Restricted Boltzmann Machine Based Autoencoders for the Classification of Faults in Rotational Mechanical Systems

Author:
Angelo DIAS

Supervisor:
Prof. Dr. José Valente DE OLIVEIRA
Prof. Dr. Chuan LI

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Departamento de Engenharia Electrónica e Informática
Faculdade de Ciências e Tecnologia

September 29, 2022

Declaration of Authorship

I, Angelo DIAS, declare that this thesis titled, "Restricted Boltzmann Machine Based Autoencoders for the Classification of Faults in Rotational Mechanical Systems" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"*The brain sure as hell doesn't work by somebody programming in rules.*"

Geoffrey Hinton

UNIVERSIDADE DO ALGARVE

Resumo

Faculdade de Ciências e Tecnologia

Departamento de Engenharia Electrónica e Informática

Master of Science

Restricted Boltzmann Machine Based Autoencoders for the Classification of Faults in Rotational Mechanical Systems

by Angelo DIAS

Contexto

O objetivo deste trabalho foi investigar a possibilidade de usar Redes Neuronais Profundas (DNN) para classificar falhas (na forma de sinais anómalos representando vibrações) em rolamentos de esferas, e comparar os resultados com os de algoritmos tradicionais.

Métodos

O conjunto de dados usado neste trabalho foi recolhido de sensores colocados nos rolamentos de esferas de um eixo conectado a um motor, sendo insuficiente para treinar eficientemente uma DNN.
Para superar esse problema, o conjunto de dados foi aumentado, primeiro separando-o em conjuntos disjuntos para minimizar a contaminação dos conjuntos de treino e teste, e em seguida um método de janela deslizante foi aplicado a esses conjuntos para gerar novos conjuntos de treino e teste.

Dois modelos de DNN baseados em Máquinas de Boltzmann Restrita (RBM) foram usados neste trabalho, com duas variantes para cada um desses modelos, usando números nítidos ou *fuzzy* para os parâmetros.
Para quantificar os resultados, foram utilizadas as métricas Exatidão e área sob a

curva (AUC). Os resultados foram analisados em conjunto com algoritmos tradicionais comuns, usados para estabelecer uma base de comparação.

Resultados

Os resultados obtidos com assim DNN foram promissores, com o melhor modelo atingindo uma exatidão média de 99.678%.
Ainda assim, os resultados obtidos com os algoritmos tradicionais foram melhores, com o melhor modelo alcançando uma exatidão média de 99.98%.
Para a outra métrica utilizada para analisar os resultados (AUC), o cenário é semelhante, com o melhor modelo de DNN a alcançar uma AUC média de 0.99995, e o melhor modelo tradicional uma pontuação perfeita de 1.

Conclusão

Os resultados obtidos revelam que, embora as DNN possam ser eficazes em problemas de classificação, elas não são a melhor escolha para problemas como este (conjuntos de dados constituídos por dados numéricos tabulares).
Isso deve-se ao desenvolvimento, afinação e volume de dados de treino necessários em comparação com os algoritmos tradicionais, que além de obter resultados ligeiramente melhores, o fizeram com tempos de treino significativamente menores e quase sem afinação.
Para outras aplicações, se houver dados de treino suficientes, as DNN frequentemente demonstram melhor desempenho, uma vez que quando os algoritmos tradicionais atingem um limite no desempenho, as DNN podem continuar a melhorar se tiverem dados de treino adicionais.

Palavras-chave: Deep Neural Networks (DNN), Restricted Boltzmann Machines (RBM), Autoencoder (AE), Classifier

UNIVERSIDADE DO ALGARVE

Abstract

Faculdade de Ciências e Tecnologia

Departamento de Engenharia Electrónica e Informática

Master of Science

Restricted Boltzmann Machine Based Autoencoders for the Classification of Faults in Rotational Mechanical Systems

by Angelo DIAS

Background

The aim of this work is to investigate the possibility of using Deep Neural Networks (DNN) to classify faults (in the form of anomalous signals representing vibrations) in ball bearings, and to compare the results with those of traditional algorithms.

Methods

The dataset used in this work was collected from sensors placed on the ball bearings of a shaft connected to a motor and was too small to effectively train a DNN. To overcome this problem, the dataset was augmented by first separating it into disjoint sets to minimize train-test contamination, then a sliding window method was applied to these sets to generate new training and test sets.

Two Restricted Boltzmann Machine (RBM) based DNN models were used in this work, and for each of these models two variants, using either crisp or fuzzy numbers for the parameters. To quantify the results, the metrics Accuracy and area under the curve (AUC) were used. The results were analyzed together with those of common traditional algorithms, used to establish a base for comparison.

Results

The results obtained with the DNN were promising, the best model achieving a mean accuracy of 99.678%. Yet, the results obtained with the traditional algorithms were even better, with the best model achieving a mean accuracy of 99.98%. For the other metric used to analyze the results (AUC), the scenario is similar, with the best DNN model achieving a mean AUC of 0.99995, and the best traditional model a perfect score of 1.

Conclusion

The results obtained reveal that while DNN can be effective at classification problems, they are not the best choice for problems such as this (datasets consisting of tabular numeric data). This is due to the development, tuning, and volume of training data required compared to the traditional algorithms, which not only obtained slightly better results, but did so with significantly lower training times, and almost no tuning. For other applications, if there is enough training data, DNN routinely show better performance, since when traditional algorithms hit a plateau in performance, DNN can continue to improve if they are provided with additional training data.

Keywords: Deep Neural Networks (DNN), Restricted Boltzmann Machines (RBM), Autoencoder (AE), Classifier

Acknowledgements

I would like to thank my supervisor, Prof. Dr. José Valente de Oliveira, for his infinite patience, guidance and support throughout this work. I am also grateful to the GIDTEC (Research and Development of Industrial Technologies Group) at the Universidad Politeécnica Salesiana, Ecuador for providing the data set used in this work.

I would also like to thank my family for their support and encouragement.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ML	**M**achine **L**earning
DL	**D**eep **L**earning
DNN	**D**eep **N**eural **N**etwork
DAE	**D**eep **A**uto **E**ncoder
DBN	**D**eep **B**elief **N**etwork
RBM	**R**estricted **B**oltzmann **M**achine
CRBM	**C**onventional **R**estricted **B**oltzmann **M**achine
FRBM	**F**uzzy **R**estricted **B**oltzmann **M**achine
STFN	**S**ymmetric **T**riangular **F**uzzy **N**umber
ATFN	**A**symmetric **T**riangular **F**uzzy **N**umber
GAN	**G**enerative **A**dversarial **N**etwork
VAE	**V**ariational **A**uto **E**ncoder
CD	**C**onstrastive **D**ivergence
RFC	**R**andom **F**orest **C**lassifier

To my little boy.

For reminding me of the joy brought on by childlike curiosity.

Chapter 1

Introduction

1.1 Context

1.1.1 Ball Bearings

Ball bearings are rolling elements used to reduce rotational friction and support radial and axial loads, and are used in the makority of devices that have moving parts, from household applicances to industrial equipment, transportation and many more.
They are typically composed of two rings separated by steel balls (the most widely used type, the single row deep groove ball bearing).

The failure of these elements, although rare, can have great impact, causing production delays, equipment damage or even injury.
The basic life of a ball bearing is defined by the number of revolutions it can withstand before failure, and is usually expressed in hours of operation.
This value (L_{10}) can be calculated using the following formula [2]:

$$L_{10} = (C/P)^p, \tag{1.1}$$

where:

- L_{10} is the 'basic life' (usually quoted in millions of revolutions) which 90% of bearings are expected to exceed.
- C is the dynamic load rating of the bearing, provided by the manufacturer.
- P is the applied load.
- p is a constant which has the value 3 for ball bearings.

The median life, or Mean Time Between Failure (MTBF), is typically about five times the basic life, but sometimes even minor problems or variations in the manufacturing process can lead to a significant reduction in the life of the bearing or even

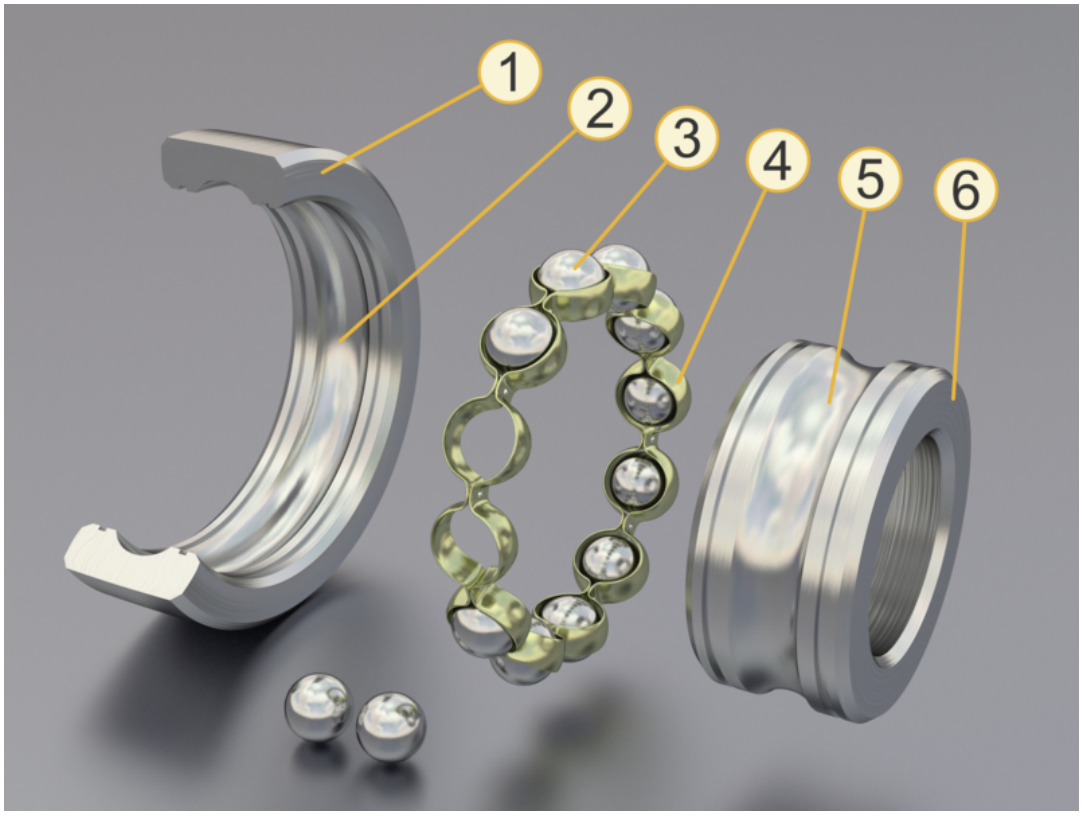


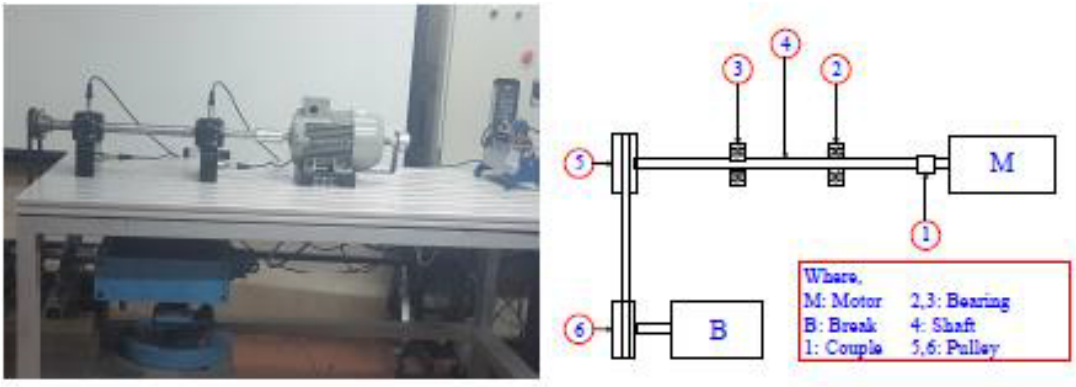
FIGURE 1.1: Structure of a single row deep groove ball bearing: 1. outer ring 2. outer race 3. ball 4. cage 5. inner race 6. inner ring, from [1]

fail suddenly.
There are many different modes of failure, such as brinelling, fatigue failure, lubricant failure, corrosion.
Most of these faults can be detected with vibration analysis [3] [4].

1.1.2 Dataset

This work will use a dataset of real-world data, containing signals obtained from an experimental setup that simulates real industrial equipment (see Figure 1.2).
The setup has a shaft supported by two ball bearings connected to a motor on one side, and on the other side there are pulleys connecting the shaft to a brake, to simulate a load.
To capture the vibrational signals accelerometers are mounted on each of the ball bearings.
For the purpose of this work only signals captured from one of these accelerometers were used.

Seven classes of signals were captured, representing different combinations of faulty



(a) the real world setup (b) Schematic setup

FIGURE 1.2: Experimental Setup

and healthy ball bearings.

The dataset is made up of 315 samples, with each sample resulting from a 20 seconds long signal captured at a sampling rate of 50 kHz, resulting in vectors with one million points.

This is an exceedingly large size to be used as the input for neural networks, and so instead features are generated from each sample, resulting in a new dataset where each vector has 787 points.

These features are from the time, frequency and time-frequency domains, and are generated using a script provided by Diego Cabrera of GIDTEC (Research and Development of Industrial Technologies Group at the Universidad Politécnica Salesiana, Ecuador).

An obvious issue to be addressed is that DNN typically require the number of training samples to be, at a minimum tens of thousands, and often in the hundreds of thousands and even millions.

This means augmenting the dataset will necessarily be the first task to be performed. This will be achieved by way of a sliding window technique that randomly extracts sub-signals from each of the original samples.

To minimize cross-contamination between the train and test sets, the original set will be first divided into 2 disjoint subsets, to each of which the the augmentation method will be applied separately.

1.2 Goals

With the explosion in popularity of Artificial Intelligence (AI), and in particular of Machine Learning (ML), new techniques and applications are discovered every day.

A subset of algorithms named Deep Neural Networks (DNN) began to gain prominence around 2012, giving rise to an area of ML called Deep Learning (DL).
Although DNN had existed for decades, the lack of computational capacity and training data volume limited their effectiveness.
These algorithms in many cases display better results than those of traditional algorithms, however, they are significantly more demanding to train in terms of time and tuning necessary owing to the large number of parameters they possess.
Currently there are many successful practical use cases of DL, like image recognition, voice recognition, automatic translation, autonomous vehicles and many more.
One particularly important use case of ML is fault detection and classification.
This work will explore the application of Deep Autoencoders (DAE), a type of (DNN) to the extraction of characteristics of vibrational signals, with the goal of performing fault detection and classification on the condition of ball bearings. Two models will be used in this work:

1. an RBM based full AE model trained in an unsupervised manner which is used to generate a reduced dimensionality dataset. This dataset is then used to train and test a Random Forest Classifier (RFC).

2. a Classifier model which consists of just the encoder half of the AE feeding into a softmax layer, the entire ensemble trained in a supervised manner.

Each of these models will have two variants, one using traditional, or crisp, RBM and the other using fuzzy RBM for each individual layer. Each layer will be pre-trained individually and then the complete network will be fine-tuned using gradient descent.
Two metrics will be used to analyze the results (Accuracy, and area under the curve (AUC)), which will be compared to those obtained by traditional algorithms.

1.3 Contributions

In this study, a novel method of implementing DAE is explored.
This method consists in the use of Fuzzy Restricted Boltzmann Machines (FRBM) whose superior performance in other problem domains such as handwritten digit recognition has been asserted [5] [6].
To make this possible, a method of augmenting the dataset must be developed, as to meet the training data volume requirements typical to DNN.
Then a study of fuzzy and conventional DAE, in terms of implementation difficulty and computational costs will be performed.
Once the implementation of the DAE is complete, a comparison with of results obtained with both types of DAE, for multiple configurations, as well as traditional ML

algorithms will be carried out.

1.4 Thesis Organization

This thesis is divided into 7 chapters.

The current chapter is the Introduction, which provides a summary of the study. In it are presented the problem, the objective and significance, and the organization of the thesis.

Chapter 2 provides a description of the problem and a theoretical framework necessary for the comprehension of the methods that will be used in the study.

Chapter 3 contains the description of the experimental setup used to capture the signal data, and the methods used to augment and prepare this data for use with the Autoencoders which are the subject of this study.

Chapters 4 and 5 contain the descriptions of the experiments performed with the Autoencoder and Classification models, in crisp and fuzzy versions.

Chapter 6 contains the presentation and analysis of the results of the experiments that were carried out.

Finally, chapter 7 summarizes the work, presenting the conclusions reached and recommendations for further avenues of research with respect to the problem.

Chapter 2

Background

This chapter provides a description of the problem and a theoretical framework necessary for the comprehension of the methods that will be used in the study.

2.1 Deep Autoencoders

This work will be conducted using deep neural networks (DNN) called Deep Autoencoders.
Simply stated, autoencoders are neural network that take an input and attempt to reproduce it as faithfully as possible at the output, after running the data through one or more intermediate layers of different (typically narrower) sizes.
If successful, then a compressed representation of the input has been learned in the middle layer, which can then itself be used as the input for another network or algorithm.
This process is called feature extraction, and is an essential step in many machine learning tasks, whereby the initial data representation is reduced in volume to a smaller set of informative and non-redundant features.

These networks are composed of two (symmetrical) halves called Deep Belief Networks (DBN) [7], trained by unsupervised learning, or by one DBN that then feeds in to a Softmax layer, forming a classifier.
The DBN themselves are composed of several individual layers of a type called Restricted Boltzmann Machine (RBM).

2.2 Restricted Boltzmann Machines

Boltzmann Machines (named after the Boltzmann distribution from statistical mechanics) are generative stochastic recurrent networks of symmetrically connected, neuron-like units.

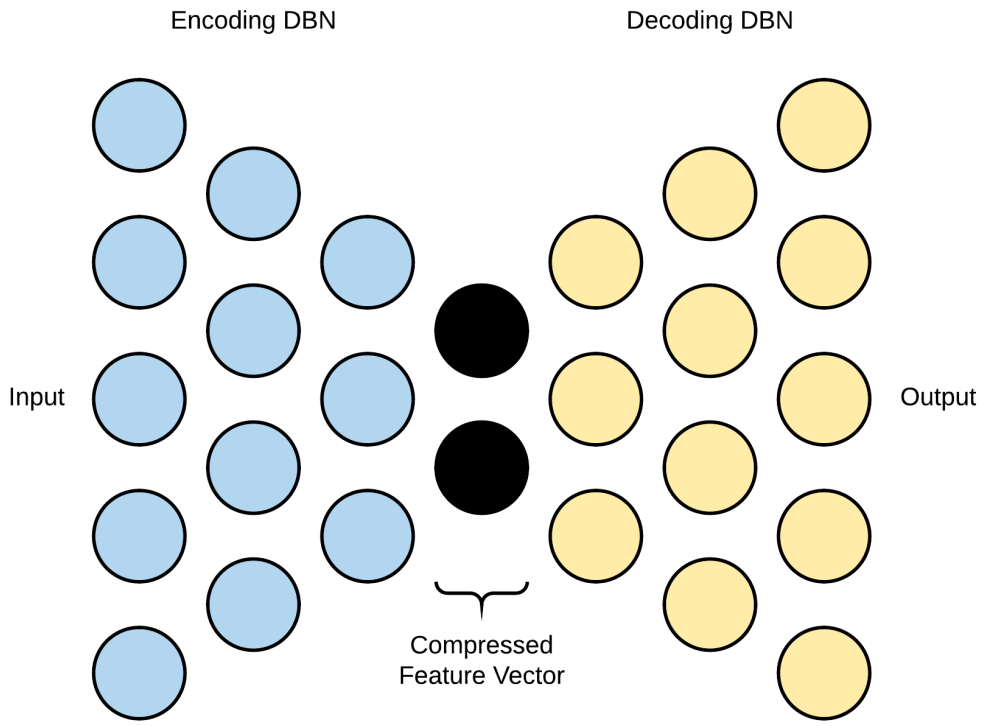


FIGURE 2.1: Structure of a Deep Autoencoder, where the left side (Encoder, represented by blue circles) maps the input to a compressed representation or code, and the right side (Decoder, represented by yellow circles) uses the code to reconstruct the input.

They were first proposed in 1983 by Hinton and Sejnowski [8], with a simple but inefficient training algorithm.

Then in 1986 Paul Smolensky proposed (under the name "Harmonium") a variation named the Restricted Boltzmann Machine [9], that includes the restriction of not having intra-layer connections.

These were popularized in the mid-2000 when efficient learning algorithms were proposed for them [10] [11], and were in part responsible for reviving interest in the field of deep learning after being largely forsaken by researchers in the late 1990s due to generalized belief that training deep networks was practically unfeasible [12].

Presently, for tasks that employ generative models (mostly image related), other types of networks such as Variational Autoencoders (VAE) and Generative Adversarial Networks (GAN) are preferred.

For classification tasks, linear models are used where possible.

Restricted Boltzmann Machines are defined as bipartite symmetrical probabilistic graphical models. Bipartite because it's composed of two parts, or layers.

Symmetrical because each node in the visible layers connects to each node in the

hidden layer. Probabilistic graphical model because it is a graph expressing the conditional dependence between random variables.
These networks are trained using unsupervised training and are adept at finding underlying patterns in the structure of the input data.
This is useful for extracting significant features and thereby obtaining lower dimensional representations of the original data, reducing redundancy and helping avoid overfitting when these representations are used as the input for other machine learning algorithms, in a process called Pretraining.

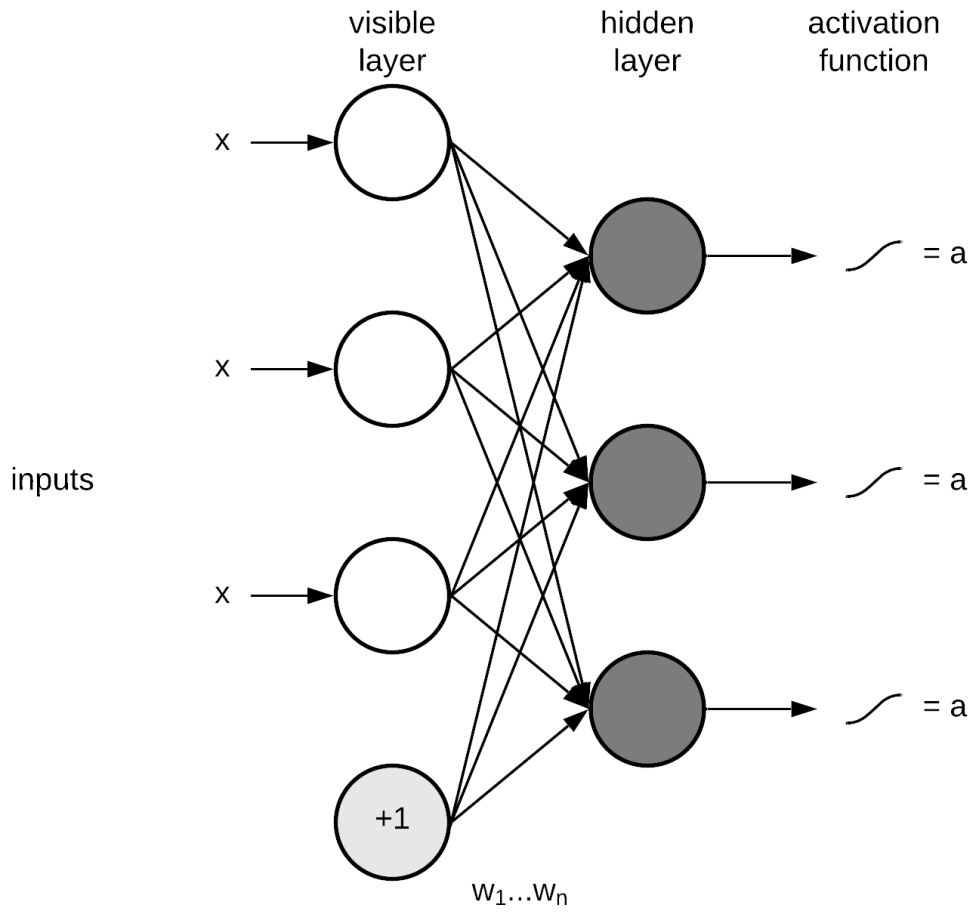


FIGURE 2.2: Restricted Boltzmann Machine structure.

RBMs are called an energy based model, as their intuition comes from the Physics concept of free energy.
This energy is equivalent to a measure of compatibility to each configuration of the variables, and learning consists in finding an energy function such as observed configurations of the variables have lower energy than unobserved ones.

This energy is calculated by the Hopfield energy function:

$$E(v,h) = -\sum_j b_j v_j - \sum_i c_i h_i - \sum_{i,j} v_j h_i W_{ij}, \tag{2.1}$$

or in matrix notation:

$$E(v,h) = -(b'v + c'h + v'hW), \tag{2.2}$$

where v denotes the visible units, h the hidden units, b and c are bias terms, and W the weights representing the strength of the connections between units.

So at any given moment, the RBM is in a state defined by the values in its visible and hidden units, and with equation (2.1) it's possible to calculate the probability that a certain state is observed, using the following joint distribution:

$$p(v,h) = \frac{1}{Z} e^{-E(v,h)}, \tag{2.3}$$

and, usually more interesting, the marginal distribution of v:

$$p(v) = \sum_h p(v,h) = \frac{1}{Z} \sum_h e^{-E(v,h)} = \frac{1}{Z} e^{-\mathcal{F}(v)}, \tag{2.4}$$

with $\mathcal{F}$ being the free energy formula:

$$\mathcal{F}(v) = -log \sum_h e^{-E(v,h)} = -b'v - \sum_i log(\sum_{h_i} e^{h_i(c_i + W_i v)}), \tag{2.5}$$

and Z is a normalizing factor used to ensure that the probability distribution sums to 1, called a partition function and defined as the sum of $e^{-E(v,h)}$ over all possible configurations:

$$Z = \sum_{v,h} e^{-E(v,h)}. \tag{2.6}$$

Like other machine learning algorithms, RBMs use gradient descent as the optimization algorithm to minimize its loss function, which is the negative log-likelihood.

Algorithm 1: Sketch of the Stochastic Gradient Descent Algorithm

foreach *sample in the dataset* **do**
- Get the predicted value
- calculate the loss using the loss function
- calculate partial derivatives of the loss function, which produce gradients
- use the gradients to update the values of weights and biases

end

The negative log-likelihood for a model of the form (2.4), with a set of parameters θ can be written as:

$$\log p(v|\theta) = \log \frac{1}{Z} \sum_h e^{-E(v,h)} = log \sum_h e^- E(v,h) - log \sum_{v,h} e^{-E(v,h)}, \tag{2.7}$$

and its gradient with respect to θ:

$$\frac{\partial \log p(v|\theta)}{\partial \theta} = -\sum_h p(h|v) \frac{\partial E(v,h)}{\partial \theta} + \sum_{v,h} p(v,h) \frac{\partial E(v,h)}{\partial \theta}. \tag{2.8}$$

The above gradient contains two terms, which are referred to as the positive and negative phase.
The titles positive and negative do not refer to the sign of each term in the equation, but rather reflect their effect on the probability density defined by the model.

The first term increases the probability of training data (by reducing the corresponding free energy), while the second term decreases the probability of samples generated by the model.
Both terms can be well factorized, but while the first can be computed exactly, the second cannot except for very small models.

Using factorization the gradient of the log-likelihood for a training sample with respect to the weight w_{ij} is:

$$\begin{aligned} \frac{\partial \log p(v|\theta)}{\partial w_{ij}} &= -\sum_h p(h|v) \frac{\partial E(v,h)}{\partial w_{ij}} + \sum_{v,h} p(v,h) \frac{\partial E(v,h)}{\partial w_{ij}} \\ &= \sum_h p(h|v) h_i v_j - \sum_v p(v) \sum_h p(h|v) h_i vj \\ &= p(h_i = 1|v) v_j - \sum_v p(v) p(h_i = 1|v) v_j. \end{aligned} \tag{2.9}$$

Similarly are calculated the gradients with respect to the bias parameter of the visible units, b:

$$\frac{\partial \log p(v|\theta)}{\partial b_j} = v_j - \sum_v p(v) v_j, \tag{2.10}$$

and with respect to the bias parameter of the hidden units, c:

$$\frac{\partial \log p(v|\theta)}{\partial c_i} = p(h_i = 1|v) - \sum_v p(v)p(h_i = 1|v). \tag{2.11}$$

Taking the average over an entire training set results in a quite simple expression for the gradient of the log-likelihood (2.9):

$$\frac{\partial \log p(v|\theta)}{\partial \theta} = \langle v_j h_i \rangle_{data} - \langle v_j h_i \rangle_{model}, \tag{2.12}$$

with the angle brackets indicating expectations related to the distributions specified by the subscripts.
This yields the following learning rule for the weights w_{ij}:

$$\Delta w_{ij} = \epsilon(\langle v_j h_i \rangle_{data} - \langle v_j h_i \rangle_{model}), \tag{2.13}$$

for the visible units bias b:

$$\Delta b_j = \epsilon(\langle v_j \rangle_{data} - \langle v_j \rangle_{model}), \tag{2.14}$$

and for the hidden units bias c:

$$\Delta c_i = \epsilon(\langle h_i \rangle_{data} - \langle h_i \rangle_{model}), \tag{2.15}$$

with ϵ being the learning rate.

Obtaining unbiased samples of the first (data) term is simple.
As there no connections betweens units in each layer, the states of the hidden units can be calculated (for a given input v) using:

$$p(h_j = 1|v) = sigm(c_j + W_j v_i) = \sigma(c_j + \sum_i v_i W_{ij}), \tag{2.16}$$

and conversely the states of the visible units (given a hidden vector h):

$$p(v_i = 1|h) = sigm(b_i + W_i h_j) = \sigma(b_i + \sum_j h_j W_{ij}), \tag{2.17}$$

where $sigm$ is the sigmoid function:

$$sigm(x) = \frac{1}{1 + e^{-x}}. \tag{2.18}$$

Obtaining samples of the second (model) term is much harder, and to obtain an exact calculation would require to perform many iterations of alternating Gibbs sampling. This makes training impossible, and Hinton's great contribution was the proposal of a training algorithm named Contrastive Divergence (CD), or more commonly, k-step Contrastive Divergence (CD-k).
The idea behind this is straightforward.

To obtain an unbiased sample of the second term of the log-likelihood gradient, Gibbs sampling has to be run for a long time, as the bias tends to disappear as $k \to \infty$.
Hinton realized that a good enough approximation can be obtained by running the Gibbs chain only for k steps (usually $k = 1$).

The Gibbs chain is initialized with a training sample $v^{(0)}$ and a sample $v^{(k)}$ is obtained after k steps.
Each step consists of sampling $h^{(n)}$ from $p(h|v^{(n)})$ and then sampling $v^{(n+1)}$ from $p(v|h^{(n)})$ subsequently.

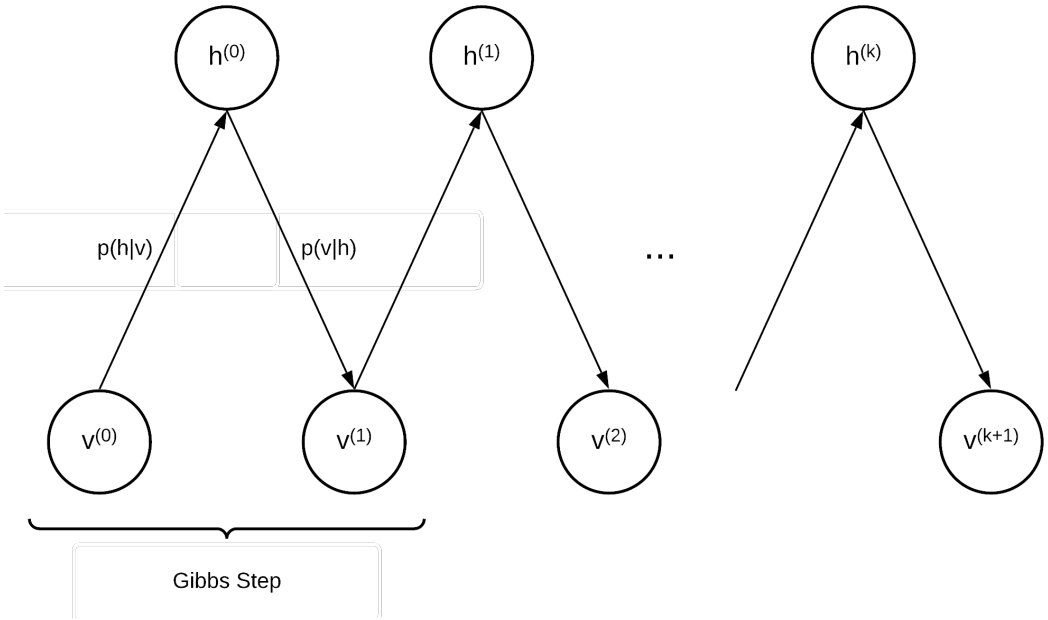


FIGURE 2.3: Gibbs Sampling. $v^{(n)}$ and $h^{(n)}$ refer respectively to the set of all visible and all hidden nodes at the n-th step of the Markov chain.

With these equations it is possible to calculate the weight updates:

$$W_{new} = W_{old} + \Delta W, \tag{2.19}$$

$$b_{new} = b_{old} + \Delta b, \tag{2.20}$$

$$c_{new} = c_{old} + \Delta c. \tag{2.21}$$

where ΔW, Δb and Δc are obtained using Algorithm 2:

Algorithm 2: A batch k-step contrastive divergence algorithm, from [10], [13]

Input: A Restricted Boltzmann machine with V_1,...,V_m visible and H_1,...,H_n hidden units; a finite training set S

Output: Weight Δw_{ij}, and bias Δb_j, Δc_i updates ($i = 1,...,n$; $j = 1,...,m$)

$\Delta w_{ij} \longleftarrow 0$, $\Delta b_j \longleftarrow 0$, $\Delta c_i \longleftarrow 0$ ($i = 1,...,n$; $j = 1,...,m$);

foreach $v \in S$ **do**
  $v^{(0)} \longleftarrow v$;
  for $t \leftarrow 0$ **to** $k-1$ **do**
    for $i \longleftarrow 0$ *to* n **do**
      sample $h_i^{(t)} \sim p(h_i|v^{(t)})$
    end
    for $j \longleftarrow 0$ *to* m **do**
      sample $v_j^{(t+1)} \sim p(v_j|h^{(t)})$
    end
  end
  for $i \longleftarrow 1$ *to* n **do**
    for $j \longleftarrow 1$ *to* m **do**
      $\Delta W_{ij} \longleftarrow \Delta W_{ij} + p(H_i = 1|v_{v^{(0)}})v_j^{(0)} - p(H_i = 1|v^{(k)})v_j^{(k)}$;
      $\Delta b_j \longleftarrow \Delta b_j + v_j^{(0)} - v_j^{(k)}$;
      $\Delta c_i \longleftarrow \Delta c_i + p(H_i = 1|v^{(0)}) - p(H_i = 1|v^{(k)})$;
    end
  end
end

2.3 Fuzzy Restricted Boltzmann Machines

Two papers have been published by a group of researchers [6] [5], where modifications to the RBM algorithm are proposed.
In these papers, the results presented suggest that these modified RBM have improved performance compared to the original algorithm.

The core idea of this modification consists in representing the network parameters using fuzzy numbers.
This way, instead of 3 sets of parameters W, b and c (weights, visible units bias,

hidden units bias), there's instead W^L, W^R, b^L, b^R, c^L and c^R in the simpler version, using symmetric triangular fuzzy numbers (STFN).
Here the superscripts L and R (left and right) designate the left and right bounds of the fuzzy numbers.
There is also a more general version, using asymmetric triangular fuzzy numbers (ATFN), with the added parameters W^M, b^M, c^M where the superscript M designates the position of the most probable value.

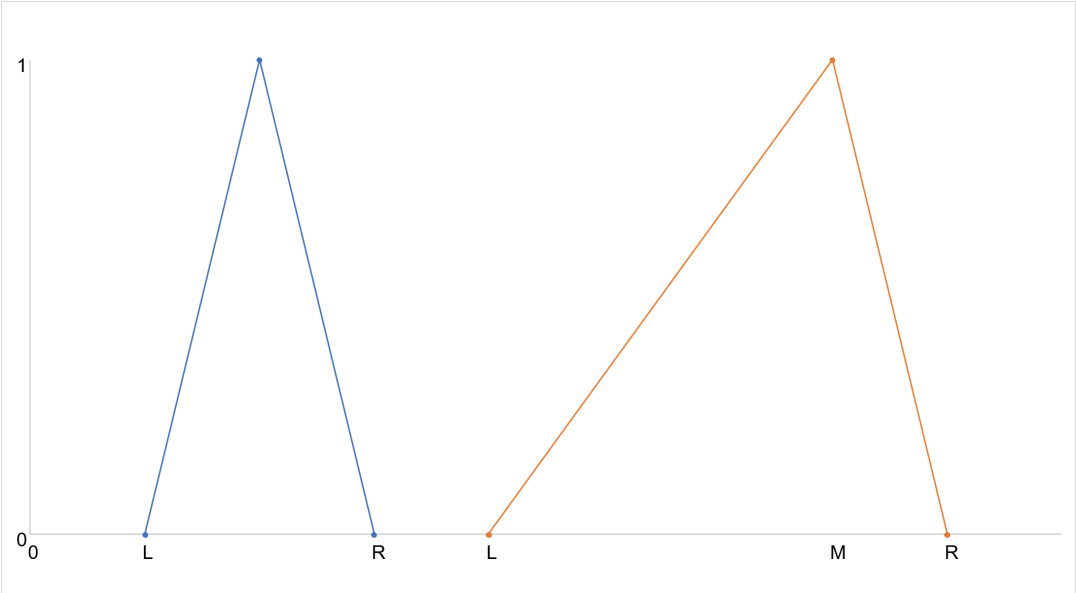


FIGURE 2.4: Triangular Fuzzy Numbers. A symmetric triangular fuzzy number is represented on the left and an asymmetric triangular fuzzy number is represented on the right.

This necessitates alterations to the training algorithm, and implementing these alterations according to [6] and [5] constituted one of the greatest difficulties in realizing this work, as no code samples were supplied accompanying the papers.
The method for the weights update computation is given by Algorithm 3:

Considering the difficulties encountered by many practitioners while training more recent network types, such as GAN and VAE, in terms of training time and training data volume, a decision was made to use conventional RBM and to try and implement the fuzzy RBM proposed by the group of Chen et al.
It should be noted that the use of this kind of networks for the classification of tabular data is a recent phenomenon, as they were typically mostly used for image problems and recommender systems.

Algorithm 3: A stochastic learning algorithm for symmetric membership function based Fuzzy Restricted Boltzmann machines, from [5]

Input: A fuzzy Restricted Boltzmann machine with V_1,...,V_m visible and H_1,...,H_n hidden units; Left bounds W^L and right bounds W^R of connection weights; Left bounds b^L and right bounds b^R of bias terms associated with visible units; Left bounds c^L and right bounds c^R of bias terms associated with hidden units; a training sample $x^{(0)}$.

Output: Parameters W^L, W^R, b^L, b^R, c^L and c^R.

for $i \longleftarrow 1$ *to* m **do**

  sample $h_i^{L(0)} \in \{0,1\} \sim p^L(h_i^{L(0)} = 1|x^{L(0)})$;

  sample $h_i^{R(0)} \in \{0,1\} \sim p^R(h_i^{R(0)} = 1|x^{R(0)})$;

end

for $j \leftarrow 1$ **to** n **do**

  sample $x_j^{L(1)} \in 0,1 \sim p^L(x_j^{L(1)} = 1|h^{L(0)})$;

  sample $x_j^{R(1)} \in 0,1 \sim p^R(x_j^{R(1)} = 1|h^{R(0)})$;

end

for $i \longleftarrow 1$ *to* m **do**

  sample $h_i^{L(1)} \in 0,1 \sim p^L(h_i^{L(1)} = 1|x^{L(1)})$;

  sample $h_i^{R(1)} \in 0,1 \sim p^R(h_i^{R(1)} = 1|x^{R(1)})$;

end

$h^{(0)} \longleftarrow \frac{h^{L(0)}+h^{R(0)}}{2}$;

$x^{(1)} \longleftarrow \frac{x^{L(1)}+x^{R(1)}}{2}$;

$h^{(1)} \longleftarrow \frac{h^{L(1)}+h^{R(1)}}{2}$;

$\Delta W \longleftarrow \frac{x^{(0)}h^{(0)}-x^{(1)}h^{(1)}}{2}$, $\Delta b \longleftarrow \frac{x^{(0)}-x^{(1)}}{2}$, $\Delta c \longleftarrow \frac{h^{(0)}-h^{(1)}}{2}$;

$W^L \longleftarrow W^L + \epsilon\Delta W$, $b^L \longleftarrow b^L + \epsilon\Delta b$, $c^L \longleftarrow c^L + \epsilon\Delta c$;

$W^R \longleftarrow W^R + \epsilon\Delta W$, $b^R \longleftarrow b^R + \epsilon\Delta b$, $c^R \longleftarrow c^R + \epsilon\Delta c$;

2.4 Gradient Descent

Gradient Descent is an algorithm used for the minimization of functions, usually of cost functions ($J(\Theta)$, where Θ are the networks parameters) during the training of feed forward neural networks.

The basic outline of the algorithm is as follows:

1. Initialize the parameters Θ randomly.
2. Compute the gradients of the cost function with respect to the parameters.
3. Update the weights according to $\Theta_j := \Theta_j - \alpha\frac{\partial}{\partial\Theta_j}J(\Theta)$.
4. Repeat until some termination criteria is reached.

In step 3, α is the learning rate, which can be interpreted as the step size in the

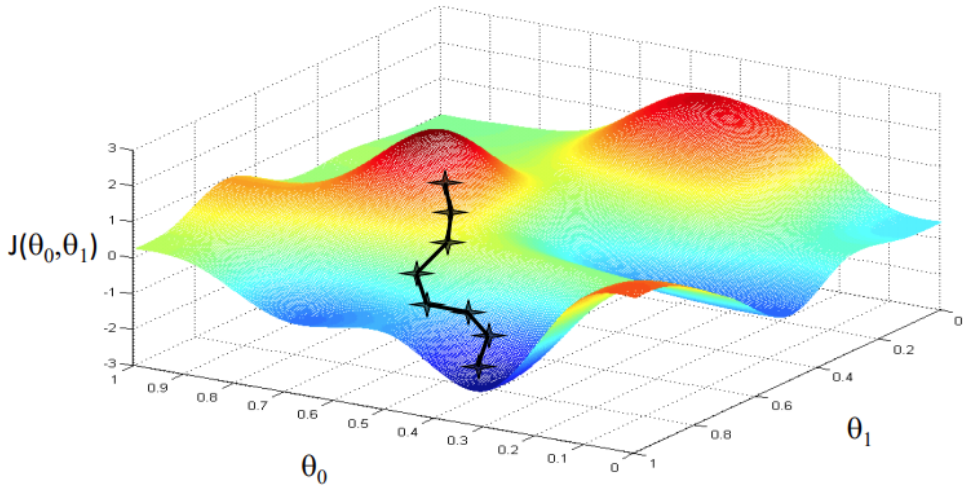


FIGURE 2.5: Gradient Descent algorithm visualization, from [14].

progression towards the minimum of the cost function.
This parameter should be chosen carefully, as if it is too small the function may take a long time to converge, and if it is too may it may even diverge.
The term $\frac{\partial}{\partial \Theta_j} J(\Theta)$ is the gradient of the cost function, with respect to the parameters Θ, and its computation is outlined in the next section.

2.5 Backpropagation

Backpropagation [15] (short for "backwards propagation (of errors)"), a generalization of the delta rule of single layer networks, applies the chain rule of derivatives to compute the gradient of the cost function, which can then be used by Gradient Descent or another optimization algorithm.

The algorithm is outlined in Algorithm 4.

In this context $\Delta_{ij}^{(l)}$ is an accumulator for the weight updates that is initialized as a zero matrix.
Then forward propagation is performed to calculate the activations $a^{(l)}$ for each layer, with $a^{(1)}$ being just the training samples.
The activations $a^{(l)}$ are calculated by applying an activation function (in this case, the sigmoid function) to the output of the previous layer:

$$a^{(l)} = sig(z^{(l)}), \tag{2.22}$$

with z defined as:

Algorithm 4: Backpropagation Algorithm, from [14]

Input: A Feedforward Neural Network with L layers; Training set $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$

Output: The partial derivatives of the cost function

Set $\Delta_{ij}^{(l)} = 0$ for all l, i, j;

for $i \longleftarrow 1$ *to* m **do**

  Set $a^{(1)} = x^{(1)}$;

  Perform forward propagation to compute $a^{(l)}$ for l = 2,3,...,L;

  Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$;

  Compute $\delta^{(L-1)}, \delta^{(L-2)}, ..., \delta^{(2)}$;

  $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$;

end

$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)}$ if $j \neq 0$;

$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)}$ if $j = 0$;

$$z^{(l)} = \Theta^{(l-1)} a^{(l-1)}. \tag{2.23}$$

The delta values δ for the last layer can be calculated as the difference between its activations and the true values y.

The delta values for the following layers are calculated from right to left according to:

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) .* a^{(l)} .* (1 - a^{(l)}). \tag{2.24}$$

Once the algorithm is finished running through the entire training set, the accumulator $\Delta_{ij}^{(l)}$ is used to calculate the terms $D_{ij}^{(l)}$, one for $j \neq 0$ which refers to the weights (which has an extra regularization term) and one for $j = 0$ which refers to the bias terms.

The terms D can be proven to be exactly the partial derivative of the cost function with respect to the network parameters:

$$D_{ij}^{(l)} = \frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta). \tag{2.25}$$

2.6 Classification Algorithms

After the data is processed by the autoencoders, a classification algorithm must be used as the output layer in order to obtain class predictions.
Two different general configurations will be used, each with different layer architectures.
Each of these configurations will use a different classification algorithm, one being the Random Forest, and the other the Softmax classifier.

2.6.1 Random Forest Classifier

The Random Forest Classifier is a supervised algorithm, that can be used for both classification and regression tasks.
It consists of a number of individual decision trees working as an ensemble, outputting either the mode of the classes (in the case of classification), or the mean prediction (in the case of regression) of all the trees.

Decision trees are themselves a very commonly used algorithm in Machine Learning, and can be used for both classification and regression, but have some important drawbacks like being very prone to overfitting and having high variance.

Random Forests were first proposed by Tin Kam Ho [16] and improved upon by Leo Breiman [17].
This was achieved by combining Ho's random feature selection with his own "bagging" (or $bootstrapaggregating$) concept, which consists of training each tree on random samples of the training set.

By averaging the output of many decision trees, performance is greatly improved by virtually eliminating overfitting and greatly reducing variance.
This comes at the expense of a small increase in bias, but with the benefits of also being robust to missing values and requiring no real parameter tuning to be performed, besides increasing the number of trees which in general leads to better accuracy, at the expense of computational complexity.

2.6.2 Softmax

The $softmax$ function, used for multiclass classification, is a smooth approximation to the $argmax$ function.
It takes as an input a vector of real numbers, and outputs a normalized probability distribution, with each probability being proportional to the exponent of the input values (2.26).

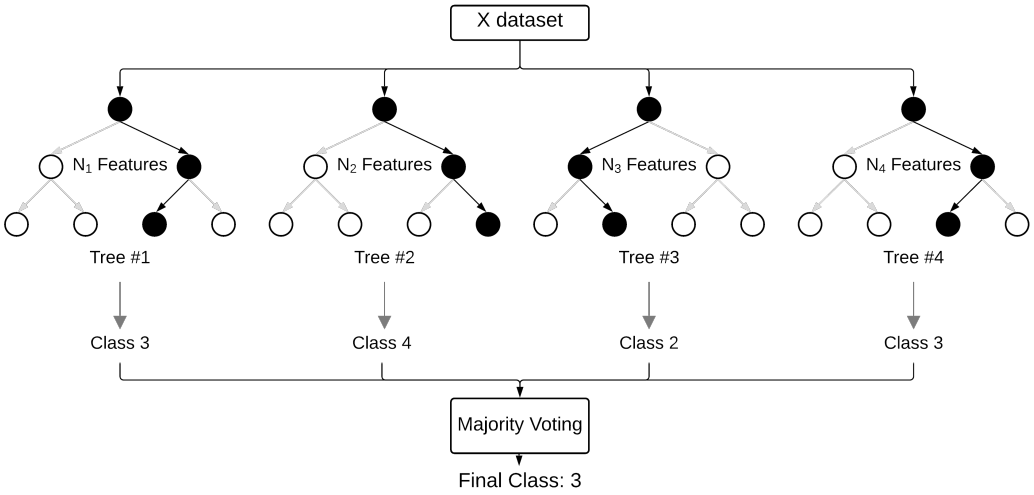


FIGURE 2.6: Random Forest Classifier - The circles represent nodes where decisions are made, using a feature. Each tree uses a sub-group of features and the final class is selected by majority vote.

$$S(y_i) = \frac{e^{y^i}}{\sum_j e^{y_j}}, \tag{2.26}$$

with y^i in the case of neural networks being equal to:

$$y_i = sigm(x^T w_i). \tag{2.27}$$

Chapter 3

Experimental Apparatus

This chapter contains the description of the experimental setup used to capture the signal data, and the methods used to augment and prepare this data for use with the Autoencoders which are the subject of this study.

3.1 Experimental Setup

The experimental setup was developed at GIDTEC, to be used in studies of bearing fault diagnosis.

The details of the experimental setup (fig. 1.2) are the following:

1. Two SKF 1207 EKTN9/C3 self-aligning bearings are mounted in SKF SE 507-606 housings and installed in a 30mm diameter shaft driven by a Siemens 1LA7 090-4YA60 2HP electric motor, controlled by a Danfoss VLT 1.5 kw driver inverter,
2. One accelerometer PCB ICP 353c03 is installed in each bearing housing for measuring the vibration signal that is collected by the NI Cdaq-9234 data acquisition card,
3. Flywheels can be mounted on the shaft for load purposes,

Each experiment is categorized by three characteristics:

1. shaft speed
2. load
3. bearing status

Three shaft speeds were tested (8, 10 and 15Hz), and also three different types of loads (0, 1 and 2 flywheels).
The bearing status characteristic constitutes the classes in the resulting dataset which

this work aims to classify.
There are 7 classes, with 6 being differents kinds of faults and one representing a healthy mechanism (for control purposes).
These different classes are listed in Table 3.1.

TABLE 3.1: Failure classes

Faulty mode	**Bearing 1**	**Bearing 2**
P1	healthy	healthy
P2	inner race fault	healthy
P3	outer race fault	healthy
P4	rolling element fault	healthy
P5	inner race fault	outer race fault
P6	inner race fault	rolling element fault
P7	outer race fault	rolling element fault

Since the results have low variability, each experiment was repeated only 5 times. This way, the aforementioned 315 samples were obtained (3x3x7x5 = 315).
A sampling frequency of 50kHz was used, well above the requirements put forth by the Nyquist-Shannon theorem (which states that the sampling frequency should be at least twice the signal's highest frequency (20 kHz)), and the duration of each sample (measurement time) was 20 seconds.

The resulting vectors are one million elements long, which if used directly as the input to a DNN would make training impractical.

To get around this issue, the samples are processed by a script that generates features in the time (shown in Table 3.2), frequency and time-frequency domains from each sample.

The time domain signal is transformed into a frequency domain one using Fast Fourier Transform (FFT).
The frequency domain signals are divided in 80 312.5Hz-bands and features comprising mean, root mean squares, standard deviation, and kurtosis are computed for both linear and logarithmic (dB) scales.
Furthermore, 15 octaves are evaluated and for each one the mean, standard deviation, and kurtosis are computed, also for both linear and logarithmic scales.
Finally, 5 wavelets packet transforms are computed: Biorthogonal (bior6.8), Coiflets (coif4), Daubechies (db7), Symlets (sym3), and Reverse Biorthogonal (rbio6.8).
These allow for the assessment of time-frequency information such as high frequency transients, that might be missed by FFT.
In total, this amounts to 787 features.

TABLE 3.2: Used time-domain features for a signal x with mean μ and duration N.

Feature	**Formula**
root mean square	$rms[x] = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$
variance	$\sigma^2[x] = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$
kurtosis	$k[x] = \frac{N\sum_{i=1}^{N}(x_i-\mu)^4}{[\sum_{i=1}^{N}(x_i-\mu)^2]^2}$
kurtosis of the speed	$ks[x] = k[\frac{d}{dt}x(t)]$
kurtosis of the derivative of the acceleration	$kda[x] = k[\frac{d^3}{dt^3}x(t)]$
skewness	$s[x] = \frac{\sum_{i=1}^{N}(x_i-\mu)^3}{N\sigma^3}$
crest factor	$cf[x] = \frac{max[x]}{rms[x]}$

3.2 Data Preparation

For the purpose of this work, a way of augmenting the dataset had to be developed as 315 samples is simply not enough to train a DNN.

To this end, portions of each signal were used to generate training samples, instead of the whole signal.
Since the rotational period of the engine is 8Hz, it would be expected that the minimum time needed to characterize a signal would be 0.125 seconds, but a longer time will account for lower frequency vibrations in the mechanical components or assembly housing caused by resonance.
Previous work at GIDTEC found that sections of 0.32768 seconds (out of the total 20 seconds length of each signal) should be enough to characterize a sample.

Then, in order to generate the training and testing datasets, the original 315 samples were divided in a 60/40% proportion in advance, to avoid cross-contamination.
Each of these parts was fed to the feature generating script, which was modified to randomly select several 0.32768 seconds windows from each input signal.

Some testing was performed using the Random Forest Classifier to assess the results obtained from training datasets generated using the full length signals versus those obtained from datasets generated using shorter sections.

Datasets containing the same number of samples as the original (315) were generated, by randomly trimming sections out of each signal, of different lengths.
The datasets were split 80/20% for training and testing, resulting in 252 training samples and 63 test samples.
Each test was repeated 10 times and averaged, with the results shown in Table 3.3.

TABLE 3.3: Performance comparison between datasets generated using different length signals

Signal length	**Classification accuracy (average)**
20s	99.157%
10s	98.848%
5s	97.948%
2s	97.051%
1s	95.991%
0.32678s	94.697%

Some loss in classification accuracy is observed, from 99.1% in the case of the full signal to 94.7% in the case of the 0.32678s sections.
This indicates there is some information loss when trimming the signals down to shorter sections, which is unavoidable since it is the only way to generate large enough datasets to train DNN.
On the other hand, this loss of accuracy should be counteracted by using the larger training datasets that this method allows.
To verify this hypothesis, larger datasets were generated from 0.32678s sections and tested using the Random Forest Classifier using a 80% training / 20% test split.
Again, each test was repeated 10 times, with the resulting averages presented in Table 3.4.

TABLE 3.4: Performance comparison between different size datasets generated from 0.32s signals

# of training samples (training/test)	**Classification accuracy (average)**
252/63	94.697%
504/126	96.291%
8064/2016	98.993%
16128/4032	99.206%

The results show that while using shorter lengths of the captures signals cause a decrease in accuracy, the increase in training data volume does indeed compensate

for this, and with a 64 times larger dataset the accuracy already surpasses that of the original dataset using full signals.

Using this process, a training set containing 60000 samples and a testing set containing 10000 samples were then generated to train the DNNs used in this work.
These sizes were chosen to be as close as possible to the sizes of the datasets provided with the example code from Professor Hinton, so it wouldn't be necessary to modify some parts of the code dealing with loading samples which have hard-coded values.

A second pair of training and tests sets with the same dimensions was prepared using the same method for use after training the DNNs.

These datasets were observed to contain very large numbers, tipically in the order of 10^6 to 10^7, but there are outliers in the 10^{12} range and probably higher, and many values close to zero.
The effect of the extremely high value outliers can be seen in the comparison between the 2 plots in Figure 3.1.

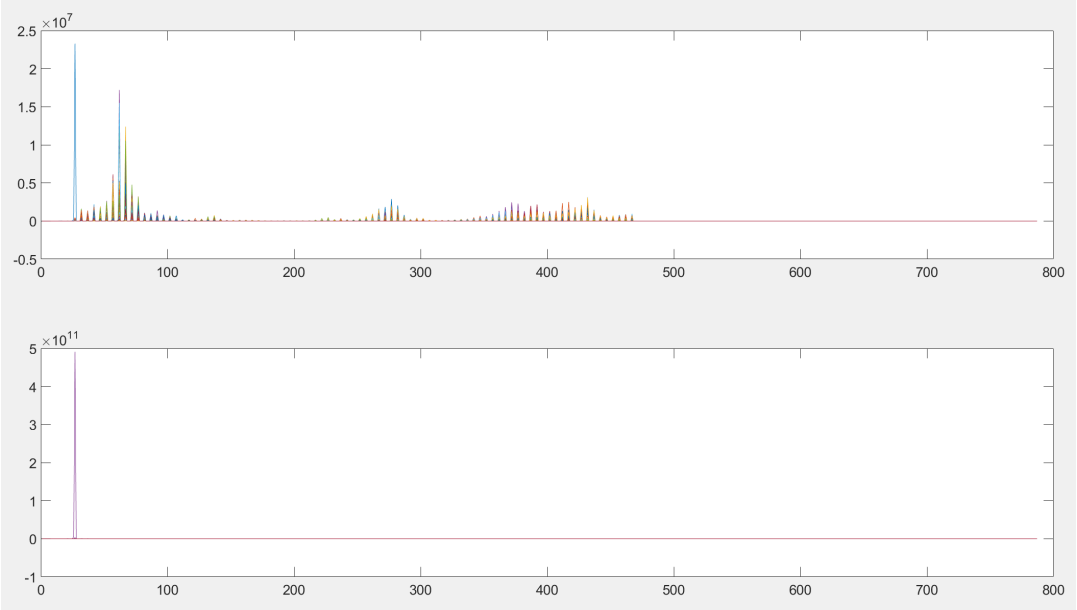


FIGURE 3.1: Plots of 2 different subsets of 1000 training samples.

Changing the scale of the second plot, the loss of detail caused by the outliers is made evident (Figure 3.2).
Even at this scale, there are features that disappear (from approximately abscissa 470 through to the end), which have values close to zero.

This is not suitable for the RBMs used in this study, which use the sigmoid function

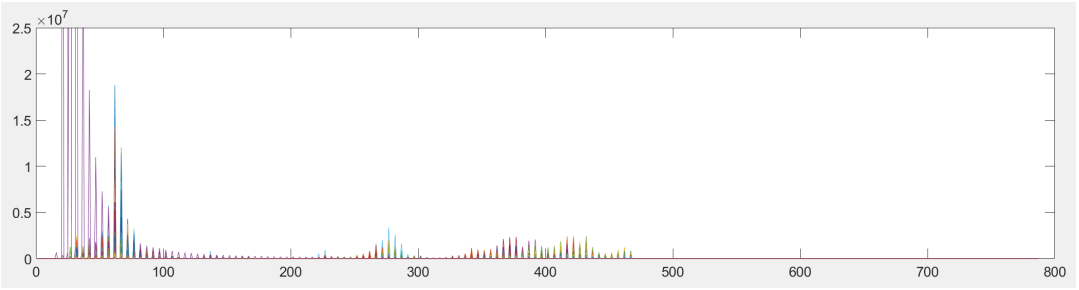


FIGURE 3.2: Detail of the second plot of Figure 3.1.

for transformations in both directions, and so are prepared to work with values between 0 and 1.

To address this issue, the dataset must be scaled down, and this end the Scikit-learn preprocessing library was used. This library has a wide variety of scaling functions, of which those that yield values between 0 and 1 were selected as candidates.

These functions were applied to some samples and the results plotted (Figure 3.3).

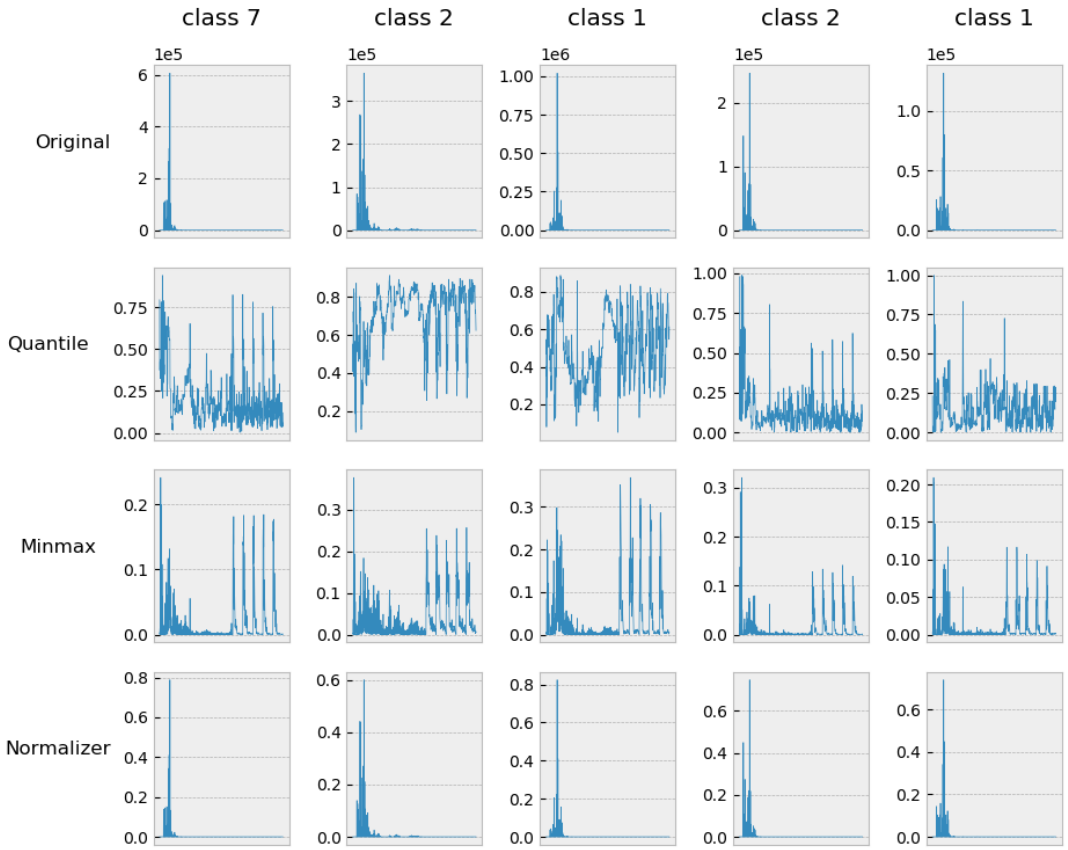


FIGURE 3.3: Examples of data transformations made using the Scikit-learn preprocessing library.

The plots show that the Normalizer scaler is not adequate for this case, as the samples

become very sparse.
This scaler simply divides all values by the maximum value to change the scale to the [0,1] interval.
The Minmax scaler was also discarded because of its small amplitude, as it yields maximum value around 0.4, which could cause issues with vanishing gradients during training.
The choice fell on the Quantile transformer, which applies a non-linear transformation, creating a dense uniform distribution that occupies most of the [0,1] interval, and is robust to outliers.

Chapter 4

Experiments using crisp Restricted Boltzmann Machines

This chapter contains the descriptions of the experiments performed with crisp Restricted Boltzmann Machines, using the Autoencoder and the Classification models.

4.1 Deep Belief Networks

The bulk of this work will be performed using code made available by Professor Geoffrey Hinton on his personal website at the University of Toronto ([18]), accompanying his paper for Science [7].

This code comprises 2 different configurations of Autoencoders.
The first of these configurations is composed of 3 layers of RBMs, and also includes a softmax layer.

Training consists of pre-training the RBMs, and then stacking them with the softmax layer and performing backpropagation (Figure 4.1).
This configuration will be referred to as the Classification model.

The output of this configuration is a class (more accurately, a vector of probabilities where the highest one is selected) corresponding to an input sample.

The other configuration is made up of 4 layers of RBMs and does not include a classifier layer.
Training consists of pre-training the RBMs, and then stacking them in 2 symmetrical halves, the first half being the Encoder and the second half the Decoder, and performing backpropagation on this complete Deep Autoencoder (Figure 2.1).
Once the training is complete, the second half (the Decoder) is discarded, and Encoder used to generate lower dimensional representation of the input samples, to which a separate classification algorithm, the Random Forest Classifier, is applied

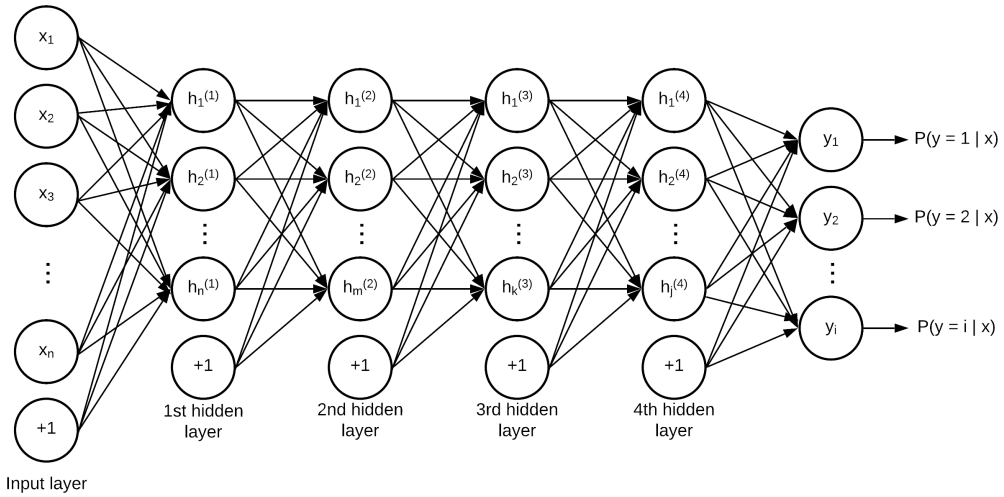


FIGURE 4.1: Autoencoder with Softmax output layer.

(Figure 2.6).
This configuration will be referred to as the Autoencoder.

These configurations were first modified to also create versions with 1, 2, 3, 4 and 5 layers, and afterwards fuzzy versions of all these were implemented according to [5] and [6].
This means that in total 20 configurations were tested.
The layers sizes were chosen arbitrarily and are as shown in Table 4.1.

TABLE 4.1: Layer configurations

# Layers	Layers sizes
1	20
2	50-20
3	100-50-20
4	200-100-50-20
5	400-200-100-50-20

4.2 scikit-learn

Some parts of this work made use of scikit-learn, which is a widely used Python language machine learning library [19] [20]. Specifically, the *sklearn.preprocessing* package for preparing the dataset, the *sklearn.ensemble* package containing the $RandomForestClassifier$ method, and the *sklearn.decomposition* package containing the PCA method.

The last two will be used to establish a classification accuracy baseline, and the Random Forest Classifier will also be used as the classifier for one of the DAE models (the other one will use a softmax classifier).

4.3 Autoencoder model

Starting with the Matlab code provided by Geoffrey Hinton [18], the first necessary modification was to the *makebatches.m* file, which is the script that loads the data for training and testing.
Since the new dataset was prepared to have the same dimensions, the modifications were minimal, essentially consisting of loading a different file. The script was renamed *makebatches_bearings.m*.

The next modification was to the *mnistdeepauto.m* file (the main execution script), or rather, multiple versions of the file were created for each of the layer configurations (5 in total).
These files were named *deepauto_c_*1.*m* through *deepauto_c_*5.*m*, where the suffix *c_*[*n*] indicates that it is a crisp model (*c*), and the number of layers ([*n*]).

The main execution scripts require corresponding backpropagation and conjugate gradient scripts, according to the number of layers.
These were named, respectively, *backpropae_c*1.*m* through *backpropae_c*5.*m* and *cg_ae_c*1.*m* through *cg_ae_c*5.*m*.

A check was introduced in the backpropagation script, which checks the progress every 10 epochs, and if the training and test accuracy haven't increased by more than 1%, the process is stopped early.

The scripts for the pre-training of individual layers(*rbm.m*) and the script for minimization of the error (*minimize.m*) did not require modifications.

A new script was created for each model, *treat_c_*[*n*].*m*, used to apply the resulting trained network to a dataset, which results in a new reduced dimensionality dataset to be used with a Random Forest Classifier.

Also, a script names *start.m* was created to simplify use. This script gets user input on model characteristics and number of layers, and starts the corresponding main execution script.

The pre-training is configured to run for 30 epochs for each layer and backpropagation is configured to run for 100 epochs.

For this model, the entire sequence is as follows:

1. Training is started by running the main execution script, corresponding to the desired number of layers (e.g, *deepauto_c_3.m* for 3 layers).
2. The *makebatches.m* script is called, which loads and divided the training data into batches.
3. The *rbm.m* script is called once for each layer, to perform pre-training.
4. Once pre-training is complete, backpropagation for fine tuning is initiated by calling the corresponding script (*backpropae_c_3.m* in the case of this example).
5. The backpropagation process calls the minimization script (*minimize.m*), and this in turn requires a minimization function adjusted to the number of layers of the model being trained (*cg_ae_c_3.m* in this example).
6. The output of the training process are 2 files: one containing the final set of weights, which represents the trained network, and another containing the training and test errors.
7. The script used to apply the trained network to create a reduced dimensionality dataset for use with the Random Forest Classifier is called (*treat_c_3.m* in this example).

The Random Forest Classifier (RFC) script was created in Python, using readily available libraries (notably, Scikit-learn [20], Numpy [21], Pandas [22], Seaborn [23] and Matplotlib [24]).

This script loads the datasets created in step 7, then using Scikit-learn and Numpy trains and tests an RFC and saves the results for analysis.
Several metrics can be recorded, like classification accuracy, precision, recall and F1-score. Additionally, using Pandas, Seaborn and Matplotlib, confusion matrices can also be calculated and displayed.

The entire process is repeated 30 times for better statistical reliability.

4.4 Classification model

This model is also a modification of the code provided by Geoffrey Hinton, similar to the Autoencoder model up to the backpropagation stage.
Because of this the data loading and main scripts can mostly be re-used.

The main scripts were named $deepclassify\_c\_[n].m$, where as in the previous model, the suffix $c\_[n]$ denotes it is a crisp model and the number of layers.

New scripts for backpropagation ($backpropclassify\_c\_[n]$) and for the conjugate gradient (two are needed for each model in this case, $cg\_classifyinit\_c\_[n]$ for the first 5 epochs and $cg\_classify\_c\_[n]$ for the rest of the process) were created.

New scripts that apply the trained models to a dataset for classification and save the real and predicted labels to a file were created, named $classify\_c\_[n].m$. These labels are used to extract performance metrics for the model.

Like the Autoencoder model, the pre-training runs for 30 epochs per layer, and the backpropagation runs for 100 epochs.

The entire sequence is slightly different from the Autoencoder model:

1. Training is started by running the main execution script, corresponding to the desired number of layers (e.g, $deepclassify\_c\_3.m$ for 3 layers).
2. The *makebatches.m* script is called, which loads and divided the training data into batches.
3. The $rbm.m$ script is called once for each layer, to perform pre-training.
4. Once pre-training is complete, backpropagation for fine tuning is initiated by calling the corresponding script ($backpropclassify\_c\_3.m$ in the case of this example).
5. The backpropagation process calls the minimization script ($minimize.m$), and this in turn requires a minimization function adjusted to the number of layers of the model being trained ($cg\_classifyinit\_c\_3$ and $cg\_classify\_c\_3.m$ in this example).
6. The output of the training process are 2 files: one containing the final set of weights, which represents the trained network, and another containing the training and test errors, both in terms of absolute error and number of incorrectly labeled examples.
7. The $classify\_c\_3.m$ script runs, applying the trained models to a test dataset and saving the predicted and real labels to a csv file.

4.5 Pre-Training

During pre-training, it is possible to observe the evolution of the process.
At the end of each epoch, the console displays the sum total of the error for that epoch, as well as the average error per batch.
This way, it is possible to observe the model's evolution, as shown in Figure 4.2 and

Figure 4.3, obtained respectively from the start and end of the training process.

```
### Epoch    1 ###
### Error Sum (Average) 12809.1459396019, Epoch Error (Average) 12.2830953825 ###
epoch 2
epoch 2 batch 100
epoch 2 batch 200
epoch 2 batch 300
epoch 2 batch 400
epoch 2 batch 500
epoch 2 batch 600
### Epoch    2 ###
### Error Sum (Average) 5869.1201169237, Epoch Error (Average) 8.5215237122 ###
epoch 3
```

FIGURE 4.2: Training Error (start of training), for the first layer of the 5 layer model (787-400 units).

```
epoch 29 batch 600
### Epoch   29 ###
### Error Sum (Average) 2376.6148166095, Epoch Error (Average) 4.4282733861 ###
epoch 30
epoch 30 batch 100
epoch 30 batch 200
epoch 30 batch 300
epoch 30 batch 400
epoch 30 batch 500
epoch 30 batch 600
### Epoch   30 ###
### Error Sum (Average) 2387.6021000154, Epoch Error (Average) 4.3463646578 ###
```

FIGURE 4.3: Training Error (end of training), for the first layer of the 5 layer model (787-400 units).

Also displayed during the pre-training process are plots of the input data and corresponding reconstruction.
This gives a visual representation of the model's progressing refinement.
Examples of these plots are shown in Figure 4.4 and Figure 4.5.
On both figures, the top plots represent the input data, so they are identical.
The bottom plots represent the reconstructions at the start and end of the training, respectively.

4.6 Fine-Tuning (Backpropagation)

The console output during the fine-tuning process is similar to that of the pre-training. At the end of each batch, the minimization function value is show (Figure 4.6), and at the end of each epoch the training and test errors.

For the Autoencoder model, the error shown is the squared error, and like in the pre-training process a plot of the input data and reconstruction is show at the end of each

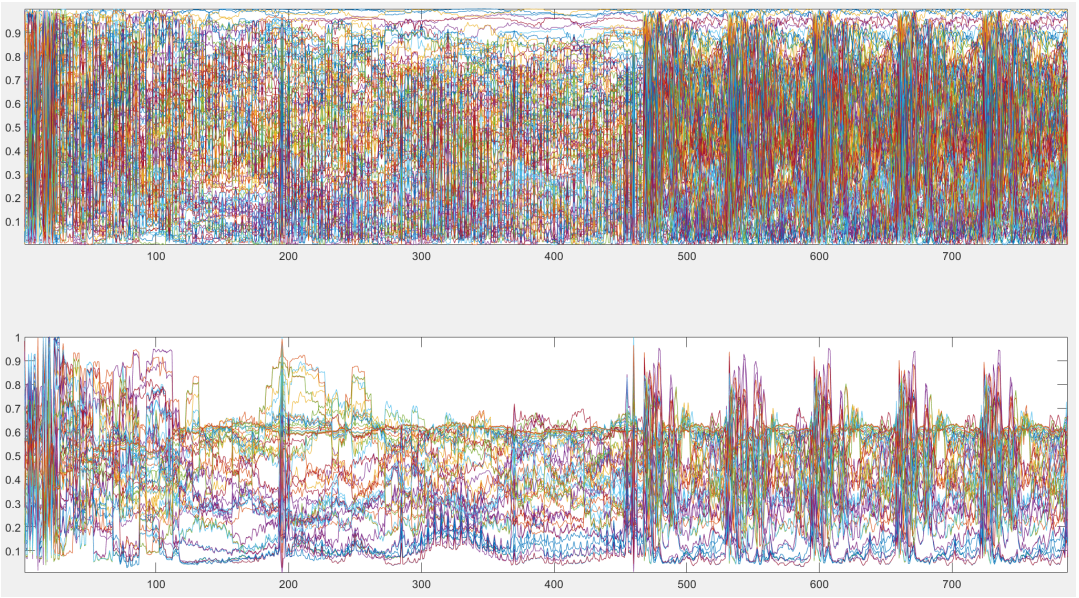


FIGURE 4.4: Input data and corresponding reconstruction (top and bottom, respectively; start of training), for the first layer of the 1 layer model (787-20 units).

epoch.
For the classifier model only the number of incorrectly labeled examples is displayed.

4.7 Results collection

After the training is done, the results are gathered, the method depending on the model in question.

For the Autoencoder model, once the training is complete, the $treat\_c\_[n].m$ script runs, applying the trained model to a training and test datasets.
This creates 2 new datasets with reduced dimensionality, which are in turn used to train and test a Random Forest Classifier implemented in Python using the Scikit-learn library.
This python script iterates through all the saved datasets, training and testing each one and saving the results to a csv file, which can then be opened in Microsoft Excel or a similar spreadsheet software for analysis.
It can also generate some metrics like confusion matrices, precision, recall and F1 scores during the process.

For the Classifier model, after the training, the $classify\_c\_[n].m$ script runs, which loads a test dataset and applies the trained model to it, outputting a csv file containing the number of correctly classified samples, as well as the ground truth and predicted

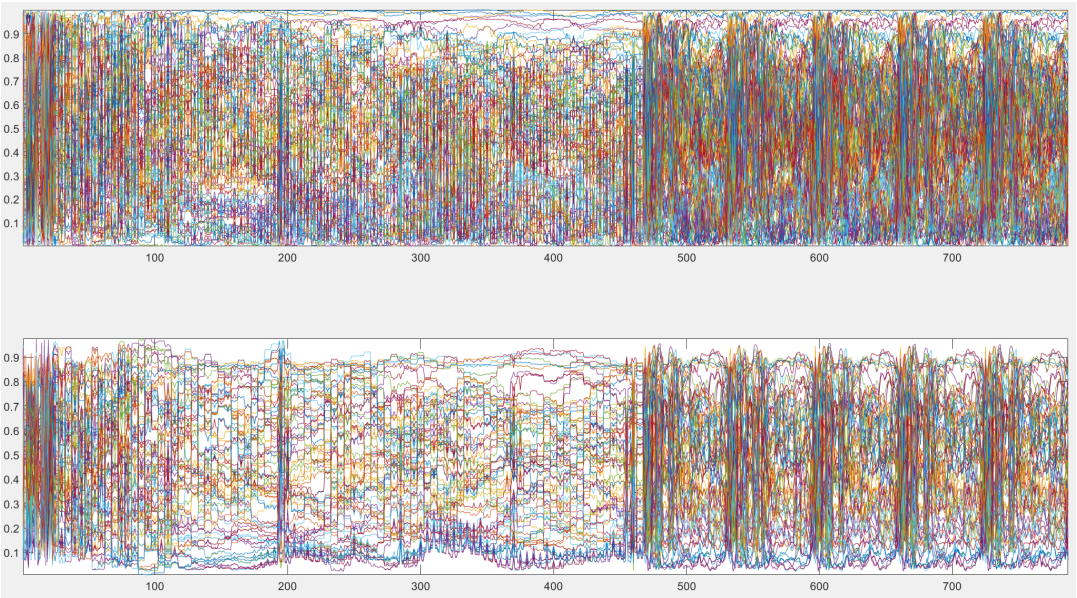


FIGURE 4.5: Input data and corresponding reconstruction (top and bottom, respectively; end of training), for the first layer of the 1 layer model (787-20 units).

class for each sample tested, which can be used to compute performance metrics.

```
Fine-tuning deep autoencoder by minimizing cross entropy error.
60 batches of 1000 cases each.
Size of the training dataset= 60000
Size of the test dataset= 10000
Before epoch 1 Train squared error: 41.752 Test squared error: 44.917
epoch 1 batch 1
Linesearch      1;  Value 4.775004e+02
Linesearch      2;  Value 4.692509e+02
Linesearch      3;  Value 4.558494e+02

epoch 1 batch 2
Linesearch      1;  Value 4.487314e+02
Linesearch      2;  Value 4.458771e+02
Linesearch      3;  Value 4.446963e+02
```

FIGURE 4.6: Backpropagation training and test errors, for the 5 layer Autoencoder model.

Chapter 5

Experiments using Fuzzy Restricted Boltzmann Machines

The successful implementation of a fuzzy Restricted Boltzmann Machine and using it to implement a Deep Autoencoder/Deep Belief Network is where the bulk of the effort applied to this study was spent.
Starting from the code developed for the crisp model (described in chapter 4), extensive modifications were required to almost every script.
Every set of weights and biases had to be doubled, and every operation involving the weights and biases, such as updates and sampling had to be modified to accommodate the new structure.

5.1 Autoencoder model

The data loading script was the only one to not require modifications.

First to be modified was the *rbm.m* script, and following algorithm 3, the variables holding the weights and biases,

```
vishid     = 0.1 * randn(numdims, numhid);
hidbiases = zeros(1, numhid);
visbiases = zeros(1, numdims);
```

are replaced by:

```
vishidL     = -0.01 * rand(numdims, numhid);
vishidR     = 0.01 * rand(numdims, numhid);

hidbiasesL = -0.01 * rand(1, numhid);
hidbiasesR = 0.01 * rand(1, numhid);
```

```
visbiasesL = -0.01 * rand(1, numdims);
visbiasesR = 0.01 * rand(1, numdims);
```

Each pair of variables (with suffixes "L" and "R", for "Left" and "Right") is initialized as very small uniformly distributed random numbers, with the "Left" side variables being negative and the "Right" variables being positive.

The variables holdings the activations for the hidden layer,

```
poshidprobs = zeros(numcases, numhid);

neghidprobs = zeros(numcases, numhid);
```

are also replaced:

```
poshidprobsL = zeros(numcases, numhid);
poshidprobsR = zeros(numcases, numhid);

neghidprobsL = zeros(numcases, numhid);
neghidprobsR = zeros(numcases, numhid);
```

The transformations also require modification, to account for the doubled weights and biases. For example,

```
poshidprobs = 1 ./ (1 + exp(-data*vishid ...
- repmat(hidbiases, numcases, 1)));
```

becomes:

```
poshidprobsL = 1 ./ (1 + exp(-data * vishidL ...
    - repmat(hidbiasesL, numcases, 1)));

poshidprobsR = 1 ./ (1 + exp(-data * vishidR ...
    - repmat(hidbiasesR, numcases, 1)));

poshidprobsD = (poshidprobsL + poshidprobsR) / 2;
```

where the suffix "D" on the last variable stands for "defuzzified", as it is a crisp output obtained from a fuzzy set. In this case, by a simple average as the fuzzy numbers used are symmetric triangular.

Once the weight updates are computed through Constrative Divergence, they are applied to each set of weights. This way, the distance between them remains constant.

```
deltaW = (posprods - negprods) / 2;
deltab = (posvisact - negvisact) / 2;
deltac = (poshidact - neghidact) / 2;

vishidinc = momentum * vishidinc ...
    + epsilonw * (deltaW / numcases);

visbiasinc = momentum * visbiasinc ...
    + (epsilonvb / numcases) * deltab;

hidbiasinc = momentum * hidbiasinc ...
+ (epsilonhb / numcases) * deltac;

vishidL = vishidL + vishidinc;
visbiasesL = visbiasesL + visbiasinc;
hidbiasesL = hidbiasesL + hidbiasinc;

vishidR = vishidR + vishidinc;
visbiasesR = visbiasesR + visbiasinc;
hidbiasesR = hidbiasesR + hidbiasinc;
```

During the implementation and testing of the fuzzy RBM, it was found that changing the lines:

```
%%%%%%%%% END OF POSITIVE PHASE  %%%%%%%%%

poshidstatesL = poshidprobsL > rand(numcases, numhid);
poshidstatesR = poshidprobsR > rand(numcases, numhid);

%%%%%%%%% START NEGATIVE PHASE  %%%%%%%%%
negdataL = 1 ./ (1 + exp(-poshidstatesL * vishidL' ...
    - repmat(visbiasesL, numcases, 1)));

negdataR = 1 ./ (1 + exp(-poshidstatesR * vishidR' ...
    - repmat(visbiasesR, numcases, 1)));

negdataD = (negdataL + negdataR) / 2;
```

to:

```
    %%%%%%%%% END OF POSITIVE PHASE  %%%%%%%%%

    %poshidstatesL = poshidprobsL > rand(numcases, numhid);
    %poshidstatesR = poshidprobsR > rand(numcases, numhid);

    %%%%%%%%% START NEGATIVE PHASE  %%%%%%%%%
    negdataL = 1 ./ (1 + exp(-poshidprobsL * vishidL' ...
        - repmat(visbiasesL, numcases, 1)));

    negdataR = 1 ./ (1 + exp(-poshidprobsR * vishidR' ...
        - repmat(visbiasesR, numcases, 1)));

    negdataD = (negdataL + negdataR) / 2;
```

yields better reconstruction results, with lower errors in training and testing.
With this alteration, the hidden units become Gaussian instead of Bernoulli, and the RBM becomes a GGRBM (Gaussian-Gaussian RBM).
This probably owes to the fact that this particular dataset is composed of continuous data, so working with Gaussian visible and hidden units is a better fit.
This change was retroactively applied to the crisp RBM and the improvement was also observed there.
Also changed in this script were the messages that are printed during the training (showing not a single error value, but 3, for the left and right bounds and the defuzzified value), and the visualization function that plots the input data and reconstruction (showing the left and right bounds and defuzzified reconstructions).

The new script was named $STFNrbm.m$ (for Symmetric Triangular Fuzzy Number Restricted Boltzmann Machine).

New main execution scripts were created, named $deepauto\_f\_[n].m$. The changes to these involved only duplicating variables, for example,

```
    hidpen = vishid;

    penrecbiases = hidbiases;

    hidgenbiases = visbiases;
```

changed to:

```
    hidpenL = vishidL;
    hidpenR = vishidR;

    penrecbiasesL = hidbiasesL;
    penrecbiasesR = hidbiasesR;

    hidgenbiasesL = visbiasesL;
    hidgenbiasesR = visbiasesR;
```

The updates to the backpropagation script and conjugate gradient function were relatively simple, after what was learned from implementing the fuzzy RBM. Once again, this consistent mostly of duplicating variables:

```
    %%%% PREINITIALIZE WEIGHTS OF THE AUTOENCODER %%%%
    w1=[vishid; hidrecbiases];
    w2=[vishid'; visbiases];
```

to:

```
    %%%% PREINITIALIZE WEIGHTS OF THE AUTOENCODER %%%%
    w1L=[vishidL; hidrecbiasesL];
    w1R=[vishidR; hidrecbiasesR];
    w2L=[vishidL'; visbiasesL];
    w2R=[vishidR'; visbiasesR];
```

and the activations to be the defuzzified product of the fuzzy set, changing for example,

```
    w1probs = 1 ./ (1 + exp(-data * w1));
```

to:

```
    w1probsL = 1 ./ (1 + exp(-data * w1L));
    w1probsR = 1 ./ (1 + exp(-data * w1R));
    w1probs = [((w1probsL + w1probsR) / 2) ones(N,1)];
```

Modifying the *minimize.m* function was a more laborious process as it is a fairly complex script, as well as the variables in it having very non descriptive names, but the actual changes necessary were minimal, and the new script named $minimizef.m$. The scripts needed to apply the trained models to create the training and testing datasets for use with the Random Forest Classifier were he last modification required,

TABLE 5.1: Training times and test error for Autoencoder crisp and fuzzy models (yellow for crisp and blue for fuzzy), according to number of layers.

Model	Time (minutes:seconds)	Test Reconstruction Error
c1	14:41	5.391
c2	4:24	4.252
c3	24:28	3.410
c4	28:04	2.149
c5	44:50	2.023
f1	29:51	4.939
f2	25:33	2.870
f3	30:58	2.577
f4	60:43	2.711
f5	113:31	2.854

and these were fairly simple to accomplish.
With the modifications to the scripts complete, the actual working of the models was exactly the same as that of the crisp models, with the same sequence of events and end result.

Some casual testing revealed significantly increased training times for the fuzzy models, which can be seen in Table 5.1, along with the final reconstruction error on the test set.
The reconstruction performance is very similar, but this can't be fully assessed with only one example and before applying the Random Forest Classifier to the resulting datasets.

5.2 Classifier model

Creating the scripts for the fuzzy Classifier model was trivial after the groundwork laid down in creating the crisp models and the fuzzy Autoencoder.
As with the Autoencoder model, some testing was performed to quickly gauge the training times and performance of the fuzzy Classifier versus the crisp version, the results of which are presented in Table 5.2.
Again, apart from an outlier in the crisp 4 layers model, the performance is similar, albeit with a small advantage to the fuzzy models.

TABLE 5.2: Training times and test classification errors (out of 10000) for crisp and fuzzy Classifier models (yellow for crisp and blue for fuzzy), according to number of layers.

Model	Time (minutes:seconds)	Test Classification Errors
c1	4:04	82
c2	4:36	154
c3	5:25	186
c4	10:47	3258
c5	10:20	161
f1	7:10	59
f2	13:44	26
f3	19:32	3
f4	36:55	40
f5	65:43	13

5.3 Pre-Training

The messages printed to the console during training differ from those of the crisp model, as now there are 3 error values being reported: the overall value, which is by defuzzification of the left and right bounds, as well as the error values corresponding to each of these bounds.

Examples of these messages are shown in Figure 5.1 and Figure 5.2, which correspond to the start and end of the training process, respectively.

```
### Epoch    1 ###
### Error Sum (Average) 17886.1346654732, Epoch Error (Average) 22.4336093754 ###
### Error Sum (Left)    27570.2610150897, Epoch Error (Left)    46.4147602276 ###
### Error Sum (Right)   27154.6860627531, Epoch Error (Right)   33.9257468904 ###
epoch 2
epoch 2 batch 100
epoch 2 batch 200
epoch 2 batch 300
epoch 2 batch 400
epoch 2 batch 500
epoch 2 batch 600
### Epoch    2 ###
### Error Sum (Average) 9851.1561011348, Epoch Error (Average) 14.0932284372 ###
### Error Sum (Left)    20562.3245322544, Epoch Error (Left)    32.6937821674 ###
### Error Sum (Right)   18235.5626345932, Epoch Error (Right)   24.4992503042 ###
epoch 3
```

FIGURE 5.1: Training Error (start of training), for the first layer of the 5 layer model (787-400 units).

The plots of the input data and corresponding reconstructions are also expanded to show the 3 reconstructions.

Examples of these plots are shown in Figure 5.3 and Figure 5.4.

```
epoch 29 batch 600
### Epoch   29 ###
### Error Sum (Average) 992.9303998673, Epoch Error (Average) 1.6922774487 ###
### Error Sum (Left)    11521.8565102026, Epoch Error (Left)    18.5055904154 ###
### Error Sum (Right)   10872.9640604999, Epoch Error (Right)   17.2713749909 ###
epoch 30
epoch 30 batch 100
epoch 30 batch 200
epoch 30 batch 300
epoch 30 batch 400
epoch 30 batch 500
epoch 30 batch 600
### Epoch   30 ###
### Error Sum (Average) 976.4376281381, Epoch Error (Average) 1.6625921910 ###
### Error Sum (Left)    11535.8837978130, Epoch Error (Left)    18.5393223744 ###
### Error Sum (Right)   10897.2310547782, Epoch Error (Right)   17.3240995679 ###
```

FIGURE 5.2: Training Error (end of training), for the first layer of the 5 layer model (787-400 units).

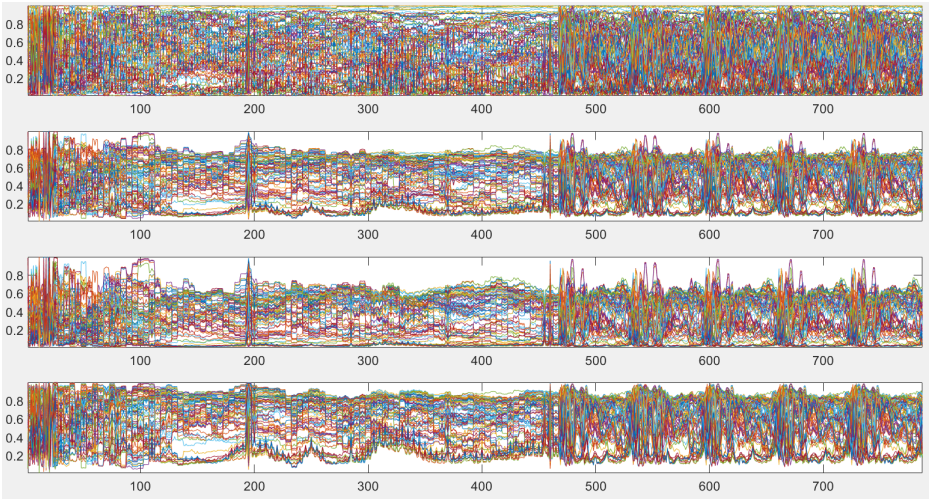


FIGURE 5.3: Input data and corresponding reconstructions (input data on top, then defuzzified reconstruction, left bound reconstruction and right bound reconstruction on the bottom; start of training), for the first layer of the 5 layer fuzzy model (787-400 units).

5.4 Fine-Tuning (Backpropagation)

The backpropagation process happens in exactly the same way, barring the differences in the scripts due to the use of fuzzy variables, as with the crisp models.
The console output is identical, the only difference the plots of the reconstructions during the Autoencoder model's fine-tuning.

5.5 Results collection

The outputs are also identical, a reduced dimensionality dataset in the case of the Autoencoder, and a csv file with the classification results for the Classifier model.

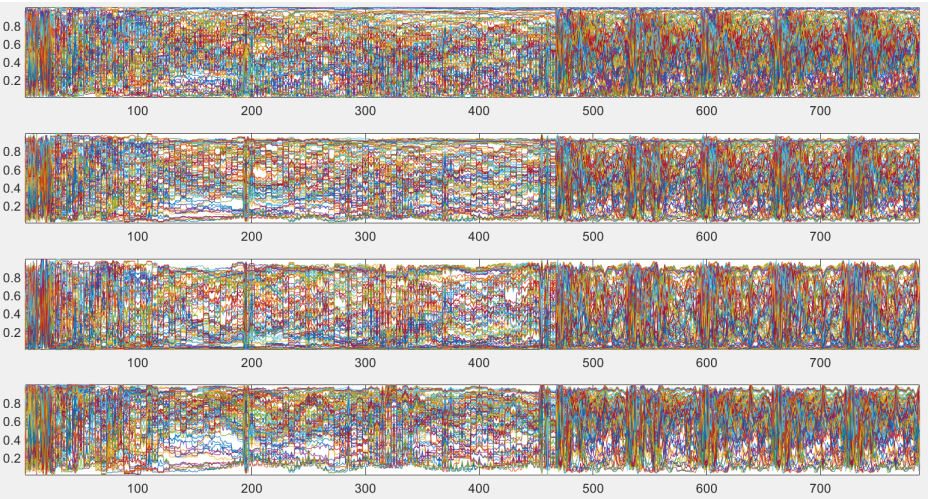


FIGURE 5.4: Input data and corresponding reconstructions (input data on top, then defuzzified reconstruction, left bound reconstruction and right bound reconstruction on the bottom; end of training), for the first layer of the 5 layer fuzzy model (787-400 units).

Chapter 6

Discussion of the experimental results

6.1 Metrics used for analysis of the results

For the presentation of the experimental results, the following structure is used: first, the results of the experiments are presented, then the results are discussed and compared with the results of the previous experiments.
Finally, the results are summarized and the conclusions are drawn.

In terms of the metrics used for analysis of the results obtained, 2 were selected. The first one is Accuracy, which is simply defined as the ratio of the number of correctly classified samples to the total number of samples:

$$accuracy = \frac{number\,of\,correctly\,classified\,samples}{total\,number\,of\,samples}. \tag{6.1}$$

More specifically, accuracy can be defined as:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN'} \tag{6.2}$$

where TP, TN, FP, FN are the number of true positives, true negatives, false positives and false negatives, respectively. These in turn are defined as:

- True Positives (TP): the number of samples that are correctly classified as positive.
- True Negatives (TN): the number of samples that are correctly classified as negative.
- False Positives (FP): the number of samples that are incorrectly classified as positive.

- False Negatives (FN): the number of samples that are incorrectly classified as negative.

Accuracy is not a good indicator of performance in the case of imbalanced datasets. For example, if a binary dataset is made up of 99% samples of class A and 1% samples of class B, a classifier that always predicts class A will have an accuracy of 99%.
In the case of the dataset used in this work, it is perfectly balanced so accuracy is a good metric to use.

The other metric selected was area under the curve, or AUC. Specifically, the curve is the receiver operating characteristic (ROC) curve.
The ROC curve is a plot of the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The TPR or recall is defined as:

$$TPR = \frac{TP}{TP + FN'}, \tag{6.3}$$

i.e, the number of correctly classified positive samples divided by the total number of positive samples. The FPR or fallout is in turn defined as:

$$FPR = \frac{FP}{FP + TN'}, \tag{6.4}$$

i.e, the number of incorrectly classified negative samples divided by the total number of negative samples. These are combined into a single metric by computing many distinct probability thresholds and plotting the TPR and FPR at each threshold in a single graph.
The curve represented in this graph is the ROC curve, and the metric considered here is the area under the curve, or AUC.
An example of the ROC curve is shown in Figure 6.1.

The higher the AUC, the better the performance of the model at distinguishing between the positive and negative classes.
A perfect classifier will have an AUC of 1, while a random classifier will have an AUC of 0.5.
The latter would be a so called no-skill classifier, essentialy a classifier that makes random guesses.
So a high AUC means that a randomly chosen classified as positive is indeed positive.
A classifier with an AUC of 0 would be one that incorrectly guesses every label. The AUC is a good metric to use even in the case of imbalanced datasets, as it is not affected by the class distribution.

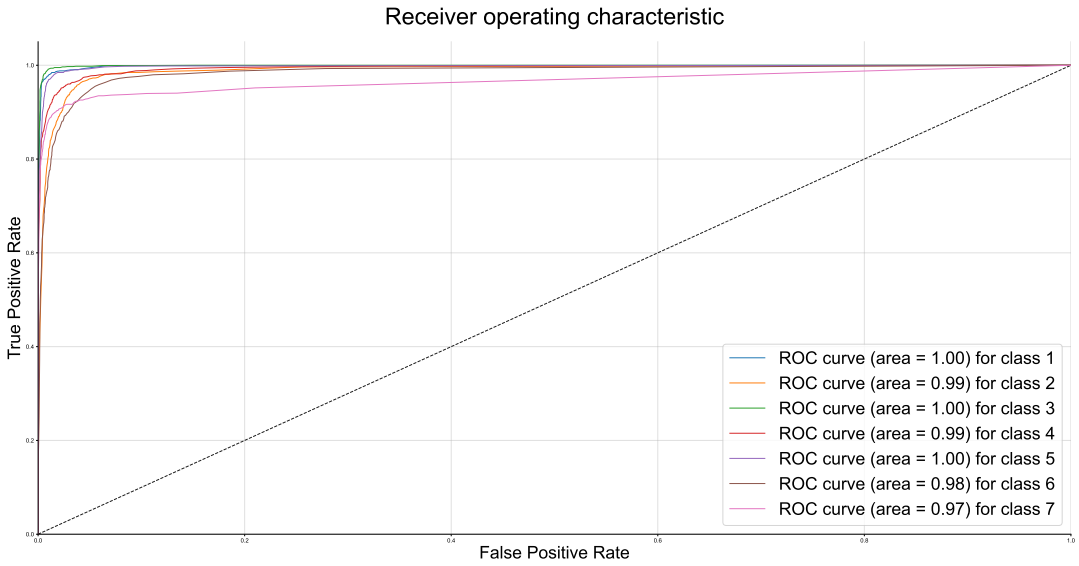


FIGURE 6.1: Accuracy Boxplot for the Crisp Autoencoder models.

AUC-ROC is tipically used for binary classification problems, but it can be extended to multi-class classification problems by using one of two methods, one versus rest (OvR) and one versus one (OvO).
In the OvR method, the ROC curve is computed for each class against the rest of the classes, and the AUC is computed for each class, and the results averaged. Conversely, in the OvO method, the ROC curve is computed for each pair of classes, and the AUC is computed for each pair, and the results averaged.
Whichever method is selected, after averaging the AUCs, the final result is a single AUC value that can be used to compare the performance of the classifier and is the same for both methods.

6.2 Results of the experiments

The results will be presented in the following order: first, the results of the experiments with the autoencoder models will be presented, then the results of the experiments with the classifier models will be presented, and finally the results of the experiments with the traditional algorithms models will be presented.
A comparison of the best models from each class will also be presented.

6.3 Results using the Autoencoder configuration

6.3.1 Crisp Autoencoder

The results for the crisp autoencoder models are shown in Figure 6.2, in the form of a boxplot. The boxplot shows the distribution of the accuracy values for each model, and the median value is shown as a horizontal line.
For this model, the best configuration is the 1 layer, with an average accuracy of 98.382%, followed closely by the 5 layer configuration with an average accuracy of 98.337%.

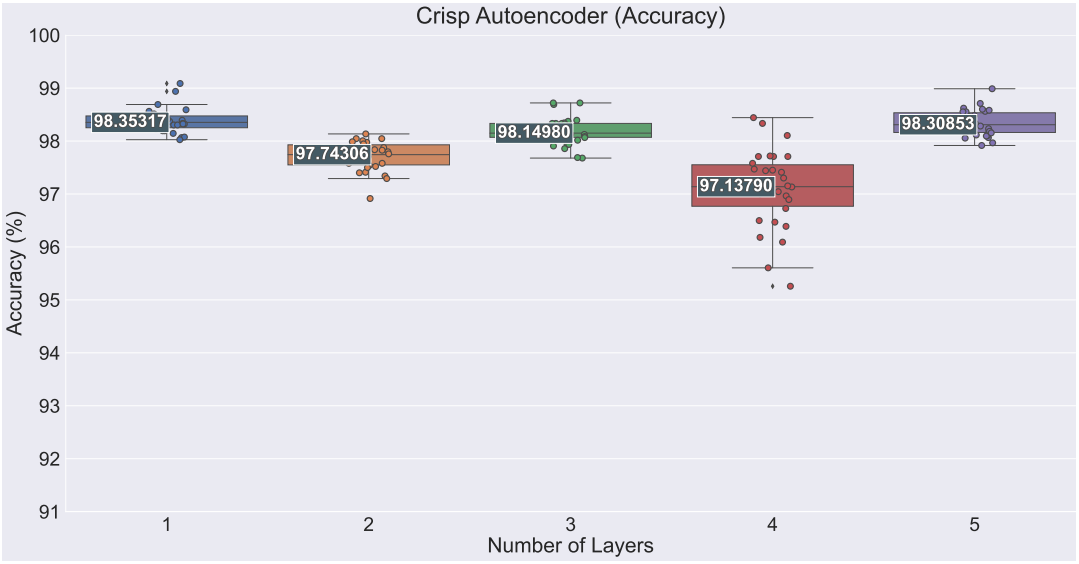


FIGURE 6.2: Accuracy Boxplot for the Crisp Autoencoder models.

In fact, all the configurations show average accuracies above 97%, which can be considered very good result.
The worst configuration is the 4 layer configuration, with an average accuracy of 97.103%. This configuration also has the lowest minimum accuracy and the highest standard deviation, which means that the model is less consistent, as the results are more spread out. The statistics for this model are summarized in Table 6.1.
Values shown in green represent the best values for each metric, and values shown in red represent the worst values for each metric.
Here it is clear that the 1 layer model is the best, as it holds the highest values for all metrics (lowest for standard deviation), and the 4 layer model is the worst, with the lowest values for almost all the metrics (highest for standard deviation).

The AUC scores are shown in boxplot form in Figure 6.3, and the statistics are shown in Table 6.2.

TABLE 6.1: Accuracy statistics for the Crisp Autoencoder models.

	# of Layers				
	1	2	3	4	5
Max	99.087	98.135	98.702	98.443	98.988
Min	98.026	96.915	97.679	95.258	97.917
Mean	98.382	97.716	98.188	97.103	98.337
Std	2.28E-01	2.67E-01	2.54E-01	7.2E-01	2.38E-01

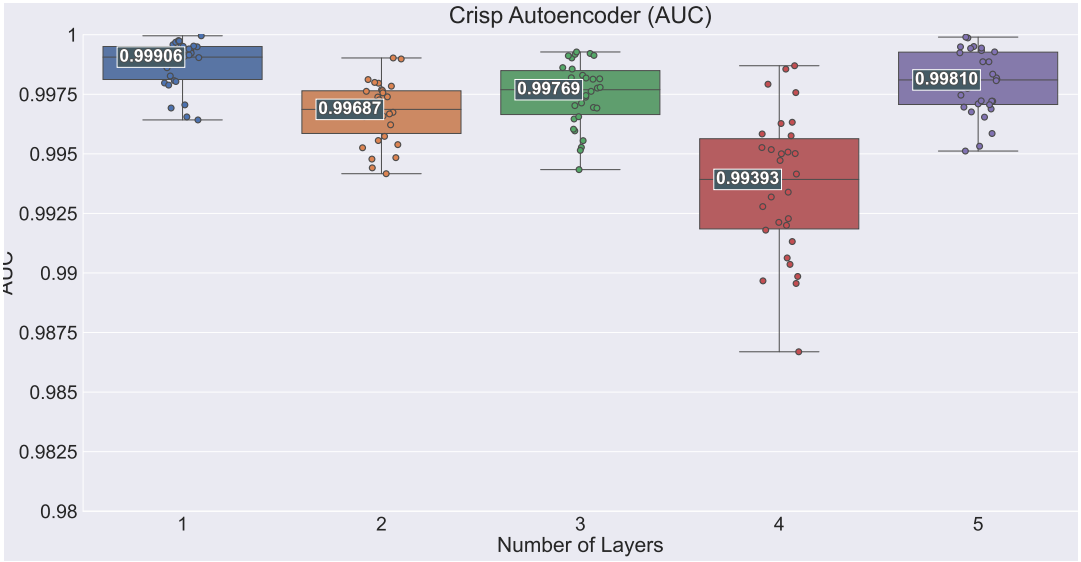


FIGURE 6.3: AUC Boxplot for the Crisp Autoencoder models.

These scores are also very good, with the best configuration having an average AUC of 0.99874, and the worst configuration having an average AUC of 0.99369, and are in fact higher than the accuracy values.
This can happen because accuracy is only measured at a specific threshold (usually 0.5), while AUC is measured for all possible thresholds, and so it is more sensitive to the performance of the model.
The AUC results reflect the accuracy results in terms of model performance, including variability.

6.3.2 Fuzzy Autoencoder

The results for the fuzzy autoencoder models are shown in Figure 6.4, in the form of a boxplot and Table 6.3, in the form of a table.

This configuration shows slightly better performance compared to the crisp autoencoder, and for this configuration the best model is now the 3 layer, with an average

TABLE 6.2: AUC statistics for the Crisp Autoencoder models.

	# of Layers				
	1	2	3	4	5
Max	0.99996	0.99902	0.99928	0.9987	0.9999
Min	0.99642	0.99417	0.99434	0.98669	0.99512
Mean	0.99874	0.99676	0.99748	0.99369	0.99795
Std	9.6E-04	1.25E-03	1.33E-03	2.89E-03	1.32E-03

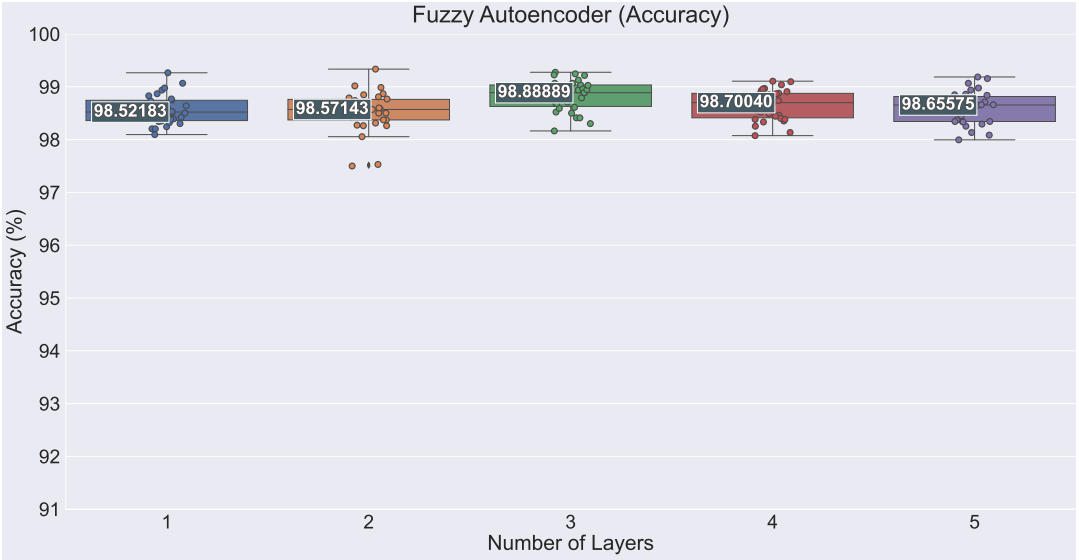


FIGURE 6.4: Accuracy Boxplot for the Fuzzy Autoencoder models.

accuracy of 98.829%, 0.447% higher than the best model from the crisp configuration.

TABLE 6.3: Accuracy statistics for the Fuzzy Autoencoder models.

	# of Layers				
	1	2	3	4	5
Max	99.266	99.335	99.276	99.107	99.187
Min	98.095	97.500	98.165	98.075	97.996
Mean	98.563	98.529	98.829	98.653	98.599
Std	2.76E-01	3.76E-01	2.87E-01	2.84E-01	3.06E-01

The AUC results for this model are very good, as seen in Figure 6.5, with the all the results for every configuration being above 0.995. In particular, the 3 and 4 layer configurations have minimum values above 0.997 and mean values above 0.999.

In Table 6.4 the statistics for the AUC scores are shown. The 3 layer configuration is clearly the best, only behind the 4 layer configuration in the maximum value statistic

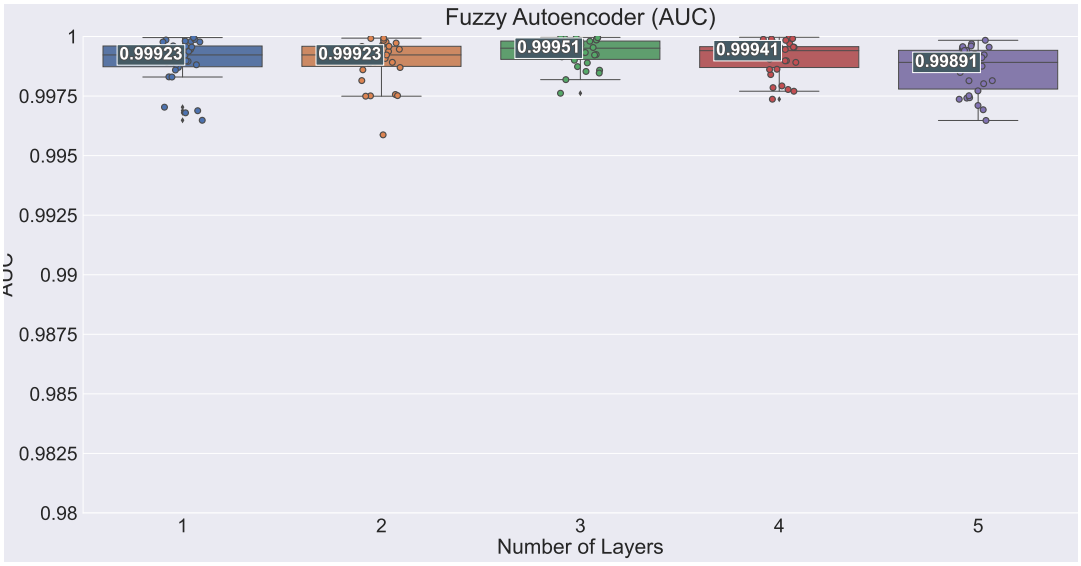


FIGURE 6.5: AUC Boxplot for the Fuzzy Autoencoder models.

by a negligible margin. It is also clear the advantage of this model in relation to the crisp variant, even in terms of variability.

TABLE 6.4: AUC statistics for the Fuzzy Autoencoder models.

	# of Layers				
	1	2	3	4	5
Max	0.99996	0.99993	0.99996	0.99997	0.99984
Min	0.99648	0.99587	0.99762	0.99737	0.99648
Mean	0.99894	0.99895	0.99934	0.99911	0.99861
Std	9.6E-04	9.2E-04	5.9E-04	7.4E-04	9.5E-04

6.4 Results using the Classifier configuration

6.4.1 Crisp Classifier

This section contains the results for the classifier model, and as in the previous section, first are shown the results for the crisp and then the fuzzy configuration.

The accuracy results for the crisp classifier are shown in Figure 6.6, in the form of a boxplot and Table 6.5, in the form of a table.
For this model, the best accuracy results were obtained with the 2 layer configuration, with 99.184% mean accuracy and 99.920% maximum accuracy, and very close to the best minimum and standard deviation values which were obtained with the 3 layer configuration.

The worst results were obtained with the 4 layer configuration, as with the crisp autoender model. In this instance, the results were so uncharacteristic compared to the other configurations that the initial conclusion was that there must have been a problem with the implementation.

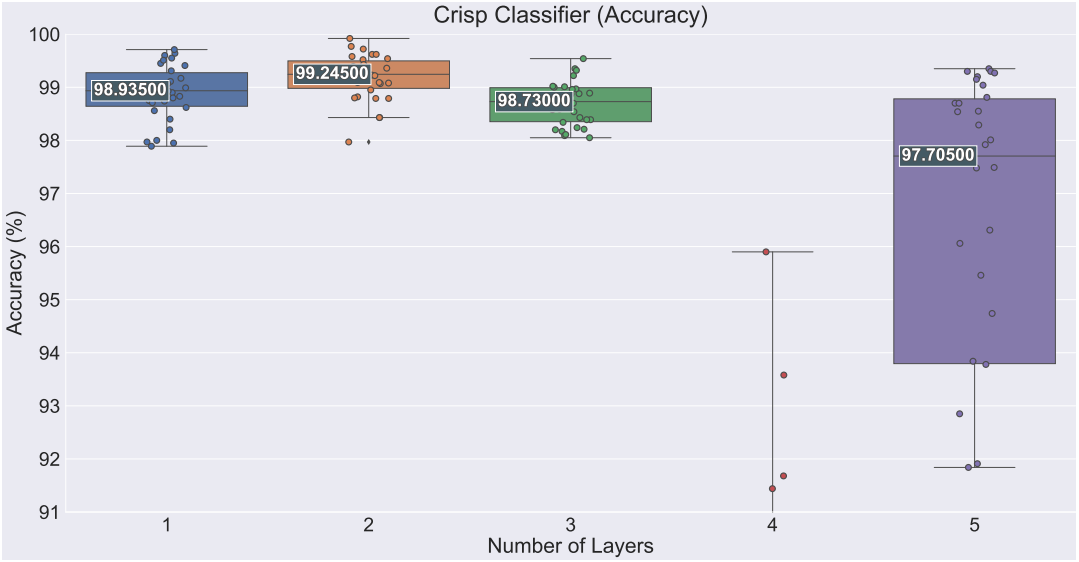


FIGURE 6.6: Accuracy Boxplot for the Crisp Classifier models.

But even after recreating the implementation twice, starting from two different well performing configurations (the 3 layer version, by adding a layer, and the 5 layer version, by removing a layer), the results were still very poor.
The configuration was also tested with different learning rates, which led to no improvement. Still, on some of the train/test iterations, the 4 layer configuration did perform well, achieving over 95% accuracy, but overall the results were not consistent enough to be considered a valid configuration. The fact that on the Autoencoder model, the 4 layer configuration was also the worst might indicate that there exists a structural incompatibility that causes the 4 layers models to perform worse than the others.
A second boxplot, zoomed out to make it possible to observe all of the results is shown in Figure 6.7. The 5 layer configuration also has several outliers, but the results are more consistent than the 4 layer configuration.

The statistics for the accuracy results are shown in Table 6.5. It is evident from the values in this table that the 4 layer configuration fails to converge on many of the train/test iterations, with minimum accuracy values around 14%, which for a 7 class classification problem represents a random guess.

The variability for this model is also very high, with the standard deviation being the highest of all the models.

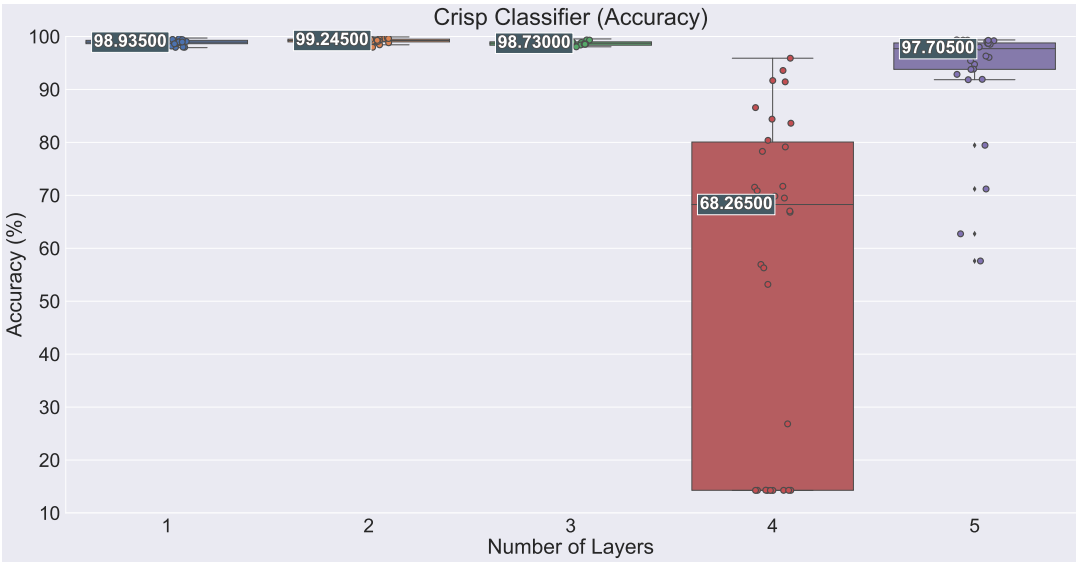


FIGURE 6.7: Accuracy Boxplot for the Crisp Classifier models (zoomed out).

TABLE 6.5: Accuracy statistics for the Crisp Classifier models.

	# of Layers				
	1	2	3	4	5
Max	99.710	99.920	99.540	95.900	99.350
Min	97.890	97.970	98.050	14.260	57.600
Mean	98.887	99.184	98.690	55.800	93.163
Std	5.17E-01	4.26E-01	4.09E-01	30.280	10.661

The AUC results follow those of the accuracy closely, with the 4 layer configuration being the worst, but now the 1 layer configuration shows the best results.

The results are shown in Figure 6.8 and Table 6.6. As like for the accuracy results, an extra boxplot is shown, Figure 6.9, zoomed out to show the whole picture.

6.4.2 Fuzzy Classifier

The results for the fuzzy Classifier model are shown in Figure 6.10 and Table 6.7. This model shows the best results yet, with all configurations achieving mean accuracy above 99% (and the 2 best ones, the 3 and 4 layers configurations, also with minimum accuracy above that value), and maximum accuracy above 99.9%.

This corresponds to single digit classification errors in a test dataset of 10000 samples. The weakest configuration for this model is the 5 layer configuration, with a mean accuracy of just under 98%.

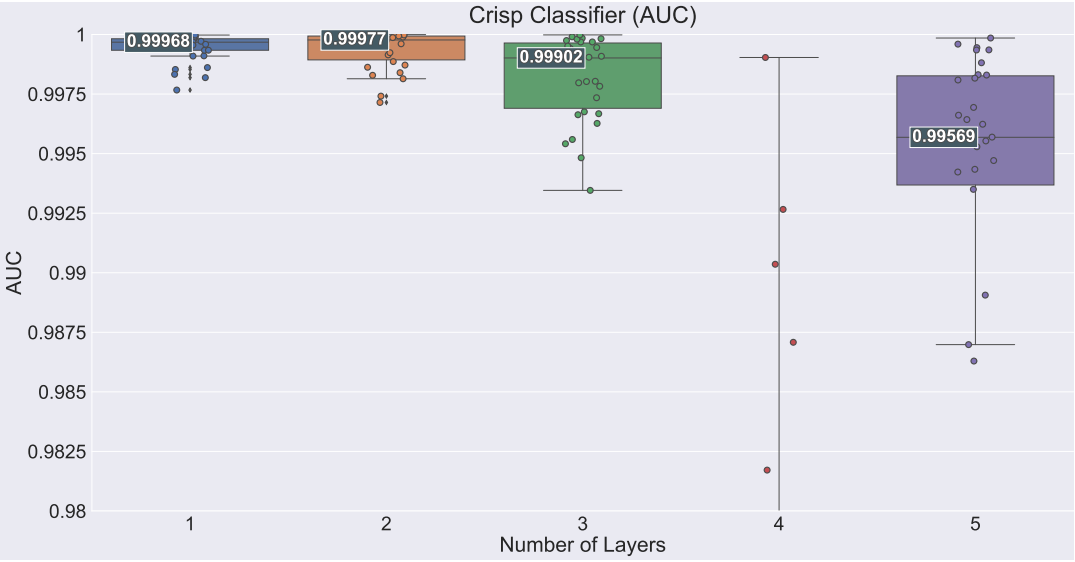


FIGURE 6.8: AUC Boxplot for the Crisp Classifier models.

TABLE 6.6: AUC statistics for the Crisp Classifier models.

	# of Layers				
	1	2	3	4	5
Max	0.99998	0.99999	0.99998	0.99904	0.99985
Min	0.99767	0.99715	0.99346	0.47292	0.89212
Mean	0.99944	0.99936	0.9982	0.8013	0.98719
Std	5.9E-04	7.9E-04	1.73E-03	2.09E-01	2.43E-02

The AUC statistics for this model, shown in Figure 6.11 and Table 6.8, are very high, with all configurations achieving mean AUC above 0.999%, and maximum values of effectively 1. In relation to this metric, the weakest configuration is the 2 layer configuration, with a single result under 0.997 and slightly higher variabililty when compared to the other configurations.

6.5 Results using traditional algorithms

Experiments using traditional algorithms were also conducted, in order to establish a baseline for comparison with the results obtained using the proposed models. The algorithms used were the Random Forest Classifier (RFC) only, and the RFC combined with the Principal Component Analysis (PCA) algorithm.

Two variants of the PCA+RFC model were used, one with 20 components, like the final layer of the Autoencoder and Classifier models, and one with only 3 components, to explore the lower bounds in terms of the number of features necessary to

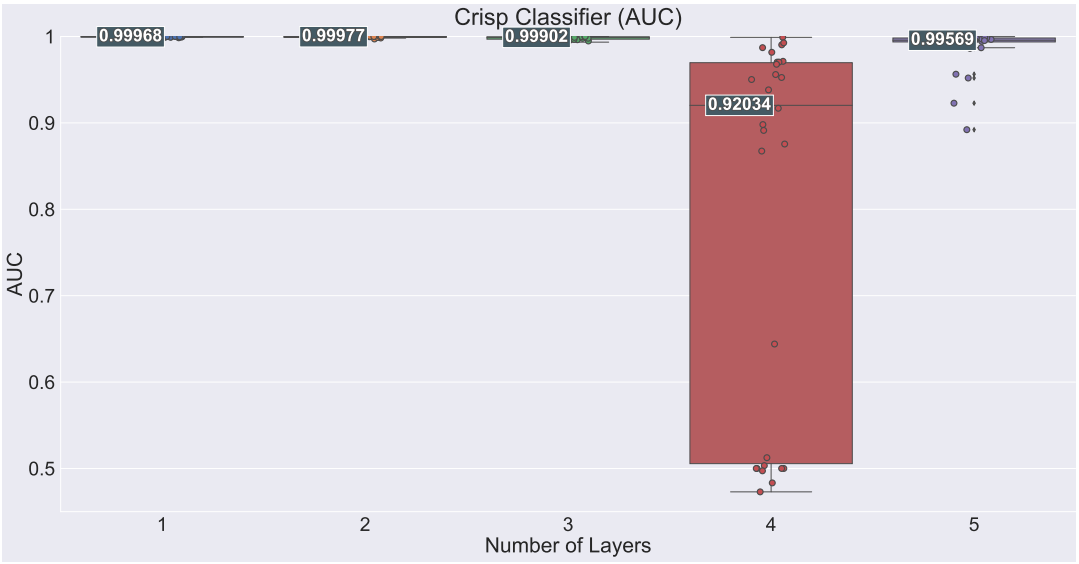


FIGURE 6.9: AUC Boxplot for the Crisp Classifier models (zoomed out).

TABLE 6.7: Accuracy statistics for the Fuzzy Classifier models.

	# of Layers				
	1	2	3	4	5
Max	99.940	99.930	99.920	99.980	99.930
Min	98.380	98.390	99.120	99.040	97.960
Mean	99.390	99.158	99.630	99.678	99.270
Std	4.1E-01	3.99E-01	2.17E-01	2.62E-01	4.24E-01

achieve good results.

Furthermore, these models were trained with 2 differents datasets.
Firstly, the original dataset made up of 315 samples, which is not enough to train DNN but for traditional algorithms could be produce workable results. Secondly, the same augmented dataset used to train the Autoencoder and Classifier models.
The accuracy results of these experiments are shown in Figure 6.12 and Table 6.9, where the superscript $\circ$ denotes models trained with the original dataset. On the left side are the 3 models trained with the original dataset, and on the right side are the 3 models trained with the augmented dataset.
While the 3 components PCA+RFC model shows markedly inferior performance compared to the other models, the 20 components PCA+RFC model was able to reach slightly higher accuracy compared to the RFC model.

It is also apparent the increase in performance resulting from the use of the augmented dataset, in terms of average and minimum accuracy, as well as much reduced

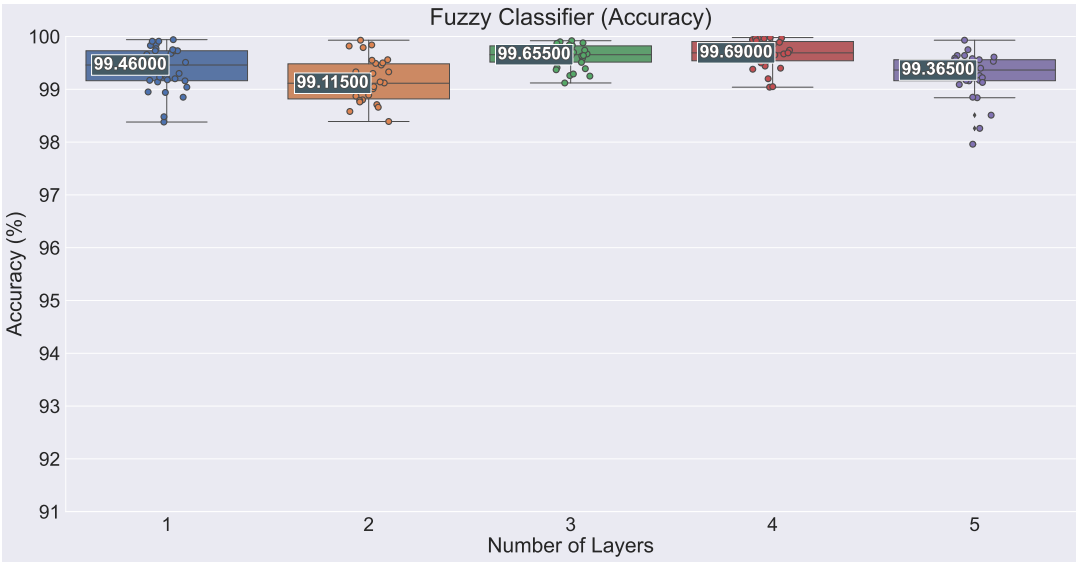


FIGURE 6.10: Accuracy Boxplot for the Fuzzy Classifier models.

TABLE 6.8: AUC statistics for the Fuzzy Classifier models.

	# of Layers				
	1	2	3	4	5
Max	0.9999992	0.9999998	1	1	0.9999996
Min	0.99848	0.9968	0.99934	0.99951	0.99852
Mean	0.99971	0.99945	0.99989	0.99995	0.9997
Std	4E-04	7E-04	1.5E-04	9.8E-05	4.2E-04

variability.

Models trained with the original dataset did manage to obtain slightly higher maximum accuracy values, but it must be taken into account that the testing dataset for these is less than 80 samples, while the augmented dataset uses 10000 samples for testing.

The AUC results for the RFC and PCA+RFC models are presented in Figure 6.13 and Table 6.10.

TABLE 6.9: Accuracy statistics for the RFC and PCA+RFC models.

	Model					
	RFC^o	PCA20+RFC^o	PCA3+RFC^o	RFC	PCA20+RFC	PCA3+RFC
Max	100	100	93.59	99.89	99.99	92.38
Min	93.59	96.154	79.487	99.6	99.96	92.06
Mean	97.436	98.376	86.88	99.771	99.979	92.212
Std	2.0406	1.2792	4.5107	5.97E-02	8.46E-03	8.24E-02

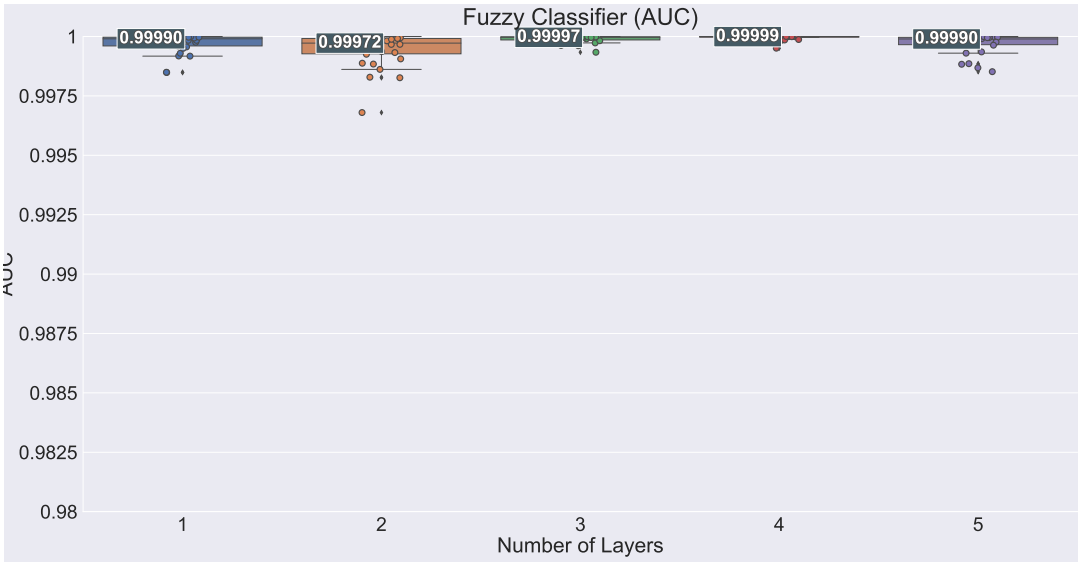


FIGURE 6.11: AUC Boxplot for the Fuzzy Classifier models.

The results are very similar to the accuracy results, with the 3 components PCA+RFC model showing inferior performance, and the 20 components PCA+RFC model showing slightly higher performance.
The use of the augmented dataset also resulted in a significant increase in performance, with 2 of the models (RFC and PCA+RFC with 20 components) achieving perfect scores.

TABLE 6.10: AUC statistics for the RFC and PCA+RFC models.

	Model					
	RFC°	PCA20+RFC°	PCA3+RFC°	RFC	PCA20+RFC	PCA3+RFC
Max	1	1	0.9953	1	1	0.99058
Min	0.98915	0.9892	0.95031	1	1	0.9892
Mean	0.99809	0.9976	0.98286	1	1	0.9899
Std	2.96E-03	3.57E-03	9.87E-03	8.54E-08	3.45E-08	3.74E-04

6.6 Comparison of Results

This section contains a comparison of the results obtained by the best models from each category, i.e., the best from the Crisp Autoencoders, Fuzzy Autoencoders, Crisp Classifiers and Fuzzy Classifiers, plus the best out of the traditional algorithms tested, for comparison.
The models selected were, ordered by increasing mean accuracy:

1. 2 layer Crisp Autoencoder.
2. 3 layer Fuzzy Autoencoder.

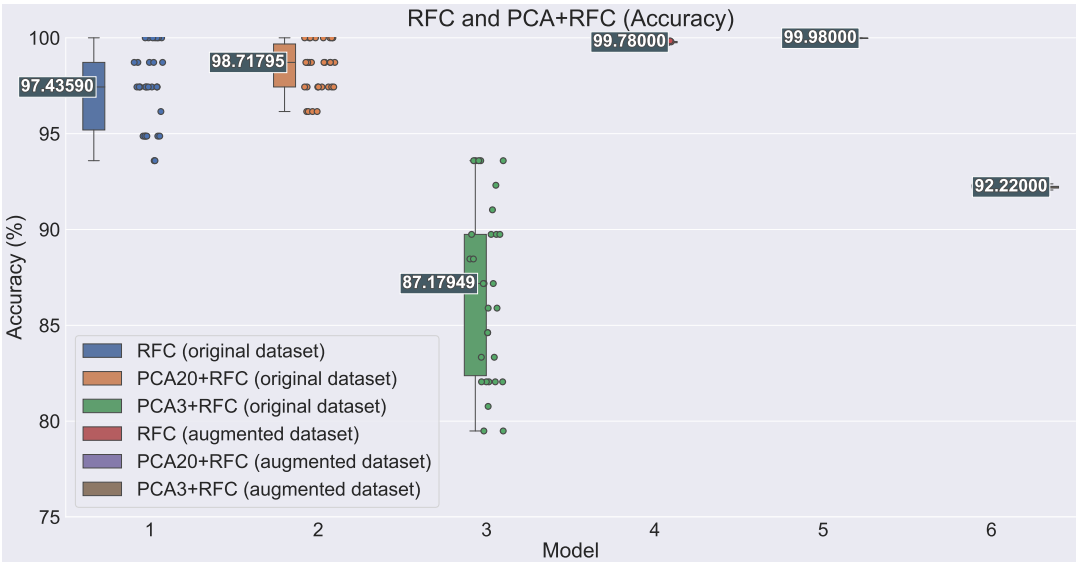


FIGURE 6.12: Accuracy Boxplot for the RFC and PCA+RFC models.

3. 2 layer Crisp Classifier.
4. 4 layer Fuzzy Classifier.
5. PCA+RFC with 20 components, trained on the augmented dataset.

The results of these models are shown in Figure 6.14 and Table 6.11. The models are ordered in the table as above, with labels that can be read as C(lassifier)-A(uto)E(ncoder)-1(layer), or F(uzzy)-CL(assifier)-4(layers).

Practically all the models have minimum accuracy above 98%, and the 2 best models (the 4 layer fuzzy classifier configuration and the PCA+RFC with 20 components) have minimum accuracy above 99%.
The 4 layer fuzzy classifier configuration achieved the best results out of all the models/configurations studied, with a mean accuracy of 99.678% and a minimum accuracy of 99.04%.
These are remarkably good results, but the the PCA+RFC with 20 components managed to overcome them, with a mean accuracy of 99.979% and a minimum accuracy of 99.96%, with both models achieving almost identical maximum values, at almost 100%
Also to note is the very low variability of this model, with a difference of only 0.03% between the minimum and maximum accuracy.

It is noteworthy to point out how the crisp models do best with only 2 layers, while the fuzzy models show their best results with 3 and even 4 layers. Also, the increase in accuracy when comparing the fuzzy and crisp variants of a same model is in both

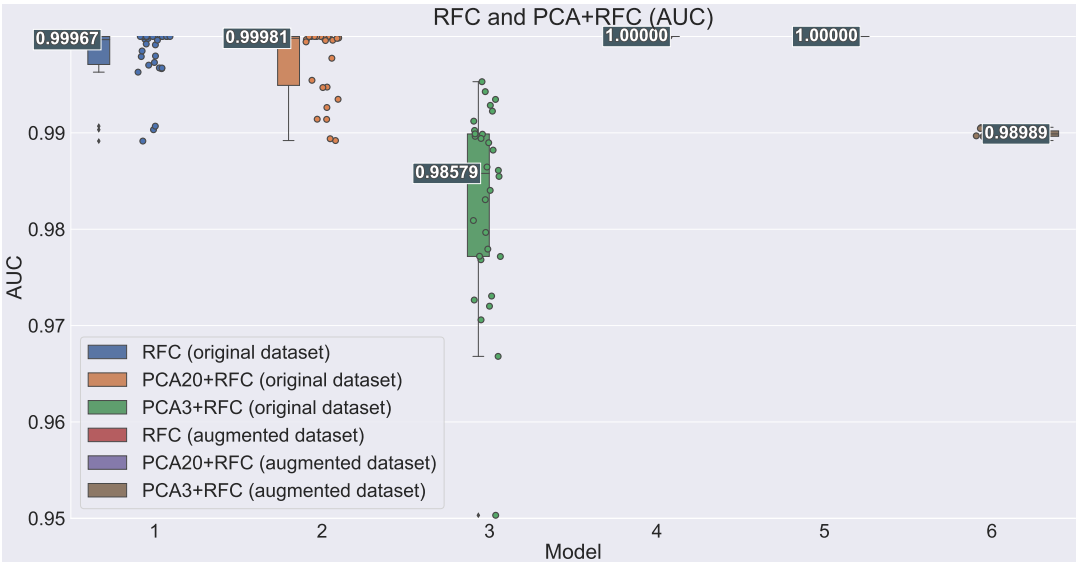


FIGURE 6.13: AUC Boxplot for the RFC and PCA+RFC models.

TABLE 6.11: Accuracy statistics for the best models.

	# of Layers				
	C-AE-1	F-AE-3	C-CL-2	F-CL-4	PCA20+RFC
Max	99.087	99.276	99.92	99.98	99.99
Min	98.026	98.165	97.97	99.04	99.96
Mean	98.382	98.829	99.184	99.678	99.979
Std	2.28E-01	2.87E-01	4.26E-01	2.62E-01	8.46E-03

cases about 0.5% (0.447% for the Autoencoder model and 0.494% for the Classifier model).

The AUC results, seen in Figure 6.15 and Table 6.12, show smaller differences between the first 3 models, particularly the Fuzzy Autoencoder and the Crisp Classifier, and also between the best 2 models. The latter achieved virtually perfect scores, the Fuzzy Classifier ten thousandths away from 1 in mean and minimum scores.
The numbers in this table were rounded to 5 decimal places, so as not to dilute the differences that are obvious in Figure 6.15. Overall, the scores in this category are very, with even the minimum values for all the models thousandths away from 1.

TABLE 6.12: AUC statistics for the best models.

	# of Layers				
	C-AE-1	F-AE-3	C-CL-2	F-CL-4	PCA20+RFC
Max	0.99996	0.99996	0.99999	1	1
Min	0.99642	0.99762	0.99715	0.99951	1
Mean	0.99874	0.99934	0.99936	0.99995	1
Std	9.6E-04	5.9E-04	7.9E-04	9.8E-05	3.45E-08

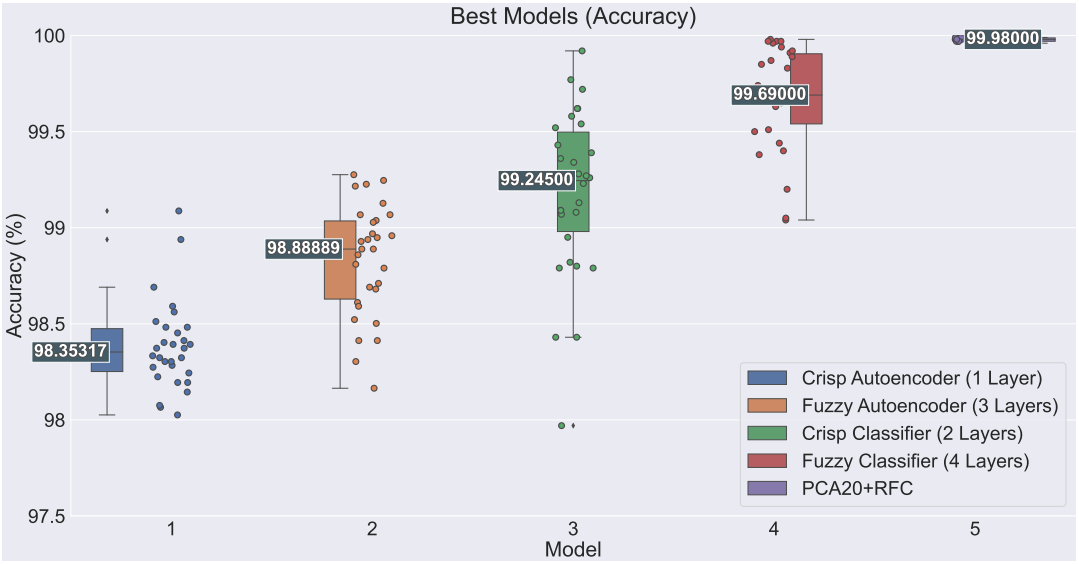


FIGURE 6.14: Accuracy Boxplot for the best models from each category.

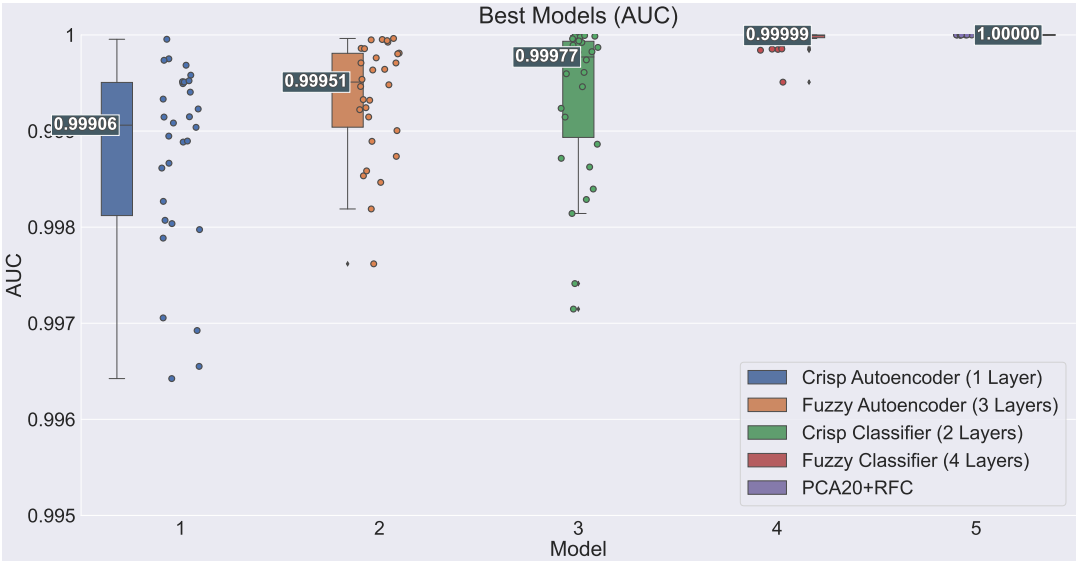


FIGURE 6.15: AUC Boxplot for the best models from each category.

Chapter 7

Conclusion and future directions

This chapter summarizes the work, presenting the conclusions reached and recommendations for further avenues of research with respect to the problem.

7.1 Summary

This work's goal was to investigate the application of Restricted Boltzmann Machine (RBM) based Autoencoders (AE) to classification problems.
Furthermore, 2 types of Restricted Boltzmann Machines were used, the standard version using crisp numbers for the parameters, using code provided by G. Hinton [18], and a version using fuzzy numbers for the parameters, implemented using the standard version as a base and following a proposal laid out in 2 papers published by C.L. Philip Chen et al [5] [6].
These 2 different types of RBM were in turn used to create 2 models of DNN, designated here as Autoencoder configuration and Classifier configuration.
The Autoencoder configuration was used to create a model of DNN that could be used to reduce the dimensionality of a dataset, to which is then applied a classification algorithm, and the Classifier configuration was used to create a model of DNN that could be used to singly classify the data.
The results achieved by these were also compared to traditional (non-DNN) classification algorithms, namely the Random Forest Classifier (RFC) and a combination of Principal Component Analysis (PCA) with RFC, the best of which was the latter.
Ultimately, it can be argued that the model implemented for this work and the PCA/RFC combination are all different ways of implementing the same workflow, where the first step is to reduce the dimensionality of the dataset, and the second step is to classify the data, so a comparison of the 3 is quite valid.

After the models were implemented, a problem immediately arose, namely that the corpus data was not large enough to train a DNN, being composed of 315 samples.

To overcome this, a larger corpus was created by applying a data augmentation technique, proposed by a group at GIDTEC, where subsections of the signal data 0.32768 seconds long are extracted from the original 20 seconds long signals, using a randomly positioned sliding window. This way, thousands of unique samples were created from the original 315 samples.

The resulting samples were then processed in order to scale their amplitude to 1.
The samples were split into 2 disjoint datasets, one for training and one for testing, in an attempt to avoid cross-contamination.
These were used to train and tests the DNN models and also the traditional algorithms. The results were then tabulated and compared.

7.2 Conclusions

The conclusions that can be drawn from the results of the experiments are the following:

• The fuzzy models show better performance than the crisp models.
This confirms the hypothesis put forth in the papers by C.L. Philip Chen et al [5] [6].
The magnitude of the improvement compared to the standard version is not the same, as in the aforementioned papers an improvement of up to 1.44% was reported, while in this work the observed improvement was just under 0.5%.
This can be caused by several factors, chiefly the fact that the the authors used a different dataset, and the results are not directly comparable.
There could also be defects in the implementation, improper tuning of hyperparameters, or just the experimental process.
The fact that the fuzzy models show better performance than the crisp models is a good indication that the implementation is correct, and that this improvement is due to the use of fuzzy numbers for the hyperparameters, but the difference not being as large as in the aforementioned papers could mean that the dataset used in this work is not as suitable for the use of fuzzy numbers.
Other observations are that the fuzzy models are more computationally expensive than the crisp models, and that they appear to be more consistent when used in deeper configurations, as in both models the crisp models achieved their best results in 2 layer configurations, and the fuzzy model in 3 and 4 layer configurations.

• The Classifier configuration in general shows better performance than the Autoencoder configuration, with just 2 exceptions (the 4 and 5 layer crisp Classifier).
The best Classifier configuration (fuzzy 4 layer Classifier) achieved a mean accuracy of 99.678%, minimum accuracy of 99.04% and maximum accuracy of 99.98%,

while the best Autoencoder configuration (fuzzy 3 layer Autoencoder) achieved a mean accuracy of 98.829%, minimum accuracy of 98.165% and maximum accuracy of 99.276%.
This could be explained by the different training process, where while the Autoencoder is trained to reconstruct the input, and after training, the encoder half is then used to reduce the dimensionality of the dataset to be classified, the Classifier is trained from the start to classify the input, fully integrated with the softmax layer.

• Assuming the data augmentation process is sound, the results show that the DNN models are able to achieve very good classification performance.
Except for 2 configurations (the 4 and 5 layer crisp Classifier), all the others, in both crisp and fuzzy variants, and in their various configurations of layers achieved minimum accuracy values above 97%, with some configurations achieving values above 99%.
The best overall configuration came very close to the best algorithm tested, the PCA+RFC with 20 components combination, and in fact, a perfect score, with a mean and maximum accuracy values of 99.678%, and 99.98% respectively.

• The best results are achieved by the PCA/RFC combination, followed by the fuzzy Classifier configuration. This means that for this particular problem, and perhaps for any dataset made up of tabular numerical data, the DNN models are not the best choice.
This raises the question of when to use DNN models, and when to use traditional algorithms.
For all their sucess, there are plainly some problems that DNN models are not the best choice for.
In this instance, the reason for this is not mainly one of performance, because the DNN models are able to achieve very good results, but mostly one of cost.
A simpler, faster, easier to use algorithm achieved better results at a fraction of the cost, owing to the nature of the data.
So before investing into the use of DNN models, it is important to consider the problem (or dataset) at hand, in terms of the type and volume of data available, as well as time, budget, computational power available and other constraints.
In some cases DNN models are a prime choice:

- Financial applications, or any other where every possible improvement in performance is valuable and DNN can provide it.
- When there is a lack of domain understanding but enough data to train a DNN.
- Some applications where DNN have proven themselves superior to traditional

algorithms (Natural Language Processing, Computer Vision, Information Retrieval and a few others).

One would do well to first test off the shelf algorithms, and only if they fail to achieve the desired results, then consider the use of DNN models.
For example, there exists a Python library called lazypredict [25], which encapsulates over 40 off the shelf algorithms, and with fewer than 10 lines of codes, can be used to test all of them on a dataset and compare their results.
Another area where traditional algorithms have an advantage over DNN is whenever explanability is required, as DNN models are notoriously difficult to explain. Using traditional algorithms, one can use a SHAP library (SHapley Additive exPlanations), to easily plot of summary of features ranked by importance which can be used to explain the results of a model, and this is not possible with DNN models.

7.3 Future work

The work presented in this paper is a proof of concept, and there some avenues to explore in order to improve it and make it a viable solution.
The first is to test the models on a other datasets, and see if the results are consistent.
Then there are possible improvements to the models themselves, such as:

- Using other activation functions, such as the ReLU function, which is thought to be more efficient and achieve better converge performance [26] than the sigmoid function.
- Testing the use of an adaptive optimizer, the best of which for most cases being the Adam optimizer. These do not guarantee better minima than the standard gradient descent, but do not require learning rate tuning and should achieve faster convergence.
- Implementing this work on a library that makes use of a GPU, such as Tensorflow or PyTorch, which should greatly reduce the training time.
- Applying regularization techniques, such as L1, L2 or dropout, to prevent overfitting and improve generalization performance.

Appendix A

Code Used

A.1 Github repository and brief explanation of code

Matlab functions and code for the Master Thesis of Angelo Dias:
github.com/amsdias/msc_thesis/

Restricted Boltzmann Machine Based Autoencoders for the Classification of Faults in Rotational Mechanical Systems . Universidade do Algarve, 2022.

Navigation

The RBM code resides in the src/matlab folder. The script can be started by running the start.m script, which will ask for inputs to select the model (Autoencoder or Classifier), variant (crisp or fuzzy), and number of layer (1-5).

The code will execute 30 runs by default. This number can be changed in the scripts for each model: deepauto_v_n.m and deepclassify_v_n.m.

At the end the data will be saved to either the saved_features or saved_variables folder. The Classifier version will output the classification results, and the Autoencoder version will output the weights of the trained models, which will then be used to generate new reduced dimensionality datasets for use with an external classifier.

The src/python folder contain 3 subfolders:

1. the data_treatment subfolder which contains the scripts used to perform data augmentation and feature generation from the original raw data, and the script used to scale the output of the first script to values between 0 and 1.

2. the rfc folder contains scripts used to classify the output of the autoencoder model, and also RFC and PCA+RFC scripts used directly on the work dataset, to obtain results to be used for comparison.

3. the statistics folder, containing scripts used to extract metrics and figures for analysis of the results.

References

[1] Niabot, CC BY-SA 3.0, via Wikimedia Commons, *Ball bearings*, [Online; accessed July 20, 2019], 2010. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/4/4a/Ball-bearing-numbered.png.

[2] D. R. Snyder. "The meaning of bearing life." MachineDesign, Ed. (2007-04), [Online]. Available: https://www.machinedesign.com/mechanical-motion-systems/bearings/article/21831664/the-meaning-of-bearing-life (visited on 2022-09-26).

[3] R. A. Tarle, N. K. Kharate, and S. P. Mogal, "Vibration analysis of ball bearing," *International Journal of Science and Research (IJSR)*, vol. 4, no. 5, pp. 2655–2665, 2015-05, ISSN: 2319-7064. [Online]. Available: https://ijsr.net/archive/v4i5/SUB154835.pdf.

[4] S. V. Shelke, A. G. Thakur, and Y. S. Pathare, "Condition monitoring of ball bearing using vibration analysis and feature extraction," *International Research Journal of Engineering and Technology (IRJET)*, vol. 3, no. 2, pp. 361–365, 2016-02, ISSN: 2395 -0056.

[5] S. Feng and C. L. P. Chen, "A fuzzy restricted boltzmann machine: Novel learning algorithms based on the crisp possibilistic mean value of fuzzy numbers," *IEEE Transactions on Fuzzy Systems*, vol. 26, no. 1, pp. 117–130, 2018-02. [Online]. Available: https://ieeexplore.ieee.org/document/7782438.

[6] C. L. P. Chen, C. Zhang, L. Chen, and M. Gan, "Fuzzy restricted boltzmann machine for the
enhancement of deep learning," *IEEE Transactions on Fuzzy Systems*, vol. 23, no. 6, pp. 2163–2173, 2015-12. [Online]. Available: https://ieeexplore.ieee.org/document/7047917.

[7] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006, ISSN: 0036-8075. DOI: 10.1126/science.1127647. [Online]. Available: https://science.sciencemag.org/content/313/5786/504.

[8] G. E. Hinton and T. Sejnowski, "Optimal perceptual inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1983-01, pp. 448–453. [Online]. Available: https://papers.cnl.salk.edu/PDFs/Optimal%20Perceptual%20Inference%201983-646.pdf.

[9] P. Smolensky, "Information processing in dynamical systems: Foundations of harmony theory.," in *Parallel Distributed Processing: explorations in the microstructure of cognition.* D. Rumelhart, J. L. McClelland, and the PDP Research Group, Eds., Cambridge: MIT Press, 1986, pp. 194–281.

[10] G. E. Hinton, "Training products of experts by minimizing contrastive divergence," *Neural Computation*, vol. 14, no. 8, pp. 1771–1800, 2002-08. [Online]. Available: https://doi.org/10.1162/089976602760128018.

[11] G. E. Hinton, S. Osindero, and Y. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, 2006-07, ISSN: 0899-7667. DOI: 10.1162/neco.2006.18.7.1527. [Online]. Available: http://dx.doi.org/10.1162/neco.2006.18.7.1527.

[12] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015-05. DOI: 10.1038/nature14539.

[13] M. Á. Carreira-Perpiñán and G. E. Hinton, "On contrastive divergence learning," in *AISTATS*, 2005.

[14] A. Ng. "Neural networks: Representation." University of California, Los Angeles, Ed. (2012), [Online]. Available: http://helper.ipam.ucla.edu/publications/gss2012/gss2012_10740.pdf (visited on 2021-03-17).

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

[16] Tin Kam Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, 1995-08, 278–282 vol.1. DOI: 10.1109/ICDAR.1995.598994.

[17] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001-10, ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. [Online]. Available: https://doi.org/10.1023/A:1010933404324.

[18] Geoffrey Hinton, Ed. "Training a deep autoencoder or a classifier on mnist digits." (2006), [Online]. Available: http://www.cs.toronto.edu/~hinton/MatlabForSciencePaper.html (visited on 2019-07-17).

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

[20] scikit-learn developers, Ed. "Scikit-learn: Machine learning in python - scikit-learn 0.21.3 documentation." (2019), [Online]. Available: https://scikit-learn.org/stable/# (visited on 2019-08-19).

[21] Numpy Web Team, Ed. "Numpy." (2019), [Online]. Available: https://numpy.org/ (visited on 2019-08-19).

[22] The pandas development team, Ed. "Pandas - python data analysis library." (2019), [Online]. Available: https://pandas.pydata.org/ (visited on 2019-08-19).

[23] Michael L. Waskom, Ed. "Seaborn - statistical data visualization." (2019), [Online]. Available: https://seaborn.pydata.org/index.html (visited on 2019-08-19).

[24] The Matplotlib development team, Ed. "Matplotlib - python plotting." (2019), [Online]. Available: https://matplotlib.org/stable/index.html (visited on 2019-08-19).

[25] Shankar Rao Pandala, Ed. "Lazypredict - pypi." (2022), [Online]. Available: https://pypi.org/project/lazypredict/ (visited on 2022-09-19).

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.