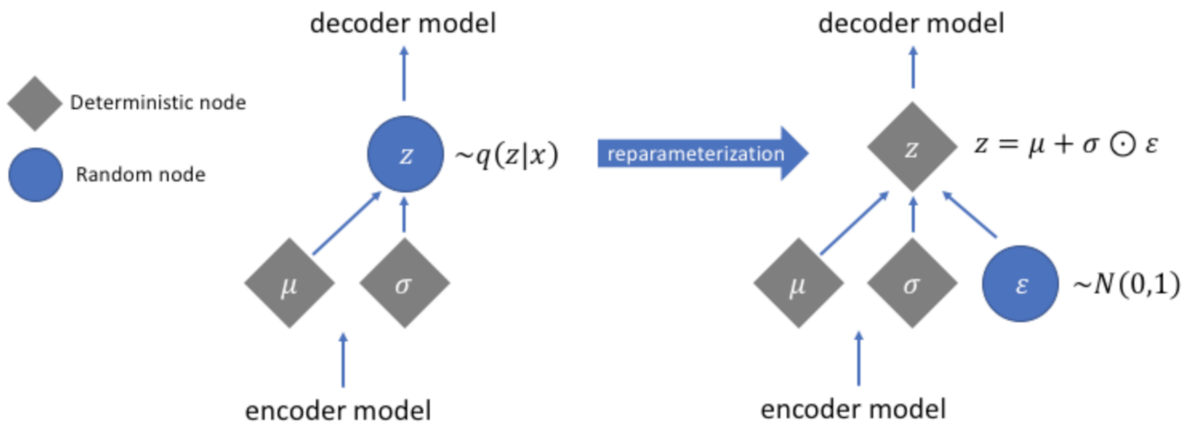


Figure 3.16: Introduction of the random node. *Source [12]*

With this reparameterization, we can now optimise the parameters of the distribution while still maintaining the ability to randomly sample from that distribution.

Chapter 4

Code refactoring

Now that all the bases on which this work is based have been introduced, it is time to explain where this thesis started. It has been explained earlier that it has not been part of nothing, but has continued the project *Study of efficient numerical tools for machine learning* ([1]) where an introduction to this topic was made and a code was developed capable of being trained and identifying what was asked, as could be numbers from the MNIST dataset or correctly carrying out the famous Iris Test.

The codes developed to build the artificial neural networks for the previous thesis are summarised in the Unified Modelling Language (UML) in figure 4.1. All the legacy code can be downloaded from this Github link.

4.1 Starting point

OOP was one of the pillars of the project to produce clean and extendable code. The strategy used to code the demand was the following:

The first step was to define the general structure of the code. The initial idea was to have three classes: Data, Network and Trainer, with network in composition with Data, and Trainer with Network. Sticking to this idea put up some hindrances during the development of the thesis.

In the end, quite a few more support classes appeared during the development of the project. Network owned a propagator, which was the central CPU where the mathematics occurred, and a plotter, which was in charge of the plots and graphics. The trainer was a template for both Fminunc from MATLAB and some gradient descent methods that inherit the functions.

The conclusions about the code structure that position the current thesis beginning, as the code has to be refactored, were the following:

"The code has been successfully implemented, it is extendable and not as easy to read as desirable but sufficient. In the end, when designing big codes is easy to get tangled up. The code can work with 8 different

data sets easily and can be extended to work with any dataset if saved in CSV. It also does what it was required, and achieves similar levels of accuracy as some libraries of deep learning." [1]

From here, the objective is to introduce the AE but also to make the code fancier and cleaner.

4.2 First steps

First of all, it is important to note that my knowledge about coding, OOP, and AI at the beginning of the project was limited, so a lot of work behind the final program has been done in order to finally perform the best I could. The training was done using *"The student's guide to clean code development"* and my tutor Alex's *teaching site*.

The first task is to learn what the code does and how to approach the refactoring needed. The UML (figure 4.1) has been created to understand how the code is organised. Also, a run test for the "Iris Test" has been done, and the Confusion Matrix has been plotted (figure 4.2, to understand what type of information this last gives us.

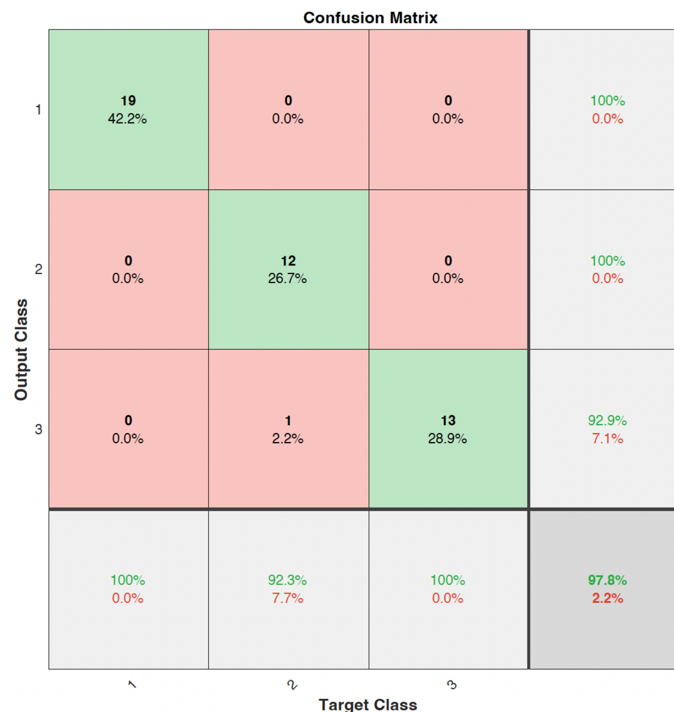


Figure 4.2: Confusion Matrix for the understanding run. *Source own*

Given that the test ratio was set at 30% in this figure and that the code was tested with 45 outputs, a total of 105 training flower classes were used. With only one error, the program mistook a "2 class" for a third class while matching 97.8% of the output classes with their targets. This gives us confidence that Toni's code is effective because both the training and the test were correctly carried out, and the software was able to almost exactly identify the Iris species (Iris setosa, Iris virginica, and Iris versicolor) that were the target just by knowing the size of the sepals and petals.

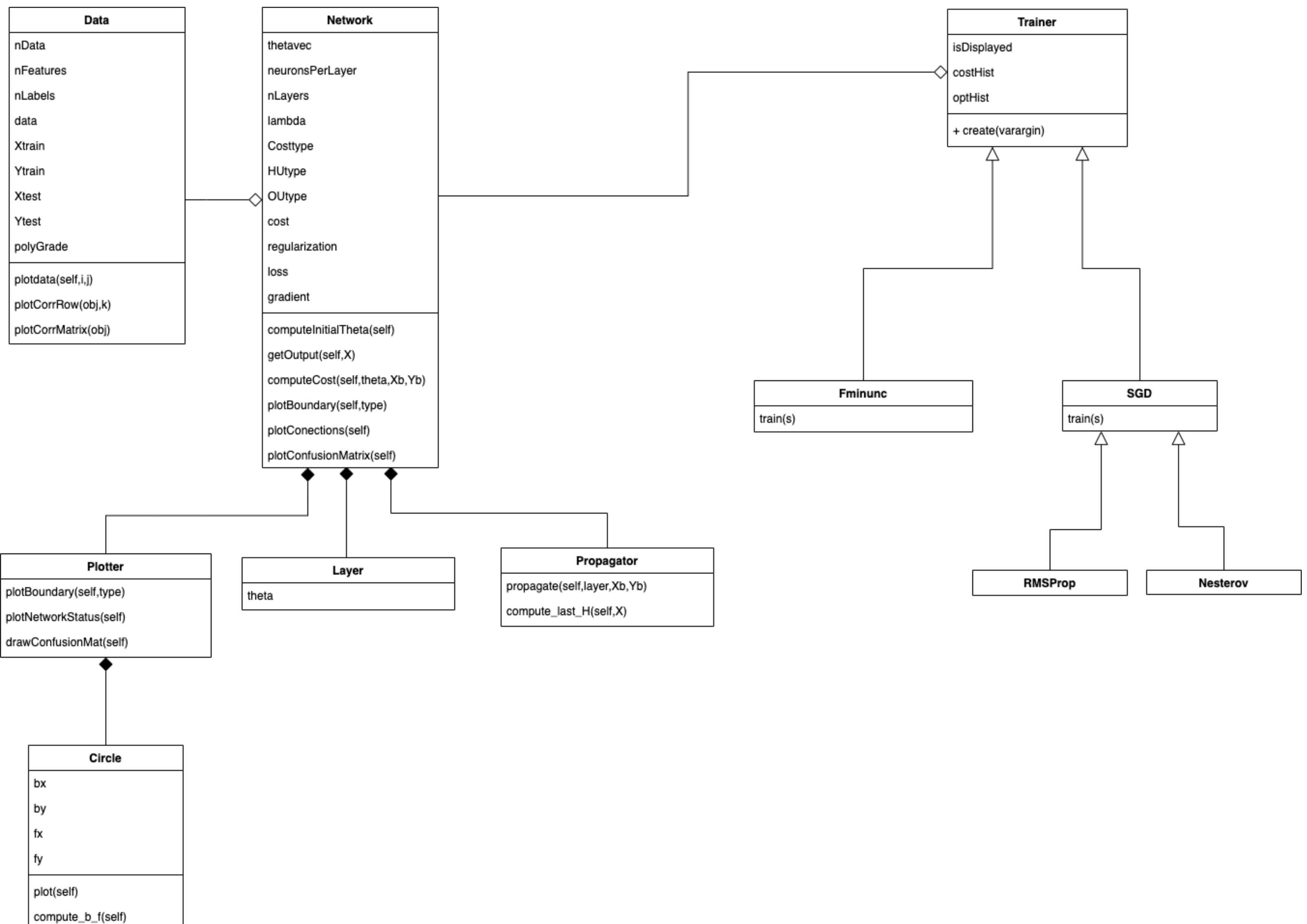


Figure 4.1: Previous thesis UML. *Source own*

After understanding what the code does and how it is organised, it is time to think about how to refactor it in order to work as an AE. For the system to be flexible and adaptable, the Trainer and Network classes must remain independent. By explicitly separating the training and generating duties, it will allow the code to build and improve its sample generation functionalities. So this is the first task to be performed.

The first idea is to create a class called "OptimizationProblem" in composition with Network, where the trainer is allocated, in order to separate everything optimisation-related from network. The structure of this intermediate refactored code is shown in figure 4.3.

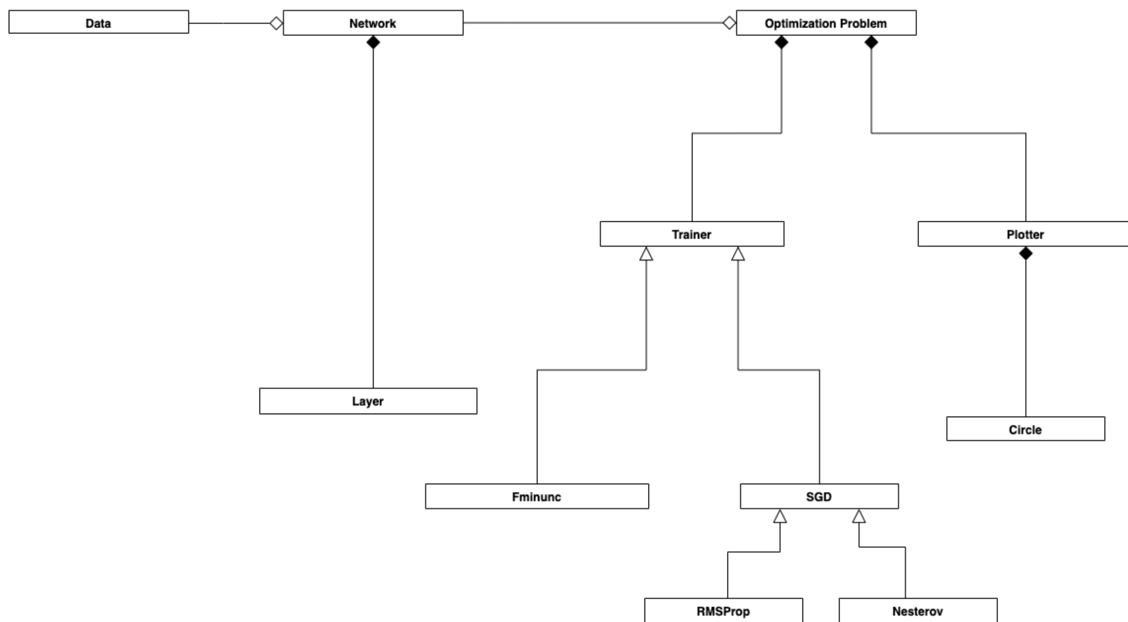


Figure 4.3: Intermediate UML. *Source own*

4.3 Final structure building

Now that "Network" and "Trainer" have been separated from each other and the propagator has been incorporated into the "OptimizationProblem" class, it is time to try to privatise the classes as much as possible in order to clear up the code and enhance its efficiency.

To accomplish this, it has been determined to develop a master class from which they will hang the rest and use it to initialise and call the required functions. The creation of this class also allows us to privatise as much as possible by allowing us to avoid moving numerous unneeded attributes from one class to another in order to reach their intended destination. From here, it has also been determined to reorganise the classes that have certain functionality so that each is responsible for its particular purpose, resulting in the structure shown in Figure 4.4.

Thus, the five classes that have been created are as follows:

The first, **LearnableVariables**, is responsible for initiating and saving the variable that is trained, as its own name indicates. It has been created in an independent class so that it is the same class that always keeps the θ and does not have to go guarding in each class and go calling from one to the other.

After that, the **Network** class finally integrates everything that the structure and navigation within it refer to. This implies then that the propagator is inside, as well as the activation functions or creating the neuronal network itself.

Now, we have a new class called **CostFunction**, which is responsible for calling the subclass that is required according to the approximation that you want to do. (L2 or a normal propagator).

The **Trainer** class is the mother of optimizers. Depending on what is selected, activate the required optimizer (usually the SGD) and interact with the rest only with cost and gradient, as AE requires.

Finally, the **Plotter** class, which has been adapted to the new structure but whose function remains the same as in the legacy code, graphically extracts certain results.

Now that we have the definitive structure, we must check that the code still works perfectly since we have only restructured it and no functionality has changed. For that, we will run an Iris test and check that the effectiveness of target vs. output remains above 90% (see results in Section 5.1).

4.4 Evolving into an AE

Once the code has the desired structure, is privatised with clean code practises, and is proven to extract valid results, it is time to transform the program into an AE. This means that now is time to move from a original "X $\rightarrow$ encoder-decoder $\rightarrow$ Y" neural network architecture to "X $\rightarrow$ encoder-decoder $\rightarrow$ X".

To do this, it is time to modify the only class that has remained intact from the inherited code, "Data". Having done all the refactoring work, a much easier scenario now arises, where what should be done is simply tell the program the expected way of producing the output. We moved from creating a vector based on the type of input we had (e.g., 10 in the MNIST, corresponding to all numbers from 0 to 9), to a matrix in the form of the input (784 columns in the MNIST, corresponding to all features of every image).

The final code for this thesis can be downloaded from this Github link.

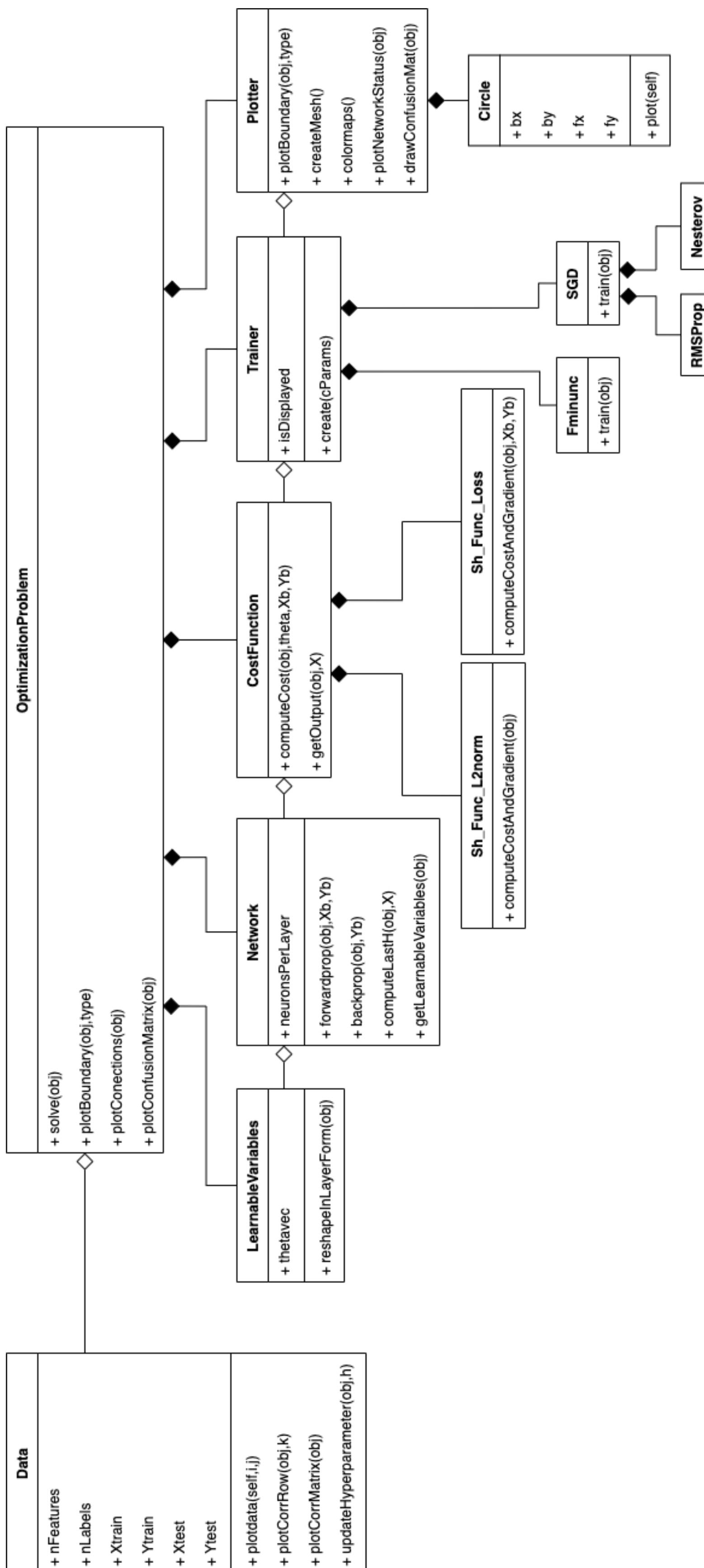


Figure 4.4: Final UML. *Source own*

Chapter 5

Results

Finally, the refactoring performed will be validated, and the performance of the AE developed in this thesis will be analysed. Two topics will be discussed: the influence of different activation functions and an analysis of the different optimisation parameters to improve the quality and performance of the program.

In this study, all the validation of refactoring has been carried out with the Iris dataset, since thanks to its reduced size, we obtained the results almost immediately. Still, for the AE, it has been decided to use the dataset MNIST, a world-known dataset containing 10,000 grey-scale images of 28x28 pixels (784 features) representing hand-written digits (see figure 5.1). The goal from the beginning was to generate images, and the fact that the images are numbers from 0 to 9 in white and black facilitates much both the visualisation of the final image as well as working with computers that are not very powerful to have a reasonable time-out. The original MNIST dataset is already separated into training and testing, 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 70/30. This has been done to reduce possible problems with memory and time.

Lastly, after all the optimisation parameters are discussed, the final results will be presented, and the effectiveness of the AE will be evaluated.

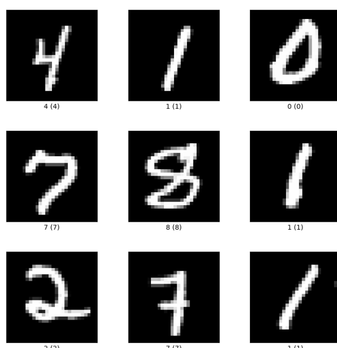


Figure 5.1: MNIST dataset. *Source [17]*

5.1 Refactoring validation

In this section, we will obtain the results of the identification program after performing the refactoring, thus validating its correct functioning.

The goal is to obtain an accuracy of more than 90%, which is the coincidence between the target stipulated by the dataset and the output that the program generates.

Until now, while we were working on the code, the continuous validations that were being made were done with the "Iris Test", as it worked with many fewer features, and that caused us to have the training almost immediately. But now that it's finished, a final validation has been made with the "Iris Test", as we knew it should work, and also with the dataset "MNIST". This has been done because it is with which the AE has been worked, so we needed to know that it identified well the images that entered it, and it has been possible to do since we had results of the code inherited in the previous thesis [1] that we have been able to compare.

Validations have been made using the loglikelihood cost type and the following activation functions: "ReLU" for hidden units and "Sigmoid" for output units.

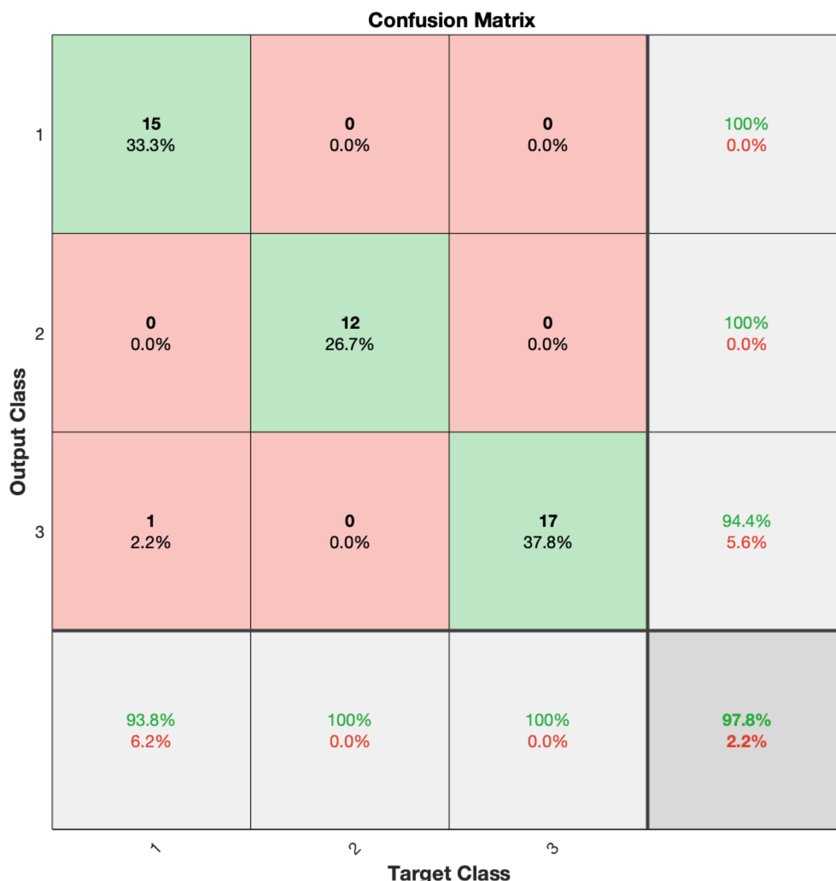


Figure 5.2: Refactoring validation: Iris Test. *Source own*

As can be seen in figure 5.2, only one of the 45 images tested has been identified wrong, giving a score of 97,8% valid answers.

Also in figure 5.3, the accuracy of the code identifying the numbers in the images is above 95%, which is considered a very good result.

Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	6	7	8	9	10	
1	291 9.7%	0 0.0%	7 0.2%	0 0.0%	1 0.0%	6 0.2%	3 0.1%	0 0.0%	1 0.0%	2 0.1%	93.6% 6.4%
2	0 0.0%	334 11.1%	0 0.0%	0 0.0%	1 0.0%	1 0.0%	0 0.0%	3 0.1%	0 0.0%	2 0.1%	97.9% 2.1%
3	0 0.0%	0 0.0%	293 9.8%	6 0.2%	2 0.1%	0 0.0%	0 0.0%	0 0.0%	2 0.1%	1 0.0%	96.4% 3.6%
4	0 0.0%	2 0.1%	3 0.1%	290 9.7%	0 0.0%	7 0.2%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	95.7% 4.3%
5	0 0.0%	1 0.0%	2 0.1%	0 0.0%	268 8.9%	2 0.1%	1 0.0%	0 0.0%	1 0.0%	9 0.3%	94.4% 5.6%
6	1 0.0%	1 0.0%	0 0.0%	5 0.2%	0 0.0%	239 8.0%	2 0.1%	1 0.0%	4 0.1%	0 0.0%	94.5% 5.5%
7	3 0.1%	0 0.0%	4 0.1%	1 0.0%	2 0.1%	4 0.1%	294 9.8%	0 0.0%	5 0.2%	1 0.0%	93.6% 6.4%
8	0 0.0%	2 0.1%	3 0.1%	4 0.1%	5 0.2%	0 0.0%	0 0.0%	291 9.7%	1 0.0%	4 0.1%	93.9% 6.1%
9	1 0.0%	1 0.0%	2 0.1%	2 0.1%	1 0.0%	2 0.1%	1 0.0%	0 0.0%	276 9.2%	1 0.0%	96.2% 3.8%
10	0 0.0%	0 0.0%	0 0.0%	1 0.0%	2 0.1%	3 0.1%	0 0.0%	6 0.2%	3 0.1%	278 9.3%	94.9% 5.1%
	98.3% 1.7%	97.9% 2.1%	93.3% 6.7%	93.9% 6.1%	95.0% 5.0%	90.5% 9.5%	97.7% 2.3%	96.7% 3.3%	93.9% 6.1%	93.3% 6.7%	**95.1% 4.9%**

Figure 5.3: Refactoring validation: MNIST. *Source own*

Taking these results into account, we can confirm that the code is working properly, so the refactoring has been done satisfactorily.

5.2 Activation Function Decision

Although it is possible to use different activation functions for different layers and even for different neurons, the NN studied for the AE will only use one type of activation function for all the hidden units and one for the output layer, as it is the only configuration where the code optimises properly.

To talk about the only valid configuration, we must understand the problem that was raised. When we first ran the code, we realised that the cost function was stable but not minimised, and we did not get the results we expected. To verify this, we decided to implement a function that would allow us to display a random image of what our code generated.

In Figure 5.4, we can see the result we obtained with this simulation next to what was expected, concluding that it was not what we were looking for. From here, we started looking for the problem and saw that ”a” (x of the intermediate layers) was not updated during the iterations. Having found the problem, it was only necessary to find the solution.

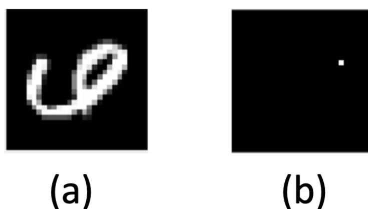


Figure 5.4: Comparison between target (a) and output (b) in the first run of the code as an AE. *Source own*

After iterating various optimisation parameters and seeing that the result remained unchanged, we realised that what might be happening is that we were using an incorrect activation function. After discussing it, it was concluded that the error was in the selected activation function (at that time, it was a sigmoid) for the hidden units.

We know that the ReLU function prevents the problem of gradient fading that can occur with the sigmoid function. Gradient fading is a phenomenon in which gradients become very small as they spread backwards through deep layers, which makes effective model training difficult. The ReLU function helps mitigate this problem by keeping gradients more stable and preventing their fading. In addition, another thing that benefits from the change is that the ReLU activation function is computationally more efficient than the sigmoid, meaning it requires less calculation time during the model training.

So the decision has been made to use the ReLU activation function for the hidden units. After applying this change, the AE extracted the first reasonable result, that can be seen in the first iteration of the optimization parameters in figure 5.5.

5.3 Optimization parameters iteration

From now on, where we already have the correct activation functions and the code works, it is time to iterate the different optimization parameters to find the minimum cost function while maintaining reasonable

training times.

After running the code several times of test and changing various parameters without following any pattern, we see that whenever it converges, with the values in which we are moving, the stop criterion is to reach the "MaxEpochs", so the rest of the parameters will be kept during the simulations as indicated in table 5.1.

Parameters	Value
Learning Rate	0,01
Lambda	0
TestRatio	30
MaxFunEvals	2000
MaxEpoch	58
OptTolerance	10^{-6}
timeStop	Inf([1,1])
fvStop	10^{-4}
nPlot	1
Batch size	200

Table 5.1: Initial values before any iteration. Green values are those that will remain constant throughout all iterations.

To evaluate the level of the result obtained in each iteration of some parameter, the easiest way is to look at the images from which we extracted the code after each run and compare them with others to see which is better. But this method is very subjective and no longer only depends on the criteria that we follow when looking at it; when plotting a random image (every time the X is created in a different order), we do not always get the same number, and some look more like its target than others. Therefore, the most appropriate way to control the proper functioning of the AE is by looking at the cost function and its final value. The smaller it is, the better the result obtained. There are no more ways to evaluate it since, unlike the inherited code where it should hit a result already stipulated, with the generation of images we do not seek to scrap the target image but learn from it and create a similar one.

Five possible parameter analyses have been performed: the Learning rate, the Test ratio, batch size, MaxFunEvals, and Lambda, all using a NN architecture of 784 - 250 - 50 - 250 - 784. Finally, an architecture analysis has been done. The number following the letter represents how many tests have been done. For example, A1_T3 represents the third test of the first architecture, and A5_T1 is the first one of the fifth architecture. In the tables, only the most significant tests are shown, but up to 30 tests with different parameters have been performed to find the optimal optimisation parameters, and 10 architectures were tried.

The analyses have been done one after the other to optimise time. There is a possibility that some combination with two values that are not optimal ends up giving a better overall result, but there is no criterion to evaluate this, and a lot of trial and error time would be needed. Also, the only optimisation method used has been SGD, as an initial trial of Fminunc did not support the amount of data needed to train this AE.

5.3.1 Learning rate analysis

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

Figure 5.5 corresponds to the first test performed, where we can clearly see a similar shape between the target and the output, but this last has a very low image quality. Figure 5.6 is the representation of the best scenario achieved by iterating the Learning Rate.

Parameters	A1_T1	A1_T2	A1_T3	A1_T9
Learning Rate	0,01	0,1	0,001	0,04
Results				
Elapsed time	43,89	0,82	43,63	42,51
Final Cost Fnc	86,83	5630,74	169,8	74,84
Nº Epochs	58	1	58	58
Nº Iterations	1994	34	1994	1994
Stop Criteria	MaxEpoch	OptTol	MaxEpoch	MaxEpoch

Table 5.2: Learning rate analysis results.

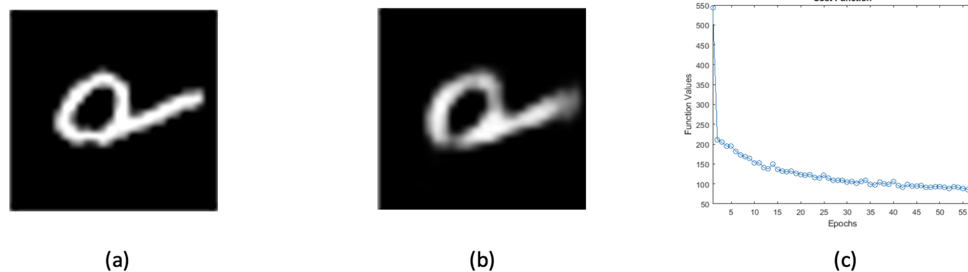


Figure 5.5: Results of test A1_T1. Target (a); Output (b); Cost Function (c). *Source own*

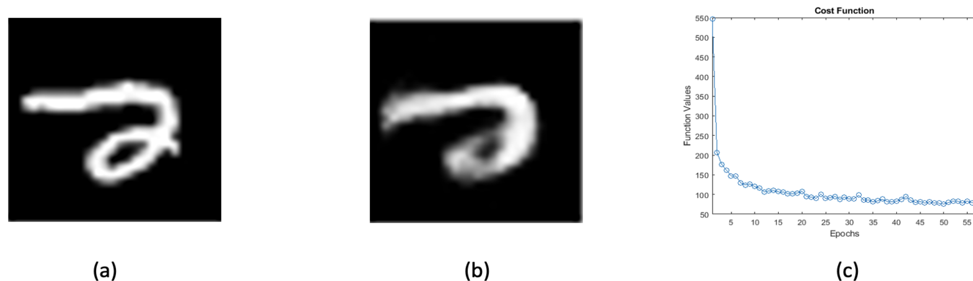


Figure 5.6: Results of test A1_T9. Target (a); Output (b); Cost Function (c). *Source own*

5.3.2 Test Ratio analysis

The Test ratio has been analysed just to see if there is a significant change in the results. This actually affects the percentage of data used to train the AE and what is left to test it.

After three different tests, the conclusion has been that the cost function does not change significantly, so the test ratio will stay at 30%.

Parameters	A1_T12	A1_T13	A1_T14
Test Ratio	10	20	50
Results			
Elapsed time	41,05	42,23	44,6
Final Cost Fnc	78,34	77,05	77,32
Nº Epochs	45	50	80
Nº Iterations	1979	1999	1999
Stop Criteria	MaxEpoch	MaxEpoch	MaxEpoch

Table 5.3: Test Ratio analysis results.

5.3.3 Batch size analysis

The concept of batch size is explained in section 3.6. What we can see here is that a very large batch size can slow down the minimization considerably, even increasing the number of MaxEpochs, so that we do not reach a final result of the cost function sufficiently well.

Figure 5.7 is a clear example of an AE with a cost function that is not sufficiently minimised. After some iterations, the optimal result was the initial one of a batch size of 200, represented in figure 5.6.

Parameters	A1_T15	A1_T16	A1_T21	A1_T9
Batch Size	2000	600	100	200
Results				
Elapsed time	276,79	98,54	24,69	42,51
Final Cost Fnc	204,5	73,7	88,52	74,84
Nº Epochs	572	172	29	58
Nº Iterations	1712	1879	1959	1994
Stop Criteria	MaxEpoch	MaxEpoch	MaxEpoch	MaxEpoch

Table 5.4: Batch size analysis results.

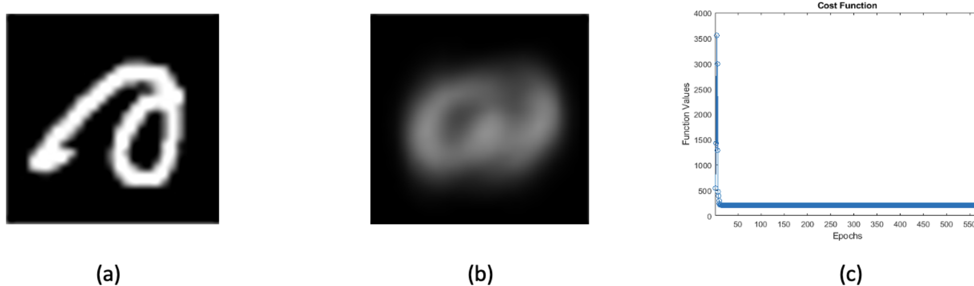


Figure 5.7: Results of test A1_T15. Target (a); Output (b); Cost Function (c). *Source own*

5.3.4 MaxFunEvals analysis

In practical terms, MaxFunEvals determines the maximum number of iterations or evaluations that the optimisation algorithm will perform before terminating. If the maximum number of function evaluations is reached without finding a satisfactory solution, the algorithm may stop and return the best solution found so far. In our case, it directly affects the MaxEpochs that our program does, so it also determines (but in a different way) how long the training process will be.

It does not affect the minimization itself, so the point here is to find a duration of the training that is good enough in terms of minimising the cost function without being too long.

Parameters	A1_T22	A1_T27	A1_T28	A1_T30
MaxFunEvals	3000	5000	15000	100000
Results				
Elapsed time	64,31	109,64	323,17	2313,33
Final Cost Fnc	72,04	64,74	59,62	53,13
Nº Epochs	86	143	429	2858
Nº Iterations	2974	4969	14980	99994
Stop Criteria	MaxEpoch	MaxEpoch	MaxEpoch	MaxEpoch

Table 5.5: MaxFunEvals analysis results.

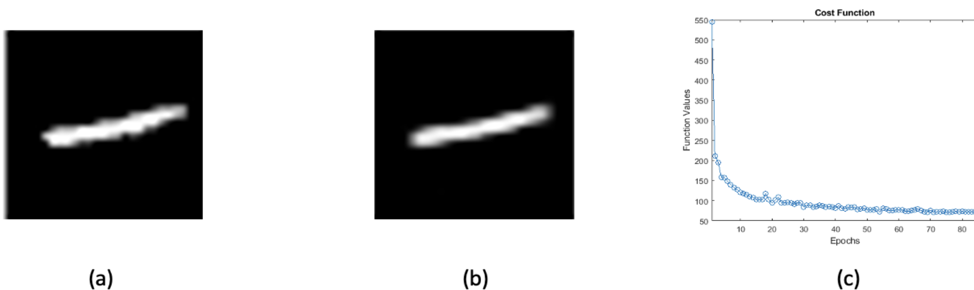


Figure 5.8: Results of test A1_T22. Target (a); Output (b); Cost Function (c). *Source own*

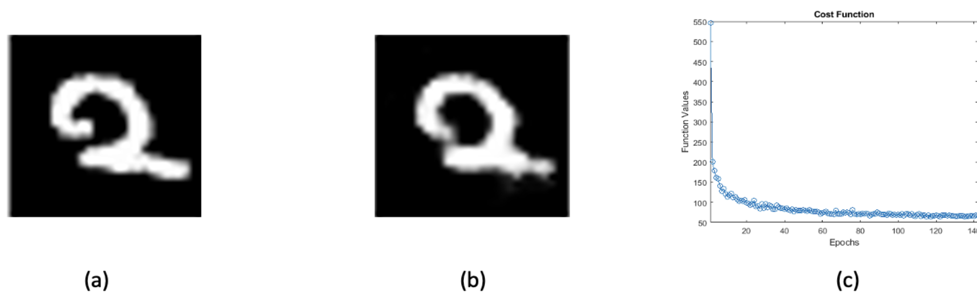


Figure 5.9: Results of test A1_T27. Target (a); Output (b); Cost Function (c). *Source own*

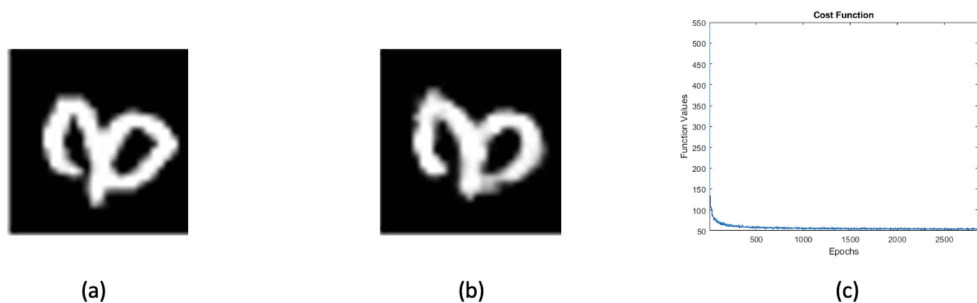


Figure 5.10: Results of test A1_T22. Target (a); Output (b); Cost Function (c). *Source own*

5.3.5 Lambda analysis

With such a large dataset as the MNIST and inside an AE that uses a lot of variables, the idea of using lambda seemed unuseful.

After doing some iterations, our hypothesis was validated, so the lambda stayed at 0.

Parameters	A1_T24	A1_T25	A1_T26
Lambda	1	0,5	0,1
Results			
Elapsed time	42,13	42,05	42,15
Final Cost Fnc	239,2	248,56	151,52
Nº Epochs	58	58	58
Nº Iterations	1994	1994	1994
Stop Criteria	MaxEpoch	MaxEpoch	MaxEpoch

Table 5.6: Lambda analysis results.

5.3.6 NN Architecture analysis

Finally, after having the optimisation parameters that give us the best result, the architecture has been iterated to find the values that give us the minimum cost function. This is achieved by avoiding the overfitting that could appear with a too complex NN.

The optimal architecture was found to be 784 - 500 - 100 - 500 - 784, giving us the minimum cost function seen from every iteration.

Parameters	A2_T1	A4_T1	A6_T1	A10_T1
Architecture	784			
	500			
	250	784		
	100	100	784	784
	50	50	500	500
	10	10	50	100
	50	50	500	500
	100	100	784	784
	250	784		
	500			
	784			
Results				
Elapsed time	175,23	75,32	148,93	153,37
Final Cost Fnc	205,18	110,371	61,81	59,64
Nº Epochs	143	143	143	143
Nº Iterations	4969	4969	4969	4969
Stop Criteria	MaxEpoch	MaxEpoch	MaxEpoch	MaxEpoch

Table 5.7: Architecture analysis results

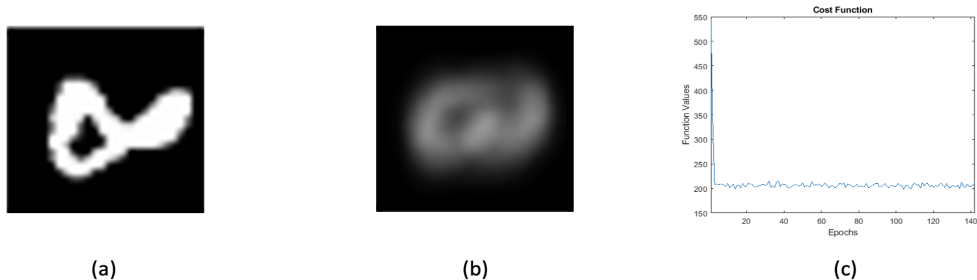


Figure 5.11: Results of test A2_T1. Target (a); Output (b); Cost Function (c). *Source own*

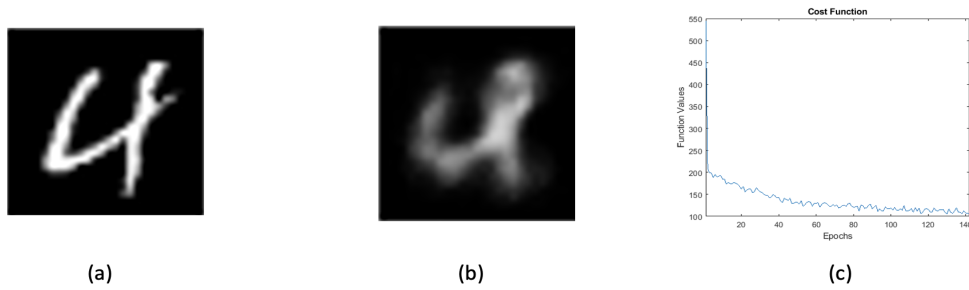


Figure 5.12: Results of test A4_T1. Target (a); Output (b); Cost Function (c). *Source own*

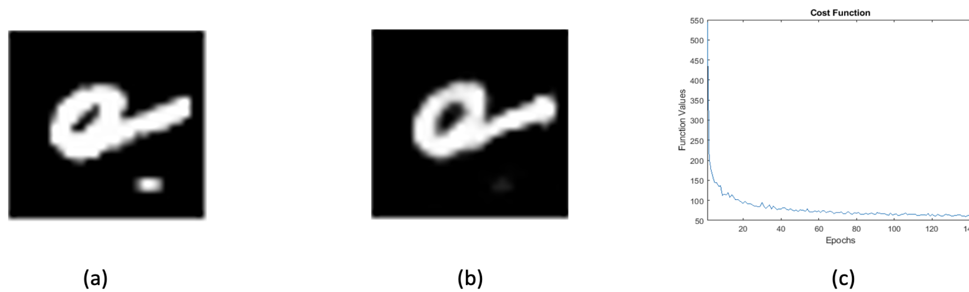


Figure 5.13: Results of test A6_T1. Target (a); Output (b); Cost Function (c). *Source own*

5.4 Optimization evolution

The purpose of this section is to show how the program is being trained and the relation between the output and the value of the cost function. In figure 5.14 the target and the cost function of one run of the code are shown. In figure 5.15, the evolution of the output is shown. Every image represents the output at the end of every epoch, except for the last two images, which represent the 80th epoch and the last one, as all these last outputs were very similar.

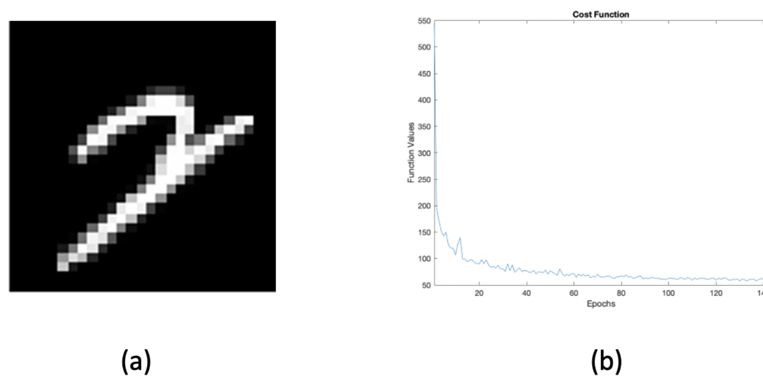


Figure 5.14: Target (a) and Cost function (b) for outputs from figure 5.15. *Source own*

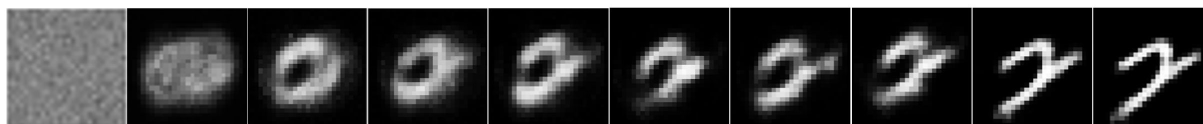


Figure 5.15: Evolution of the output during the training process. *Source own*

5.5 Final results

The final results are shown in this section. In table 5.8 the values of the parameters are exposed. AF_TF represents the optimal AE, and AF_Tmax corresponds to a much larger run performed to see the results, knowing it was not worth it to wait such an amount of time for a very similar result.

The images extracted and the cost function evolution from both simulations are shown in figures 5.16 and 5.17 respectively.

Parameters	AF_TF	AF_Tmax
Learning Rate	0,04	0,04
Lambda	0	0
TestRatio	30	30
MaxFunEvals	5000	15000
MaxEpoch	143	429
OptTolerance	10ˆ(-6)	10ˆ(-6)
timeStop	Inf([1,1])	Inf([1,1])
fvStop	10ˆ(-4)	10ˆ(-4)
nPlot	1	1
Batch size	200	200
Architecture	784 100 50 10 50 100 784	
Results		
Elapsed time	153,37	450,61
Final Cost Fnc	59,64	54,26
Nº Epochs	143	429
Nº Iterations	4969	14980
Stop Criteria	MaxEpoch	MaxEpoch

Table 5.8: Final parameters values

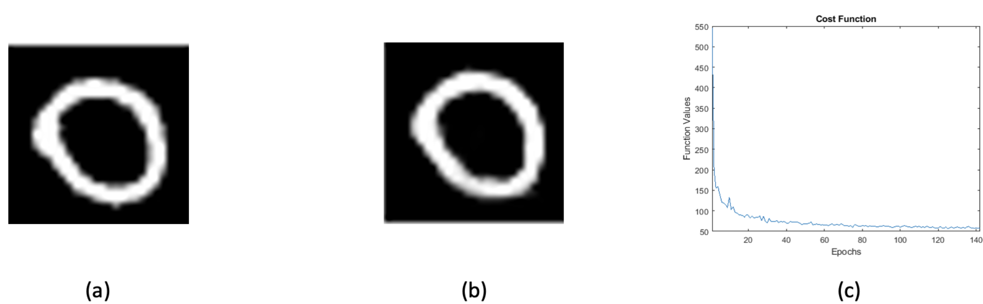


Figure 5.16: Results of test AF_TF. Target (a); Output (b); Cost Function (c). *Source own*

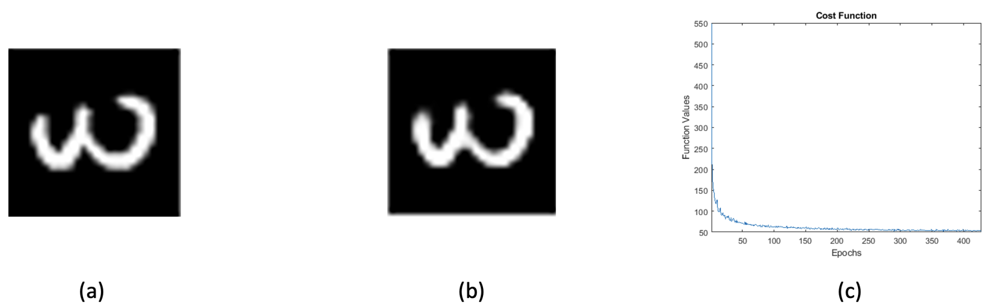


Figure 5.17: Results of test AF_Tmax. Target (a); Output (b); Cost Function (c). *Source own*

Chapter 6

Conclusions

The aim of this thesis was to conduct a comprehensive examination of the various techniques and algorithms related to VAEs in Machine Learning. At the end, the implementation of AE has been the most realistic objective and the one achieved successfully. We were able to create a model that not only successfully detected photos but also showed the potential to create new images based on the learned representations by utilising the power of AEs. This generative feature of the AE improves the model's potential use cases and creates intriguing new opportunities for creative applications.

The code can be run on every computer without having a large elapsed time, despite requiring more capacity than the legacy code as it works with much more variables. This fulfils an important objective of this project.

Rethinking the architecture as part of the refactoring process helped us increase clean code practises and optimise the performance of the code. All of the results were above 95%, demonstrating the refactored code's effectiveness in recognising images.

The final results demonstrated how well the AE captured and represented the key elements of the input photos while enabling the creation of new samples that were visually plausible. The project also demonstrated the advantages of using an AE for image generation, including the capacity to explore the latent space, regulate the generation process, and improve the interpretability of the model's internal representations.

All these objectives were successfully met. The codes used during this project have been coded from a legacy code from thesis [1], maintaining a robust and simple-to-use program. Nevertheless, during the development of this thesis, the scope was reduced as we faced our optimistic schedule predictions, leaving the implementation of VAEs out of the scope because of a lack of time.

The natural path to continuing this project is to finally implement VAEs into the code. Also, some random data generation in the latent space might be interesting to perform, as together with VAE's implementation, controlled output generation might be done. Another point to work on might be implementing the Fourier transform for intelligent data classification to reduce the features in every image and enable our code to work with larger and more complex images.

Chapter 7

Social & Environmental impact

In this chapter, the social and environmental impacts derived from the study of generative AI and AE in the field of machine learning will be analysed in detail. As technology advances and becomes increasingly integrated into our lives, it is critical to understand how our research and development can affect society and the environment.

7.1 Social Impact

AI's impact on society is widely debated. Many argue that AI improves the quality of everyday life by doing routine and even complicated tasks better than humans can, making life simpler, safer, and more efficient. Others argue that AI poses dangerous privacy risks, exacerbates racism by standardising people, and costs workers their jobs, leading to greater unemployment.

But what is certain is that we are reaching a point where we must regulate and control this sector, as one can easily deceive a person, for example, with voice, image, or video generative AI.

7.2 Environmental Impact

The specific environmental impact of this work is minimal. You could count on the consumption of electricity for the computer or WiFi, but we would be talking about a ridiculous and despicable amount.

However, in the already widespread field of AI, there is one factor to consider. Now we are working on a program where we have many degrees of freedom, which implies we need a lot of computing ability, but we have very little data trained. At the time we evolve and want to give a much more complex and comprehensive training to our program, we will need a lot more calculation capacity because we will be talking about many degrees of freedom, which implies a huge energy consumption.

Therefore, on this latter point, it would be interesting to work either with methods that reduce the needs of the program or on how to make this necessary energy no problem.

Chapter 8

Budget summary

The aim of this section is to present a summary of the associated costs of the project. The budget is divided into five different groups, which are the ones presented in Table 8.1: Documentation, Technical Development, Software, Other Direct Costs, and Indirect Costs. All of these are related to the hours of work of the engineer and the cost of the tools used.

The indirect costs are estimated at 20% of the direct costs (i.e., office rental, energy supplies, etc.). The mean hourly cost has been set to 13€/h, as it is the mean salary in engineering in Spain.

BUDGET			
Item	**Unit Cost [€/h]**	**Quantity [h]**	**Total [€]**
Documentation	**13**	**170**	**2210**
Technical Development	**13**	**400**	**5200**
Others	**-**	**30**	**715**
Software	**-**	**-**	**816**
DIRECT COST	**-**	**-**	**8941,0**
INDIRECT COST	**20%**	**8941 €**	**1788,2**
TOTAL	**-**	**-**	**10729,2**

Table 8.1: Project's budget summarised

References

1. DARDER BENNASSAR, A. *Study of efficient numerical tools for machine learning.* 2022. Available also from: https://discovery.upc.edu/permalink/34CSUC_UPC/rdgucl/alma991005064413806711.
2. *ChatGPT.* OpenAI. Available also from: https://chat.openai.com.
3. *Gantt Project.* BarD Software S.R.O., 2021.
4. ROCCA, Joseph. *Understanding Variational Autoencoders (VAEs).* Towards data science, 2019. Available also from: https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73.
5. *What is machine learning?* IBM. Available also from: https://www.ibm.com/topics/machine-learning.
6. MAINI, Vishal. *Machine Learning for Humans.* Medium, 2017. Available also from: https://medium.com/machine-learning-for-humans/why-machine-learning-matters-6164faf1df12.
7. *Dartmouth Workshop.* Wikipedia, the free enciclopedia, 2018. Available also from: https://en.wikipedia.org/wiki/Dartmouth_workshop.
8. WATTS, Rob; HAAN, Kathy. *24 Top AI Statistics & Trends in 2023.* Forbes Advisor, 2023. Available also from: https://www.forbes.com/advisor/business/ai-statistics/#:~:text=AI%5C%20is%5C%20expected%5C%20to%5C%20see,technologies%5C%20in%5C%20the%5C%20coming%5C%20years..
9. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning.* Massachusetts Institute of Technology, 2016.
10. *Generative artificial intelligence.* Wikipedia, the free encyclopedia, [n.d.]. Available also from: https://en.wikipedia.org/wiki/Generative_artificial_intelligence.
11. CHOLLET, Francis. *Building Autoencoders in Keras.* Keras, 2016. Available also from: https://blog.keras.io/building-autoencoders-in-keras.html.
12. JORDAN, Jeremy. *Variational autoencoders.* Jeremy Jordan, 2018. Available also from: https://www.jeremyjordan.me/variational-autoencoders/.
13. HUBENS, Nathan. *Deep inside: Autoencoders.* Towards data science, 2018. Available also from: https://towardsdatascience.com/deep-inside-autoencoders-7e41f319999f.
14. PRAKASH, Abhinav. *Diferent types of Autoencoders.* Open Genus, 2023. Available also from: https://iq.opengenus.org/types-of-autoencoder/.

15. SHAFKAT, Irhum. *Intuitively Understanding Variational Autoencoders*. Towards data science, 2018. Available also from: `https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf`.
16. TIU, Ekin. *Understanding Latent Space in Machine Learning*. Towards data science, 2020. Available also from: `https://towardsdatascience.com/understanding-latent-space-in-machine-learning-de5a7c687d8d`.
17. *MNIST*. Tensor Flow, [n.d.]. Available also from: `https://www.tensorflow.org/datasets/catalog/mnist`.

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	$\mathbb{M}(nD, nS_i)$
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
Iris	150	4	3

https://www.kaggle.com/datasets/uciml/iris

Name	Data points	Features	Classes
MNIST	10,000	784	10

https://ch.mathworks.com/help/deeplearning/ug/data-sets-for-deep-learning.html