NOVA IMS Information Management School

MEGI

Mestrado em Estatística e Gestão de Informação

Master Program in Statistics and Information Management

Hyperparameters Optimization on Neural Networks for Bond Trading

Francisco Urbano Fonseca

Trabalho de Projeto apresentado como requisito parcial para obtenção do grau de Mestre em Estatística e Gestão de Informação

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação

Universidade Nova de Lisboa

Universidade Nova de Lisboa
Information Management School

Hyperparameters Optimization on Neural Networks for Bond Trading

Francisco Fonseca

Submitted in part fulfilment of the requirements for the degree of
Master of Science in Statistics and Information Management
of the Universidade Nova de Lisboa. 2018

Abstract

Artificial Neural Networks have been recently spotlighted as *de facto* tools used for classification. Their ability to deal with complex decision boundaries makes them potentially suitable to work on trading within financial markets, namely on Bonds. Such classifier faces high flexibility on its parameters in parallel with great modularity of its techniques, arising thus the need to efficiently optimize its hyperparameters. To determine the most efficient search method to optimize almost the majority of the Neural Networks hyperparameters, we have compared the results obtained by the manual, evolutionary (genetic algorithm) and random search methods. The search methods compete on several metrics from which we aim to estimate the generalization capability, i.e. the capacity to correctly predict on unseen data. We have found the manual method to present better generalization results than the remaining automatic methods. Also, no benefit was found on the direction provided by the genetic search method when compared to the purely random. Such results demonstrate the importance of human oversight during the hyperparameters optimization and weight training phases, capable of analyzing in parallel multiple metrics and data visualization techniques, a process critical to avoid suboptimal solutions when navigating complex hyperspaces.

Acknowledgements

I am profoundly grateful to everyone who have inspired or helped me throughout the development of this work, either directly or indirectly. This achievement would not have been possible without them.

To my supervisor Ivo Gonçalves and co-supervisor Mauro Castelli for all the guidance.

To my parents and family for the constant encouragement and optimism.

To Inês for all the joy and never-ending support.

To Guilherme and Paulo for the friendship.

To Diogo, Paulo, Berto, Di and Ivo for the companionship and camaraderie.

To João, Ana, Rafa, Vera and Vanda for the motivation and coaching.

To Sérgio for introducing me to the world of Machine Learning and for all the guidance and discussions which influenced deeply the course of this work.

To Carina, Tiago and João for the cheerfulness and good spirit.

To Ricardo for believing in me and always teaching me how to be a better person.

To the Python community for all the open-minded distribution of contents.

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**

1.1 Motivation and Objectives 1

1.2 Contributions and Importance of the Topic 1

1.3 Methodology 2

1.4 Structure 3

1.5 Research Questions 3

2 Neural Networks: An Introduction **4**

2.1 Architecture 5

2.1.1 Representation 5

2.1.2 Activation Functions 5

2.2 Training 7

2.2.1 Weight Initialization 7

2.2.2 Loss Function 8

2.2.3 Optimization Algorithm 9

2.2.4 Regularization 12

3 State of the Art **17**

3.1 Dropout 17

3.2 Batch Normalization 18

3.3 Self Normalizing Neural Networks 19

3.4 Software and Tools 19

4 Bonds **20**

4.1 Bond Pricing and Life cycle 21

4.2 Categorical Characteristics 24

5 The Database **26**

5.1 Data Life Cycle 26

5.2 Collection, Selection and Cleaning 27

5.3 Data Preprocessing 29

5.3.1 Class-Imbalanced Data 29

5.3.2 Categorical Data 30

5.3.3 Scaling 31

5.4 Database Creation: The Process 33

5.4.1 Early Steps 33

5.4.2 Inputs 36

5.4.3 Overall Frame 39

6 The Model **41**

6.1 Problem Definition 41

6.2 The Features and Feature Engineering 41

6.3 Generalization, Over-fitting and the Learning Curve 43

6.3.1 Generalization and Learning curve 43

6.3.2 Cross-validation 45

6.3.3 Statistical Analysis 46

6.3.4 Metrics 50

7 Deployment **53**

7.1 Dimensionality Reduction 53

7.2 Hyperparameters Optimization Strategies 56

7.2.1 Introduction 56

7.2.2 Comparison and Preliminary Results 58

7.2.3 Exploding Combinations 60

8 Results **62**

9 Conclusions **70**

9.1 Research Answers 70

9.2 Summary of Thesis Achievements 72

9.3 Applications 73

9.4 Future Work 73

A Appendix **75**

A.1 Mathematics and Statistics 75

A.1.1 Bernoulli Distribution 75

A.1.2 Hyperbolic Secant 76

A.1.3 Sample vs Population Standard Deviations 76

A.2 Metrics Surface Plots 76

List of Tables

2.1 Comprehensive list of activation functions considered in the hyperparameters optimization search. 15

2.2 The Cost Functions considered. 16

4.1 Fair Value with constant interest rates. 21

4.2 Fair Value with parallel increase in interest rates in $t = 1$. 22

4.3 Positively-sloped Interest Rate Curve. 23

4.4 5-year, 4%-coupon bond present value with increasing interest rates. 23

5.1 Ordinal, *1 out of N* and *1 out of N-1* categorical encoding techniques. 30

5.2 Examples of bond prices for target definition. 38

5.3 Optimal decisions for the trading scenario. 38

5.4 Creation of the target by differencing, applying a target function and temporal shift. . 38

6.1 $2x2$ confusion matrix. 51

6.2 Comprehensive list of metrics from confusion matrix. 52

7.1 Mean Squared Error of reconstructing the multiple datasets with an Autoencoder. . . 55

7.2 List of optimized hyperparameters with categories. 61

List of Figures

2.1 A simple fully connected neural network with one hidden layer. 6

2.2 Neural Network Backpropagation Training Life Cycle. 11

4.1 PV of Bonds A and B throughout their life cycle. 22

4.2 Present Value with positively-sloped interest rate curve. 24

5.1 Overall dataset in matrix form. 35

5.2 Graphic interpretation of sampling the dataset. 35

5.3 Graphic interpretation of passing a dataset sample throughout the neural network. . . 36

6.1 Extra Trees, Linear SVM and Perceptron features weights/ importances. 47

6.2 Pearson's correlation of the variables. 48

6.3 A Recursive Feature Elimination study using Logistic Regression. 49

6.4 2D and 3D PCA per class label. 50

7.1 Representation of an autoencoder. 54

7.2 Learning Curvers of the Autoencoder training. 56

8.1 Accuracy, LogLoss and Sensitivity box plots. 64

8.2 Specificity, False Positive Rate and Precision box plots. 65

8.3 AUC, Youden's Index and Discriminant Power box plots. 66

8.4 F-Score and Profit box plots. 67

8.5 Testing Period Financial Gain by Search Method. 67

8.6 Positive and Negative Likelihoods box plots. 68

8.7 True Positive vs False Positive Results per Search Method. 69

A.1 Contour Plots of the Discriminant Power metric. 77

A.2 Contour Plots of the F-Measure metric. 77

A.3 Contour Plots of the Youden's Index metric. 77

Chapter 1

Introduction

1.1 Motivation and Objectives

The ultimate goal of this work is to create an effective and practical trading tool for an Asset Management company. The search for alternatives to the traditional investment decision methods, e.g. trading based on newspaper articles or on classical economic theory models, gives machine learning techniques an important spotlight for their consolidation as *de facto* tools in the industry. Accordingly, by incorporating state-of-the-art neural networks' data processing capabilities, we aim to implement the algorithm as the ultimate investment decision maker.
Considering the generalized recognition of neural networks as classifier tools, we will take advantage of their modular design of multiple tuning (hyper)parameters and study their combinatorial optimization.

1.2 Contributions and Importance of the Topic

The overall contribution of this work is dual within the fields of Economics (in particular, Financial Markets) and Data Science (focusing on Machine Learning).
Within the Data Science field, the work aims to compare the Evolutionary, Random and Manual hyperparameters optimization search procedures through their ease-of-use, efficiency and best solution encountered. The applicability on Neural Networks is highly relevant due to their vastitude of optimization parameters, along with the techniques that keep appearing within the field which are modular-like and easy to implement. The work hereby produced aims to provide a relevant baseline

for both academia and industry on implementing neural networks and their optimization. Their construction as modular building blocks is, at the same time, a advantage and a curse — such will be the main topic of our work. The flexibility will be proven to be a positive characteristic up to a certain point where it starts to become a burden. Our work aims to provide a complete description on how to implement a machine learning approach to problem-solving, as well as to describe and test the main framework on implementing such techniques.

With what concerns the Financial Markets, we aim to study the possibility and the consequent capacity of predictability of a Bond Trading Algorithm. Bonds are financial securities that usually have low relevance on centralized trading markets, being traded more on Over-The-Counter, and usually higher Bid-Ask spreads [1] when compared to Stocks. This creates a degree of difficulty which we intend to overcome and, possibly, counteract. We will study the ability to predict the future movement on Bonds' pricing which, if found to be successful, may provide market deepening and increase liquidity, by increasing portfolio rotation (when comparing to a passive strategy of buy-and-hold to receive the coupons periodically).

1.3 Methodology

The work hereby produced aims to achieve the following properties: simplicity, reproducibility, ease-of-use and *pipeline-ability*.

Simplicity aims to gain intuitive control over the processes of the learning scheme. The logic behind it is that if we keep the algorithm simple and clear enough for the human mind, we can counteract adversities easily and efficiently, because we will be able to identify the source of the problem.

The practical approach requires *ease-of-use* of the overall algorithm, such that all the necessary steps for the deployment (creation, debugging and implementation) can be shared within the company. This often relies on having the script well documented and simple.

In harmony with the script being easy to use, it must also be *reproducible*. All the necessary steps to its successful implementation (from collecting the data to making the investment decisions) must be stress-tested and robust enough for maintaining stability when facing changes.

The *Pipeline-ability* is a characteristic of the coding itself, which forces us to, when confronted with multiple solutions for the same problem, favor simpler solutions which can be pipelined, as a combination of multiple blocks, conjugating multiple modules.

[1] The Bid-Ask spread can be measure by the difference between the Ask and Bid prices of the security.

1.4 Structure

We first present individually the main topics of our work: **Neural Networks** (Introduction and State of the Art) and **Bonds**, on chapters 2, 3 and 4, respectively, to provide the non-specialist reader sufficient knowledge to comprehend the remainder of the text.
The practical development is then divided into three main topics, by their order of appearance on the work: **The Database** (5), where we explain the process and specificities of creating the database; **The Model** (6), which contains the core topics on developing the algorithm, and **Deployment** (7) where we present the implementation of the work.
Finally, the discussion the **Results** (8) and the **Conclusions** (9) present the main findings and future work.

1.5 Research Questions

The work is constructed to provide answers to the following questions:

H1. Are the Evolutionary and Random search procedures efficient, in their time-cost of application?

H2. Does the direction provided by the Evolutionary Search procedure provides an advantage over a pure Random Search procedure?

We believe such enquiries may provide relevant directions for future practitioners on how to address the problem of optimizing a large number of hyperparameters of a learning algorithm.

Chapter 2

Neural Networks: An Introduction

This section intends to provide a presentation and brief review of the general concepts that surround the field of Artificial Neural Networks (ANNs). This is particularly aimed to allow non-experts on the field to comprehend the work done below.

Neuronal Networks are computational means which mimic the mechanisms and behaviour of the human brain (P. Fonseca, 1995), in particular, the ability to deal with complex, non-linear pattern recognition tasks (Fitkov-Norris et al., 2012). Similar to a human brain, the ANN are parallel processing structures (Mojarad et al., 2011) of densely connected multiple neurons receiving, processing and outputting information. In feedforward ANNs, which will be our focus, all connections are directed from inputs towards the outputs (Mojarad et al., 2011), contrasting with recurrent networks in which connections amongst nodes are allowed to retain information about past inputs (Pascanu et al., 2013).
Neural networks are well established tools for classification (Fitkov-Norris et al., 2012; Janocha and Czarnecki, 2017) with good performance on diverse fields, from which we include medicine (Kaguara et al., 2014; Lu et al., 2001; Mojarad et al., 2011), computer-aided detection and design (Zur et al., 2009), hidrology (Piotrowski and Napiorkowski, 2013). Along with the capacity to capture complex and non-linear interactions, they can also take into account the inter-relations between variables (Mojarad et al., 2011). Their success is also a result of the development of simple, broadly applicable techniques (Neelakantan et al., 2015), such as dropout (see section 3.1), innovative activations functions (see 2.1.2) or weight initialization (see 2.2.1).

2.1 Architecture

The architecture of an ANN, i.e. the pattern between the neurons (Ng, 2011), which includes its connectivity and the activation functions of each node, has a great impact on a network's information processing capabilities (Stanley and Miikkulainen, 2002; Yao, 1993; Yao, 1999), as it defines the number of parameters to be optimized (Piotrowski and Napiorkowski, 2013).

2.1.1 Representation

Regarding the way we can represent our network's architecture, there is no dominant method that outperforms the remaining and the choice of the representation relies primarily on the application (Yao, 1993). Therefore, for practical reasons we are using an indirect encoding of the ANN architecture: only specifying in the chromosome the most important parameters (such as the number of layers or the number of nodes at each layer), instead of specifying all the details, i.e. detailing every node and its connections within the architecture. This allows a more compact representation of the network's connectivity (Stanley and Miikkulainen, 2002; Yao, 1993), enabling us to search a larger hyperspace of parameters, to the detriment of fine-tuning a smaller architecture.
To illustrate the construction of an ANN connectivity by specifying indirectly their components, we have drawn an example of a 3-layer network — with 1 input layer (3 nodes), 1 hidden layer (5 nodes) and 1 output layer (1 node) — in Figure 2.1. Considering all nodes are connected throughout the layers, we define it as a fully connected neural network. The hidden layers enable the neural network to extract high order statistics (Mojarad et al., 2011).

The architectural choice of the ANN will be made by the hyperparameter optimization search process. Nonetheless, it is relevant to mention that generalization is more constricted due to small networks than large ones (Caruana et al., 2000), so bigger ANNs should be naturally favored.

2.1.2 Activation Functions

Activation functions transform the activation state of each neuron to an output. They introduce non-linearity (Njikam and Zhao, 2016) which gives ANNs non-linear capabilities (LeCun et al., 1998) and can significantly impact the ANN perfomance (Xu et al., 2016; Yao, 1993).
It is thus relevant to note activation functions face multiple problems when dealing with backprop-

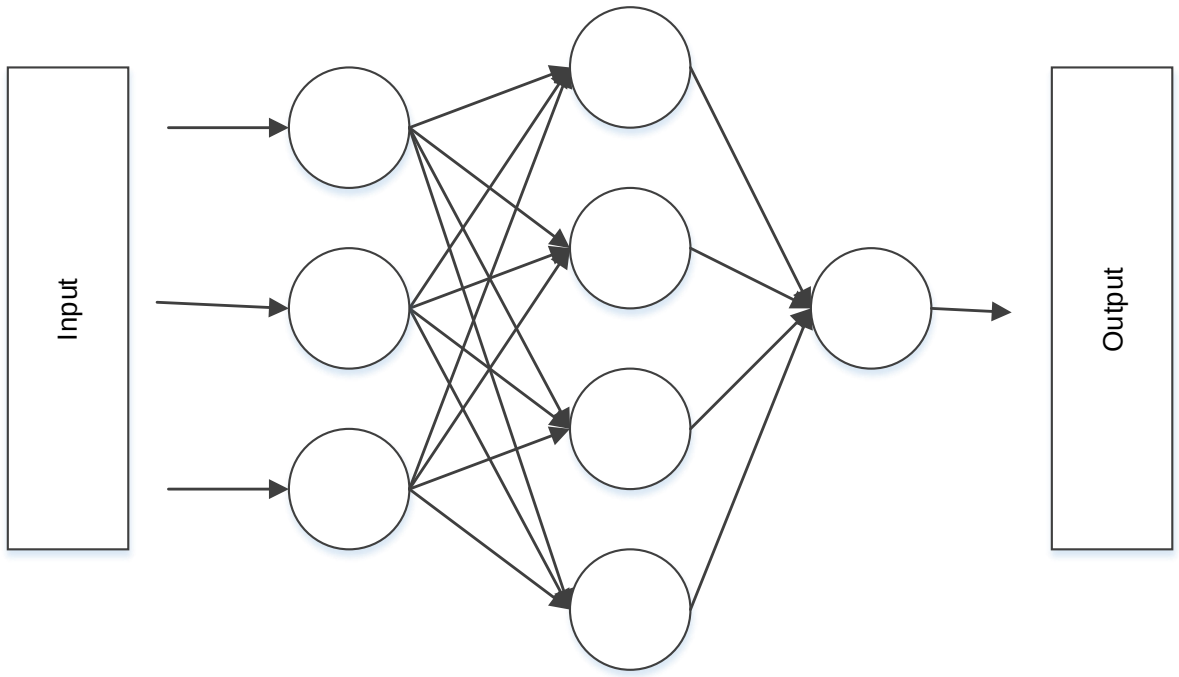

Figure 2.1: A simple fully connected neural network with one hidden layer.

agation: overly linear units do not compute interesting results (Glorot and Bengio, 2010); excessive saturation[1] can cause the gradients to vanish or explode (Xu et al., 2016); and activation functions not symmetric around 0 should be avoided when initializing from small random weights, because they yield poor learning dynamics (Glorot and Bengio, 2010), due to their proximity to the null. On the positive side, symmetric functions are believed to yield faster convergence (LeCun, 1989).

The poor performance of the traditional activation functions (Njikam and Zhao, 2016) requests an investigation of new functions and other potentiating techniques, such as the weights initialization (as discussed on section 2.2).

Therefore, for the choice of the activation functions we tested commonly used transformations (natively present in the Keras source code (Chollet et al., 2015)) along with some other functions from the review of literature. A comprehensive list is detailed in table Table 2.1 on page 15. For the last activation we have to choose only the transformations that match our expected output — if we are classifying a scenario on a binary output (0 or 1) and we want a continuous value that approximates with some confidence degree of such scenario, it is natural that the last activation functions outputs percentage

[1] We talk about *function saturation* when the argument is too positive or negative that causes the function to become very flat and insensitive to small changes (Goodfellow, Bengio, et al., 2016). Using the logistic sigmoid function as an example, a change in the argument near the asymptotes will have less effect than changes near the origin $x = 0$.

values between $[0, 1]$, such as the logistic sigmoid function.

A complimentary Python code for activation functions not usually found in the usual neural network libraries can be found in (F. Fonseca, 2017a).

2.2 Training

2.2.1 Weight Initialization

The starting values of the weights can have a significant impact on the training process (LeCun et al., 1998). They should be chosen in such a way that (LeCun et al., 1998):

- The activation function is activated on its linear region,
- The standard deviation of the inputs is close to 1.

We will consider multiple initialization methods, which include the *LeCun Normal*, *LeCun Uniform*, *Glorot Uniform* and *Glorot Normal*.

The **LeCun Normal Initializer**, presented in (LeCun et al., 1998), considers the weights being randomly drawn from a zero mean distribution, with the standard deviation given by:

$$\sigma_w = m^{-\frac{1}{2}} \tag{2.1}$$

where m is the number of inputs to the unit.

The **LeCun Uniform Initializer** is presented in (LeCun, 1989). In this case, the weights are uniformly initialized from:

$$W \sim U\Big[-\frac{2.4}{F_i}, \frac{2.4}{F_i}\Big], \tag{2.2}$$

where F_i is the number of inputs to the connection.

The **Glorot Uniform** is presented in (Glorot and Bengio, 2010). The weights at each layer are initialized from:

$$W_{ij} \sim U\Big[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\Big], \tag{2.3}$$

where n is the size of the previous layer.

Glorot and Bengio also present a **Glorot Normalized Initialization**, with the premise of maintaining the activation and gradient variances across the network layers. To achieve this:

$$W \sim U\Big[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\Big] \tag{2.4}$$

Considering having different magnitudes in the gradients may slower the training process (Glorot and Bengio, 2010), the normalized initializations which counteract that problem have theoretical advantages.

We will further study the implementation of the **Orthogonal** (Saxe et al., 2013) and the **He Normal** and **He Uniform** (He et al., 2015) initializers, natively present in the Keras (Chollet et al., 2015) library.

2.2.2 Loss Function

During the training process, the neural network is given some feedback on its performance to orient the training scheme to the best possible scenario. By comparing the network's output with the desired output, such cost function (usually denoted $J(\theta)$ where θ represent the parameters) is minimized with respect of the network's parameters (Mojarad et al., 2011) by the optimization algorithm (see section 2.2.3). A brief overview of the considered loss functions is detailed below.

Some works found the **conditional log-likelihood**, or Cross Entropy (CE), cost function to work much better for classification than the mean squared error (MSE), presenting less plateaus in the training criterion (Glorot and Bengio, 2010) and offering faster convergence (Golik et al., 2013).

The works of (Janocha and Czarnecki, 2017) found the log loss to be a poor choice for the loss function, with a good performance of the **squared hinge loss** or the surprisingly **mean squared error**.

In (Golik et al., 2013), the CE outperforms the MSE. This is often caused by the vanish gradients of using MSE with the softmax activation function and with random weight initialization. Nonetheless, with a good initialization the MSE criterion seems to consistently improve the CE-based solution.

The non-dominance of a loss function is studied by considering them all. By unifying the scoring function in the evolutionary and random search processes, we can input the loss function as a tunable hyperparameter of the network. This implies having two loss functions during the training stage: one for updating the network's weights and another to measure the network's capability of prediction. Otherwise, the different natural ranges of each loss function would block the comparison between

networks with different functions.

The considered loss functions are detailed in table 2.2 on page 16, as considered in (Golik et al., 2013; Janocha and Czarnecki, 2017), where $\mathbf{y}$ is the true label, σ is the probability estimate and $\mathbf{o}$ is the output of the last layer.

A faster convergence is expected on combining the architecture (particularly the output activation function) with the loss function (Golik et al., 2013; Janocha and Czarnecki, 2017). While this is conceivable in a manual hyperparameters optimization setting, in a random or evolutionary search it is not. Nonetheless, theoretically they will tend to this 'natural pairing' if it indeed induces a better performance.

In a practical tip, it is important to provide the true labels (i.e. the classes) for the scoring function, specially in a multi-class problem with imbalanced data. If we do not feed such information, the scoring function must assume that all of the existing possibilities are within the range it sees in the predicted y vector, which unique values can be less than in the true y vector, causing the cost function to falsely report the true performance. This can be easily demonstrated within a cross-validation scenario, a topic further detailed in 6.3.2. Suppose that of 4 true labels unevenly balanced across the dataset, a cross-validation fold only contained 3. This has the impact of negatively bias loss functions, such as the log loss.

2.2.3 Optimization Algorithm

An important part of training a Neural Network is optimizing the weights between the neurons. The two main considered methods are evolutionary training and backpropagation.

According to (Yao, 1993), evolutionary training is usually slower than gradient descent techniques, more computation intensive and more indicated to work with feedback connections or deep feedforward ANNs. Nonetheless, due to the recent works of (Ioffe and Szegey, 2015; Klambauer et al., 2017) on self-normalizing properties of the networks, the disadvantages brought by back-propagating throughout deep networks are reduced, which allows us to work with a faster alternative to evolutionary training. The **Backpropagation** refers to the errors of the training phase which are backpropagated to alter the network parameters (i.e. the connections' weights) if the predicted output is found to be deviant from the desired, after the inputs are forward-passed throughout the network.

Despite our choice for the learning scheme, it is important to bear in mind that back propagation is prone to getting trapped in local minima, being very inefficient to search for a global optimal (Yao, 1993).

Gradient descent aims to minimize the cost function $J(\theta)$ by updating the parameters according to the gradient $\nabla_\theta J(\theta)$ of the objective function w.r.t. the parameters (Ruder, 2016). Intuitively, we can imagine a gradient descent algorithm as a guide indicating within the cost function hyperplane the direction towards the steepest descent. The learning rate η — the length of the steps we take at each iteration — is somewhat difficulty to tune (Ruder, 2016), so we are expecting the adaptive-learning methods, which automatically regulate their learning rate, to perform better on the unaware Automatic search processes because we will not consider the learning rate as an hyperparameter.
We will consider the gradient descent methods of Stochastic Gradient Descent (SGD), Adagrad, Adadelta, RMSProp, Adam, AdaMax and Nadam. A simple overview of these optimizers can be found in (Ruder, 2016).

We can divide the frequency at which the parameters (i.e. the weights) are adjusted into: batch, stochastic and mini-batch. The first requires that each update is performed for the whole dataset, i.e. a whole batch, which can be quite cumbersome. Stochastic (SGD) is quite the opposite, by updating the parameters for each training sample. By analyzing each sample individually, it will naturally compute quite variant updates that cause the cost function to fluctuate heavily (Ruder, 2016). The in-between solution is the mini-batch gradient descent, where the parameters' updates are computed for mini-batches of n examples, a parameter we manually define. Due to the advantages of efficiency and stability, we will use this last approach of **mini-batch gradient descent**.

The training is thus performed for each n examples of a mini-batch for a specified number of epochs. An epoch represents a forward and consequent backward pass on all the training examples. At each epoch we perform $\frac{len(X)}{n}$ updates, where $len(X)$ is the number of total samples in the dataset and n is the size of the mini-batch.

The overall process is thus:

1. Select the optimization algorithm;
2. If needed, define the learning rate η;
3. Define the mini-batch size n and the number of epochs;

4. Initialize the weights;

5. Forward pass and compute the errors;

6. Update the parameters ($\frac{len(X)}{n} \times \#epochs$) times or until early stopped.

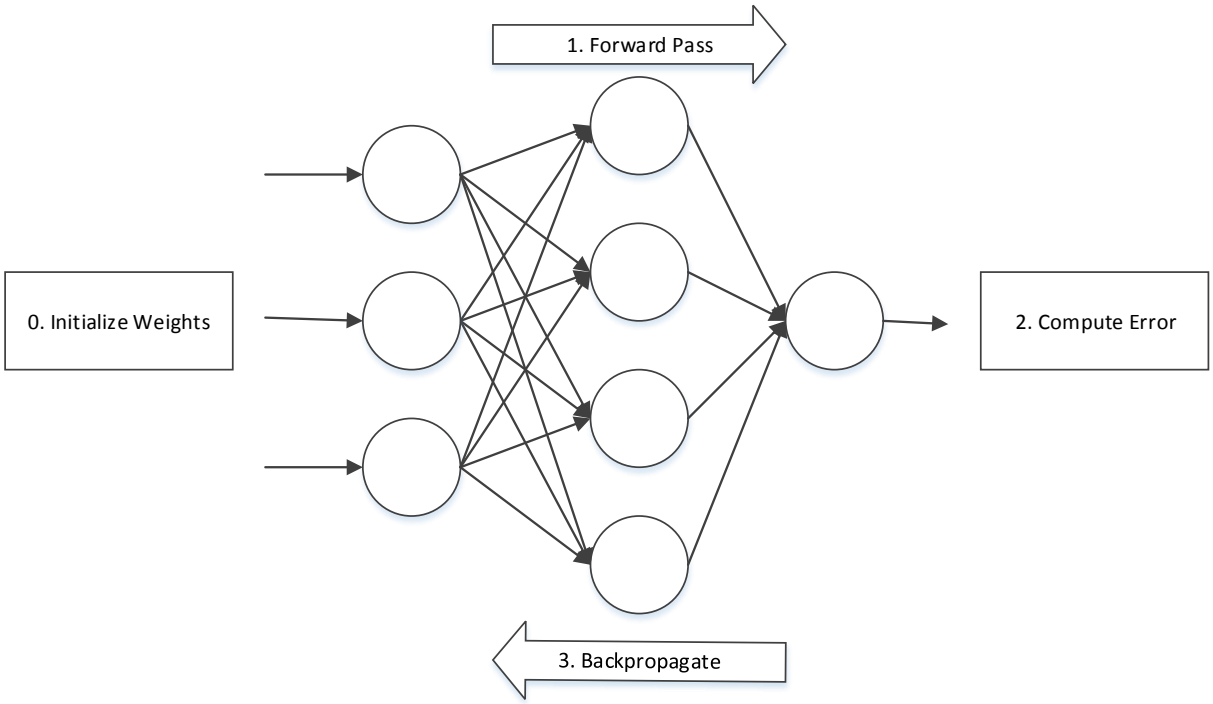

Figure 2.2: Neural Network Backpropagation Training Life Cycle.

Given that ANNs learn the fastest from the most unexpected sample (LeCun et al., 1998), we set the shuffle parameter of the Keras $fit(shuffle = True)$ to guarantee the shuffling of the samples at each epoch. Also, considering our training examples do not have a meaningful order or an implicit degree of difficulty, shuffling the training data after each epoch is advised (Ruder, 2016) and, therefore, we will not pursue some form of Curriculum Learning (Bengio et al., 2009), i.e. a technique which forces the learning to accompany a gradually increasing degree of difficulty of its examples.

A specific form of overfitting is overtraining the weights in the network, by running too many epochs. This can be avoided using early stopping or other regularization techniques (Lu et al., 2001). For further details on regularization see 2.2.4 and on overfitting see 6.3.

2.2.4 Regularization

Regularizers are modifications in the learning algorithm aimed to reduce generalization error but not the training error (Goodfellow, Bengio, et al., 2016) by preventing overfitting (Domingos, 2012; Zhang et al., 2016). They are standard tools when using neural networks which can also help avoid overtraining (Lu et al., 2001). They can act implicitly (e.g. early stopping) or explicitly (e.g. dropout, weight regularization). While, when properly trained, they can improve generalization performance, bigger gains may derive from changes the architecture of the network itself (Zhang et al., 2016). Note that regularizers often imply that overfitting is a global phenomena, but it can vary significantly within different regions of the model (Caruana et al., 2000).

Weight regularization, such as L_1 or L_2, forces the weights to become small by creating an artificial penalty α against large values of the weight vector. A third penalty can be constructed with the sum of the prior two. The three weight losses are calculated as:

$$E_{L_1}(w_i) = E(w_i) + \alpha \sum |w_i| \tag{2.5}$$

$$E_{L_2}(w_i) = E(w_i) + \alpha \sum w_i^2 \tag{2.6}$$

$$E_{L_1 L_2}(w_i) = E(w_i) + \alpha \sum |w_i| + \alpha \sum w_i^2 \tag{2.7}$$

The intuition behind weight regularization is that forcing small values for the parameters yields simpler hypothesis which are eventually less prone to overfitting.

As demonstrated in (van Laarhoven, 2017), the combination of regularization with normalization can influence the learning rate of a backpropagation scheme. A possible workaround is to set the norm of the weight matrix to unit.

Note that Keras (Chollet et al., 2015) also supports regularizing on the bias and on the activity, i.e. the output of the neurons. Despite the potential benefits on the performance of these last two, our literature review mostly refer only to weight regularization, reason why we shall focus only on that one.

Noise Injection adds noise artificially to the ANN input during training and can indirectly penalize complex models (Zur et al., 2009). Intuitively, adding noise encourages the active exploration of the parameter space (Neelakantan et al., 2015). This is usually done with white gaussian noise, namely $w_i = w_i + N(0, \sigma^2)$, where w_i is the weight vector on layer i and it has been found to improve

the generalization capability (Piotrowski and Napiorkowski, 2013). Such deviation measure of the noise $h = \sigma^2$ has impact on the performance of the technique, and consequently on the overall ANN performance (Piotrowski and Napiorkowski, 2013; Zur et al., 2009), reason why we need to be careful when using this technique. (Zur et al., 2009) found that Noise Injection has a comparable, or better than, performance from the methods of weight decay or early stopping. Considering it is difficult to choose a single best value of variance h, and because we are searching for easy, modular and pipeline-able techniques, we will not be considering *Noise Injection* within our scheme. Nonetheless, given its promising applicability it is worth the mention.

The **Norm Clipping** technique is considered as proposed in (Pascanu et al., 2013). It limits the norm of the gradient whenever it exceeds a defined threshold. This is a simple and computationally efficient technique, which is natively found in the Keras package (Chollet et al., 2015). It has, however, a disadvantage of introducing an hyperparameter — the threshold.

Also, given the possibility that large, complex ANNs dominate smaller ones in the training phase because they overfit, one possible regularizer (which we will not use) is incorporating the network size into the cost functions itself (Stanley and Miikkulainen, 2002).

Within the implicit regularizers category, we can use the early stopping technique (Prechelt, 1998) to avoid over-fitting — when the gap between the training and the test error is too large (Goodfellow, Bengio, et al., 2016) — in the specific problem of over-training — when the network learns the noise intrinsic to the dataset (LeCun et al., 1998) by running on too many epochs (Liu et al., 2008). This technique stops the weight-training mechanism of the neural networks, it is simple to implement, has been reported to be successful avoiding overfitting (Piotrowski and Napiorkowski, 2013) and superior to other regularization methods in many cases (Prechelt, 1998).

Considering that deciding *when* to stop can be a difficult challenge, we will use two possibilities to trigger the stopper, both inspired in the works of (Prechelt, 1998). First, a simpler criteria to trigger the stopper which will call *Blunt Patience* — the training process stops if the validation loss (loss function measured in the validation set) does not decrease after n successive epochs. The 'patience' parameter n must be manually introduced due to the complex learning curves a neural network faces in the training process, with validation error curves presenting more than one local minimum. Secondly, we compute the generalization loss as:

$$GL(t) = 100 * (\frac{E_{va}}{E_{opt}} - 1) \tag{2.8}$$

where E_{va} is the error on the validation data and E_{opt} is the best error encountered within the training process so far. Such mechanism anchors the best result and measures the online[2] divergence towards the optimal, stopping the training process when $GL \geq treshold$. This threshold, also known as α, is considered to be 20%, as per the work of (Piotrowski and Napiorkowski, 2013).
In all the early stopping scenarios, it is important to notice that the training process must return to the prior *best* scenario. Otherwise, the ANN would have been stuck in the last training epoch before it was stopped and would naturally learned too much of the training data. If we find to be the errors to be deviating from the 'best' model — measured by the difference of performances on the training and validation datasets — we go back to the network weights which presented the lowest validation error and we apply the model to the test dataset.

An interesting point is made on (Caruana et al., 2000), where the authors claim the advantage of large backpropagated neural networks with an early stopping mechanism. Along the training of these excess-capacity networks, they will encounter smoother models similar to the ones smaller nets would have learned and, as such, they can stop and recreate the others whenever they want. In the end, this implies generalization capacity can be surprisingly insensitive to the network's excess capacity, so there is no disadvantage (besides the time cost) of using too large models.

Other explicit regularizers may be used, such as Dropout (Srivastava et al., 2014) or Batch Normalization (Ioffe and Szegey, 2015), which are further detailed on 3.1 and 3.2.

[2]Online as an immediate stochastic update.

Activation Function	Abbreviation	Function	Reference
Logistic Sigmoid	LogSig	$f(x) = \frac{1}{1+e^{-x}}$	(LeCun et al., 1998) (Glorot and Bengio, 2010) (Goodfellow, Bengio, et al., 2016) (Dugas et al., 2000) (Krizhevsky et al., 2012) (Ioffe and Szegey, 2015) (Janocha and Czarnecki, 2017) (van Laarhoven, 2017)
Hyperbolic Tangent	tanh	$f(x) = tanh(x)$	(LeCun et al., 1998) (Glorot and Bengio, 2010) (Krizhevsky et al., 2012)
Softsign	soft	$f(x) = \frac{x}{1+\|x\|}$	(Glorot and Bengio, 2010)
Softplus	soft+	$f(x) = log(1 + e^x)$	(Dugas et al., 2000)
Rectified Linear	relu	$f(x) = max(0, x)$	(Krizhevsky et al., 2012) (van Laarhoven, 2017)
Scaled Exponential Linear	selu	$f(x) = \begin{cases} x & x > 0 \\ \alpha e^x - \alpha & x \leq 0 \end{cases}$	(Klambauer et al., 2017)
LeCun Sigmoid	lecun	$f(x) = 1.7159\ tanh(\frac{2}{3}x) + \alpha x$	(LeCun, 1989) (LeCun et al., 1998)
ScaledSigmoid	scalsg	$f(x) = \frac{4}{1+e^{-x}} - 2$	(Xu et al., 2016)
HardSigmoid	hardsigm	$f(x) = \begin{cases} 0 & x < -2.5 \\ 0.2 * x + 0.5 & -2.5 \leq x \leq 2.5 \\ 1 & x > 2.5 \end{cases}$	
PenalizedTanh	pnltnh	$f(x) = \begin{cases} tanh(x) & \text{if x} > 0 \\ \alpha\ tanh(x) & \text{otherwise, } \alpha \in [0, 1] \end{cases}$	(Xu et al., 2016)
Rectified Hyperbolic Secant	resech	$f(x) = x * sech(x)$	(Njikam and Zhao, 2016)
Truncated Sin	tr.sin	$f(x) = \begin{cases} 0, & -\frac{\pi}{2} > x \\ sin(x), & -\frac{\pi}{2} \leq x \leq \frac{\pi}{2} \\ 1, & \frac{\pi}{2} < x \end{cases}$	(Parascandolo et al., 2017)
Sin	sin	$f(x) = sin(x)$	
Linear	lin	$f(x) = x$	
AlphaLinear	alphlin	$f(x) = \alpha * x$	
Step	step	$f(x) = \begin{cases} 0 & \text{if x} \leq \text{threshold} \\ 1 & \text{otherwise} \end{cases}$	

Table 2.1: Comprehensive list of activation functions considered in the hyperparameters optimization search.

Loss Function	Formula
Log Loss / Cross Entropy	$\sum_j y^{(j)} log\, \sigma(\mathbf{o})^{(j)}$
L_1 Loss / Least Absolute Errors	$\lvert\mathbf{y} - \mathbf{o}\rvert$
Mean Squared Error	$(y - \mathbf{o})^2$
Squared Hinge	$\sum_j max(0, \frac{1}{2} - \hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})$

Table 2.2: The Cost Functions considered.

Chapter 3

State of the Art

The more recent theories and techniques are now presented. They include innovations such as Dropout, Batch Normalization or Self-Normalizing Neural Networks, and Python libraries, such as Keras and Scikit-learn. Considering the innovations in the field of Neural Networks are primarily modular and easily compatible with the state of the art so far, we too are going to present them in such way.

3.1 Dropout

Dropout is a regularization technique presented in (Srivastava et al., 2014) which prevents overfitting. By temporarily removing a determined percentage of the units from the layer by setting to zero the output of the neurons (Krizhevsky et al., 2012; Wager et al., 2013; Zhang et al., 2016), the technique adds noise to the training process (Njikam and Zhao, 2016) and provides an inexpensive simple way to combine an ensemble of models by averaging their predictions (Goodfellow, Warde-Farley, et al., 2013).

Dropout forces the neural network to learn more robust features by reducing complex co-adaptations of neurons (Krizhevsky et al., 2012) which lowers the generalization error and prevents overfitting without the need to dimensionality reduction (Srivastava et al., 2014).

The original paper (Srivastava et al., 2014) refers some practical tips, which we will state and use as a pendulum for our hyperparameter search. For the dropout probabilities, the recommended 20% for the inputs layer and 50% for the hidden units are considered, along with the max-norm regularization — constraining the maximum norm of the incoming weight vector at each hidden unit to a fixed

constant — between 3 and 4. Also, large decaying learning rates and high momentum (from 0.95 to 0.99) are advised.
We will consider the dropout probability and the max-norm constant as additional tunable hyperparameters.

Along with the classical Dropout technique, we will also consider the **Alpha Dropout** as presented in (Klambauer et al., 2017), a technique that fits well to the SELU activation function, along with the **Gaussian Dropout** also presented in the original paper (Srivastava et al., 2014). The difference between the original and the Gaussian Dropout is that while the first multiplies the hidden activation functions by Bernoulli distributed random variables[1], the lather adds Gaussian noise with zero mean and standard deviation of σ, which becomes another hyperparameter.

An interesting application of this concept is the extension of the concept of dropout as a generic learning method that can be applied to any learning algorithm, as found in (Wager et al., 2013).

3.2 Batch Normalization

Throughout the training of a neural network, the distribution of each layer's inputs change due to precedent change of the previous layer. This is known as the *internal covariate shift* and has been know to slow the training process. Batch Normalization (BN) (Ioffe and Szegey, 2015) is a technique that acts as a regularizer (for further details on regularizers see 2.2.4). This technique works by whitening the inputs at each layer, i.e. by normalizing the means and variances of batches in the training data (van Laarhoven, 2017). BN has multiple benefits: it accelerates the training process and makes it more resilient to parameter scale, prevents saturation in the network and reduces the need for dropout. BN is widely adopted and it is often found to improve the generalization performance (Zhang et al., 2016).

One of the main advantages of BN and Dropout is that they can be coded as layers we add onto the neural network in a modular way. Nevertheless, it arises thus the need to pick the right ordering of their combination for a proper application. The original paper of Dropout (Srivastava et al., 2014) refers to applying the technique after the activation function. Regarding the position of the BN though, its original paper (Ioffe and Szegey, 2015) advises to use it after a fully connected layer but before the

[1]For more information on Bernoulli distribution see A.

activation function. However, recent yet unpublished works seem to suggest using BN after dropout might be a promising scenario, so we will test them both (with the order as another hyperparameter).

3.3 Self Normalizing Neural Networks

Self Normalizing Neural Networks (SNNNs) (Klambauer et al., 2017) are based on the "Scaled Exponential Linear Units" (SELU) activation function which induces self-normalizing properties such as variance stabilization, thus avoiding exploding and vanishing gradients. SNN can keep the normalization throughout multiple layers with many units both in the mean and the variance, which speeds up the convergence (LeCun et al., 1998). Continuing the original Dropout technique (Srivastava et al., 2014), Klambauer et al. propose an *Alpha Dropout* which keeps the mean and variance after the dropout to also keep the self-normalizing property when using SELUs. The original paper recommends dropout rates of 5% or 10% for good perfomance.

SNNNs are able to work with many layers because they do not face the activation function saturation (vanishing or exploding gradients) by enforcing activations towards zero mean and unit variance.

3.4 Software and Tools

The coding processes were implemented with Python, with specific dependency on the **Keras** library (Chollet et al., 2015) for the neural networks, **Scikit-learn** (Pedregosa et al., 2011) for general machine learning purposes (cross-validation and others operations), **Pandas** (McKinney, 2010) for data manipulation, **sklearn-deap** (*sklearn-deap* 2017) for the evolutionary search using the **DEAP** (Fortin et al., 2012) evolutionary computing framework, **Numpy** (Walt et al., 2011) for scientific computing and **Matplotlib** (Hunter, 2007) for the graphical environments. In parallel, there was the need for developing some complementing libraries, such as **Normalizator** (F. Fonseca, 2017c) for normalization of continuous variables, **confusion_matrix_cv** (F. Fonseca, 2017b) for creating confusion matrixes of cross-validated algorithms and **abnormal_activations** (F. Fonseca, 2017a) for unusual ANNs activation functions.

Chapter 4

Bonds

For deeper understanding of the computational difficulties and specificities, it is necessary to understand the underlying theory surrounding the financial securities. The simplistic approach provided below should allow a full comprehension of the thesis.

A bond is a financial security which entitles the bondholder to receive from the issuer the principal borrowed amount plus periodic interest (Hull, 2012; Martellini et al., 2003). For the issuer, the cost of financing will be the coupon rate inherent to the security. Ceteris paribus, the larger the company's stability, the lower the coupon it needs to pay to attract investors.

For the bondholder, the rate of return to maturity is given by the quoted yield, which takes into account both the coupon rate and the price of the security. The yield to maturity is the discount rate of return that equals current price with the future cash flows (Hull, 2012) and it is the rate of return an investor earns from investing in such security if he holds it until the maturity (Martellini et al., 2003).

Classic economic theory states that a bond price can be calculated as the sum of the future cash-flows discounted by their appropriate discount rate (Hull, 2012). The discount rate must be a real quantification of the risk a bondholder incurs on lending the money to the company, i.e. buying the bond. Such risk can be either caused by market movements (an increase in the overall rates causes a fixed coupon bond to be less attractive, thus diminishing its price) or by the credit risk (a company might fail to repay any of the periodic coupons or the underlying principal, incurring in default).

The bond fair value is, in mathematical form:

$$B = \sum_{t=1}^{t} \frac{CF_t}{(1+i_t)^t} \tag{4.1}$$

where CF_t is the cashflow at period t (interest or principal payment) and i_t is the relevant discount rate for the bond's risk at period t.

In the case of a fixed coupon rate bond, an investor knows at inception which are going to be the future cash flows, simply by multiplying the coupon rate by the principal, leaving the uncertainty of the pricing to the quantification of the discount rate.

When dealing with floating rates, the coupon rate normally follows a market index plus a spread for the company's risk, making their price closer to par (the redemption value).

4.1 Bond Pricing and Life cycle

If we consider a constant interest rate, the bond's fair value will be the sum of each cash-flow discounted at the same rate. This implies that the price tends to the par from different directions depending on whether it's coupon rate starts above or below it's appropriate discount rate.

Consider two coupon-paying bonds with 3-year maturity on a 5% appropriate discount rate. For some reason, Bond A was priced with a 6% coupon rate and Bond B with 4%. At inception, their value is:

$$B_A = \frac{6\%}{(1+5\%)^1} + \frac{6\%}{(1+5\%)^2} + \frac{6\%+100\%}{(1+5\%)^3} = 102.72\% \tag{4.2}$$

$$B_B = \frac{4\%}{(1+5\%)^1} + \frac{4\%}{(1+5\%)^2} + \frac{4\%+100\%}{(1+5\%)^3} = 97.28\% \tag{4.3}$$

Solving for the next years, each bond's fair value is thus:

Fair Value	t=0	t=1	t=2	t=3
Bond A	102.72%	101.86%	100.95%	100%
Bond B	97.28%	98.14%	99.05%	100%

Table 4.1: Fair Value with constant interest rates.

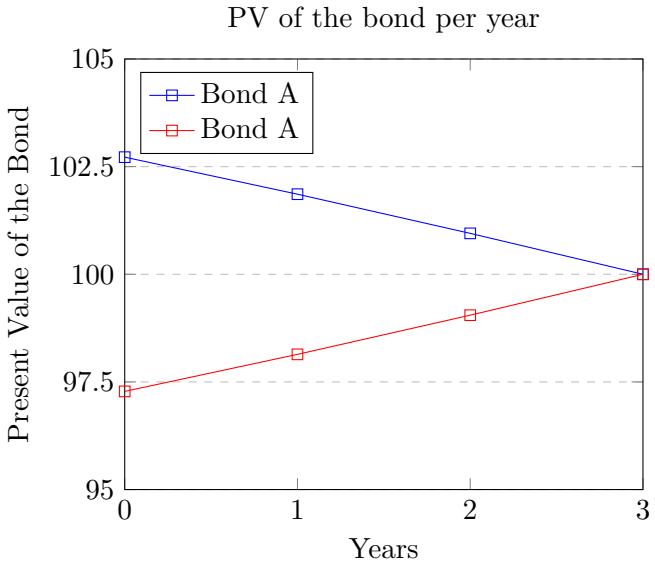

Figure 4.1: PV of Bonds A and B throughout their life cycle.

This would imply that, regardless the year, the Bond A price would always decrease and Bond B the reverse, which we could easily predict just by comparing the initial coupon and discount rates.

When excluding such assumption on the discount rate stability, we observe that the interest rate curves are non-horizontal and they can even jump on unexpected interest rate changes along the years, which will create some disruption on the previously explained price trends.

As an example, a sudden parallel increase the market interest rates also increase a bond's discount rate, which will ultimately decrease it's fair value. Consider a 1% increase in the discount rate on the previous bonds, just before $t = 1$. This would make Bond A's discount rate equal to its coupon rate and, as such, it's price will be always 100%. On the other hand, Bond B would be affected with a sudden decrease in its value for both $t = 1$ and $t = 2$.

Fair Value	t=0	t=1	t=2	t=3
Bond A	102.72%	100%	100%	100%
Bond B	97.28%	96.33%	98.11%	100%

Table 4.2: Fair Value with parallel increase in interest rates in $t = 1$.

As we can see, Bond's B fair value suffers a sharp decrease and only then starts to increase. In an investor's point-of-view, and specifically in Asset Management, it is important to know what are the expected changes in the interest rates because they influence the investment behavior. If the 1% rise is

expected at the start, waiting for $t = 1$ to buy the bond is an optimal decision because avoids having the downside on the bond's value on the first year. Note that this does not influence the rate of return if one waits until the maturity. When measuring in terms of yield to maturity, both bonds perform at the discount rate, if valued at their fair price. Nonetheless, given the trading approach and the necessity to avoid downsides for commercial interest, the relevant measure must be in terms of pricing and not in terms of yield, because we may want to sell before the maturity.

When leaving the assumption of a constant interest rate scenario, the evolution of a bond price along its life cycle can be interesting for a trading approach. Consider a 4% coupon-rate, 5-year bond, within a positive-sloped interest rate curve, which is constant in time, i.e. despite the increase in the interest rate for longer maturities, such behavior will persist in the future. If we consider the following curve:

t	1	2	3	4	5
i	0.5%	1.5%	2.5%	3.5%	4.5%

Table 4.3: Positively-sloped Interest Rate Curve.

by the discounted cash flows method we will have the following fair values (their present value (PV)) for the bond, at each year:

t	0	1	2	3	4	5
PV	98.52%	102.21%	104.44%	104.93%	103.48%	100%

Table 4.4: 5-year, 4%-coupon bond present value with increasing interest rates.

It is clear that, from a trading perspective, it is only interesting to go long, i.e. buying the bond, between the years 0 and 3, given that the price only lowers thereafter.
This period between which is relevant to invest in a trading perspective, which we name the *good trading period*, is influenced both by the coupon rate and by the interest rate curve. If the bond was issued with a 6% coupon rate, investing from year 2 onwards would not be advantageous. Also, if the interest rate was reversed, i.e. negatively sloped from 4.5% to 0.5%, the bond would never have a price increase.
These examples serve to present theoretical evidence for the possibility of trading on bonds — if we can capture these relationships and movements, we might be able to use them in our favor.

When dealing with bonds, the absence of a centralized markets, such as Stock Markets, creates at the same time a problem and an opportunity. Given the fact quoted prices are, by no means, absolute

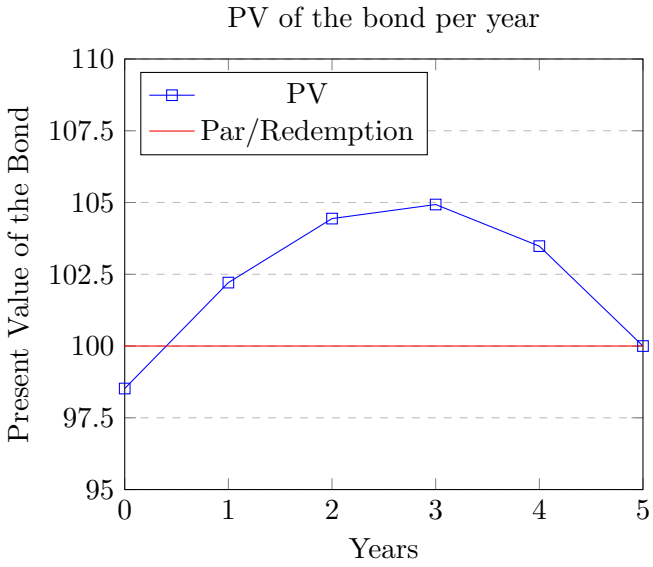

Figure 4.2: Present Value with positively-sloped interest rate curve.

— multiple OTC [1] traders quote different prices which will ultimately be negotiated — such implies that, for every price we consider, we will not know the total certitude of its value. This can be either advantageous for the algorithm, i.e. the prices can swing at our favor, or the reverse. An ultimate possible effect of trading models based on Artificial Intelligence is introducing market depth [2] which can eventually create the conditions for a more rigorous pricing by the market players.

4.2 Categorical Characteristics

Bonds have some categorical characteristics we believe can be helpful in evaluation their fair value and their price movements. For being inherently present in defining the financial security, we considered the Coupon Type, Maturity Type, Call Option and Payment Rank characteristics as categorical variables in our database. Further details on their implementation can be found in the Categorical Data subsection (5.3.2).

Bonds can be Callable, i.e., the issuer can buy them back from the bond holds at pre-defined prices on pre-defined call dates (Ding et al., 2012), which can lead to some uncertainty on its pricing.

Bonds have different ranking on their promptitude of payment in the case of default, so if the default scenario is included in the bond's pricing valuation (which it should), lower-ranking bonds should also

[1] An Over-The-Counter (OTC) market is a decentralized exchange where the market players talk directly between themselves, instead of placing the orders on a formal exchange.

[2] Market depth measured by the liquidity: ease at which one investor can enter (buy) or leave the market (sell).

be priced lower than the higher- ones. Nonetheless, we are not studying a regression problem (e.g. pricing the bond) but a classification one in respect of future movement; so it will be interesting to see if such ranking has an influence on the predictability capacity.

The type of the coupon can also have an influence on the pricing. Floating coupons are less sensitive to market risk because they fluctuate accordingly, so in periods of interest rate rises fixed coupon bonds are expected to have a decrease in their price in a greater extent than floating.

Bonds can have their maturity defined at a certain date or have no maturity defined at all. If the maturity date is defined, the issuer is obliged to repay the remaining cash flows to the bondholder at that specific date. If not, the Bond is considered to be Perpetual where no principal is ever repaid and the only cash flows received by the bondholder are coupons payments. Bonds who have inherent options (call or put) allow either the issuer or the bondholder to exercise its maturity at a time different from the original maturity. Bonds can also have extendible maturities, an inherent option which delay its end.

Chapter 5

The Database

5.1 Data Life Cycle

The data life cycle considered in this work will follow the work of (Fayyad et al., 1996) to provide a process of identifying potential useful patterns in the data. This intends to provide a systematic approach to the process of Knowledge Discovery in Databases (KDD). In accordance, we will proceed through the following steps:

1. Understand relevant prior knowledge, application domain and identify the goal;
2. Create a target data set;
3. Data Cleaning and Preprocessing;
4. Data Reduction;
5. Match the goals of the of the KDD process to the learning algorithm *(classification through neural networks)*;
6. Exploratory Analysis, Model and Hypothesis Selection;
7. Data mining (learning algorithm);
8. Interpreting mined patterns;
9. Deployment of discovered knowledge.

Such procedure allow us to methodically implement a process for treating the data.

5.2 Collection, Selection and Cleaning

Collection

The data collection step is extremely relevant for a good deployment, considering the quickest path to success is often to get more data (Domingos, 2012). The practical approach of this work implies guaranteeing the reproducibility of the model. To achieve this, the database which the model will learn must be constructed from scratch within the on-demand capabilities of the company.
In accordance to the interests of the company, the search criteria will meet non-defaulted bonds in euro currency. Furthermore, some types of coupon and maturities were also excluded:

- Coupon Types: Flat Trading[1], Pay-in-Kind[2], Zero Coupon[3]
- Maturity: Convertibles[4], Sinkable Bonds[5]

Selection and Cleaning

After collecting the raw data, there is often the need to transform (and potentiate) the data to a form amenable to learning (Domingos, 2012).
After the database is constructed, it is necessary to ensure data consistency throughout the time series, as multiple features are time-dependent. Otherwise, we would be overfitting the algorithm to a specific time period, instead of learning the overall relationships that provide a good generalization capability. Unlike stocks, bonds usually have a defined lifetime and their emissions are not synchronised. As such, as time goes by the number of bonds available in the market differs, which forces us to guarantee some type of control on this stability. Also, taking into account the multiple lifetimes of bonds, we need to ensure that we train our model in a general approach and do not overfit to specific bonds which dominate others on the number of trading days. Therefore, to guarantee the model validity and to counter the problem of the imbalaced number of trading days between bonds, we will randomly sample a certain user-defined number of scenarios (samples) for each bond. Such technique allows us to augment the diversification of the database because it increases the number of different bonds

[1]Bond trading without considering the accrued interest.
[2]Payment in other forms than cash.
[3]Does not pay coupons, usually issued at discount. The bond's value is totally reflected on the principal.
[4]Bonds that can be converted into the issuing company's equity.
[5]Debt securities which have their liabilities secured by a sinking fund.

present in the training data, in contrast with the alternative of selecting only the bonds which had a longer lifetime, while controlling a uniform distribution of examples for each bond. This is the same principle of Random Under Sampling we will further discuss on 5.3.1 to counter class imbalanced distributions in classes of the dataset.

Additionally, we will manually cleanse some data which is either not of our interest or which is faulty (e.g. coupon rate below 0% or a negative time to maturity). This is highly relevant as it ensures such non-relevant outliers do not cause noise to the dataset nor waste unnecessary processing time. To guarantee we are dealing with multiple similar datasets, we will erase the data entries which are outside the maximum and minimum margins of the training dataset variables range. For the sake of simplicity, we will consciously forgo some examples to maintain the consistency throughout the results.

A key aspect in any learning algorithm is that it must see changes to capture relationships — constant variables provide us no information, so we must remove them. Due to computational limitations a constant variable may be stored with infinitesimal imprecisions, leading the variance of a variable to be non-zero. Therefore, we will eliminate the variables which have a variance below a certain threshold (1E-20), for which we will consider the variable as constant. Note that a learning algorithm such as the neural network must theoretically adjust its weights to ignore the influence of constant variables. Nonetheless, removing them reduces the dataset size and, consequently, increases the computational performance.

It is important to note that a category which is not of our interest due to the low ratio of samples to cumbersomeness is perpetual bonds[6]. Due to their characteristics, a continuous value cannot be achieved to define their remaining lifetime, causing the dataset feature 'Time to Maturity' to have multiple categories — continuous value for non-perpetual bonds and the 'Perpetual' category for perpetual bonds. Due to their reduced size on the dataset, we have decided to exclude them. Alternatively, we could encode the continuous values to predefined bins and create 'false categories' to include the perpetual bonds as one of the bins (e.g. 0-2 years, 3-5 years, and so forth until 'perpetual'). Nonetheless, we will favor the continuous inputs.

[6]Perpetual bonds do not have a defined time to maturity, only paying interest and never repaying the principal if not called.

5.3 Data Preprocessing

5.3.1 Class-Imbalanced Data

It is relevant to ensure prediction capability in the cases which are a minority in the data distribution on highly imbalanced datasets. These observations are less frequent but are, by no chance, less relevant (Haixiang et al., 2017). A failure in rare event detection, such as fraud detection, severe weather, rare diseases detection (Tahir et al., 2009), defective product detection (Murphey et al., 2004) or software defects can have large impact. This is known as the class imbalance problem (Tahir et al., 2009).

The problems of learning from imbalanced datasets are dual: the learning algorithms can perform poorly on the minority class and it might mislead conclusions with certain metrics (Jeni et al., 2013). This is relevant to our work because neural networks tend to ignore features representing classes that have a small number of examples in the training set (Murphey et al., 2004). This is also true for other machine learning classifiers (Haixiang et al., 2017; Tahir et al., 2009).

Multiple techniques can be applied to minimize this problem, which can be found in (Haixiang et al., 2017; Japkowicz, 2000; Jeni et al., 2013). These include preprocessing techniques, such as resampling, feature selection and extraction, cost-sensitive learning, ensemble methods or modifications to the algorithmic classifier. For practical reasons we will choose the resampling preprocessing techniques, because they allow us to minimize (and hopefully correct) the problem before applying the learning algorithm, thus enabling to 'pipeline' the process. They include over-sampling, under-sampling or hybrid methods (Haixiang et al., 2017; Japkowicz, 2000). Over-sampling involves creating new samples by replicating the minority examples, under-sampling discards samples from the majority class and hybrid methods implement a combination of both (Haixiang et al., 2017; Jeni et al., 2013; Tahir et al., 2009). In specific, we will use the Random Under Sampling (RUS) technique due to its superior performance on large domains (Japkowicz, 2000) and simplicity of use.

Another technique we could possibly use to counter imbalanced datasets is controlling the misclassification (Murphey et al., 2004). We could force the neural network to, when in doubt, choose a passive market strategy by deciding not to invest or the reverse. This can be achieved by defining the relationship between the output of the neural network and the investment decision — normally, the real-valued output x for a binary decision ($y = 0$ for sell/do not buy and $y = 1$ for buy/hold) oscillates between those values and it is rounded to 1 if $y_{predicted} \geq 0.5$ and 0 otherwise, but such threshold can

Classes	Ordinal	N			N-1	
Class A	1	1	0	0	0	0
Class B	2	0	1	0	0	1
Class C	3	0	0	1	1	0

Table 5.1: Ordinal, *1 out of N* and *1 out of N-1* categorical encoding techniques.

be altered manually.

An initial statistic analysis is performed on the dataset to analyze the data distribution. While the class-imbalanced problem usually refers to uneven target distributions, we suspect this can be also a potential problem for our categorical categories. A potential future work relies on studying the effectiveness of applying the RUS technique to categorical classes within the dataset.

5.3.2 Categorical Data

Some of the features we considered within the dataset are categorical (see 4.2), which are natively stored and extracted as text. This creates a problem since multiple machine learning algorithms, such as neural networks, only work with numeric inputs (Potdar et al., 2017). Therefore, we will have to convert the categorical into numeric values.

This can be achieved through multiple encoding techniques, which include *1 out of N* (also known as One Hot encoding), *1 out of N-1* or *Ordinal*. In the *Ordinal Encoding*, each unique category is represented with a numeric code, which implies a certain order or rank. Since not all of the features have a theoretical order, this technique is expected not to be preferable. The *1 out of N* creates N input variables for a feature with N categories, which can be burdensome for the dataset as it augments the data dimensionality and can slow down the performance by introducing multicollinearity(Fitkov-Norris et al., 2012). This can be useful, though, for transforming outputs in multiple classes classification scenarios.
A theoretically more feasible alternative is the *1 out of N-1* techniques, which creates N-1 binary variables for a N-category field (Lai et al., 2006). Comparing with the previous technique, this reduces data dimensionality by having one category as reference with all 0s and avoids the multicollinearity problem.
A visualization example of the categorical encoding variables is available in Table 5.1.

After the categorical encoding is complete, the features will have binary values, which may cause the

saturation of weights due to the possible proximity of the boundaries of the activation functions. AS such, we will transform to 0.1 and 0.9 instead, in accordance with (Fitkov-Norris et al., 2012).

The categorical encoding must be a preprocessing step technique, due to the dimension variability it introduces when used inside a pipeline. For further details see section 6.3.2.

5.3.3 Scaling

By changing the location and scale parameters, the scores from different distributions are transformed into a common domain (Jain et al., 2005; Latha and Thangasamy, 2011), which avoids features in greater numeric ranges dominating others in smaller ones (Huang and C. Wang, 2006), reduces computing time by initializing the training process for multiple features on similar scales (Jayalakshmi and Santhakumaran, 2011) and avoids numerical difficulties during the calculation (Hsu et al., 2010). Furthermore, in the specific cases of backpropagation, (LeCun et al., 1998) advises to average each input to zero mean and scale the variables so that their covariances are similar. The first trick forces the weights to be updated on both signs (+ and -), in contrast to only using positive weights, while the second helps to stabilize the rate at which the weights are updated.

An example of multiple scales can be found in the coupon rate and the bond minimum piece[7] features, which differ naturally in their values. While the coupon rates typically vary in small percentages (between 0% and 10%), the minimum pieces can range from 1 cent (1% of an euro unit) and 1 million euros.

On a practical note, it is important to save the scaling parameters before applying the transformation to both the training and test data as a preprocessing layer. Otherwise, the addition of new data could change the scaling which would, without re-training the model, cause the pre-trained model to predict on wrong inputs. This is confirmed in (Hsu et al., 2010).

Several scales were considered, based on the works of (Jain et al., 2005; Jayalakshmi and Santhakumaran, 2011; Latha and Thangasamy, 2011).

1. Standard Scaler

   Also know as Z-Score normalization, this technique uses the arithmetic average and the standard

[7] Smallest amount allowed in a market transaction.

deviation of the data to scale the data to zero mean and unit variance.

$$x'_i = \frac{x_i - \mu_x}{\sigma_x} \tag{5.1}$$

Given that both the mean and standard deviation are sensitive to outliers, the technique is not robust. Also, the parameters are only optimal for a Gaussian distribution, being only reasonable for other distributions.

2. Min-Max Scaler

   It is best suited for cases where the minimum and maximum bounds are known. Otherwise, scaling with the estimated parameters it will return a non-robust method, concentrating the remaining data to a smaller range in the presence of outliers. Shifts the minimum and maximum bounds to 0 and 1.

$$x'_i = \frac{x'_i - min(x)}{max(x) - min(x)} \tag{5.2}$$

3. Decimal Scaling

   Applied in the assumption that different features vary by a logarithmic factor. It is non-robust.

$$x'_i = \frac{x_i}{10^n} \tag{5.3}$$

   where $n = \log_{10} max(x_i)$.

4. Median

   By normalizing each sample by the median of the raw inputs, the scaler becomes insensitive to extreme deviations.

$$x'_i = \frac{x_i}{median(x)} \tag{5.4}$$

5. Median-MAD

   The *median and median absolute deviation* scaler is insensitive to points in extreme tails in the distribution and outliers. It does not provide a common numerical range and has poor performance when the distribution is not Gaussian, because it relies on the *median* and *median absolute deviation* as estimates of the location and scale parameters of the distribution.

$$x'_i = \frac{x_i - median(x)}{MAD} \tag{5.5}$$

where $MAD = median(|x_i - median(x)|)$.

6. Max Scaler Inspired by the good performance on (Latha and Thangasamy, 2011). It is similar to the Min-Max, with the $min = 0$.

$$x'_i = \frac{x_i}{max(x)} \tag{5.6}$$

7. Modified tanh

   As proposed in (Latha and Thangasamy, 2011), it is a simplified version of the *tanh-estimators* introduced by Hampel (Hampel et al., 1986). Because it does not need the genuine score distribution given by the Hampel estimators, its complexity is reduced and its speed increases.

$$tanhx'_i = \frac{1}{2} * \left(tanh\left(0.01\frac{(x_i - \mu_x)}{\sigma_x}\right) + 1\right) \tag{5.7}$$

We could use the multiple Scalers as another hyperparameter of the learning algorithm to be optimized, as a preprocessing step. Nonetheless, for simplicity and to reduce the number of combinations of hyperparameters, we will use the Z-Score (Standard Scaler). Note that this step is only applicable to the continuous-valued variables, e.g. coupon rates or market indexes. The categorical variables converted to binary features are not scaled.

5.4 Database Creation: The Process

One interesting aspect of our work is the process of creating a database from scratch, a topic we believe lacks documentation and in which we aim to document the thinking process. Thus, to provide some insights into this area, we now describe a diary-like log on our procedures.

5.4.1 Early Steps

Before any construction, we need to understand what is the main goal of our work and its specifications. Being the aim of the project implementing a trading decision-maker, the simplest way to modelize such decision is a binary output — either we invest/hold the financial security or we sell/do not buy. The definition of the Y variable labels $(0, 1)$ will thus depend on the occurrence of an event we will define as being a good trading opportunity or not. Since we have the possibility of creating an historical

registry, we can incur on supervised learning because on top of creating the **X** dataset, which holds all the features, we can label each instance accordingly on the **Y** vector, which holds the target.

Regarding the definition of the target, the Yield to Maturity was one of the metrics thought as the critical indicator, but it implies a temporal stability, i.e. holding until maturity, we will not achieve due to the need of a rotating portfolio, for company purposes [8]. The rotation period was defined, in accordance to the company, with a 5 business days period span, to achieve a weekly rotation. On a theoretical side, we do not see any reason to contest the applicability of such rotation period, despite our concerns on being somewhat shortsighted. For the sake of simplicity, the temporal span we consider to define the **Y** labels vector will be the same to calculate the temporal changes on the **X** historical features.

The choice of the decision trigger thus relies on the Clean Price of the bond, which does not include any accrued interest. This is not the preferable choice because the alternative, the Dirty Price, may provide better insights into the trading profitability — a stable clean price can still provide a trading opportunity due to the accrued interest we receive when holding the bond. Nonetheless, the Clean Price is the *de facto* standard. Also, we can consider as negligible the accrued interest over such a short span, by the way we define the threshold for the variation needed to consider a positive outcome and, consequently, define an instance as of label $y = 1$. As an example, a 10% and 5% coupon bonds have approximately 0.139% and 0.069% of 5-day span accrued interest. On top of that, we have to quantify our transaction costs on a 2-way basis — if a bond is bought on one week (label 1) and sold on the next, the last transaction cost must be inputed on the priors' week decision. The example follows with a 25 basis points (bps) [9] transaction cost for a 1-way transaction. Therefore, the minimum threshold we have for considering a 2-way transaction is, when dealing with clean prices and for the worst case scenario (highest coupon rate):

$$\begin{aligned} min(threshold) &= \frac{10\%}{360} * 5 + 2 * 25bps \\ &\approx 0.64\% \\ &\approx 64bps. \end{aligned}$$

In addition, we can consider a confidence margin so that all the algorithm does learn will undoubtedly,

[8] Note that, for commercial purposes on the buy side, a company has interest in carrying a continuous trading volume because a lot of vendors tend to apply minimum transaction volumes which need to be fulfilled, otherwise the line of trading is closed.

[9] A basis point is percentage of a percentage.

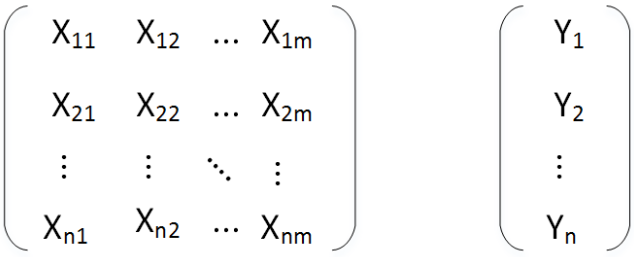

Figure 5.1: Overall dataset in matrix form.

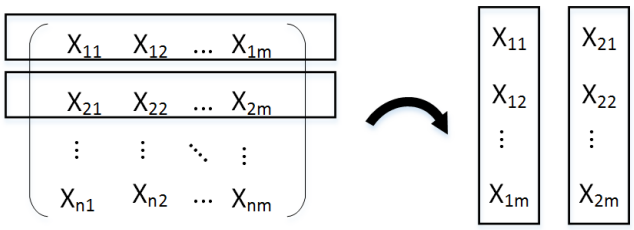

Figure 5.2: Graphic interpretation of sampling the dataset.

and with a risk-margin, be good examples. This margin is manually defined, for simplicity at 16 bps, so that the final threshold is:

$$\begin{aligned} Y_{threshold} &= TimeSpanAccruedInterest + 2WayTransactionCosts + ConfidenceMargin \\ &= 80bps \end{aligned}$$

Having theoretized over the $\mathbf{Y}$ vector, it is now time to demonstrate the development of the $\mathbf{X}$ dataset, also known as the independent variables dataset. The overall layout will be a single dataset which comprises both $\mathbf{X}$ and $\mathbf{Y}$, as demonstrated in Figure 5.1, where n are the number of samples and m the number of features of X.

On applying the machine learning algorithm, we are going to explore the relationships of X, both with Y and internally within different features, that allow us to explain up to some extent the behaviour of Y. Graphically, we can think of training the neural network in relation with the database as drawn on Figure 5.2.

By transposing [10] the original dataset and separating each columns, we can isolate each instance and feed the neural network on the input side and, at the same time, providing the output answer on the other. This is demonstrated on Figure 5.3.

[10] We can define transposing as flipping a matrix over its diagonal, changing the rows per columns.

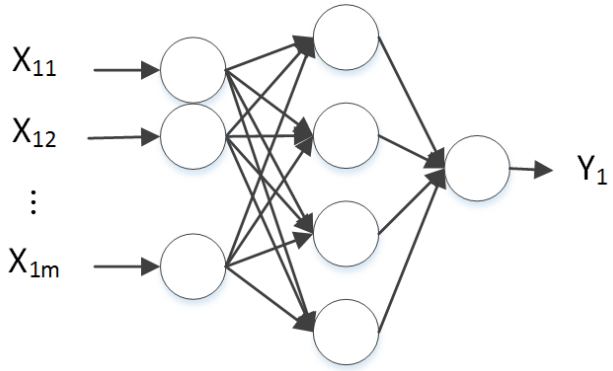

Figure 5.3: Graphic interpretation of passing a dataset sample throughout the neural network.

Having approved the architecture of the final structure of the database, we will now describe how to achieve it.

5.4.2 Inputs

The inputs we consider are 3-fold: Supporting Features, Bond Features and Bond Prices. These will be described further on.

The first, Supporting Features, encompasses market variables which come in raw values, such as the NASDAQ index value or the price of gold, on a daily basis. The final structure shall be a dataset of multiple feature values per day. Since we are dealing with absolute values but we intend not to calculate a regression but to classify, we preprocess such variables by calculating the percentage of change in relation to the same feature n days into the past, being n defined manually, to tailor the features representation to the objective (Shen et al., 2012). The time-index of such calculations is extremely important because we want to clearly define, at each day in the past, what was and what was not available information at that time, so that we do not internally overfit somehow the data. Thus, at each day (T), we will have the percentage change between (T-1) and (T-1-n) days, where n is for simplicity equal to the forecasting period. This aspect of time-awareness of the data will be ubiquitous to the database creation process and we will further return to this subject.

Next, we have the Bond Features dataset, which includes the details of each bond (e.g. coupon rate, maturity and so forth) and intends to provide the learning algorithm with intrinsic details in parallel with the market ones. For practical reasons, some of the entries will be deleted to coerce the dataset to elements of our interest, as referenced on section 5.2. Unlike the Support Features dataset, the Bond Features is not time dependent. For the sake of simplicity, we will consider such details are largely

constant, despite they can alter at some point, e.g. a bond can change some of its details along its life cycle, such as the coupon type or the coupon rate. Such stability allows us to have a matrix of i rows of bonds, which have j columns of features, which do not change.

The third input are the historic Clean Bond Prices, for which we create a time dependent table of prices per day (i columns of bonds and j rows of days). Despite being only capable to buy at the Ask and sell at the Bid price (where $Ask_{Price} \geq Bid_{Price}$), we will consider the Mid price as the relevant ($Mid_{Price} = Average(Bid, Ask)$) to the target vector and use the spread (Spread = Ask - Bid) as an input to the X. The motive behind creating the spread as a feature relies on the flexibility of the prices on the Bond Markets, given they are greatly traded over-the-counter, which leads to some instability on the exact price and use the spread as a proxy for the liquidity[11] of the security, in the hope such property can add informational value to the model.
The shifting operation must also be done for the spread vector. Unlike using the percentage change on the Support Features, we will apply for this feature a moving average for the last n days, with a minimum of 1 day, and shift for 1 day after, so that the database includes these values only for the next day. Similar to the present,if we consider to be in-between the trading period of day T (i.e. intraday period), we only know the moving average of n days for yesterday's close and not as of today, for (T-n-1) to (T-1). Similar to the precautions we had with the time dependent data series of Support Features, we too have to be careful on defining temporarily the movements. Since we are able to develop within a supervised learning environment, our creation of the label must be clear and thoughtful.

Our target vector will thus be the difference of the bonds' prices, followed by an evaluation (a conversion on an interesting movement (label 1) if the change is greater than a threshold and 0 otherwise) and a consequent temporal shift backwards of n days, with n being the forecasting period. This can intuitively understandable by the following example.
Imagine that we have a 5-day week of historic bond prices and we want to define their target, i.e. to classify them as worthy of investment or not, for a 1-day and a 2-day rotation horizon. We are going to use the schematic stated before of differencing, applying a *target function* and a temporal shift. Consider the following prices on table 5.2. If we consider the minimum threshold of 0.8 for going long on the trading decision, we will have the optimal decisions, per rotation schedule, on table 5.3.

[11]The liquidity property refers to the capacity to easily and efficiently enter (buy) or leave (sell) the security on the market.

Day	T-4	T-3	T-2	T-1	T
Price	100.0	100.5	101.5	100.0	102.0

Table 5.2: Examples of bond prices for target definition.

Day	T-4	T-3	T-2	T-1	T
1-Day	0	1	0	1	nan
2-Day	1	0	0	nan	nan

Table 5.3: Optimal decisions for the trading scenario.

We now replicate, for the 2-day rotation, the operations described before, on table 5.4, where **Diff** calculates the absolute difference between values separated by n days (Diff(T-2) = 101.5 - 100), **Tgt Fn** converts the difference to 1 if $Diff \geq treshold$ and 0 otherwise, and the shift operation moves the original target function by n periods backwards. Bear in mind not to override the nan's with zeros because they signal an absence of information, which we cannot interpret as of label 0. Also, the 2-day period cleary demonstrates the need to shift in regard to the forecasting period.

The combination of the datasets will thus rely on using the day T as a pendulum and using the percent changes of the previous n days for the independent variables of X, shifted by one period, and the percent changes of the next n days for the target Y.

Having the different datasets available, it is now time to combine them into one unique X and Y. A crucial step for computational efficiency is selecting the relevant data before creating the soon-to-be-large dataset which will concatenate all the information. One way to achieve this is to take into consideration a minimum number of days of data available for each bond and erase the ones who do not meet such condition. This minimizes the probability that, for any sampled bond, we will not encounter them in a specific life cycle of its price. As an example, a bond issued at a higher coupon rate than its real risk rate will increase its price on the early days and vice-versa, which despite the potential for a good performance, it does not interest us due to its short-sightedness and possible

Day	Price	Goal	Diff	Tgt Fn	Shift
T-4	100.0	1	nan	nan	1
T-3	100.5	0	nan	nan	0
T-2	101.5	0	+1.5	1	0
T-1	100.0	nan	-0.5	0	nan
T	102.0	nan	+0.5	0	nan

Table 5.4: Creation of the target by differencing, applying a target function and temporal shift.

overfitting ability. This was priorly demonstrated on section 4.1.

5.4.3 Overall Frame

In practical terms, the merge will be made by days and, consequently, we will gather all days' datasets into one. As such, for each day we anchor the construction on the dataframe that is constant throughout the entire time series: the Bond Features. Within it, the only calculation is to estimate, for each day, what was the time to maturity of each bond at that time, to have an input which reflects the remaining lifetime and its impact on the trading capability.
After this, we append the Supporting Features (which includes stock indexes, commodities prices and other indexes) relevant row for that specific date. Note that it includes the percentage changes for the previous n days we have chosen, which implicitly take into consideration the final value for the day we are calculating upon. This is something we want to avoid because, at each day of the training dataset, all the information must reflect prior events, as including any calculus which regards the events of that same day may be implicitly correlated with the target itself. Making such mistake would result in overfitting the training data and, consequently, lowering the generalization capability. To avoid this we shift the supporting features vector of data by one trading day — remember this is a matrix of features values per days, so for each day we have a vector. Shifting by one day is the lowest value we can shift to both overcome the overfitting problem (possible direct correlation between the dependent and the independent datasets), while giving us the most recent data possible, at the same time.
Afterwards, we append the Y vector for that day, which contains each bonds' future movements true prediction, already shifted in regard to the temporal over-fitting. Iterating the above operations for each day, we will end with a list of datasets to merge and, again, manually clean non-relevant data. For this time, we will exclude absurd values, such as negative time to maturities or coupon rates, as discussed in 5.2.

A key step on finalizing the dataset is the categorical conversion. We will create dummies on the *1 out of N-1* technique; for further details please see section 5.3.2. From the full dataset, we are going to divide it into two: before and after 2017. We will leave the last as a temporal-continuous validation dataset on where we are going to evaluate our learned model, as an out-of-sample testing. For the remaining dataset, prior to 2017, we will balance the importance each bond has on the training dataset by performing a Random Under Sampling technique, briefly described on 5.3.1, only this time it does not concern the distribution of the Y labels but the number of samples for each bond. This technique

will, for the whole dataset, randomly select u unique number of samples from each bond, so that all have the same weight overall on the learning scheme. We will also drop the *Date* feature on this dataset to force the algorithm to learn intrinsic relationships within the data without providing a key identifier such as the date feature.

On summary, our subsetting of training and test datasets goes as follows. A major split is performed on the end of the year 2016: all data after December 31st, 2016 is considered as out-of-sample test set. This will be the dataset on which we will draw our conclusions. Before 2017, we will split the dataset in 4 subsets: weight training, encoding/dimensionality reduction, hyperparameter optimization and validation.The validation dataset will show us the expected generalization capability of the predictors withing the training time frame but outside the data points on which the learning algorithm has trained. Also, the cross-validation operator (further detailed below on 6.3.2) will create temporarily overridable test sets within the before mentioned subsets of weight training, encoding and hyperparameter optimization.

Chapter 6

The Model

6.1 Problem Definition

A crucial step for meaningful machine learning is the definition of the learning environment. Taking into account we want to predict what are the bonds which are going to be worthful of our investment, we are incurring in a classification problem. A classification task can be defined as specifying which of the k categories some of the inputs belong to (Goodfellow, Bengio, et al., 2016). It is also important to notice that given that the dataset feeded to the algorithm contains labeled past experience, we are working on supervised learning.

Thus, the Evolutionary Neural Network algorithm will train in a supervised learning environment, where the target is a classification problem. The main answer we will test is the ability for the network to predict a future movement on a specific bond price: 1 for an upward movement - the positive class we want to detect (Jeni et al., 2013) - and 0 for a downward movement or a small upward movement (below a threshold that represents the transaction costs). This is the definition of the fifth step of the Knowledge Discovery in Databases approach, as presented in section 5.1.

6.2 The Features and Feature Engineering

One of the most important factors of machine learning are the features (Domingos, 2012). Due to crescent globalization, the movements of worldwide financial markets should have an (expected) influence on the European market, our object of study. In accordance, we take as inputs changes on

both the main stock market indexes (such as the DAX or the FTSE) and other proxies for investment sentiment, such as the price of commodities or currency trading. As discussed in section 5.4, to ensure we do not overfit the model by internally providing the answer and to train in real conditions, we shift temporarily these movements in relation to our target. This allows us to create a row, for each day and for each bond, that contains the previous period movements (day, week or month) of the specific bond market movement (changes in price), the previous movement for the supporting macroeconomic features (e.g. forex, stock markets) and, at last, a final column signaling the target value (0 or 1). In summary, in relation to a certain date, the bond and market movements are defined *a priori* and the target *a posteriori*.

When creating a database from the ground up, the amount of possible inputs is virtually unlimited. Despite feeding the model with more data, there is no guarantee that the data is not redundant or of limited relevance — the benefits may be outweighted by the curse of dimensionality[1] (Domingos, 2012) — which eventually may deteriorate the performance of the learning algorithm (Piotrowski and Napiorkowski, 2013). As the neural network dimensionality size increases with the number of features, more training data is required for maintaining the generalization capability (LeCun, 1989) which consequently increases the training time (E. I. Chang and Lippman, 1991; Yao, 1999). Therefore, selection, creation and elimination of features is a central problem in machine learning (E. I. Chang and Lippman, 1991; Langley, 1994).
Feature selection (or dimensionality reduction) is often required and it can improve the performance and/or reduce the computation time. Nonetheless, selecting features without using dimensionality reduction techniques can be a cumbersome activity due to the exponential growth of the number of possible combinations of features within the original dimensionality (E. I. Chang and Lippman, 1991). When creating new features, it is important to understand the ability each machine learning model has on its own to capture relationships in the data. New features which are created with relationships natural to a specific learning algorithm will be expectedly redundant.

As we are dealing with neural networks, deciding the promising new features can be accomplished by studying the data transformation operations within it, which are mainly within the layers. A layer in a neural network can be normally represented by

$$Y = \varphi(\sum(w_i * x_i + b)) \tag{6.1}$$

[1] While many algorithms work well in low-dimensions, they can become intractable on high-dimensional inputs (Domingos, 2012)

where φ is an activation function, b is a bias, w_i is a vector of weights that multiply by the x_i vector of values. So, basically, a neural network is a complicate connection of sums and multiplications. This orients our feature creation to relationships that are not combinations of these operations, such as ratios and ratios of differences, which are expected to contribute with novel insights. Otherwise, engineered features of powers, counts, differences or rational polynomials would duplicate the capability the neural network already has (Heaton, 2017). For simplicity, we will only create new features if we find our learning model to have high bias, i.e. underfitting the data.

As explained above, the dimensionality of a neural network is important parameter to consider because it can impact the training time and the generalization ability. As such, to guarantee wide applicability, we will have to limit the features of the dataset by either deleting already existent ones, replacing them by engineered features (in which we include data dimensionality reduction techniques) or carefully add new features up to a certain feature size, in the event of high bias.

6.3 Generalization, Over-fitting and the Learning Curve

There is an open debate between the relationship of the capacity/complexity of an ANNs and it's tendency to overfit, some supporting (Piotrowski and Napiorkowski, 2013) while others reject it (Caruana et al., 2000). Intuitively, we can easily associate the overfitting phenomena within small datasets — a complex learning algorithm learns all the data instead of the relationships. Nonetheless, overfitting can occur within large datasets when the underlying distribution is complex or when the features space is large (Zur et al., 2009).

6.3.1 Generalization and Learning curve

Generalization is a central issue both in designing and training the network (Lu et al., 2001). The ultimate goal of a machine learning classification task is the capacity to generalize - i.e. to correctly perform on data it has not seen before (Domingos, 2012; Goodfellow, Bengio, et al., 2016; LeCun et al., 1998; Piotrowski and Napiorkowski, 2013; Prechelt, 1998). Estimating the generalization capacity is important to predict the future prediction capacity (Kohavi, 1995). Considering this generalization error is unknown, we use the performance on a validation set, a subset of the training data, as a proxy for the real world performance of the model. The generalization error is proxied by the difference

between the error on the training dataset and this validation set (Zhang et al., 2016). The capability to generalize can be degraded by overfitting (Liu et al., 2008).

The modular design of ANNs can often result in a curse, where the number of trainable parameters is larger than the number of samples they are trained on (Zhang et al., 2016). The multitude of different parameters of the ANNs increase the complexity of the ANN, which make them prone to overfitting (Lu et al., 2001), i.e., classifying better the training data than the population of cases at large (Zur et al., 2009). The effective capacity of several successful ANNs architectures is large enough for them to completely learn the training data (Zhang et al., 2016), reason why we need to be extremely careful and attentive to performance In fact, our total number of hyperparameters combinations is over 31 million, a value quite superior to the dataset size. If we decrease the network processing capability by reducing the number of free parameters, we can increase the generalization capability (LeCun, 1989) because we reduce the learning on specific noise and concentrate the learning on fundamental relationships.

A major problem in machine learning is balance between under- and over-fitting the model to the data. If we under-fit, our model will under-perform on its processing capabilities. If we over-fit, the learning scheme learns the noise instead of the signal present in the training data (Piotrowski and Napiorkowski, 2013), which yields a poor applicability on the real world. Another ever-present problem is finding the global optimal. This can be a though task because the cost surface is often high-dimensional with many local minima and/or flat regions (LeCun et al., 1998), which can trap gradient descent optimization techniques. In order to test the balance between under- and overfitting the data, we will split the training dataset into three subsets: training, validation and test datasets. For the test dataset, considering the data we are analyzing may be time-dependent, our out-of-bag test sample will be dual: random points along the training dataset and continuous data points for a specific time period (we will use the 1^{st} of January, 2017 onwards). This gives us the model performance on both stochastic and continuous prediction. Therefore, to measure the model's generalization capacity, we have extracted all the data from 2017 from the training dataset, a technique also found in (Piotrowski and Napiorkowski, 2013). This ensures the ability to both predict on unseen data and to predict on data further on the time series. Such derives from the necessity that with very flexible classifiers, which neural networks are, a strict division must be enforced between the training and testing datasets (Domingos, 2012).

After all the necessary steps are taken — from collecting data to training, testing and evaluating —

the deployment of the final model is in order. For such, all of the dataset set aside for training and testing will be merged and a final training is performed on all the data (Domingos, 2012), considering we can mitigate overfitting issues within the model itself.

6.3.2 Cross-validation

Cross-validation is a resample technique used to estimate the expected capacity of a predictive algorithm to generalize (Barrow and Crone, 2013) which can help prevent overfitting (Domingos, 2012). By averaging the scores on repeated training and testing processes on multiple subsets of the original dataset (Domingos, 2012; Goodfellow, Bengio, et al., 2016), Cross-validation aims to artificially simulate the process of predicting on unknown scenarios, by momentarily force the known test data class labels into oblivion, train the algorithm and match the predictions against the known labels.
In particular, *K-Fold Cross-validation* splits the dataset into k non-overlapping subsets and averages the test errors across the folds. This technique allows us to test the network capability to generalize and ensures a more robust classification (Fitkov-Norris et al., 2012). Stratifying the folds so that they contain approximately the same proportions of labels as the original dataset (Lu et al., 2001) seems to be uniformly better than not to (Kohavi, 1995; Mojarad et al., 2011). As recommended in (Kohavi, 1995), we will use a 10-fold stratified cross-validation. A major advantage of *K-Fold* is that all training observations are used with equal weights, with each observation being validated exactly once.
Other forms of Cross-validation are available, such as *Holdout* or *Leave-One-Out* (Barrow and Crone, 2013; Lu et al., 2001). The first is the simplest — equivalent to having $k = 1$ on *K-Fold Cross-validation* — the dataset is split into training and test dataset, on a proportion that often follows heuristic rules of thumb, such as 70%/30%. The latter is extremely cumbersome because for the n rows of samples it creates $k = n$ folds, meaning every sample is tested on a learning algorithm trained on the remaining data.

From the observation of the variables distribution analysis of our datasets, the i.i.d. (independent and identically distributed) random variables assumption is not encountered, specifically when it comes to categorical variables. Their lack of uniformity in the distributions blocks the use of commonly used Cross Validation iterators[2], such as the *K-Fold* or the *Stratified K-Fold* because of their unawareness of feature distribution. To overcome this we will create a non-trainable feature, called 'eras', which will be the identification of the specific combinations of the categorical variables' dummies. In practical terms,

[2]The iterators generate the indexes which we will use to subset the datasets.

we will have to convert (revert) the dummy encoding to a ordinal encoding — a single identifier (era1, era2, and so forth) from combinations of multiple dummy variables — where the 'era' amount does not have a meaning or order, i.e. era3 is not quantifiably comparable to era2. Not having a specific era(s) within the training dataset creates constant variables, which we should delete for reducing the data dimensionality since they do not provide additional information, which in the end creates dimensionality concerns for the training of the neural network by varying the number of features at each training circumstance. To simplify such process, and facing a lack of relevant alternatives, we will not erase the constant variables by expecting the ANNs will eventually disregard their importance. For the group cross-validation, i.e. cross-validation with awareness of the eras, we will have the *Group K-fold*, *Group Leave One Out* and *Group Leave P Out* cross-validation iterators, natively present in (Pedregosa et al., 2011).

The fitness function applied over the cross-validation process is the binary cross entropy, due to its awareness of the predictor's confidence (proxied by the predicted probability) and heavy penalties on very bad predictions.

6.3.3 Statistical Analysis

A major component of the generalization ability of a learning scheme is to impede the model from over-fitting, i.e. learn the training dataset so well it does not learn the intrinsic data relationships, which thwart generalization capacity. To do so, we will study univariate and multivariate feature relevance algorithms which will allow us to grant a certain validity to the dataset. If a variable has too many relevance, or it is too correlated with the target, it can be a sign that the variable is implying the answer in its values, in some way.

Random Forests is a popular, efficient algorithm for classification and regression (Genuer et al., 2010). In particular, the Extra-Trees algorithm (Geurts et al., 2006) generates an ensemble of randomized decision trees from which we can infer each variable relevance. Considering the choice of parameters can influence discriminating useful from useless variables, we will use the default values considered in (Genuer et al., 2010; Geurts et al., 2006) of $M = 500$, $K = \sqrt{n}$ and $n_min = 2$; where M is the number of trees in the forest/ensemble, K is the number of attributes considered at each node, n the number of attributes/features in the dataset and n_min the minimum sample size for splitting a node. This is natively available in the Scikit-learn package (Pedregosa et al., 2011). The Extra-Trees Classifier shows an absence of dominant features, with uniformity in their importance. This enable

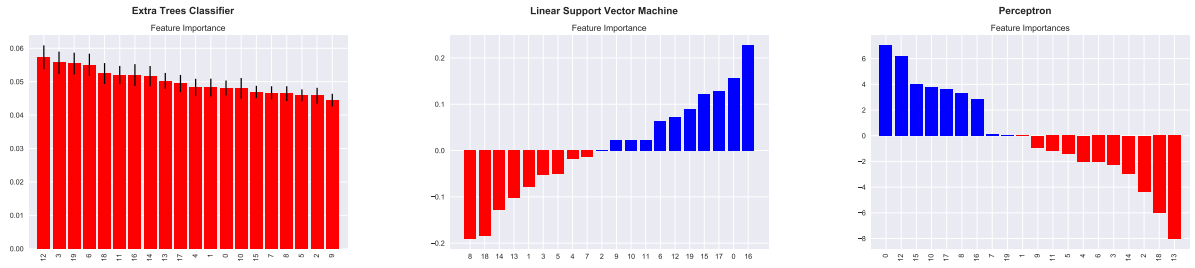

Figure 6.1: Extra Trees, Linear SVM and Perceptron features weights/ importances.

us to validate that no feature is, in the light of this model, specially dominant and that, as such, no overfit to the target is made. We can also observe a relative stability in the standard deviations of the features importances (black vertical lines) throughout the multiple (500) estimators used.

Another way to measure a feature relevance is to use Support Vector Machines, an algorithm which finds a separating hyperplane with the maximal margin in the dimension space (Hsu et al., 2010). As described in (Y.-W. Chang and Lin, 2008), we train a Linear SVM on a L-2 loss and sort weights in the model. The SVM graph shows an absence of a dominant feature importance, with an unevenly distribution throughout the signal (positive or negative) and throughout the absolute feature importance value. Such results validate the absence of overfitting and, thus, the use of the dataset.

On the subject of Neural Networks, we can use a 1-neuron classifier, capable of recognizing linearly separable patterns, called the Perceptron (Rosenblatt, 1958). Decoding the value of the weights will not be our final aim, we just intend to check for a non-extreme-balanced distribution. The Perceptron graph shows diverse results on the features importances, both in amount as in signal. There is also an absence of specially dominant features, which could imply a direct relationship of such feature(s) with the target, what would ultimately lead to overfitting the model. We have found the results to validate the use of the dataset.

The weights of the SVM, ExtraTrees and Perceptron linear classifiers are depicted in Figure 6.1.

Another way to test our dataset validity is to calculate the Pearson's correlation coefficient — which measures the linear dependence between variables of each variable (Guyon and Elisseeff, 2003; Mojarad et al., 2011) — of the features set $\mathbf{X}$ against the target $\mathbf{Y}$ and between themselves. The Pearson's correlation between variables X and Y varies between -1 and +1 and it is calculated by:

$$\rho_{XY} = \frac{\mathbf{cov}(\mathbf{X}, \mathbf{Y})}{\sigma_x \sigma_y} = \frac{E[(X - \mu_x)(Y - \mu_y)]}{\sigma_x \sigma_y}; \tag{6.2}$$

where σ is the standard deviation, μ is the average and E is the expected value. The correlation of

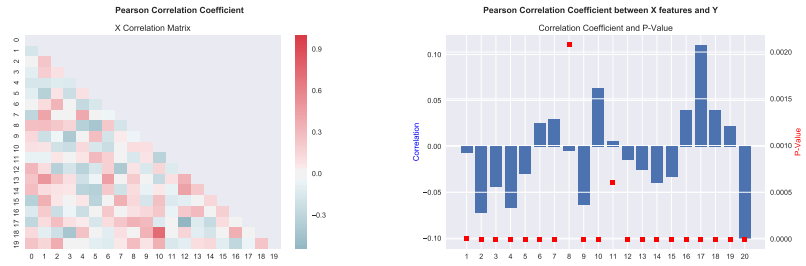

Figure 6.2: Pearson's correlation of the variables.

the variables between themselves and against the target is depicted Figure 6.2.

This allows us to check if we are somehow overfitting the model by providing an implicit answer on the $\mathbf{X}$, which would be noticed by an abnormal correlation between the variables. One might use such test to reduce data dimensionality by discard variables which have near-zero correlation (no relation between both variables) or for which the p-value for the null hypothesis is above a certain threshold $p > \alpha$, being α a certain significance level (that the relationship is not statistically significant). Nonetheless, as we are dealing with neural networks capable of capturing complex relations between the data, removing features which may not seem relevant in a linear way may result in lowering the network's capacity.

From the Pearson's correlation graph between $\mathbf{X}$ and $\mathbf{Y}$, we validate that there is no strong abnormal relation of a specific features, with the maximum absolute coefficient close to 0.1 and that all feature's correlation coefficients are statistically significant at, at least, 1% (their p-value is less than this threshold). The Correlation Matrix between the features of $\mathbf{X}$ studies their internal correlation within the dataset. We observe an absence of clear patterns of correlations, despite observing higher values for the positive than for the negative coefficients. This test validates that within our dataset we do not have abnormal relations which prevent its use.

The Logistic Regression (Cox, 1958) is a linear binary classifier which we will use as the estimator of a Recursive Feature Elimination (RFE) study, a backwards feature eliminator. By iteratively eliminate features and test the performance, we can plot the predicting performance per number of features selected which allow us to sense the overall performance of the classifier when the dataset is reduced at each step. Again, our interest relies more on the overall figure than studying deeply this behavior — theoretically, we can consider the dataset as balanced if no single feature has an extreme predictability capacity and the performance increases, at naturally different paces, with the number of features. Starting with all the features allow us to study their combination in full provides

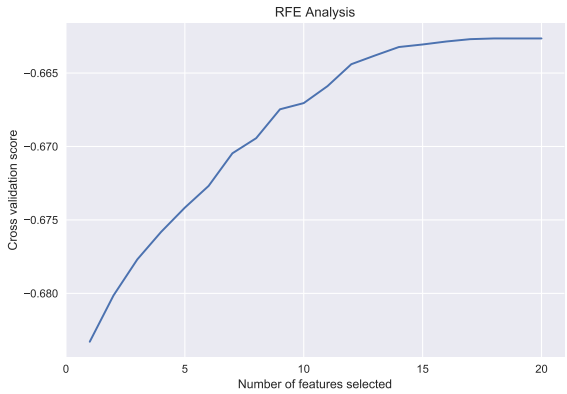

Figure 6.3: A Recursive Feature Elimination study using Logistic Regression.

stronger combinations despite being more slow, when comparing to working the other way around with a Forward Feature Selector (Guyon and Elisseeff, 2003). The Recursive Feature Elimination performance is shown on image 6.3. In accordance with the expected behavior, the RFE has an adequate performance, i.e. not extremely good, for a reduced number of features, a rapidly increasing performance that ends up stagnating when there are still several features to be selected. Note that the considered error function is the negative cross entropy, for which higher values represent a lower error and, thus, a better result. The overall behavior illustrates the existence of different feature's relevances for linear models.

The Principal Component Analysis (PCA) is a technique for reducing dimensionality while preserving as much statistical information as possible (Jolliffe and Cadima, 2016). While this can be used as a preprocessing step, we intend here to validate the dataset. As we are running a classifier, the study of running the PCA per class can provide an insightful and interest result. To allow the visualization of the results, we will run the PCA for 2 and 3 components, creating two- and three-dimensional plots, respectively. After transforming the original dataset, we assign each Principle Component to the graph's axis and differentiate their class label by color. This will return a distribution of class labels on the PCA's axis. Thus, if the dataset is imbalanced, we are expected to see clearly separable clusters. The 2D and 3D PCA are depicted on Figure 6.4. From the figure we can observe generally distributed values, both overall and with relation to the class labels. Despite the existence of some outliers, we consider the dataset to be adequate according to the PCA by the overall non-existence of class clusters.

Overall, the results found in the Statistical Analysis tests, both univariate as multivariate, validate the use of the features ($\mathbf{X}$) dataset. Despite presenting some high values for internal correlations within

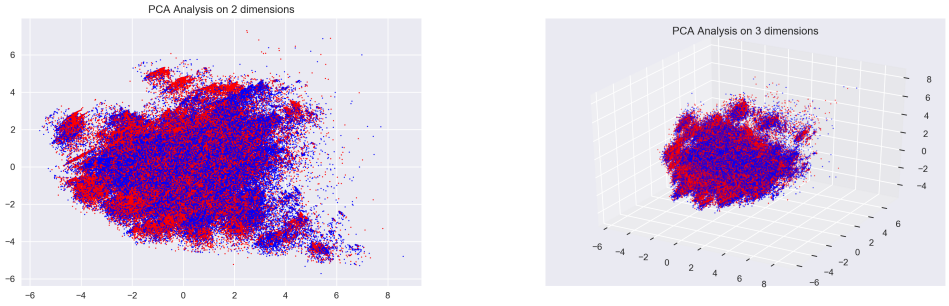

Figure 6.4: 2D and 3D PCA per class label.

features, all of the tests failed to identify a severe overfitting of dataset features to the target which could have implications on the model's final capacity of prediction.

6.3.4 Metrics

After the learning algorithm has trained, it is relevant to measure its performance on known data to get a glimpse of the generalization capacity — the ability to perform on unseen data. A metric alone is not sufficient to detail all of the capacities of the model. As such, a combination of measures is needed to give a balanced evaluation of the algorithm's performance (Sokolova et al., 2006). The complimentary discussion of results is available in chapter 8.

Measuring the accuracy of the model, i.e. the percentage of correctly predicted samples (Fitkov-Norris et al., 2012; Goodfellow, Bengio, et al., 2016; Kohavi, 1995), is very simple and easy to implement but is often a poor choice to evaluate performance Fawcett and it can be misleading (Jeni et al., 2013). For example, when leading with an unbalanced data set (Fitkov-Norris et al., 2012), the accuracy will tend to correctly classify the majority class and ignore the remainder. Despite its advantages, the accuracy is a very common performance measure (Lu et al., 2001). A common metric for categorical classification accuracy is the confusion matrix (Fitkov-Norris et al., 2012), which records the correctly and incorrectly instances for each class (Kohavi, 1995) and allows to construct a multiplicity of metrics. A binary classification problem generates a $2x2$ matrix, as show on Table 6.1. A comprehensive list of the Confusion Matrix used metrics is detailed on Table 6.2. A Python application for the confusion matrix can be found in (F. Fonseca, 2017b; Pedregosa et al., 2011).

Other relevant measure is the Area Under Curve of the ROC (Receiver Operating Characteristics) curve (Fawcett, 2006). The ROC curve plots the classification results from the most positive to the most negative (Sokolova et al., 2006) with the *true positive rate (tpr)* on the $y - axis$ and the *false*

True \Predicted	1	0
1	True Positive	False Negative
0	False Positive	True Negative

Table 6.1: $2x2$ confusion matrix.

positive rate (tpr) on the $x - axis$. By comparing the *recall (tpr)* against the *fpr*, we can study our learning algorithm's prediction capacity against a pure random classifier. The further we are from the diagonal random line to the northwest, the better. Classifiers that output discrete values create single points on the ROC space. Nonetheless, as we can output a continuous (probabilistic) score, we may study the impact of considering multiple thresholds: if the score is above a certain threshold, return 1; otherwise return 0. This generates multiple points in the ROC space which eventually merge to form a line, below which we calculate the Area Under Curve. The AUC-ROC is intuitively *'the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance'* (Fawcett, 2006).
The *Discriminant Power* has a performance of poor when $DP < 1$, limited for $1 \leq DP \leq 2$, fair for $2 \leq DP \leq 3$ and good if $DP \geq 3$.(Sokolova et al., 2006)

In parallel with the metrics already discussed, it is relevant to measure the performance of the trading models throughout the testing period. As such, we will sample the top decisions for each week, considering the model is designed for a weekly portfolio rotation, and measure the profitability of those trading decisions. The buying decision will be made for the highest probabilities predicted — as we are trading on a binary decision of 0 or 1, the neural network outputs a continuous value between those limits for which higher values, i.e. closer to 1, have higher probability of being interesting buying opportunities.
The results of the metrics stated above are available on chapter 8.

Metric	Formula	Citation
Sensitivity / Recall	$\frac{tp}{tp+fn}$	(Sokolova et al., 2006) (Fawcett, 2006) (Jeni et al., 2013) (Lu et al., 2001) (Mojarad et al., 2011)
Specificity	$\frac{tn}{tn+fp}$	(Sokolova et al., 2006) (Fawcett, 2006) (Lu et al., 2001) (Mojarad et al., 2011)
Precision	$\frac{tp}{tp+fp}$	(Sokolova et al., 2006) (Fawcett, 2006) (Jeni et al., 2013)
False Positive rate	$\frac{fp}{fp+tn}$	(Fawcett, 2006) (Lu et al., 2001)
Accuracy	$\frac{tp+tn}{tp+fn+fp+tn}$	(Sokolova et al., 2006) (Fawcett, 2006) (Jeni et al., 2013) (Lu et al., 2001) (Mojarad et al., 2011)
AUC_b (Balanced Accuracy)	$\frac{Sensitivity+Specificity}{2}$	(Sokolova et al., 2006)
F-measure	$2 * \frac{Precision*Recall}{Precision+Recall}$	(Fawcett, 2006) (Jeni et al., 2013)
Youden's index	$J = sensitivity - (1 - specificity)$	(Youden, 1950) (Sokolova et al., 2006)
Likelihoods	$\rho_+ = \frac{Sensitivity}{1-Specificity}$; $\rho_- = \frac{1-Sensitivity}{Specificity}$	(Sokolova et al., 2006)
Discriminant Power	$DP = \frac{\sqrt{3}}{\pi}(logX + logY)$, $X = \frac{Sensitivity}{1-Sensitivity}$, $Y = \frac{Specificity}{1-Specificity}$	(Sokolova et al., 2006)

Table 6.2: Comprehensive list of metrics from confusion matrix.

Chapter 7

Deployment

7.1 Dimensionality Reduction

One of the focus points of our work is the applicability for the general reader who wants to implement such techniques without having to resource to dedicated servers (which have more computational capacity than the common laptop). The need to increase the performance thus raises the question to subset the original features space and/or reduce their dimensionality. Reducing the dimensionality of the dataset is thus a necessary step for fasten the overall process and it can be advantageous since it facilitates classification tasks (G. E. Hinton and Salakhutdinov, 2006).

Dimensionality reduction relies on the assumption of a lower dimensional intrinsic dimensionality. We will consider dimensionality reduction in three categories (Y. Chang, 2014): subspace, manifold and kernel. They differ on the assumption of the underlying topology of the data, with *subspace learning* focusing on linearity, *manifold learning* on non-linearity and the *kernel* has an hybrid approach combining both worlds. Both PCA and LDA (Linear Discriminant Analysis) are computationally efficient under a linear subspace, but fail when the structure of the data is not, i.e. when the low-dimensionality lies on a non-linear manifold. Manifold techniques of Isomap or Locally Linear Embedding (LLE) suffer from the curse of dimensionality to characterize a manifold and do not have easy out-of-sample extensions. An interesting alternative on this category is t-SNE (van der Maaten and G. Hinton, 2008).

The problem with such techniques is that they exploit fixed relationships in original dimension of the data to learn, which may not be valid (W. Wang et al., 2014). As such, we believe using an Autoen-

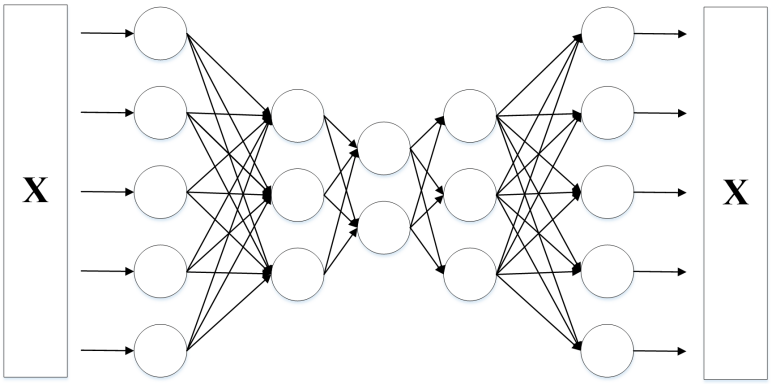

Figure 7.1: Representation of an autoencoder.

coder (Rumelhart et al., 1986) can be useful due to its flexibility and because it has been found to be better and more flexible than PCA and LLE (G. E. Hinton and Salakhutdinov, 2006; Kaguara et al., 2014).

An autoencoder is an unsupervised neural network trained to learn a compressed representation of its input by minimizing its reconstruction error (W. Wang et al., 2014). The intuition behind the undercomplete autoencoder — whose code dimension is less than the input dimension (Goodfellow, Bengio, et al., 2016) — is that if we reduce the number of nodes across the layers to expand afterwards, we are forcing the neural network to learn the intrinsic structures which will have to be present on the smaller layers of the network. Having such complete network trained with proper results, we can split it to use only the encoder section to reduce the dataset dimensionality. Such technique is more flexible that manifold learning because it also learns sparse, overcomplete feature representations of the data (Ng, 2011). We have decided to start with a simple autoencoder and test its results. If those do not meet our needs for efficacy, we will explore further adaptations and innovations of the technique. A graphic representation can be found in Figure 7.1.

The autoencoder hyperparameters were manually defined due to the good efficiency of the results obtained on the early implementation. The layout on which we will center our development is a 20-nodes-mid-layer with a kernel regularizer (an extra loss penalty on the weights matrix) of $L_1 = 1E-5$ and *selu* activation functions (Klambauer et al., 2017). Similar to subsetting the original dataset to optimize the hyperparameters combinations, we too have subset some examples dedicated to train the autoencoder. The main idea behind it is to avoid training the encoder and the consequent weights/architecture optimization of the neural network on the same dataset, as it would double adapt the learning algorithm to the specific dataset noise.

We have tested the influence on the number of epochs, activation functions and the number of the

nodes on the mid-layer by measuring their combinations' MSE errors on the original dataset. The main observations of our work are:

- The autoencoder only works after scaling the continuous and 'soft-binarizing' the discrete binary features;
- The most diffuse PCA results are obtained on low number of epochs (1, 2, 5) and low number of nodes in the mid layer (5, 10);
- High number of epochs and number of nodes in the mid layer result in PCAs either near-zero or with a clearly defined shape;
- The error values stagnate approximately at 50 epochs;

To guarantee the stability and accuracy of the autoencoder dimensionality reduction technique we have plotted the learning curves, i.e. train and validation loss throughout the epochs on weight training, measuring the error on the multiple datasets available: weight training, hyperparameters search, cross-validation, encoding and test dataset. The learning curves can alert us for problems of high bias (underfitting) or high variance (overfitting). The quality tests are depicted in Table 7.1 and Figure 7.2.

Dataset	Reconstructing Error
Weight Training	0.0881
Encoding	0.0887
Hyperparameters Search	0.0906
Cross-validation	0.0884
Testing	0.07707

Table 7.1: Mean Squared Error of reconstructing the multiple datasets with an Autoencoder.

From Table 7.1, we observe a similar error values for the training datasets (Autoencoder Training, Weight Training, Hyperparameters Search and Cross-validation), so there is no problem of high variance. The graph shows an evolution towards a low training error, which signals a sufficient capacity

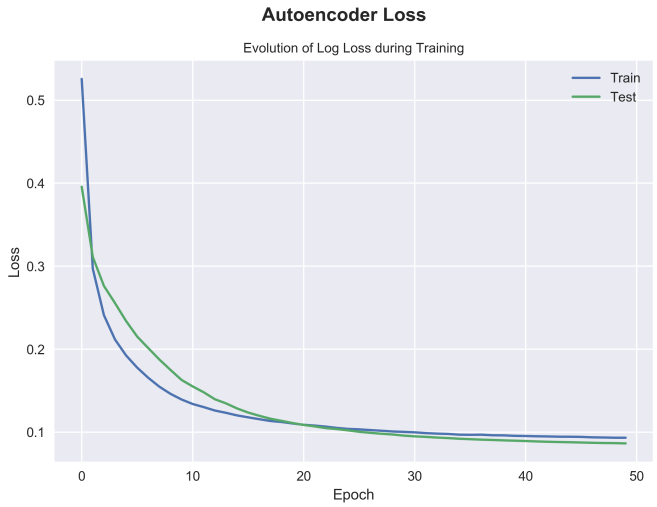

Figure 7.2: Learning Curvers of the Autoencoder training.

of the autoencoder, along with a similarly low test error which signals no overfitting is occurring. The application of such technique is thus validated by the observance of low bias (low underfitting) and low variance (low overfitting).

7.2 Hyperparameters Optimization Strategies

7.2.1 Introduction

Most machine learning algorithms have settings that must be defined externally, outside the learning environment (Goodfellow, Bengio, et al., 2016). These are called hyperparameters and they can have a deep impact on the algorithm performance. Considering a major advantage of neural networks is their modular design (Janocha and Czarnecki, 2017), the way we search the flexible parameters is highly relevant. Despite being defined externally, we can internalize the hyperparameters combination optimization within a learning process itself by applying different schemes of searches of those combinations. These include the Manual, Grid, Random and oriented searches described below.

Both grid and manual searches are widely used strategies for hyper-parameter optimization (Bergstra and Bengio, 2012). Grid-search is a naive search method which tries every combination possible, which is computationally expensive. Even in the cases of a low-dimensional hyperspace search, some orientation is advised by combining two grid searches: a broader one to identify potential good regions and a finer grid on the "better" region (Hsu et al., 2010). On the other hand, the manual search is

very selective by using human intuition but it is expectedly difficult to reproduce.

A more efficient approach is the Random Search (Bergstra and Bengio, 2012), where the hyperparameters hyperspace is randomly searched. This has multiple advantages in relation to the prior methods: it does not allocate so many trials on regions that do not matter, it can find models similar in performance within a small fraction of computational time and giving the same time random search can explore a larger hyperspace. Nonetheless, as the authors mindfully discuss, this is a non-adaptive strategy: the optimization does not change its course with consideration for the results it encounters. In accordance, some works (Bergstra, Bardenet, et al., 2011) propose the need to search the problem space more efficiently. Thus, we will use random search as a baseline but we will test an adaptive search method which theoretically has a better performance due to this mindfulness of the results it encounters along the process.

Evolutionary Algorithms (EA) are a class of population-based stochastic search algorithms that are developed from principles of natural evolution (Yao, 1999). One of the strategies, Genetic Algorithms, will be our choice for the optimization problem. The artificial evolution of neural networks using genetic algorithms is called *Neuroevolution* (NE) (Stanley and Miikkulainen, 2002). Genetic algorithms (GA) are an adaptive optimization search methodology based on the Darwinian principle of 'survival of the fittest' (Huang and C. Wang, 2006). GA generates populations of alternative solutions as chromosomes, evaluates their quality by a fitness function and then applies three operators to evolve the population: *selection*, *crossover* and *mutation*. The *selection* operator grants the fittest individuals to survive by eliminating bad solutions; the *crossover* generates *offsprings* of two individuals, hoping to take advantage of useful parts of both parents; and *mutation* introduces some noise to avoid the population from becoming too similar and thus avoids convergence towards local optima (E. I. Chang and Lippman, 1991; Hertz and Kobler, 2000). We used an uniform crossover, in which each hyperparameter of the offspring's chromosome is randomly selected with equal probability from its parents. For the mutation operator, we applied the same method for all the parameters, regardless of their data type (numerical, categorical or binary), for the sake of simplicity and ease of implementation. By defining a probability of mutation, we provide the search method the possibility to alter, with such probability, the original value of the hyperparameter on the chromosome. We believe such type-aware mutation may provide a further improvement on the results obtained by the Evolutionary algorithm. GA have been shown to be effective in exploring a large, complex space (Yao, 1993; Yao, 1999) in an adaptive way (Kim, 2006) with less chance to get trapped in a local optimal (Huang and C.

Wang, 2006), and thus are suitable for our algorithm to optimize the search in the hyper-parameters hyperspace.

A disadvantage of searching the hyperparameters hyperspace with GAs is that we can spend too much training time on poor combinations, which despite evolving through successive epochs, still have a very poor performance overall. To counteract this problem, we created a customized callback mechanism implemented in Keras (Chollet et al., 2015) that ensures very bad architectures are early stopped (see 2.2.4 to accelerate the hyperparameters search). Such callback stops the neural network weight training if, after 50 epochs, the validation loss is still above a certain threshold, for which we defined $threshold = 1.00$ using the cross entropy loss function. To implement the evolutionary search we will use a python package based on DEAP (Distributed Evolutionary Algorithms in Python) (Fortin et al., 2012), called **sklearn-deap** (*sklearn-deap* 2017).

Optimizing the hyperparameters on the same training dataset used for weight training would result on the solution with the maximum capacity, which would lead to overfitting (Goodfellow, Bengio, et al., 2016). As such, we will dedicate part of the training dataset only to the hyperparameter optimization search.

A relevant problem when dealing with the optimization of neural networks' parameters arises. Due to their modular design and extensive parameter tuning, such overparametrization may cause a significant negative impact on the performance (Piotrowski and Napiorkowski, 2013). We will expect to encounter this problem when combining all the possibilities for architectures and techniques within the non-manual search methods. Considering the advantages and disadvantages of the methods stated above, we will study three search methods: Manual, Random and Evolutionary (specifically using GAs). We intend to study the influence of their different approaches on the obtained results.

7.2.2 Comparison and Preliminary Results

A key element of our work is to compare the Evolutionary, Random and Manual hyperparameters combination search methods, with regard to three main aspects: efficiency, ease-of-use and best solution encountered. We consider *efficiency* as the trade-off between results and the cost to obtain them (in our case, the most valuable resource is computing time): a search method is efficient only if it achieves good results within a small cost. This property is highly subjective, so we can only consider it when comparing between methods and not in absolute value. The *ease-of-use* can be defined as the

ability to quickly and efficiently implement the search procedure on a practical approach — we consider here the necessary changes to the coding script and their impact. The *best solution encountered* will be measured by the performance of the best hyperparameters combination each search procedure encounters on the out-of-bag test dataset, to measure their generalization capability.

To allow a meaningful comparison, we need to define the rules of the contest. For the automatic search processes, we have studied three numbers of scenarios within each search: 20, 50 and 100. This means that, for the Evolutionary and Random searches each, we have 5 runs where 20 different combinations were tested, 5 runs with 50 combinations and 5 runs with 100. This is relevant for further applications because it may signal interesting developments on the number of scenarios needed for the search methods to be sufficient.

Given that the Manual implementation has awareness of the state-of-the-art techniques, higher acquired knowledge and a more flexible approach for model validation (e.g. measuring fitness on multiple metrics at the same time), we have only allowed for 5 models to compete. We have limited the manual search method environment to similar rules of the least capable automatic search scenarios. As such, only 20 combinations of hyperparameters were allowed. Also, similar to the other search methods, the manually oriented search could only be tested within the validation subset of the training dataset, so no performance on the final test set was observed.

For the evolutionary process, we will run 4x5, 5x10 and 10x10 generations of individuals, respectively. The parameters of the evolutionary search itself are 20% for the probability of mutation and 50% of crossover, with an hall of fame of 1 individual[1] and a tournament size of 3[2]. To test the efficiency of the evolutionary search orientation, we test the same number of scenarios (generations * population) but with a randomized scenario, i.e. each combination is purely chosen at random.

Regarding the *ease-of-use*, the manual search is the easiest search method to implement because it does not require almost any construction in comparison with the remainder. Both the Evolutionary and the Random search methods rely, in terms of Python script, on the same coding principle of creating a dictionary which will hold multiple lists of all the possible values for each of the parameters, for which each hyperparameter has a 'key'. Both automatic methods are already implemented and documented on Python libraries, such as 'scikit-learn' (Pedregosa et al., 2011) and 'sklearn-deap' (*sklearn-deap* 2017), allowing a quick and relatively easy implementation. Therefore, regarding the

[1]The hall of fame refers to propagating the best k individuals throughout a new generation. The value is forcibly $k = 1$, due to library coding restrictions; we believe a higher value may be beneficial.

[2]The tournament size randomly samples k individuals of a population for which the best are considered to become a parent

ease-of-use, all of the methods have similar implementation.

From the perspective of *efficiency* and *best solution encountered*, we must take into account the results further discussed on chapter 8. Nonetheless, regarding the aspect of time cost of implementation, we observe a much quicker implementation for the manual than for the automatic search methods. Both evolutionary and random search methods rely on training multiple learning algorithms, a process that is computationally intensive and time-consuming. Manually searching throughout the hyperparameters combination hyperspace is also a costly process, but the implementation of the state-of-the-art techniques allowed quickly reaching good results.

Considering the fundamental aspects of ease of implementation and efficient deployment which underly the development of our work, we have found that the optimization scheme with automatic search methods must take into account only one metric at a time. We do realize that such decision may lead into a suboptimal solution when compared to multiobjective optimization algorithms, such as the NSGA-II (Deb et al., 2002). Nonetheless, in practical terms, using only one metric provides a much faster and easier implementation.

We advise for future work the investigation of the multiobjective search algorithms in comparison both to the results obtained by a single objective search and to the manual search method, in the context of low population sizes and limited computing capability.

7.2.3 Exploding Combinations

One of our expected main problems will be to efficiently navigate the enormous hyperspace of hyperparameters combinations such modular approach on developing neural networks creates. An early test on this subject return a total of ≈ 290 million possible combinations. Due to the slow training process experience on training the neural networks, we can hardly increase our number of tested scenarios for both the Evolutionary and the Random search methods, which harshly limits the search. Even if we could test 1.000 scenarios in each method (which we cannot due to the computational limitations), this still represents $\approx 0.00034\%$ of the total possible combinations. We are thus concern that such infinitesimal range of values tested is not capable of capturing the overall picture of the loss function hyperplane and that, as such, it will not be able to properly orient to the best direction.

On a practical tip, to test that all the possible values for the parameters are working properly we have decided to run a large population (1.000 individuals) across multiple generations (100) but just on one epoch, with a faster cross-validation iterator, to check if the model accepts such inputs. This

enable us to quickly validate if the learning algorithm script is working properly to avoid errors which could cost us computing time and lost of work if they appeared further on, e.g. a typo on a parameter causes the code to crash and with it all the work done so far.

For future reference, a list of the optimized hyperparameters and their category is detailed on Table 7.2. The hyperparameters possible values are divided into three categories, according to their possible values: numeric, categorical and binary.

Hyperparameter	Category	Value Range
Number of hidden layers	numerical	[1, 2, 3, 4, 5]
Number of nodes per layer	numerical	[5, 10, 20, 30, 40, 50]
Activation function of mid-layers	categorical	(see 2.1.2)
Activation function of output layer	categorical	(see 2.1.2)
Weight optimizer	categorical	(see 2.2.3)
Weight initializer	categorical	(see 2.2.1)
Weight regularization technique	categorical	[None, L_1, L_2, L_{12}]
Weight regularization lambda	numerical	[1E-4, 1E-3, 1E-2]
Use of Batch Normalization	binary	[true, false]
Use of Dropout	binary	[true, false]
Batch Normalization prior to Dropout	binary	[true, false]
Probability of first-layer dropout	numerical	[0.0, 0.1, 0.2,..., 0.7, 0.8, 0.9]
Probability of mid-layers dropout	numerical	[0.0, 0.1, 0.2,..., 0.7, 0.8, 0.9]
Loss (error) function	categorical	(see 2.2.2)

Table 7.2: List of optimized hyperparameters with categories.

Chapter 8

Results

In order to compare the different hyperparameters optimization strategies, we have tested the results of the multiple models found by the search methods on the out-of-sample testing dataset of 2017. The analysis of the results over different metrics intends to provide a basis for answering the research questions presented in section 1.5. We will mainly use two desirable characteristics to compare the results: best solution encountered and stability. The first is found by the highest value (or lowest, depending on the metric) and it gives us the best possible scenario. The stability characteristic is concerned with finding concentrated distribution of results, which represent an inherent steadiness. The formulas for calculating the metrics and their referencing are available in the Metrics subsection 6.3.4.

In parallel with the metrics previously presented in subsection 6.3.4, it is relevant to measure the financial performance of the trading models throughout the testing period. As such, we will sample the top 10 decisions for each week, considering the model is designed for a weekly rotation, for a specific weekday which best suits the company's interests, and measure the profitability of the trading decisions. The decision for buy/hold will be made by the highest predicted probabilities — as we are trading on binary decisions of 0 (sell/do not buy) and 1 (buy/hold) and the neural network outputs contains values between those limits, higher values have higher probability of being interesting trading opportunities.

Accuracy

The accuracy metric measures the percentage of correctly predicted outcomes, either positive (1) or negative (0). *Ceteris paribus*, the higher the accuracy the better.
The manual search method is dominant on the accuracy results, having an higher minimum and than the remainder maximum values and a more concentrated distribution of results. Between the evolutionary and the random search method, they both share the same range of values, approximately, with the genetic search having slight less stability.

Log Loss

The Cross Entropy metrics measures the closeness of the model's predictions to the target. It is relevant to know if a "buy/hold" decision, i.e. $predicted\ target = 1$, is due to a small degree of confidence above random ($predicted\ probability = 0.55$) or due to a high confidence (closer to 1). The reverse for the "sell/ do not buy" decision is also true. The threshold for a good log loss result is below $0.6931 = -ln(0.5)$.
The best Log Loss (or Cross Entropy) results are found by the manual search method, along with the best stability (despite having a poor outlier). Between the Evolutionary and the Random search methods, the second presents higher stability and better results, but both the automatic search methods present several negative results.

Sensitivity

The Sensitivity (or Recall) metric measures the percentage of the true buy decisions (positive scenarios) correctly predicted, i.e., from the true opportunities, how much did we predicted correctly.
The best scenario is found by the Evolutionary search method, followed by the Random. Despite having the best results, their dispersion of results is quite high. The manual presents lower valued results but with higher stability.
It is important to note the shortcomings of using this metric isolated. A classifier who only predicts positive scenarios will have the highest value of sensitivity, despite not providing useful insights. Thus, we believe this metric should be analised in relation to its equivalent for the negative examples — specificity — which we will further discuss combining both in the appropriate metrics: AUC, Youden's Index, Likelihoods and Discriminant Power.

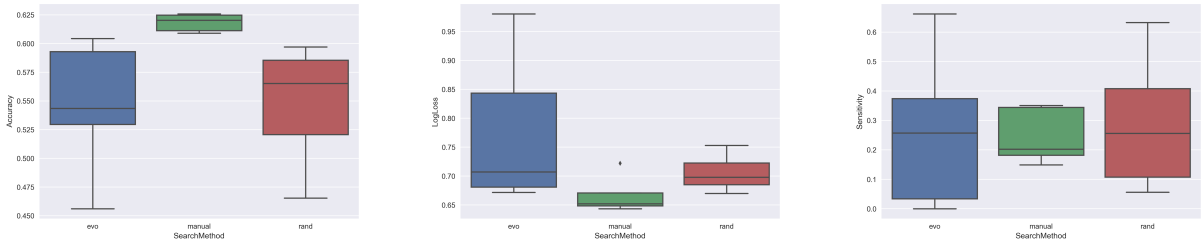

Figure 8.1: Accuracy, LogLoss and Sensitivity box plots.

Specificity

The specificity measures the percentage of the true negative results correctly predicted. As with the sensitivity metric, this measure can be misleading if we take it into consideration alone.
Both Random and Evolutionary search methods have high dispersion of results, slightly higher for the second method, but both with very good and very bad results. On the contrary, the manual search provides a low dispersion within a good region of values.

False Positive Rate

The false positive rate shows the wrongly positive predicted examples, from all the true negative. The desirable behaviour for this metric is having the lowest possible error. Nonetheless, this can be a misleading measure of performance — a model which only predicts 0's has a false positive rate of 0 but it is not of interest. As such, only the stability can be analised and the metric's value must be taken into account along with other performance metrics.
The manual search presents the highest stability, with the random coming in second and the evolutionary search with the worst (highest) spread of results. The direction provided by the genetic-based search method does not provide a visible advantage on stability of the false positive rate metric.

Precision

The precision measures the correctly predicted examples from all the positive predicted samples.
The best result is found by the evolutionary search method, which seems to be an outlier. The manual search provides a relative low dispersion within a good region of results. The random search has similar results to the evolutionary, both in value and in dispersion.

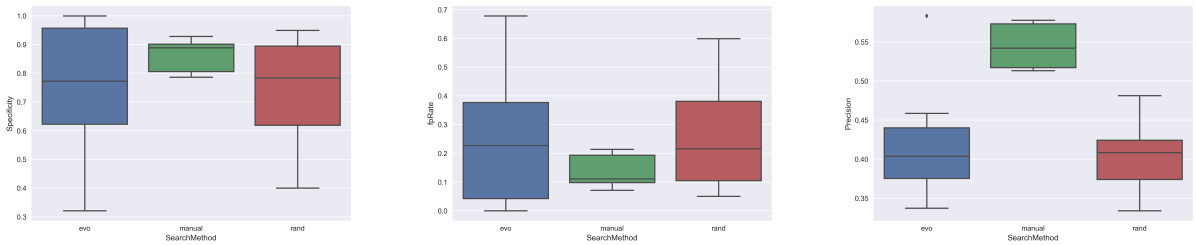

Figure 8.2: Specificity, False Positive Rate and Precision box plots.

Balanced Area Under Curve

The balanced Accuracy (approximated Area Under Curve) is the average between the specificity and the sensitivity metrics. To achieve interesting results, the values must be above the 50% threshold — this is the value a model which predicts the same target every time achieves (e.g. a model that never predicts a positive scenario).

The overall best results are found by the manual search, with a reasonable stability. Between the evolutionary and the random, the first has an equivalent performance of results with lower dispersion (higher stability), so we believe it to be preferable. Nonetheless, the two are both very near, and sometimes below, the threshold.

Youden's Index

The Youden's Index formula is $J = sensitivity + specificiy - 1$, with the last operator forcing its values to oscillate between -1 and 1, since both metrics are between $[0, 1]$. The last operation (-1) also transforms the metric to have value of 0 when the predictor is not useful (e.g. predicts all samples into the same class). Thus, a useful predictor presents a positive Youden's Index and, in reverse, a worse-than-random predictor presents a negative index value. Despite taking both simpler metrics into account, it doesn't punish all 1's or all 0's type predictions. A visualization of the contour plot is available in the appendix A.2.

The manual search method dominates the remainder, with a similar stability and a distribution on higher values. The Evolutionary search results are similar to the Random ones, but they do present an higher stability, which makes them preferable. Both automatic methods present results near threshold.

Discriminant Power

The Discriminant Power metric is a zero-centric metric (i.e., a random classifier has $DP = 0$) which punishes predictions who do not present good results on the sensitivity and specificity metrics, simultaneously. A visualization of the contour plot is available in the appendix A.2.

The manual search method presents the better results (higher values) and a better stability (lower dispersion) than the remaining methods. The Evolutionary have slight better results and higher dispersion than the Random method. It is relevant to note that only the manual search method has clear results above the random threshold ($DP = 0$) and that the other methods present some bad results ($DP < 0$)

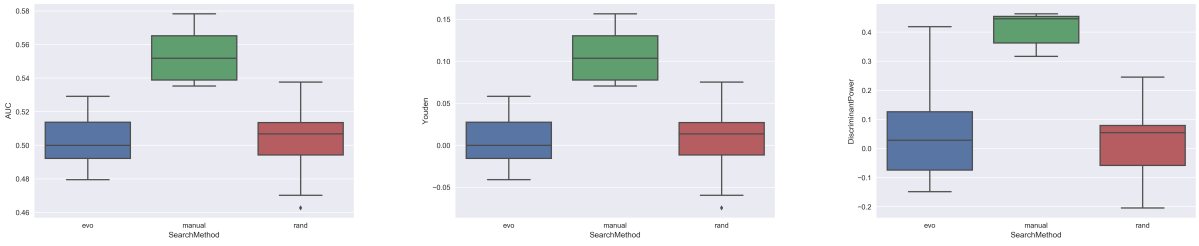

Figure 8.3: AUC, Youden's Index and Discriminant Power box plots.

F-Measure

The F-measure takes into account the Precision and Recall (a.k.a. Sensitivity) metrics, so that both have to present interesting results to have a high F-score. A visualization of the contour plot is available in the appendix A.2.

The Random and Evolutionary have similar results on the best performance (highest value), but both have low stability. The manual search method presents a slightly lower best result but with better stability, being thus preferable.

Profit

The profit metric is the financial gain made by the models during the test period. Naturally, the higher the profit the better the model. The financial gain throughout the testing period can be found in Figure 8.5.

The best profit is achieved by the manual search method, followed by the random and, at last, the

Evolutionary. With regard to the results dispersion, the manual is appears the best search method, similar stability to the evolutionary but within an higher range of values. The Random method does not only have a low stability (high variance) of results, but also presents several negative results.
Therefore, we conclude the manual search method to present the best result for the Profit metric, since it displays the best result and its distribution is the only within positive values.

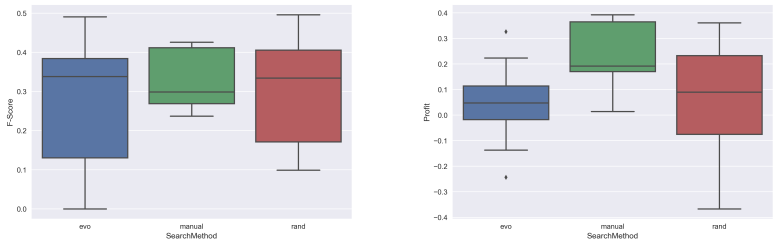

Figure 8.4: F-Score and Profit box plots.

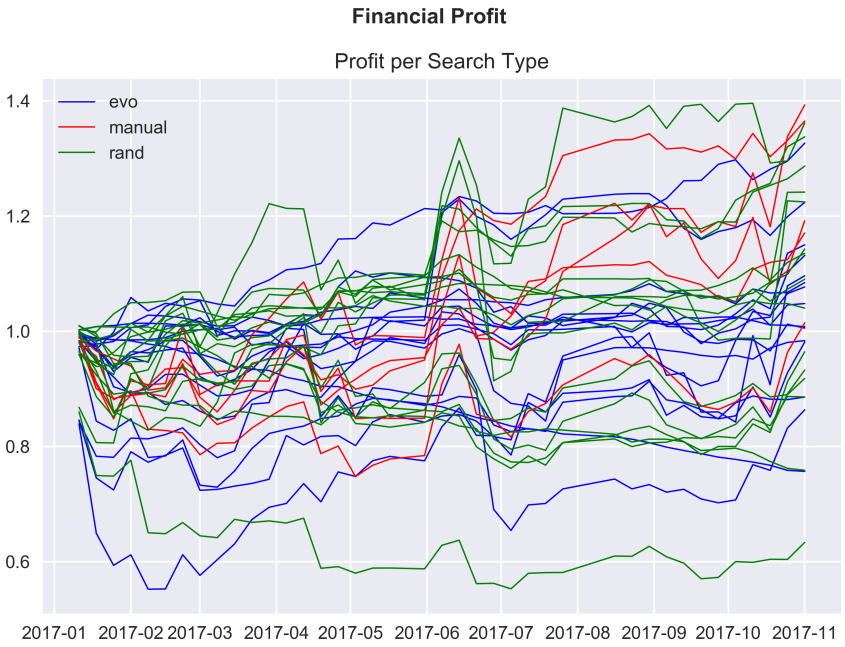

Figure 8.5: Testing Period Financial Gain by Search Method.

Likelihoods

The likelihoods metrics evaluate the classifier performance on the positive and negative classes separately. A higher positive likelihood and a lower negative likelihood mean better performance on positive and negative classes, respectively.
Regarding the positive likelihood, the evolutionary and manual search methods presents the best re-

sults (higher positive likelihood value), despite the first seems to be an outlier. The negative likelihood results are favorable to the manual search method, presenting the best results and the lower dispersion.

Overall, the manual search method has superior performance than the Random and Evolutionary methods on the likelihood metrics.

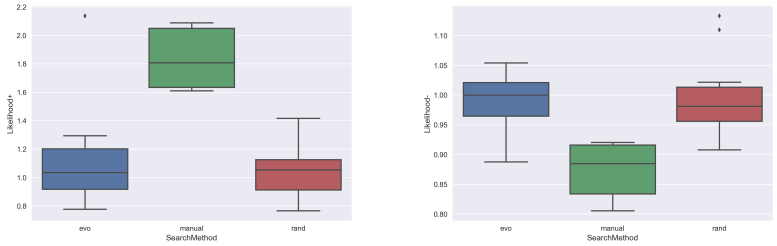

Figure 8.6: Positive and Negative Likelihoods box plots.

True Positive vs False Positive

By plotting the results of the models' False Positive Rate and True Positive (Sensitivity) rates, along the x- and y-axis, respectively, we can graphically observe and compare the models' capability of prediction with the scope of such metrics. For this comparison we used the results from the Manual, Evolutionary and Random search methods, along with a Logistic Regression classifier to serve as baseline.

The best possible classifier predicts correctly all of the 'positive' scenarios ($y = 1$) and does not make commit errors on predicting 'negative' scenarios ($y = 0$), which locates himself in the top-left corner of the graph. The worst case classifier does not correctly classify one 'positive' scenario and mistakenly predicts all the 'negative', which gravitates its results towards the bottom-right corner of the graph. If a classifier is purely random, it will eventually gravitate towards the diagonal dotted line which connects the bottom-left to the top-right corners of the plot. In this linear and simple way, we can condense the overall result of the models: the closer to the top-left corner, the better.

By observing the Figure 8.7, we can note an absence of the number of expected data points for the Evolutionary method, as we only see 13 of the expected 15. This is due to an overlap between results located on the lower-left corner, specifically with $fp_{rate} = 0$ and $tp_{rate} = 0$.

From the graph we observe that only the manual search method results are clearly distant from the diagonal random line and that they are the closest to the top-left corner. While some of the Evolutionary and Random search methods results are better than a random classifier, they are not

stable enough to present clearly good results. Also, the baseline logistic regression presents a bad result, being located below the diagonal line.

Therefore, we conclude to the dominance of the Manual in relation to the Evolutionary and Random search methods, in their results on the combination of True Positive (a.k.a. Sensitivity, Recall) and False Positive rates metrics.

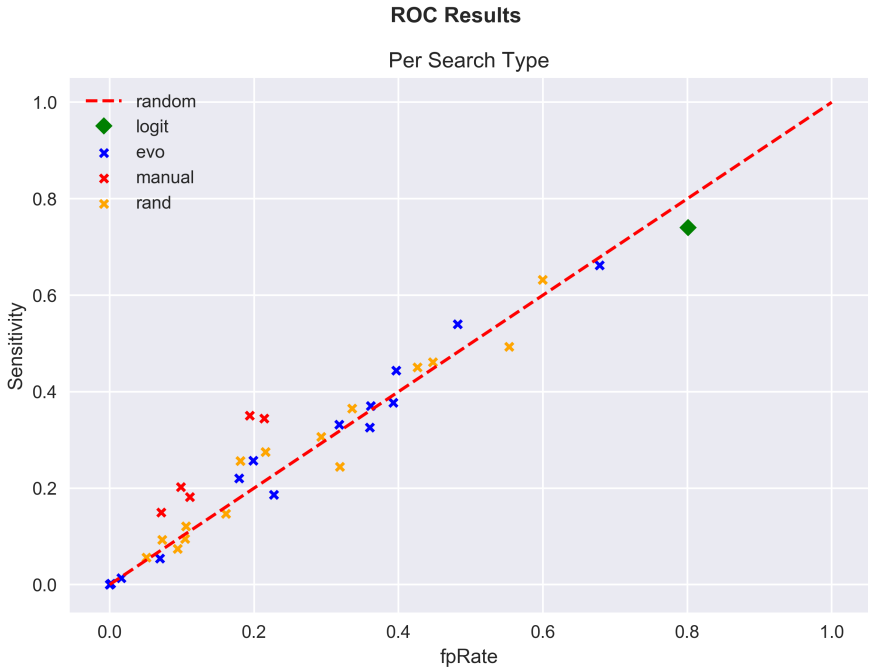

Figure 8.7: True Positive vs False Positive Results per Search Method.

Chapter 9

Conclusions

9.1 Research Answers

The results above detailed present sufficient evidence to answer the initial research questions to which we propose to answer:

H1. Are the Evolutionary and Random search procedures efficient, in their time-cost of application?

H2. Does the direction provided by the Evolutionary Search procedure provides an advantage over a pure Random Search procedure?

From the 13 metrics discussed on the chapter 8, the manual search method is preferable on the most majority (11) than the remaining methods. Overall, the manual method presents the best results (if not, they are close) and the lowest dispersion/higher concentration of results, which indicate higher stability of the metrics results. We thus conclude the dominance of the manual search method in relation to the evolutionary and random methods, in the context of optimizing a large number of hyperparameters. Considering both automatic search methods have a high time cost of application, as previously discussed on section 7.2, we conclude such methods are not not efficient on optimizing complex hyperparameters combinations hyperspaces. Regarding the benefits given by the search orientation provided by the evolutionary search method, we have not found a dominance of the genetic-based approach in comparison to a pure random search method. By observing the metrics results above, we face similar results on both methods, with no significant differences. Therefore, we conclude that the evolutionary search method provides no advantage over a pure random search method in the

context of a large number of hyperparameters optimization and a limited computing capability.

It is important to note that the dominance of the manual search method may not be valid for optimizing a small combination of parameters. Everything else defined, we believe the random and evolutionary search methods are relevant to optimize more specific combinations of parameters, e.g. only the number of hidden layers and activation functions. This is also applicable to the non-dominance of the evolutionary over the random — in the context of a smaller hyperspace of parameters, the direction given by the genetic-based approach may provide interesting results over a pure random.

Considering we are optimizing practically all the hyperparameters available in the neural network, the increase in the learning algorithm complexity makes it prone to overfitting the data, as discussed previously on chapter 2. Such tendency must be counteracted with significant regularization and/or taking into account multiple metrics at the same time during training, to guarantee generalization capacity. Despite estimating the generalization capacity through cross-validation, the automatic search methods only consider one metric at a time which will decide eventually decide the 'best' model. This shortsightedness of the evolutionary and random methods reflects poorly on the results obtained above, as we observed better generalization results for the manual search method which allows taking into consideration multiple metrics and visualization techniques during training. Our results ultimately defend human interaction and the value of accumulated knowledge for optimizing a learning algorithm with large number of hyperparameters. It is relevant to note the possibility that all the accumulated knowledge gathered to produce this document was eventually beneficial to the good results provided by the manual implementation.

A possible reason for the fact that the obtained results do not indicate a dominance of the Evolutionary over the Random method is that the GA conditions may not be properly adequate to assess its performance. Both the number of generations and the size of their population may be too small to observe a proper evolution in the results. Also, the size of the tournament may also be very high (reaching 75% of the population in some cases). This may cause the genetic search method to unsuitably search the parameters' hyperspace. Such possibility is paramount to take into consideration the results obtained in the context of a limited computing capability.

A potential advantage of the manual search method is the human awareness of the state-of-the-art techniques (e.g. Batch Normalization, Dropout and so forth), which are theoretically expected to deliver better results to the algorithm. As such, the manual search can be initiated on such expectably more suitable hyperparameters, contrarily to the randomly-initiated automatic search methods which may be initiated far from their optimal and, in a limited capacity context, they might not have the

conditions to effectively search the hyperspace.
It is relevant to note that our conclusions of a dominance of the manual search method against random and evolution algorithms are opposed to the common literature. This may be due to the constraints and context we have developed our work upon.

9.2 Summary of Thesis Achievements

We found the State of the Art techniques, originally discussed on chapter 3, to be useful and relevant for the deployment of our application. The regularizing properties of the presented techniques — Dropout, Batch Normalization and SNNs — in parallel with the available programming capabilities, were successfully implemented throughout the present study with relative ease and speed.

Some of the good results obtained give feasibility to the profitability of a Bond trading algorithm, based on a machine learning algorithm. We thus conclude that, without any privileged information besides market data, one can take full advantage of the available data to correctly predict, to some extent, the future behavior of such financial securities.

The autoencoder dimensionality reduction technique revealed to be useful for accelerating the training and hyperparameters optimization search processes, while maintaining sufficient data knowledge which allowed the classification learning algorithm to effectively define the decision boundaries with good results on the out-of-sample dataset. We thus conclude on the validity and usefulness of constructing a neural network for unsupervised data dimensionality reduction.

The complex error hyperspace faced by multi-parametered neural networks provide dangerous suboptimal solutions traps which shortsighted optimization techniques, that only consider one metric at a time in our context, may fall into. Such vulnerability provides an interesting opportunity for a manual search, which can take at the same multiple metrics and visualization techniques into account.

Despite the absence of feature engineering, the neural networks learning algorithm revealed aptitude to capture the intrinsic relationships underlying the data. The positive results obtained confirm the algorithm's complex capacity and demonstrate its potential as both a dimensionality reduction technique and a classifier.

A main acknowledgment of our work, if not the most relevant, is the importance of human supervision and guidance throughout the implementation of a machine learning model. A simpler choice of

defining a range of possible values for each parameter and then using an automatic hyperparameters combinations optimization technique presented worse results than manually inputing such parameters. There is an exploding number of combinations generated by having optimized automatically all (or the great majority of) the neural network hyperparameters, which may thwart the automatic search process. As such, arises the importance of human knowledge, both in orienting towards state of the art hyperparemeters and on controlling the training scheme.

9.3 Applications

A main concern of our work is to orient future practitioners on how to quickly and efficiently deploy a learning algorithm which suffers from the curse of dimensionality on its parameters, as is the case of neural networks whose number of parameters, due to the modularity of its techniques, can be easily extended. The applications of our work are dual to the fields of Economics and Data Science, with specialization in the subfields of Financial Markets and Machine Learning, respectively. Considering we focused on also providing intuition for deploying a learning algorithm as a tool *per se*, we intend to also broaden some of our conclusions for all the fields in which the machine learning algorithms are applicable.

Regarding the financial sector, we have provided reasonable doubt for the applicability of a trading algorithm focused on Bonds. The results found by the optimization search methods have provided viable claims for the models profitability in a real-world scenario.

For Machine Learning purposes, we recommend the experienced reader to manually input its accumulated knowledge on optimizing a large number of hyperparameters, as we found such technique to be preferential than automatic search methods (evolutionary and random) on high dimensionality context. Also, for a future practitioner that does not have experience or knowledge on the subject matter, we advise for the choice of a learning algorithm simpler than neural networks, or for the automatic optimization of a smaller number of hyperparameters.

9.4 Future Work

Throughout the development of our work, we have decided to test a blunt approach on handling the original dataset, as no feature engineering was performed (mainly for simplicity and quickness

of deployment) which was eventually foregone due to the good results obtained. Despite using the autoencoder as a dimensionality reduction technique, which eventually compressed and preprocessed some of the data, we believe some changes on the original dataset might provide an enhancement of the overall capacity of the learning algorithm. By exploring deeper insights in the original dataset trough Exploratory Data Analysis (a step of the previously discussed KDD on section 5.1), such knowledge of the dataset features may be advantageous.

An interesting aspect of working with neural networks is the concept of the learning algorithm as a 'black-box', for which we cannot easily analyze feature importances or how the model classifies, considering classification algorithms work on creating decision boundaries within hyperspace whose dimensionality is created by the amount of features of the dataset. An interest aspect of simpler classifiers, such as the Logistic Regression used as baseline, or other ensemble models, such as the Random Forest used for Statistical Analysis, is the possibility to get insights from the parameters tuned in the training process. Considering this can be useful to understand the decisions made by the model, it is our belief some future work on comprehending the decision boundaries constructed by neural networks is of extreme relevance.

For simplicity, we have not deeply studied the impacts of the practical details of deploying the learning model, such as the impact of the weekday in which we rotate our portfolio or on how the rotation schedule (e.g. weekly, biweekly, monthly and so on) influences the profit result. Such details are critical to the practical application and we believe they can thwart or exponentiate the final result. Hence, we advise for a further study prior to the live deployment of the algorithm.

Appendix A

Appendix

This chapter serves to present mathematical formulations and other demonstrations which may not be immediately perceptible to the reader.

A.1 Mathematics and Statistics

A.1.1 Bernoulli Distribution

The Bernoulli Distribution is a discrete distribution with two possible outcomes - $x = 0$ with $1 - p$ probability and $x = 1$ with p probability. We say that X follows a Bernoulli distribution if :

$$P(x) = p^x * (1 - p)^{1-x} \tag{A.1}$$

or, equivalently,

$$P(x) = \begin{cases} 1 - p & \text{x} = 0 \\ p & \text{x} = 1 \end{cases} \tag{A.2}$$

A.1.2 Hyperbolic Secant

The hyperbolic secant function is an argument of the (Njikam and Zhao, 2016) ReSech activation function. It can be defined as:

$$sech(x) = \frac{1}{cosh(x)}, cosh(x) = \frac{e^x + e^{-x}}{2}, sech(x) = \frac{2}{e^x + e^{-x}} \tag{A.3}$$

A.1.3 Sample vs Population Standard Deviations

The standard deviation aims to measure the spread of a distribution's values. This can be calculated using the population, if we know all of its elements, or with a sample. The distinction is made on the denominator, which is lower on the sample's standard deviation to create an higher deviation value, that considers this uncertainty. Both calculations are detailed below:

$$\sigma_p = \sqrt{\frac{\sum (X - \mu)^2}{n}}$$

$$\sigma_s = \sqrt{\frac{\sum (X - \mu)^2}{n - 1}}$$

where σ_p and σ_s are the population and sample standard deviations, X is a vector of values, μ is their mean and n its length (number of examples).

A.2 Metrics Surface Plots

To allow a better comprehension of the relationships between the combined metrics (which use more than one metric) to their simpler components, we have plotted their surfaces. Thus, we demonstrate how the simpler metrics, on the x- and y-axis, can influence the combined metrics value, on the z-axis.

Discriminant Power

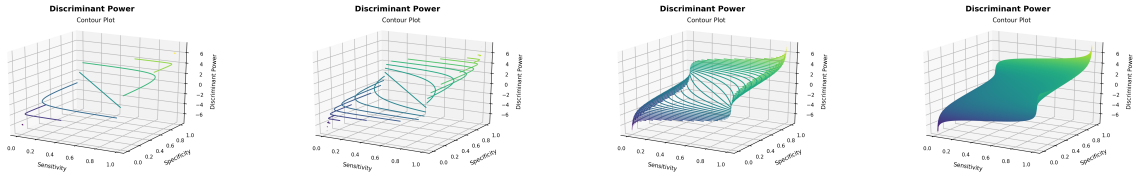

Figure A.1: Contour Plots of the Discriminant Power metric.

F-Measure

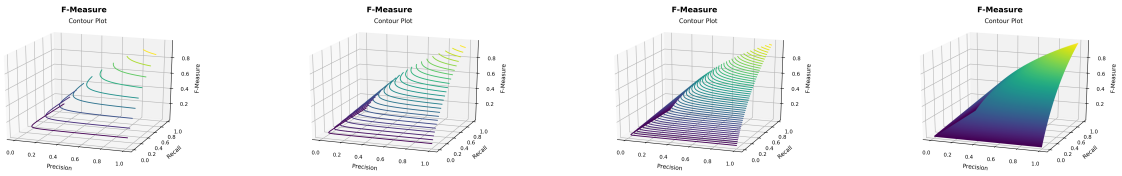

Figure A.2: Contour Plots of the F-Measure metric.

Youden's Index

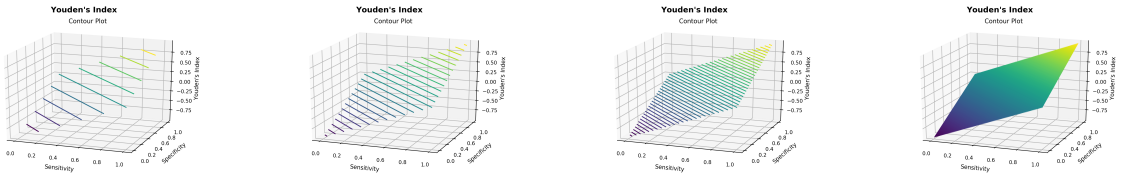

Figure A.3: Contour Plots of the Youden's Index metric.

Bibliography

Barrow, D. K. & Crone, S. F. (2013, August). Crogging (cross-validation aggregation) for forecasting: a novel algorithm of neural network ensembles on time series subsamples. In *The 2013 international joint conference on neural networks (ijcnn)* (pp. 1–8). doi:10.1109/IJCNN.2013.6706740

Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning* (pp. 41–48). ICML '09. Montreal, Quebec, Canada: ACM. doi:10.1145/1553374.1553380

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems 24* (pp. 2546–2554). Retrieved from http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

Bergstra, J. & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, *31*, 281–305.

Caruana, R., Lawrence, S., & Giles, L. (2000). Overfitting in neural nets: backpropagation, conjugate gradient, and early stopping. In *Proceedings of the 13th international conference on neural information processing systems* (pp. 381–387). NIPS'00. Denver, CO: MIT Press. Retrieved from http://dl.acm.org/citation.cfm?id=3008751.3008807

Chang, E. I. & Lippman, R. P. (1991). Using genetic algorithms to improve pattern classification performance. In *Advances in neural information processing systems* (Vol. 3, pp. 797–803). Retrieved from http://papers.nips.cc/paper/381-using-genetic-algorithms-to-improve-pattern-classification-performance.pdf

Chang, Y. (2014, March). *Graph embedding and extensions: a general framework for dimensionality reduction.* Department of ECE, Northeastern University. Retrieved from http://www1.ece.neu.edu/~ychang/notes/dim_reduction.pdf

Chang, Y.-W. & Lin, C.-J. (2008, March). Feature ranking using linear svm. In I. Guyon, C. Aliferis, G. Cooper, A. Elisseeff, J.-P. Pellet, P. Spirtes, & A. Statnikov (Eds.), *Proceedings of the workshop*

on the causation and prediction challenge at wcci 2008 (Vol. 3, pp. 53–64). Proceedings of Machine Learning Research. Hong Kong: PMLR. Retrieved from http://proceedings.mlr.press/v3/chang08a.html

Chollet, F. et al. (2015). Keras. https://github.com/fchollet/keras. GitHub.

Cox, D. R. (1958). The regression analysis of binary sequences (with discussion). *Journal of the Royal Statistical Society*. B (Methodological), *20*, 215–242.

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002, April). A fast and elitist multiobjective genetic algorithm: nsga-ii. *IEEE Transactions on Evolutionary Computation*, *6*(2), 182–197. doi:10.1109/4235.996017

Ding, D., Fu, Q., & So, J. (2012). Pricing callable bonds based on monte carlo simulation techniques. *Technology and Investment*, *3*, 121–125.

Domingos, P. (2012, October). A few useful things to know about machine learning. *Commun. ACM*, *55*(10), 78–87. doi:10.1145/2347736.2347755

Dugas, C., Bengio, Y., Bélisle, F., Nadeau, C., & Garcia, R. (2000). Incorporating second-order functional knowledge for better option pricing. In *Proceedings of the 13th international conference on neural information processing systems* (pp. 451–457). NIPS'00. Denver, CO: MIT Press. Retrieved from http://dl.acm.org/citation.cfm?id=3008751.3008817

Fawcett, T. (2006). An introduction to roc analysis. *Pattern Recognition Letters*, *27*(8), 861–874. ROC Analysis in Pattern Recognition. doi:https://doi.org/10.1016/j.patrec.2005.10.010

Fayyad, U., Piatetsky-Shapiro, G., & Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI magazine*, *17*(3), 37.

Fitkov-Norris, E., Vahid, S., & Hand, C. (2012). Evaluating the impact of categorical data encoding and scaling on neural network classification performance: the case of repeat consumption of identical cultural goods. In C. Jayne, S. Yue, & L. Iliadis (Eds.), *Engineering applications of neural networks: 13th international conference, eann 2012, london, uk, september 20-23, 2012. proceedings* (pp. 343–352). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-32909-8_35

Fonseca, F. (2017a). Abnormal activations. https://github.com/UrbanoFonseca/abnormal_activations. GitHub.

Fonseca, F. (2017b). Confusion matrix cross validation. https://github.com/UrbanoFonseca/confusion_matrix_cv. GitHub.

Fonseca, F. (2017c). Normalizator: python package for normalization of continuous variables. https://github.com/UrbanoFonseca/Normalizator. GitHub.

Fonseca, P. (1995, July). Neural networks: a survey.

Fortin, F., De Rainville, F., Gardner, M., Parizeau, M., & Gagné, C. (2012, July). DEAP: evolutionary algorithms made easy. *Journal of Machine Learning Research*, *13*, 2171–2175.

Genuer, R., Poggi, J.-M., & Tuleau-Malot, C. (2010, October). Variable selection using random forests. *Pattern Recogn. Lett. 31*(14), 2225–2236. doi:10.1016/j.patrec.2010.03.014

Geurts, P., Ernst, D., & Wehenkel, L. (2006, April). Extremely randomized trees. *Machine Learning*, *63*(1), 3–42. doi:10.1007/s10994-006-6226-1

Glorot, X. & Bengio, Y. (2010, May). Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh & M. Titterington (Eds.), *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (Vol. 9, pp. 249–256). Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR. Retrieved from http://proceedings.mlr.press/v9/glorot10a.html

Golik, P., Doetsch, P., & Ney, H. (2013). Cross-entropy vs. squared error training: a theoretical and experimental comparison. In *Interspeech*.

Goodfellow, I. J., Bengio, Y., & Courville, A. (2016). *Deep learning*. http://www.deeplearningbook.org. MIT Press.

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., & Bengio, Y. (2013, June). Maxout networks. In S. Dasgupta & D. McAllester (Eds.), *Proceedings of the 30th international conference on machine learning* (Vol. 28, *3*, pp. 1319–1327). Proceedings of Machine Learning Research. Atlanta, Georgia, USA: PMLR. Retrieved from http://proceedings.mlr.press/v28/goodfellow13.html

Guyon, I. & Elisseeff, A. (2003, March). An introduction to variable and feature selection. *J. Mach. Learn. Res. 3*, 1157–1182. Retrieved from http://dl.acm.org/citation.cfm?id=944919.944968

Haixiang, G., Yijing, L., Shang, J., Myngyun, G., & Yuanyue, H. (2017). Learning from class-imbalanced data: review of methods and applications. *Expert Systems with Applications*, *73*, 220–239.

Hampel, F., Rousseeuw, P., Ronchetti, E., & Stahel, W. (1986). *Robust statistics: the approach based on influence functions*. Wiley.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: surpassing human-level performance on imagenet classification. *CoRR*, *abs/1502.01852*. arXiv: 1502.01852. Retrieved from http://arxiv.org/abs/1502.01852

Heaton, J. (2017). An empirical analysis of feature engineering for predictive modeling. *Computing Research Repository*, *abs/1701.07852*. Retrieved from http://arxiv.org/abs/1701.07852

Hertz, A. & Kobler, D. (2000). A framework for the description of evolutionary algorithms. *European Journal of Operational Research*, *126*(1), 1–12. doi:https://doi.org/10.1016/S0377-2217(99)00435-X

Hinton, G. E. & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, *313*(5786), 504–507. doi:10.1126/science.1127647. eprint: http://science.sciencemag.org/content/313/5786/504.full.pdf

Hsu, C., Chang, C., & Lin, C. (2010). A practical guide to support vector classifier. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.224.4115

Huang, C. & Wang, C. (2006). A ga-based feature selection and parameters optimization for support vector machines. *Expert Systems with Applications*, *31*, 231–240.

Hull, J. (2012). *Options, futures, and other derivatives* (Eight Edition). Prentice Hall.

Hunter, J. D. (2007). Matplotlib: a 2d graphics environment. *Computing In Science & Engineering*, *9*(3), 90–95. doi:10.1109/MCSE.2007.55

Ioffe, S. & Szegey, C. (2015). Batch normalization: accelerating deep network training by reducing internal covariance shift. In *Proceedings of the 32nd international conference on machine learning* (Vol. 37, pp. 448–456). Proceedings of Machine Learning Research.

Jain, A., Nandakumar, K., & Ross, A. (2005). Score normalization in multimodal biometric systems. *Pattern Recognition*, *38*, 2270–2285.

Janocha, K. & Czarnecki, W. M. (2017). On loss functions for deep neural networks in classification. *CoRR*, *abs/1702.05659*. arXiv: 1702.05659. Retrieved from http://arxiv.org/abs/1702.05659

Japkowicz, N. (2000). Learning from imbalanced data sets: a comparison of various strategies. In *Proceedings of the aaai'2000 workshop on learning from imbalaced data sets* (pp. 10–15). AAAI Press.

Jayalakshmi, T. & Santhakumaran, A. (2011). Statistical normalization and back propagation for classification. *International Journal of Computer Theory and Engineering*, *3*(1), 89–93.

Jeni, L. A., Cohn, J. F., & De La Torre, F. (2013). Facing imbalanced data–recommendations for the use of performance metrics. In *Proceedings of the 2013 humaine association conference on affective computing and intelligent interaction* (pp. 245–251). ACII '13. Washington, DC, USA: IEEE Computer Society. doi:10.1109/ACII.2013.47

Jolliffe, I. T. & Cadima, J. (2016). Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, *374*(2065). doi:10.1098/rsta.2015.0202. eprint: http://rsta.royalsocietypublishing.org/content/374/2065/20150202.full.pdf

Kaguara, A., Nam, K. M., & Reddy, S. (2014). A deep neural network classifier for diagnosing sleep apnea from ecg data on smartphones and small embedded systems. Retrieved from https://doi.org/10.13140/2.1.4174.5448

Kim, K. (2006). Artificial neural networks with evolutionary instance selection for financial forecasting. *Expert Systems with Applications*.

Klambauer, G., Unterthiner, T., A, M., & Hochreiter, S. (2017). Self-normalizing neural networks. *CoRR*, *abs/1706.02515*. arXiv: 1706.02515. Retrieved from http://arxiv.org/abs/1706.02515

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th international joint conference on artificial intelligence - volume 2* (pp. 1137–1143). IJCAI'95. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1643031.1643047

Krizhevsky, A., Sutskever, I., & Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems 25* (pp. 1097–1105). Curran Associates, Inc. Retrieved from http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Lai, K. K., Yu, L., Wang, S., & Zhou, L. (2006). Neural network metalearning for credit scoring. In D.-S. Huang, K. Li, & G. W. Irwin (Eds.), *Intelligent computing: international conference on intelligent computing, icic 2006, kunming, china, august 16-19, 2006. proceedings, part i* (pp. 403–408). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/11816157_47

Langley, P. (1994). Selection of relevant features in machine learning. In *In proceedings of the aaai fall symposium on relevance* (pp. 140–144). AAAI Press.

Latha, L. & Thangasamy, S. (2011). Efficient approach to normalization of multimodal biometric scores. *International Journal of Computer Applications*.

LeCun, Y. (1989). Generalization and network design strategies. In R. Pfeifer, Z. Schreter, F. Fogelman, & L. Steels (Eds.), *Connectionism in perspective*. Elsevier.

LeCun, Y., Bottou, L., Orr, G., & Müller, K. (1998). Efficient backprop. In Springer (Ed.), *Neural networks: tricks of the trade* (pp. 9–48).

Liu, Y., Starzyk, J. A., & Zhu, Z. (2008, June). Optimized approximation algorithm in neural networks without overfitting. *IEEE Transactions on Neural Networks*, *19*(6), 983–995. doi:10.1109/TNN.2007.915114

Lu, C., De Brabanter, J., Van Huffel, S., Vergote, I., & Timmerman, D. (2001). Using artificial neural networks to predict malignancy of ovarian tumors. In *Engineering in medicine and biology society, 2001. proceedings of the 23rd annual international conference of the ieee* (Vol. 2, pp. 1637–1640). IEEE.

Martellini, L., Priaulet, P., & Priaulet, S. (2003). *Fixed income securities: Valuation, risk management and portfolio strategies*. John Wiley & Sons Ltd.

McKinney, W. (2010). Data structures for statistical computing in python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th python in science conference* (pp. 51–56).

Mojarad, S. A., Dlay, S. S., & Sherbet, G. V. (2011). Cross validation evaluation for breast cancer prediction using multilayer perceptron neural networks. (Vol. 4, pp. 576–585).

Murphey, Y., Guo, H., & Feldkamp, L. (2004). Neural learning from unbalanced data. *Applied Intelligence*, *21*(2), 117–128.

Neelakantan, A., Vilnis, L., Le, Q. V., Sutskever, I., Kaiser, L., Kurach, K., & Martens, J. (2015). Adding gradient noise improves learning for very deep networks. *CoRR*, *abs/1511.06807*. arXiv: 1511.06807. Retrieved from http://arxiv.org/abs/1511.06807

Ng, A. (2011). Cs294a lecture notes. Retrieved from https://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf

Njikam, A. & Zhao, H. (2016). A novel activation function for multilayer feed-forward neural networks. *Applied Intelligence*, *45*(1), 75–82. doi:10.1007/s10489-015-0744-0

Parascandolo, G., Huttunen, H., & Virtanen, T. (2017). Taming the waves: sine as activation function in deep neural networks.

Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *Proceedings of the 30th international conference on international conference on machine learning - volume 28* (pp. III-1310–III-1318). ICML'13. Atlanta, GA, USA: JMLR.org. Retrieved from http://dl.acm.org/citation.cfm?id=3042817.3043083

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Piotrowski, A. P. & Napiorkowski, J. J. (2013). A comparison of methods to avoid overfitting in neural networks training in the case of catchment runoff modelling. *Journal of Hydrology*, *476*(Supplement C), 97–111. doi:https://doi.org/10.1016/j.jhydrol.2012.10.019

Potdar, K., Pardawala, T. S., & Pai, C. D. (2017, October). A comparative study of categorical variable encoding techniques for neural network classifiers. *International Journal of Computer Applications*, *175*(4), 7–9. doi:10.5120/ijca2017915495

Prechelt, L. (1998). Early stopping - but when? In G. B. Orr & K. Müller (Eds.), *Neural networks: tricks of the trade* (pp. 55–69). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/3-540-49430-8_3

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *CoRR*, *abs/1609.04747*. arXiv: 1609.04747. Retrieved from http://arxiv.org/abs/1609.04747

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Parallel distributed processing: explorations in the microstructure of cognition, vol. 1. In D. E. Rumelhart, J. L. McClelland, & C. PDP Research Group (Eds.), (Chap. Learning Internal Representations by Error Propagation, pp. 318–362). Cambridge, MA, USA: MIT Press. Retrieved from http://dl.acm.org/citation.cfm?id=104279.104293

Saxe, A. M., McClelland, J. L., & Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *CoRR*, *abs/1312.6120*. arXiv: 1312.6120. Retrieved from http://arxiv.org/abs/1312.6120

Shen, S., Jiang, H., & Zhang, T. (2012). Stock market forecasting using machine learning algorithms. Retrieved from http://cs229.stanford.edu/proj2012/ShenJiangZhang-StockMarketForecastingusingMachineL pdf

sklearn-deap. (2017). https://github.com/rsteca/sklearn-deap. GitHub.

Sokolova, M., Japkowicz, N., & Szpakowicz, S. (2006). Beyond accuracy, f-score and roc: a family of discriminant measures for performance evaluation. In A. Sattar & B.-h. Kang (Eds.), *Ai 2006: advances in artificial intelligence: 19th australian joint conference on artificial intelligence, hobart, australia, december 4-8, 2006. proceedings* (pp. 1015–1021). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/11941439_114

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, (15), 1929–1958.

Stanley, K. O. & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99–127. Retrieved from http://nn.cs.utexas.edu/?stanley:ec02

Tahir, M., Kittler, J., Mikolajczyk, K., & Yan, F. (2009). A multiple expert approach to the class imbalance problem using inverse random under sampling. (pp. 82–91).

van der Maaten, L. & Hinton, G. (2008). Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, *9*, 2579–2605.

van Laarhoven, T. (2017). L2 regularization versus batch and weight normalization. *CoRR*, *abs/1706.05350*. arXiv: 1706.05350. Retrieved from http://arxiv.org/abs/1706.05350

Wager, S., Wang, S., & Liang, P. S. (2013). Dropout training as adaptive regularization. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems 26* (pp. 351–359). Curran Associates, Inc. Retrieved from http://papers.nips.cc/paper/4882-dropout-training-as-adaptive-regularization.pdf

Walt, S., Colbert, S. C., & Varoquaux, G. (2011, March). The numpy array: a structure for efficient numerical computation. *Computing in Science and Engg. 13*(2), 22–30. doi:10.1109/MCSE.2011.37

Wang, W., Huang, Y., Wang, Y., & Wang, L. (2014). Generalized autoencoder: a neural network framework for dimensionality reduction. In *Proceedings of the 2014 ieee conference on computer vision and pattern recognition workshops* (pp. 496–503). CVPRW '14. Washington, DC, USA: IEEE Computer Society. doi:10.1109/CVPRW.2014.79

Xu, B., Huang, R., & Li, M. (2016). Revise saturated activation functions. *CoRR*, *abs/1602.05980*. arXiv: 1602.05980. Retrieved from http://arxiv.org/abs/1602.05980

Yao, X. (1993). A review of evolutionary artificial neural networks. *International Journal of Intelligent Systems*, *4*, 539–567.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, *87*(9), 1423–1447.

Youden, W. J. (1950). Index for rating diagnostic tests. *Cancer*, *3*(1), 32–35. doi:10.1002/1097-0142(1950)3:1⟨32::AID-CNCR2820030106⟩3.0.CO;2-3

Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2016). Understanding deep learning requires rethinking generalization. *CoRR*, *abs/1611.03530*. arXiv: 1611.03530. Retrieved from http://arxiv.org/abs/1611.03530

Zur, R. M., Jiang, Y., Pesce, L. L., & Drukker, K. (2009). Noise injection for training artificial neural networks: a comparison with weight decay and early stopping. *Medical Physics*, *36*(10), 4810–4818. doi:10.1118/1.3213517