UNIVERSITÀ DEGLI STUDI DI TORINO

SCUOLA DI MANAGEMENT ED ECONOMIA

FINANZA AZIENDALE E MERCATI FINANZIARI

ANALYSIS AND PERFORMANCE OF A NEURAL NETWORK BASED TRADING ALGORITHM

Francesco Lomio

Supervisor: Prof. Pietro Terna

Counter Examiner: Prof. Sergio Margarita

Academic Year

2016/2017

To my family
and loved
ones

A big thank you goes to the ESCP Europe Business School, for allowing me to spend the last 3 years in such a stimulating and international environment, as well as for providing me the tools used to obtain the data needed for my thesis.

Abstract

In this work, after defining *algorithmic trading* and the features of a *neural network*, I built a simple trading algorithm which uses a *Feedforward* neural network to forecast the USD/GBP exchange rate.

I used it to predict the data for two different time periods, all referring to the year 2016, one characterised by low volatility and one characterised by high volatility. In this case, the latter corresponds to June 2016, during the Brexit referendum.

For both the periods, the performance of the neural network has been tested for different train and test set configurations as well as for different time horizons (to check how far in time it can give a reasonable prediction).

The forecasted data have been used in a simple trading algorithm in order to decide whether to take a long or short position on USD/GBP. In this way I could confront the profit of such strategy over the two periods described before. The simulation has been performed under two hypothesis: the first allows the algorithm to open a position for every data point, while the second allows it to open a position only when the difference of a forecasted price is greater than the previous data by at least a certain threshold.

Contents

Part 2: The Model

Part 3: Results

List of Figures

Introduction

The use of algorithms in financial markets has grown exponentially in the past years. These algorithms have become more and more complex over time, thanks to a more powerful technology and the need to keep up with the ever faster trading environment. Most of the algorithms are used in order to take investment decisions based on the estimated future price of the asset. Amongst these, we can find the trend following, pair trading and High Frequency Trading algorithms, just to cite some.

The most recent use another "innovative" mathematical tool: the neural networks. These are networks that model the functioning of the neurons in human brain and, if correctly implemented, after learning important features from the past data, can be used to forecast the future prices of an asset.

The aim of my thesis work is to verify if an algorithm based on predictive neural network can be used, and are profitable, in period of high uncertainty, specifically during the period following the UK's EU membership referendum held on 23 June 2016.

To do so, I used the data of the USD/GBP exchange rate relative to the year 2016 (collected from Bloomberg Terminal at ESCP Europe Paris Campus), and I individuated two different periods, one of low volatility and one of high volatility, and I built a model based on a neural network to forecast the data related to these two periods.

Once I obtained the data, I simulated two investment strategies based on the forecasted data, to verify if this approach could be profitable regardless of the volatility of the period analysed.

Literature Review

With the neural networks becoming more and more common for constructing trading algorithms, a range of academic articles have been published on the subject. Unfortunately, even though it is a fast-growing topic, most of the research is based on analysing how diverse

types of neural network perform when forecasting financial time series. Following are some of the articles that I found useful for my research.

Levendovszky and Kia [6], in their paper on *Prediction Based - High Frequency Trading on Financial Time Series*, showed that through the use of a simple Feedforward neural network, it can be created an algorithm capable of forecasting financial time series and taking investment decision based on the forecasted data.

Other research did the same, sometimes being able to build algorithm with an accuracy of over 60%. Arévalo, Niño, Herández and Sandoval [7], in their work on *High-Frequency Trading Strategy Based on Deep Neural Network*, showed that it is possible, through the use of a more complex deep neural network, to build a trading algorithm which is capable of yielding 81% successful trades, with an accuracy of 66%.

Due the multiplicity of neural networks available at the moment, some other works have been done in order to analyse the performances of diverse types of neural network. Guresen, Kayakutlu and Daim [8], in their paper *Using Artificial Neural Network Models in Stock Market Index Prediction*, compared dynamic artificial neural networks, multi-layer perceptron and hybrid networks in order to evaluate their performances in predicting stock market indices.

Even if the literature is full of articles related to the use of neural networks in financial markets, I wasn't able to find works related to the analysis of the performance of a neural network used to make prediction in high volatile periods, hence the subject of my thesis.

THE CONTEXT

My analysis, will be carried during a particular moment in time, the Brexit, a period in which the markets turmoiled and experienced unexpected (and unpredicted) movements, which led the main indices as well as the currency pairs to "jump" minutes after the result of the referendum were announced. In particular, the GBP went down more than 10% against the USD at $1.33, a 31-year record low, as soon as the polling stations closed [9]. We can see in figure 1 the sudden movement of the USD/GBP currency pair, minutes after the results became public.

Figure 1: The USD gained 10% in the moments after the announcement of the referendum results (source: Bloomberg Terminal at ESCP Europe, Paris campus)

Thesis Structure

In my thesis, the topic analysed, will be discussed as described below.

- In *Chapter 1*, I will introduce the topic of algorithmic trading, giving a brief history of how computers and software have influenced financial markets since the 1970*s*, to later explain some of the most famous and used trading algorithm.
- In *Chapter 2*, I will give a deeper introduction on the neural networks. After giving a brief history of their development, I will present some network architectures, focusing especially on the *feedforward*, and its characteristics, being the neural network used in my model.
- In *Chapter 3*, I will present the model I built, explaining how it was made, how it works and giving a brief description of the code used.
- In *Chapter 4*, I will introduce how the investment was simulated based on the results obtained thanks to my model explained in chapter 3, giving a brief description of the code used to implement it.
- In *Chapter 5*, I will describe the analysis of the results of my model. This chapter will be divided into three parts: first I will show the results obtained when forecasting the data in periods of high and low volatility; secondly I will show how these results change

when the data set used to test the algorithm is part of the data set used to train it. Finally I will give the results obtained when forecasting the exchange rate at different time horizons.

- In *Chapter 6*, I will present the results obtained from the investment simulation described in chapter 4. The results will be divided in multiple parts, in order to show how the investment simulation behaved according to the volatility of the data considered and to the time horizon of the prediction used.

Moreover, in *Appendix A* we find the whole code I wrote, described in Chapter 3 and Chapter 4, while in *Appendix B* we can find the results of a preliminary analysis made on the same data.

Theory

Chapter 1

Algorithmic Trading: Brief History and Most Used Algorithms

Before moving further in my thesis subject, I find it useful to define the context in which the neural networks are used: algorithmic trading. Also known as automated trading, “algo trading” is nothing but a specific set of rules made to carry various operations in the daily task of a trader.

In this chapter, we are going to see more specifically what it is meant by algorithmic trading, how they influence the markets and various example of algorithms currently used by institutional and retail traders.

1.1 Definition and some history

As said before, algorithmic trading is about giving a computer a set of well define instructions, based on mathematical assumption, in order to carry activities at with a speed and an accuracy which would be unthinkable for human being. In this way, complex analysis and decision can be made in a fraction of a second, simplifying the life of the traders, and taking away the emotions related to the human nature of the decisions.

Contrary at what one may think, computers for trading purposes have been used since at least the ’70s, when the New York Stock Exchange started using an innovative order transmission system called *DOT* (Designated Order Turnaround). With this system, the orders were transmitted electronically to traders, who then executed them manually. This system was later substituted by the *SuperDOT*, who was automatically transmitting market order and limit order to the traders, bypassing the need of the brokers to pass the same informations.

Another step ahead in the use of computers and algorithms in the markets was in the ’80s,

when *program trading* became vastly used for trading in the $S\&P500$ Equity and Futures markets. Program trading allowed traders to automatically place orders, becoming more and more attractive for institutional investors aiming at carrying some simple arbitrage strategies. As for the last report published by the *NYSE* in 2016, program trading still accounted for the 20.5% of the total buy-sell volume on the *NYSE* itself for the trading week of the 25 to 29 April 2016 (this is the last available report published by the *NYSE* on program trading statistics).

The advances of electronic execution programs et simila in the '80s and '90s, and the following decimalisation of the USD, led to a reduction of the bid-ask spread, and therefore an increase liquidity in the market. For this reason, many investors felt the urge to build system capable of splitting up orders to take advantage of better average prices.

The real turning point for algorithmic trading came in 2001, when an important research paper [10] was published showing how two simple algorithmic strategies were outperforming human traders.

With the introduction of other electronic markets, the algorithms have become more complex and more common amongst institutional investors as well as retail investors, creating many different algorithm based strategy which account for more than 75% of the total exchanged volume in the financial markets [11].

1.2 Algorithmic Trading Strategies

Let's now have a look at the most important and diffuse strategies which can be implement though the use of algorithms. We can divide these strategies into two categories:

- in the first one, we'll describe three of the most common strategies: Trend Following, VWAP and Pairs Trading;
- in the second we'll describe the more sophisticated strategies, like High Frequency Trading and Price Predicting strategies.

1.2.1 Trend Following

Trend Following is one of the most common strategy which can be implement through algorithms. This kind of analysis relies only on the long-term market trend, and not on any external information or forecast.

Once a clear trend has been identified by the algorithm, it strictly adheres to the rules specified by the program, and it follows them until the trend changes or another trend has

been recognised. In some of the most basic trend following algorithms, the moment in which the program will buy or sell are identified using the *moving average*: this is nothing but the mean of the previous n data available. The set of n data is then shifted forward as time goes on, in order to have a complete set of data which can be used to understand a trend. The reason for which moving average is used is that it allows to smooth the fluctuation in the data due to short-term event. When the Moving Average (which can be weighted, meaning that some data weight more than others) crosses above or below a certain threshold, the algorithm triggers the buy or sell instruction.

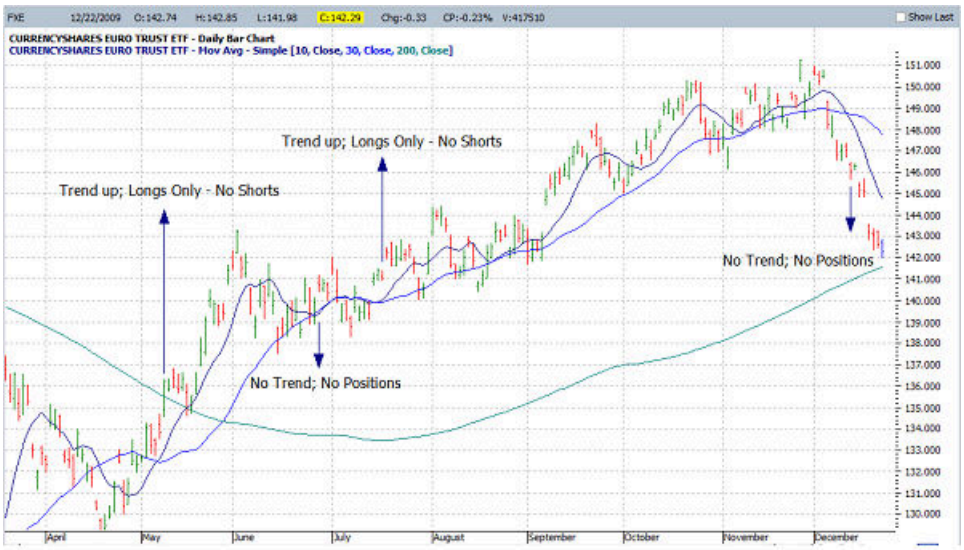


Figure 1.1: Example of trend following strategy [1]

In figure 1.1 we can see an example of trend following strategy. In this case, we have an algorithm which detects an up-trend every time that the 10-day moving average goes above the 30-day moving average, and at the same time the latest close is placed above the 200-day moving average. The algorithm tells than in this case, we should open a long position on the underlying. In the same way, it detects a down-trend when the 10-day moving average is below the 30-day moving average and when the 200-day moving average is above the latest close. As before, the algorithm suggests that a short position must be open on the underlying.

1.2.2 VWAP

Another very commonly used algorithm is the VWAP, *Volume Weighted Average Price*. The aim of this indicator is to highlight the moment in which the liquidity is high: it provides a measure of the price weighted with the volume.

The algorithms based on this indicator have the purpose of dividing a large order in multiple, small orders: once the trader has decided the time intervals in which to place the orders, the algorithm, taking into account historical and real-time volume, decides their size, in order to take advantage of better prices.
The use of the VWAP allows traders to save time and cost when dealing with large size orders. Slicing it in many smaller orders, it reduces the impact that a larger size order would have on the market. Moreover, these smaller batches enable the trader to "mask" the real order size to the other participants in the market.

1.2.3 Pair Trading

As Leshik and Cralle say in their book, *An Introduction to Algorithmic Trading*, the *pair trading* strategy is the strategy "which initially provided the main impetus to algorithmic trading". This is a market-neutral strategy, and it depends on the correlation properties of the assets taken into account.

It was originated by a quant at Morgan Stanley in the '80s, when he noticed that some stocks, within the same industry, were showing a very high correlation between with each other. What really is interesting is that he noticed that when the correlation was diminishing, it would eventually return to its original status.

The algorithms based on this strategy, looks at the past in order to find highly correlated assets. These pair of assets are continuously observed to check for a temporary weakening of their correlation. When this happens, the strategy consists in selling the asset which is over-performing and buying the one under-performing. The algorithm will then close the positions once it sees that the correlation is turning back to its original status.

In figure 1.2 we can see one of the most famous stock pairs used for this strategy: *Coca-Cola* and *PepsiCo*. We can see highlighted the moments in which the two stocks temporarily diverge from each other, and therefore their correlation weakens. In this example [2], a good strategy would have been to buy *PepsiCo* at the end of May while selling *Coca-Cola*, for later closing the positions in the second half of June, when the stock returned to their original behaviour.

Figure 1.2: Example of pair trading strategy [2]

1.2.4 High Frequency Trading

High-Frequency Trading, also referred to as HFT, is an algorithmic trading form, whose peculiarities are the highly rate of sophistication of the algorithms and the very short-term investment horizon.

As we can imagine, HFT is a "quantitative" form of trading, which bases its decisions on models that run and update in fractions of a seconds, analysing an extremely large amount of data. Thanks to these characteristic, HFT can be highly versatile and there are diverse types of strategy that can be put in place with high frequency trading. One of these strategies is *Statistical Arbitrage*. Being able to analyse large amount of data in very short time, it is possible for HFT algorithms to find some statistical relationship between securities of any asset class. Once the relationship has been identified, the algorithm can arbitrate taking advantage of temporary deviation from these relationships.

Another strategy very common in High-Frequency Trading are the *Low Latency Strategies*, meaning that these strategies are implemented in the market in a time period express in terms of milliseconds [12]. This strategy is based upon the fact the HFT firms have usually direct access to the market, which allow them to gain an advantage by, for example, allowing to arbitrate a price discrepancy. To allow this kind of strategy, it is important that the algorithm first, and all the other steps involved in receiving and transmitting the information, run extremely fast. For this reason, often the server running the HFT algorithm are located as close as possible to the exchange venue's servers. In this way, the time that the information takes to go from one to the other is drastically reduced.

In figure 1.3, we can see the prices of the *SDPR S&P 500 ETF* and the *E-mini S&P 500* futures.

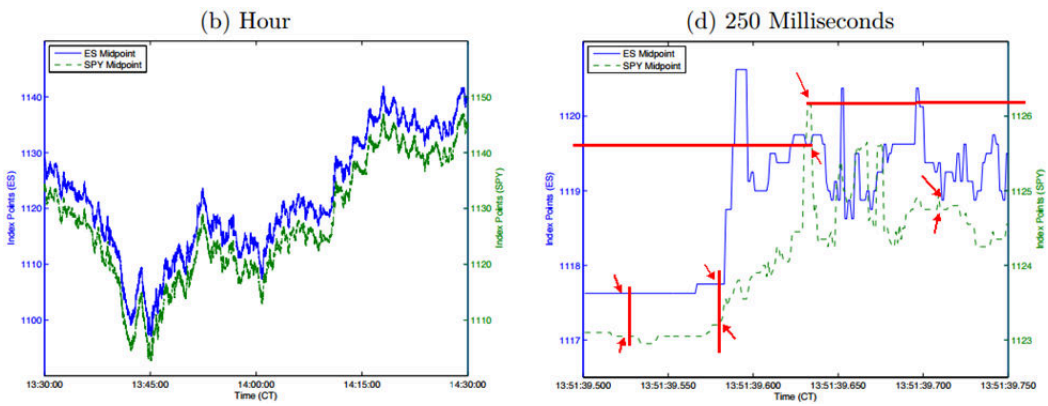


Figure 1.3: Example of High Frequency Trading strategy [3]

What is clear from the hourly chart is that the price of the two are perfectly correlated, not giving any possibility of arbitrage between the prices. When we consider the time frame used by HFT low-latency strategies, we see that this is not true any more: in fact, in the 250-millisecond chart, we can see how the prices of these two assets continuously diverge from each other, creating an arbitrage opportunity only visible by HFT low latency algorithms [10].

1.2.5 Price Forecasting

Many of the strategies presented have as objective the one to predict the direction that the price of an asset will take in the future. The price forecasting techniques are based on the *efficient market hypothesis*. According to this theory, the price of an asset includes all the information available [13]. Based on this assumption, if we know the historical price of an asset, we can predict what its expected future price will be. In order to do so, a range of different techniques has been created, which allow traders to have an idea of how the price of the asset will behave in the future. The most commonly diffused techniques are the *fundamental* and the *technical* analysis. While the first focuses on the intrinsic characteristics of an asset, such as its economic and financial context, the technical analysis focuses on the statistics of the price of the security, like its price and its volume. Both of them try to extrapolate from these factors indications about future price of the asset analysed.

More recent techniques involve the use of sophisticated mathematical tools, one of these are the *neural networks*. These tools, frequently used in machine learning, taking as input the historical price of the asset, can learn from its behaviour important features which can be used to estimate the future price.

In the following chapters, we'll see more in details how a neural network work and how it can be implemented in a trading strategy.

Chapter 2

Neural Networks: What They Are, How They Work and How They Can Be Used.

As we saw in the previous chapter, one of the tools used to build price forecasting trading algorithms are the *Neural Networks*. In the lasts years, these have become vastly used in different disciplines as an analysis and predictive tools. During my work, to build the predictive algorithm, I will use a FeedForward neural network. Therefore, I find it useful to give a brief history and to introduce the basic and most important features of neural networks.

2.1 A BRIEF HISTORY

The firsts researchers introducing the concept of neural networks (NN) were McCulloch and Pitts in 1943 [14], where they stated that because of the "all-of-none" behaviour of neuronal firing, neural activity can be treated with logical calculus and therefore they modelled the first NN using simple electrical circuits in order to try to simulate the way our brain works.

This idea has then been extended, in 1949, by Hebb in his book *The Organization of Behaviour* [15], where he discovered and explained that the strength of neural pathway is enhanced every time these connections are used, a fundamental concept to understand the human learning.

After the advent and the development of computers, B.G. Farley and W.A. Clark [16], in 1954 at MIT, firstly used computational machine to simulate *Hebbian learning* in a neural network. The use of computational features to study neural networks gave birth to the concept of Artificial Neural Networks (ANN).

In 1958, Rosenblatt created the perceptron [17]. This algorithm is still used today as the basis of many types of neural network, and it's mostly used for pattern recognition. The real

turning point came when the perceptron was first used as the basis of machine learning: in 1969, M. Minsky and S. Papert [18], published the book *Perceptrons: An Introduction to Computational Geometry.* This book stated two key issues of computing neural networks: the first was that the simple perceptron was not able to solve the *exclusive-or* (figure 2.1) circuit, and the second was that at that moment there were not enough powerful machines to compute large neural networks.

A	B	output
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.1: Exclusive-Or function: it an output only when the inputs differ from each other

A speed-up to the research was given in 1975, when Werbos in his PhD thesis at the Harvard University [19], defined the *back propagation* (described in detail further) which solved the exclusive-or problem and made possible the computation of fast training multi-layer neural networks.

After the definition of the back propagation, the research in the field of neural networks stopped until 1986, when D. E. Rumelhart and J. McClelland published a text book [20] where they introduced the concept of parallel distributed processing.

In the last decade, thanks to the improvements in parallel computing and GPU-fast computing, the research in neural networks have had a fast advancement and a lot of fast algorithm have been developed, such as recurrent neural network and feed-forward neural networks.

Neural networks are now used in a vast range of fields, from biomedical studies to social behaviour studies, such as people flux in the word or studies on flight routes, to financial and economic studies, such as studies on pattern recognition and price forecasting on trading data-sets.

2.2 Artificial Neural Network

Neural networks models are frequently referred as *Artificial Neural Networks* (ANN). The models are mathematically described by a function or a distribution. The network is characterised by *nodes*, whose biological respective are neurons, and the ANN is the function that describes the connections between the nodes. The synapses are described with the *weight* assigned to the connections.

Let's take the example of the simplest type of neural network, consisting of a single node.

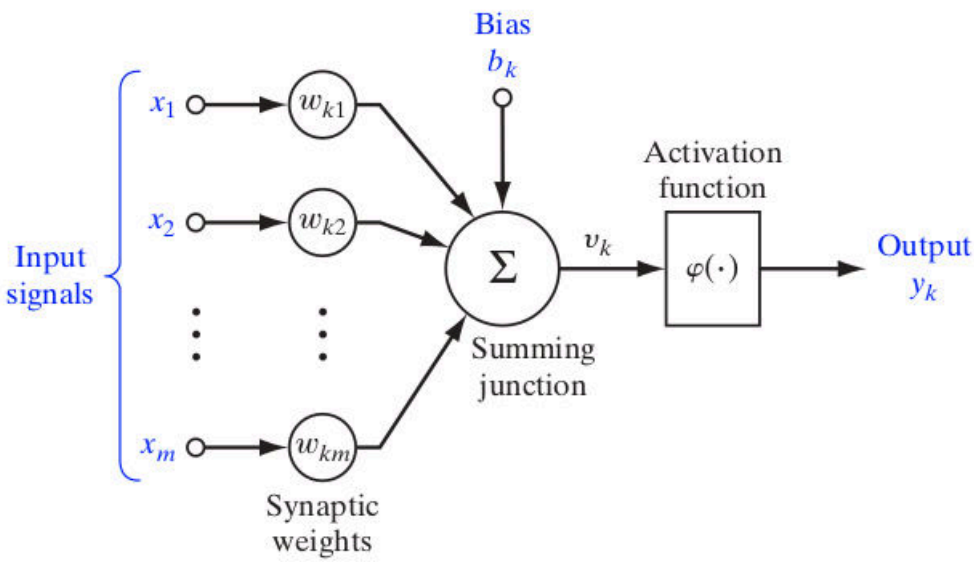


Figure 2.2: Model of a Neuron [4]

We can see in figure 2.2, the various components of a neural network:

- $x_1, ..., x_m$ represents the signals applied to the input of the synapsis;
- $w_1, ..., w_m$ are the weights associated with each synapsis;
- Σ indicates a linear combiner, a junction in which the input signals multiplied by the weights of their respective synapsis, are summed together;
- b_k is a bias term, used to increase or lower the net input (after they are summed). It can be greater, lower or equal to 0;
- $\varphi(\cdot)$ is the activation function which limits the amplitude of the output in order to have a range of finite values;

- y_k is the output produced by the network.

Once the signals x_j have been applied to the inputs of the synapsis, they are multiplied by the corresponding weights w_j and then summed as follow:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j, \tag{2.1}$$

where k indicates the neuron and j the synapsis to which a particular signal and weight correspond. After this operation, u_k is summed with the bias element to form the *activation potential* of the neuron v_k:

$$v_k = u_k + b_k \tag{2.2}$$

This is then used as the argument of the activation function which defines the output of the neuron:

$$y_k = \varphi(v_k). \tag{2.3}$$

2.2.1 Activation Function

We can identify two basic types of activation function $\varphi(v)$:

- Threshold Function;
- Sigmoid Function.

Threshold Function

Commonly referred to as the *Heaviside Function*, the threshold activation function (figure 2.3) is defined as:

$$\varphi(v) = \begin{cases} 1, & \text{if } v \geq 0 \\ 0, & \text{if } v < 0 \end{cases} \tag{2.4}$$

The output of a neural network with a neuron k, utilising a threshold activation function, will therefore be:

$$y_k = \begin{cases} 1, & \text{if } v_k \geq 0 \\ 0, & \text{if } v_k < 0 \end{cases}, \tag{2.5}$$

where v_k is represented by the equation 2.2.

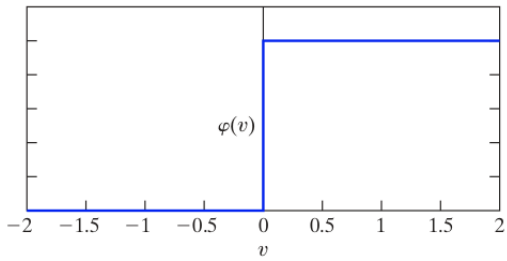


Figure 2.3: Example of Threshold function, also referred to as Heaviside step function [4]

Sigmoid Function

One of the most used activation function, the sigmoid is a differentiable, strictly increasing function, which can assume all the values from 0 to 1. One example of the sigmoid function is the *logistic function* (figure 2.4), described by the following equation:

$$\varphi = \frac{1}{1+e^{av}}, \tag{2.6}$$

where a is a parameter which modifies the slope of the sigmoid function, and v the activation potential.

For some types of neural networks, is desirable having an activation function which can range from -1 to $+1$. For this reason it is sometimes preferable to used the *hyperbolic tangent function* instead of the logistic function:

$$\varphi(v) = \begin{cases} 1, & \text{if } v > 0 \\ 0, & \text{if } v = 0 \\ -1, & \text{if } v < 0 \end{cases}, \qquad \varphi(v) = tanh(v) \tag{2.7}$$

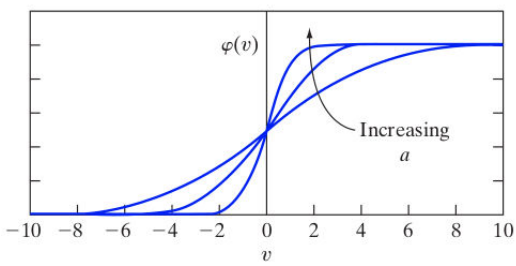


Figure 2.4: Example of a Sigmoid function at different value of a [4]

2.3 NEURAL NETWORK STRUCTURE

Designing an artificial neural network, implies creating a specific set of rules which define the structure of the network. We can identify 3 major types of structure:

- Single layer Feedforward network;
- Multilayer Feedforward network;
- Recurrent network.

For my thesis work I used a Multilayer Feedforward neural network, but I find it useful to briefly show the other types of network available and currently used in related researches.

2.3.1 Single Layer Feedforward

This is the simplest kind of *layered* neural network: the input nodes (non-computation nodes, therefore *single layer*) injects the signals directly into the output layer (computation nodes). The information can only flow in this direction, hence the name of *feedforward*. In figure 2.5, we can see an example showing 4 input nodes fully connected with 4 output nodes.

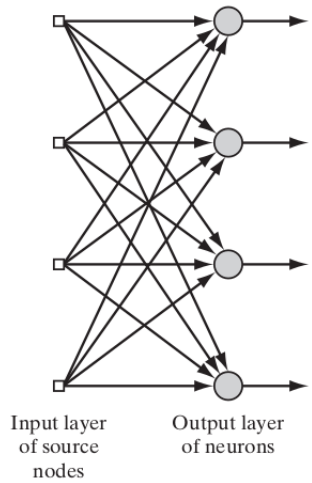


Figure 2.5: Representation of a Single layer Feedforward neural network [4]

2.3.2 Multilayer Feedforward

Compared to the previous type of neural network, the Multilayer Feedforward network have 1 or more layers between the input and the output. These are called *hidden* layers,

composed of computation nodes, which help to extract more features and therefore to have a better understanding of the statistics of the input data. The name *hidden* comes from the fact that neither the input nor the output layer can directly see these middle layers.

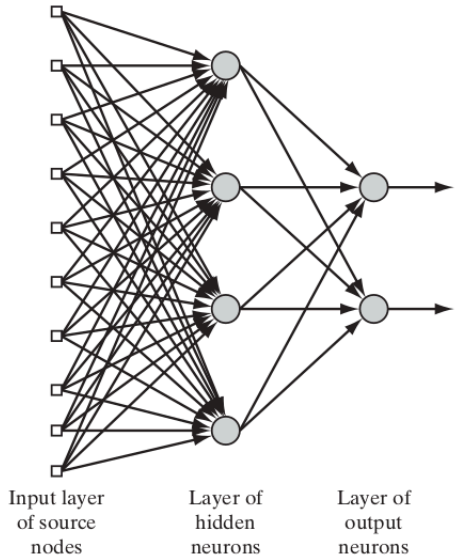


Figure 2.6: Representation of a Multilayer Feedforward neural network [4]

As we can see in figure 2.6, the signal from the input layer, are passed on to the nodes of the hidden layer (note that signal from all the nodes is passed to every other node of the next layer). Once the computation in this layer are made and the features extracted, the information is than passed to the nodes of the output layer.

2.3.3 Recurrent Network

The peculiarity of the Feedforward networks, as we saw, is that the information is passed only in one direction, the *forward* direction. This approach has some limitations in term of features extraction, even when hidden layers are present in the network. For this reason, a new class of network has been introduced in the recent years: the *Recurrent Neural Network*.

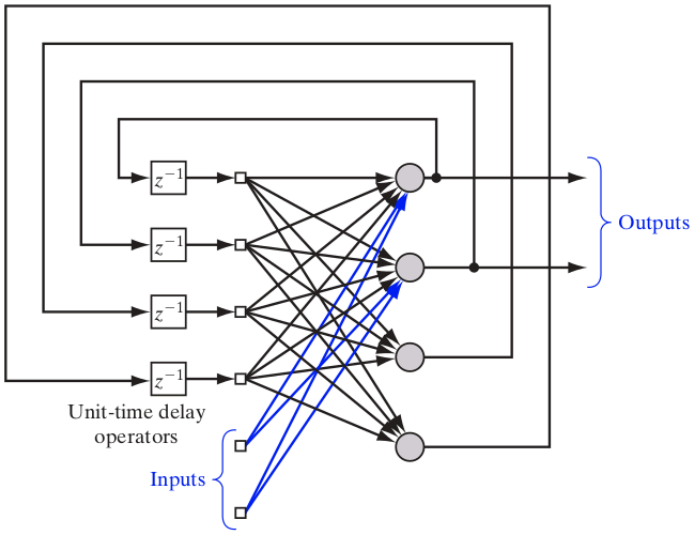


Figure 2.7: Representation of a Recurrent Neural Network, with a hidden layer [4]

Recurrent networks are characterised by the presence of a *feedback loop*. As we can see in figure 2.7, once the signal is passed by the input nodes to the hidden layer, their output is fed back to the input of the other nodes, before being passed on to the output layer. It is important to note the presence of a time delayed term (z^{-1}), which gives a non-linear dynamic behaviour to the network.

2.4 Learning Algorithms

One of the most important characteristics of the neural networks, are their capabilities to *learn*. This means that given a set of data, or observations, the network is *trained* to extract features from these data, which can later be used on data which have never been seen by the network. The input data can be either *labeled* or *unlabeled*. Being labeled, means that each data example, is paired with the desired response expected by the network: in few words the solution is already given, therefore the network learns to associate each input to its results, and then apply the knowledge drawn to a test set of unlabeled results. Labeled examples are harder to find, as this implies that we already know what each data sample means. For this reason in most of the scenarios, the input data will be unlabeled, hence without any additional information related to the data.

Given the different nature of the input data, we can distinguish two different way a neural network can *learn*, corresponding to whether the data is labeled or unlabeled.

2.4.1 Learning with Labeled Data

To best describe what dealing with labeled data means for a neural network, let's imaging a *teacher* which helps the network providing the expected answer. The network can compare the results it gets with the results given and therefore adjust the weights of the synaptic connection in order a similar answer to the one expected.

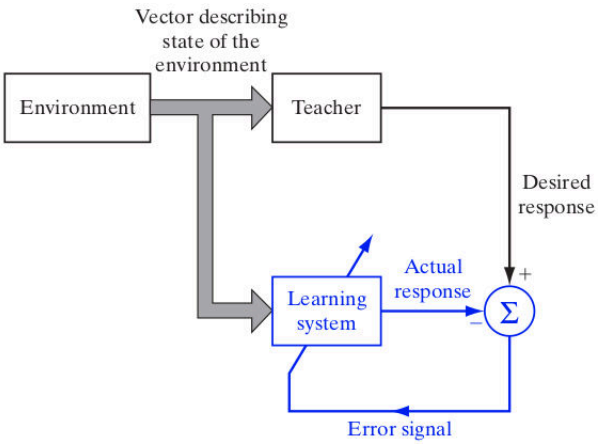


Figure 2.8: Representation of a Supervised learning process [4]

In figure 2.8, the *Environment* represents the data provided to the neural network, of which the *teacher* knows the expected result. Through a feedback loop, an *error signal*, representing the difference between the expected results and the one computed by the network, is fed to the neural network and it is utilised to adjust the synaptic weighs in order to minimise. This is an iterative process, and it stops once the network is able to compute the correct answer alone. At this point, the network can be provided with unlabeled data and ideally it is able to compute the correct results.

2.4.2 Learning with Unlabeled Data

Contrary to the supervised learning, in this situation there is no aid to the neural network in acquiring the knowledge to compute the correct answer. The network is still capable of learn the important features thanks to two different learning methods: *reinforcement learning* and *unsupervised learning*.

2.4.2.1 Unsupervised Learning

In an unsupervised learning algorithm, the task is to deduce a function to describe *hidden* structure from unlabelled data. Contrary to supervised learning, there is no way to check the accuracy of the output obtained with the network. These types of algorithm are used when there are problems in which is necessary to estimate some parameters. Some of these applications include: clustering, the estimation of statistical distributions, compression and filtering. In other words, the parameters of the network are optimised with respect to a *task-independent measure*: once the network has learnt the statistical regularities of the data provided, it is capable to encode the features of the data and therefore it can create new classes automatically.

One way to perform unsupervised learning, is to use *competitive-learning* approach. Given a generic neural network, the input nodes send their signals to the a layer of competitive nodes, which follow the *winner-take-all* rule: the node which better respond to the features of the data is activated, while the other get deactivated.

2.4.2.2 Reinforced learning

Contrary to the other learning methods described, in *reinforced learning* there is a continuous interaction between the *environment* (the input data) and the network.

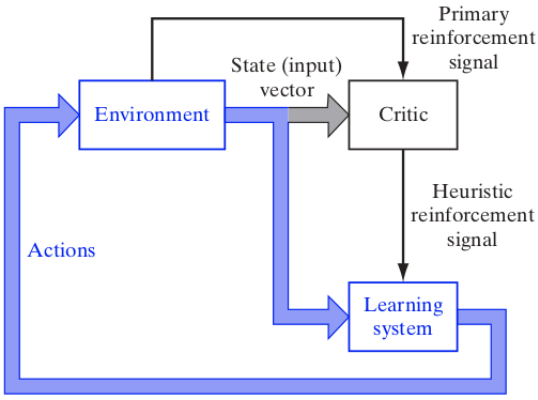


Figure 2.9: Representation of a Reinforced learning process [4]

As it can be seen from figure 2.9, the input signal passes both through the network *raw* and after being processed by a unit called *critic*. The latter, converts a *primary reinforcement signal* into a *heuristic reinforcement signal*, which is fed into the network as well.

In this way, the neural network learns to perform a task based on its interaction with the environment.

2.5 Back-propagation Algorithm

The learning process of the networks, as described before, is all about updating the weighs of the synapsis in order to minimise the error between the expected output and the obtained output. One of the algorithms used to perform this task, and also the one that I used in my model, is the *back-propagation algorithm*.

It's aim is to reduce the error of each single node, and it acts backwards (hence the name), starting from the output error. In order to calculate the error of the output obtained with respect to the desired output, it needs to know the latter *a priori*. For this reason it is used mainly in *supervised learning*.

Technically, it calculates the gradient of the *loss function*. The loss function, also referred to as *cost function*, is a function which *maps* the values of some variables into real number, which represents the cost associated with each value. In the specific case of neural network, the loss function calculates the error between the network output and the expected output. An example of loss function is:

$$E = \frac{1}{2j} \sum_{x_j} ||y_k(x_j) - y(x_j)||^2, \tag{2.8}$$

where j is the number of input signals, x_j is the input signal, y_k the output of the network and y the expected output.

Once the value of the error at the output layer is known, it is propagated back through the network until each node is associated with an error. At this point, we can modify the weighs

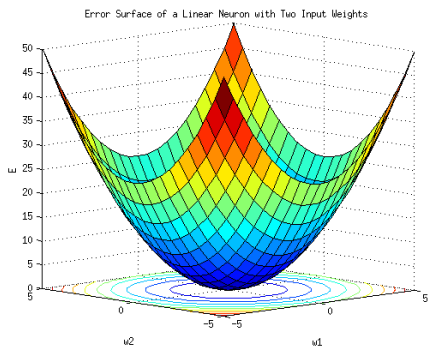


Figure 2.10: Example of an error surface of a linear neuron with two input weights [5]

in order to minimise the error in each node. The most common way to do so is through the *gradient descent*. Mathematically it calculates the first derivative of the error function with respect to the weighs in order to reach the minimum of the error surface (figure 2.10

After the gradient has been calculated, the weighs are updated according to the following variation:

$$\Delta w_{jk} = -\eta \frac{\partial E}{\partial w_{jk}}, \tag{2.9}$$

where j is the synapsis corresponding to the weigh, k is the node considered, $\frac{\partial E}{\partial w_{jk}}$ is the derivative of the loss function with respect to the weighs, and the sign $-$ is used to indicated that the weighs have to be updated in the direction of the minimum of the loss function.

2.6 How Neural Networks Are Used

With the development of powerful algorithms, and the technological progress which allowed better performing machines, the neural networks today are used mainly in application concerning *machine learning* and *artificial intelligence*.

These find great application in many fields: from pure scientific fields, like biology and physics, passing through social sciences, like economics or finance, to arrive at our daily life. Some example may include *computer vision*, which is being applied in self-driving cars or industrial machine, or also pattern recognition, heavily used by the most recent algorithm used in finance to track and interpret signal from financial data.

From the algorithmic side, many advancement have been made in the recent years, like the release of TensorFlow library (by Google) and the creation of Keras library, both of which I used to create the neural network used in my algorithm. These will be better described in the following chapters.

The Model

Chapter 3

Characteristics and Analysis of the Model

In order to make the forecast utilising the 10-minutes data of the USD/GBP exchange rate for the year 2016, I built an algorithm utilising *python* and *Keras API*.

In this chapter we will see how the part of the algorithm which makes the prediction was built. It can be divided into three parts:

- data preprocessing, in which the data is selected, reshaped and made *readable* by the neural network;
- neural network definition, in which the heart of the model is built, after defining all the paramethers utilised to predict the data;
- results visualisation, to visualise the prediction and the paramenters of the neural network, in order to be able to graphically and analytically assess the performance of the model.

3.1 Data Preprocessing

Before being able to run the model and obtain the forecast, I needed to modify the data so that the neural network could analyses. The dataset was made of a singe column of the 10-minutes exchange rate value for the USD/GBP from 01/01/2016 to 31/12/2016, paired with the corresponding data and time. The problem was that the dataset was not clean: it contained duplicates in the *date* column, corresponding to the weekend in which the Forex market is closed, for this reason was necessary to remove this duplicates, and *clean* the data.

```
raw_data = pd.read_csv('./Data.csv')
```

```
clear = raw_data.drop_duplicates(['date'])
clear.to_csv('./Data_clean.csv', index = False)
data = pd.read_csv('./Data_clean.csv')
data['date'] = pd.to_datetime(data['date'])
```

As we can see from the code above, the program reads the data, and after checking for duplicates, it create a new *.csv* file with the *clean* dataset. Moreover, to ease the subsetting of the dataset, it converts the *date* column to a format easily readable by python code: *YYYY-MM-DD hh:mm:ss*. In figure 3.1 there is an example of how the dataset changed after this operation.

date	last_price
01/04/2016 05:20:00	0.6758
01/04/2016 05:30:00	0.6761
01/04/2016 05:40:00	0.6760
01/04/2016 05:50:00	0.6761
01/04/2016 06:00:00	0.6764
01/04/2016 06:10:00	0.6764
01/04/2016 06:20:00	0.6766
01/04/2016 06:30:00	0.6769
01/04/2016 06:40:00	0.6769
01/04/2016 06:50:00	0.6771

(a) Raw dataset

date	last_price
2016-01-04 05:20:00	0.6758
2016-01-04 05:30:00	0.6761
2016-01-04 05:40:00	0.6760
2016-01-04 05:50:00	0.6761
2016-01-04 06:00:00	0.6764
2016-01-04 06:10:00	0.6764
2016-01-04 06:20:00	0.6766
2016-01-04 06:30:00	0.6769
2016-01-04 06:40:00	0.6769
2016-01-04 06:50:00	0.6771

(b) Processed dataset

Figure 3.1: Differencese in the format of the *date* column before and after it has been processed

After celansing the dataset, the model creates two subset, which will be utilised by the network for the train and the test.

```
training_period = ('2016-01-09 00:00:00','2016-03-11 23:50:00')
mask_train = (data['date'] >= training_period[0]) & (data['date'] <=
    training_period[1])
train_set = data.loc[mask_train]
train_set = np.array(train_set['last_price'])

print(train_set.shape)

test_period = ('2016-03-10 00:00:00','2016-04-01 23:50:00')
mask_test = (data['date'] >= test_period[0]) & (data['date'] <= test_period
    [1])
test_set = data.loc[mask_test]
test_set = np.array(test_set['last_price'])

print(test_set.shape)
```

In the code before we can see that, for both the subsets, I had to manually specify the period in which I was interested (*training_period* and *test_period*), so that the code could then select the exchange rate values corresponding to this period (*mask_train* and *mask_set*). Once this is done the to new sets are creating, *train_set* and *test_set*.

Once I have divided the two dataset, I need to reshape the data: as said at the beginning of the section, the data consists of a column containing all the data for the period considered. In order for the neural network to predict the data n utilising the t data points before, I need to reshape the datasets created before as follow:

$$[x_1, ..., x_n], [x_2, ..., x_{n+1}], [x_3, ..., x_{n+2}]... \tag{3.1}$$

In this way I obtained rolling windows made of n data point, comprising the t data utilised for the prediction.

This was done in the following lines of code:

```
new_train=[]
for x in range(0, len(train_set)-n):
    new_train.append(train_set[x:x+n])
train = np.array(new_train)

new_test=[]
for x in range(0, len(test_set)-n):
    new_test.append(test_set[x:x+n])
test = np.array(new_test)
```

Once created the windows of data, I had to further modify these sets in order to separate the input data (the t data points described before) and the target data (the data point n). Therefore, every data window showed in equation 3.1, is modified as follow:

$$[x_1, ..., x_n] \rightarrow [x_1, ..., x_t], [x_n] \tag{3.2}$$

This process can be seen in the following part of the code, where *x_train* and *x_test* are the input data, while *y_train* and *y_test* are the target data:

```
x_train = np.array(train[:,:t])
x_test = np.array(test[:,:t])

print('X Train - shape: ' +str(x_train.shape))
```

```
print('X Test - shape: ' +str(x_test.shape))

y_train = np.array(train[:,n-1:n])
y_test = np.array(test[:,n-1:n])

print('Y Train - shape: ' +str(y_train.shape))
print('Y Test - shape: ' +str(y_test.shape))
```

Once this is done, the preprocessing of the dataset is concluded, and therefore the data is ready to be fed into the neural network, and to be utilised for the prediction. It is important to notice that for both the train set and the test set, the input data utilised for the prediction is the real data.

3.2 Neural Network

This is the core of the model: the feedforward neural network, in order to compute predictions given the dataset, first updates the weights of the synapsis so as to minimise the error of it's output as compared to the expected output of train set. Secondly, once the error has been minimised, the test set is presented to the network, which computes the outputs: our predictions.

The following is the code defining the neural network as well as the parameters characterising it:

```
randomnormal = keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed
    =1000)
model = Sequential()
model.add(Dense(10, kernel_initializer=randomnormal, input_dim = t, activation
     = 'sigmoid'))
model.add(Dense(1,  kernel_initializer=randomnormal, activation = 'sigmoid'))

sgd = optimizers.SGD(lr=0.01, decay=0.0001, momentum=0.9)
model.compile(loss='mean_squared_error', optimizer=sgd, metrics=['mae'])

fit = model.fit(x_train, y_train, epochs=500, batch_size=32)

train_score = model.evaluate(x_train, y_train)
test_score = model.evaluate(x_test, y_test)
```

```

train_forecast = model.predict(x_train)
test_forecast = model.predict(x_test)
```

The first thing that this code is to initialise the weights: the variable *randomnormal* gives a initial value to each weight generated randomly using a normal distribution of mean 0 and standard deviation 0.05. I also set a *seed* in order to gain the same results every time that I run the simulation with the same parameters.

Secondly, thanks to the *Sequential* model derived from Keras API, I defined the architecture of my neural network:

- one hidden layer, composed of 10 nodes (arbitrary, found as a number of nodes between the numbers of inputs and the number of output nodes), which accept t amount of inputs (corresponding to the number of elements of the input datasets created before, and set to 17 for all the simulations); the activation function for these elements is the *sigmoid* sigmoid, which was described in chapter 3;

- One output layer, composed of 1 node (as we only expect one output for each step of the prediction), which accepts as inputs the outputs of the hidden layer. The activation function for this layer is again a *sigmoid* function.

After having defined the neural network, I defined the way it updates the weights: the variable *sgd* indicates the Stochastic Gradient Descent, which is the optimizer I used for my network. As described in section 3.5, the stochastic gradient descent calculates the first derivative of the error function with respect to the weighs in order to calculate of how much they have to be adjusted to reduce the error. This optimizer takes a number of parameters:

- *lr* defines the *learning rate*, which is a value indicating how fast the neural network can update it's weights in each iteration: the higher the value, the faster it learns, the higher the probability of overtraining the network, and vice versa;

- *decay* defines a parameter which helps avoiding overtraining problems, reducing the learning rate;

- *momentum* is a term between 0 and 1 which indicates the dimension of the step that the neural network takes iterating to adjust its weights.

The *sgd* just defined, is used as a parameter in the next line of code, *model.compile*, which defines how the weights are optimized, and what are the metrics used to estimate the precision of the model. In this case the metrics used are the Mean Squared Error and the Mean Absolute Error.

The Mean Squared Error [21] calculates the mean of the squared errors of the single outputs related to the test set, and divides it by all the number of elements in the test set. It is defined as follow:

$$mse = \frac{\sum_{i=1}^{n} (y_k(x_i) - y_i(x_i))^2}{n}, \tag{3.3}$$

where $y_k(x_i)$ is the expected output for the element x_i of the test set, y_i is the output of the neural network for the same element, and n is the total number of elements of the test set.

The Mean Absolute Error [21] calculates the mean of the absolute errors of the single output related to the test set, and divides it by all the number of elements in the test set. It is defined as follow:

$$mae = \frac{\sum_{i=1}^{n} |y_k(x_i) - y_i(x_i)|}{n}, \tag{3.4}$$

where $y_k(x_i)$ is the expected output for the element x_i of the test set, y_i is the output of the neural network for the same element, and n is the total number of elements of the test set.

Going on in the analysis of the neural network, the variable *fit*, in line 9 of the code, is what makes the network train. We can see that in its parameters the set to use for the prediction (*x_train*) and the set used to confront the output to update the weights (*y_train*). Moreover, I set the parameter *epochs* to 500, which indicates how many times the network has to iterate over the set to update the weights, and the parameter *batch_size* to 32, meaning that the weights are updated every time that 32 inputs are fed into the network.

Finally, from line 11 to line 15 of the code shown, we can see the parts in which the neural network work is evaluated (*train_score* and *test_score*) and, most important, the command to make the network predict the data, which are *train_predict* and *test_predict*.

In figure 3.2 we can see the structure of the network created.

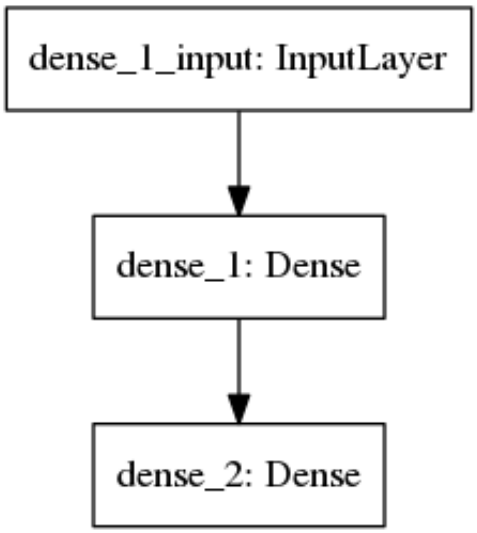


Figure 3.2: Neural Network structure

It shows exactly what I explained previously in this section: the inputs are fed into the *dense_1* layer, which is the hidden layer, whose outputs are fed into the *dense_2* layer, the output layer. It is important to notice *Dense*, written for both the hidden and the output layer: it indicates that the nodes are fully connected between each layer, a situation similar to the one seen in figure 2.6, when I described the multilayer feedforward in chapter 3. We can also see, in figure 3.3, the characteristics of the neural network created.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 10)                180
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 11
=================================================================
Total params: 191
Trainable params: 191
```

Figure 3.3: Neural Network characteristics

For each layer it is shown , the shape of the output, which is a 10 element array for the hidden layer and a unidimensional array for the output layer, but as well the number of parameters for each layer. This consists of the number of weights which need to be trained, and we can see that are 180 for the hidden layer, and 11 for the output layer. This is due to the fact that the structure is *dense*, and therefore each node of each layer is connected with each

other node of the following layer. This creates a total of 191 connection whose weights need to be trained.

3.3 Results Visualisation

Once the neural network has calculated the prediction, in order to visualise and analyse the results, I wrote the following line of code:

```
plt.figure(dpi=1200)
plt.plot(poi, 'b', label = 'Real')
plt.ylabel('Exchange Rate')
plt.xlabel('Time Steps')
plt.legend()
plt.show()

print(poi.shape)
print(train_forecast.shape)
print(test_forecast.shape)

shift_array = [[None]*(len(poi)-len(test_forecast)+n)]
shift_array = np.array(shift_array)
shift_array = shift_array.T
test_shift = np.concatenate((shift_array,test_forecast[n:]))

plt.figure(dpi=1200)
plt.plot(poi, 'b', label = 'Real')
plt.plot(train_forecast[n:], 'g', label = 'Train')
plt.plot(test_shift, 'r', label = 'Test')
plt.ylabel('Exchange Rate')
plt.xlabel('Time Steps')
plt.legend()
#plt.savefig('forecast-LowLow.png')
plt.show()

plt.figure(dpi=1200)
plt.plot(train_set[n:-n], 'b', label = 'Real')
plt.plot(train_forecast[n:], 'g', label = 'Train')
plt.ylabel('Exchange Rate')
```

```
plt.xlabel('Time Steps')
plt.legend()
#plt.savefig('forecast_train-LowLow.png')
plt.show()

plt.figure(dpi=1200)
plt.plot(test_set[n:-n], 'b', label = 'Real')
plt.plot(test_forecast[n:], 'r', label = 'Test')
plt.ylabel('Exchange Rate')
plt.xlabel('Time Steps')
plt.legend()
#plt.savefig('forecast_test-LowLow.png')
plt.show()

plt.figure(dpi=1200)
plt.plot(fit.history['loss'])
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.annotate('Mean Squared Error: '+"{:.4e}".format(test_score[0]), (0,0), (0,
    -35), xycoords='axes fraction', textcoords='offset points', va='top')
#plt.savefig('mean_squared_error-LowLow.png')
plt.show()

plt.figure(dpi=1200)
plt.plot(fit.history['mean_absolute_error'])
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.annotate('Mean Absolute Error: '+"{:.4e}".format(test_score[1]), (0,0),
    (0, -35), xycoords='axes fraction', textcoords='offset points', va='top')
#plt.savefig('mean_absolute_error-LowLow.png')
plt.show()
```

This code generates a series of data, which will be shown in the results (chapter 5). Starting from the beginning, it plots firs of all the *poi*, *Period Of Interest*, which is nothing but the real data of the period analysed, from the beginning of the train set to the end of the test set. After this there are 3 plots which show in order: the train and the test forecast confronted with the real data of the whole period, the train forecast compared to the train train set, and the test forecast compared to the test set.

The last two plots, show the evolution of the Mean Squared Error and the Mean Absolute Error with respect to the iterations. Both these plots are characterised by a steep slope during the first iterations and then become constant during the last iterations: in fact, going on in the training of the network, the error reduces gradually until it reaches a moment in which it is not possible to reduce it any more. In these plots it is also printed the value of these error when calculated with respect to the test set. We will see it better when analysing the results in chapter 5.

Chapter 4

Investment Simulation Algorithm

After having obtained the forecasted data and having analyse the statistical performance of the neural network used, I decided to test an automatised trading algorithm on the real data used, based on the prediction made by my model. What I did was to use the simulated data as an indicator of whether to open or not and to decide if this position should be long or short. Then I calculated the profit on utilising the real data.

I made two different simulation with two slightly different assumption:

- continuous investment, where the algorithm invests continuously and open a position for each data point;
- threshold investment, where the algorithm invests only when the percentage difference of two forecasted data points is greater than a certain threshold (to mimic transaction costs).

4.1 Continuous Investment

Again, for this simulation I wrote a small code in python.

```
real = test_set[n:]
forecast = test_forecast
l = len(real)

print(len(real))
print(len(forecast))

```

```
'''Creates a list of 1 and -1 according to the price movement of the forecasted data:
if at t+1 the value is > than the one at t, I write 1, and vice versa'''

indicator = []

for j in range(0,l-1):
    if forecast[j+1]-forecast[j]>0:
        indicator.append(1)
    elif forecast[j+1]-forecast[j]<=0:
        indicator.append(-1)

indicator.insert(0,1)
#print(len(indicator))

'''I do the same on the real data set, in order to check the performance
of the algorithm related to the price movement (% of times it got the right movement)'''

indicator_real = []

for j in range(0,l-1):
    if real[j+1]-real[j]>0:
        indicator_real.append(1)
    elif real[j+1]-real[j]<=0:
        indicator_real.append(-1)

indicator_real.insert(0,1)
#print(len(indicator_real))
```

In this first part of the code, first it creates two different sets of data, the one including the real data (*real*) and the one including the forecasted data (*forecast*). Both the periods correspond to the test set. Once this is done, it runs a control on both the dataset: it checks whether each data point in the forecasted data set is greater or lower than the previous data point, and it creates two lists, *indicator* and *indicator_real*, in which there is a 1 every time that the data point at time $t+1$ is greater than the data point at time t, and a -1 otherwise.

These are needed by the algorithm to decide if it needs to open a long position (if there is a 1) or a short position (in case there is a -1).

Once this is done, the next part of the code works on the real data to define the returns on which to calculate the profit:

```
'''Creates a list containing the % difference between two consecutive values
    in the real data'''

returns = []

for j in range(0,l-1):
        returns.append(((real[j+1])/real[j])-1)

returns.insert(0,1)
print(len(returns))
```

In the code above, we can see that the algorithm creates a list called *returns*, which includes the returns calculated for each consecutive data point.

Working with data at 10 minutes distance from each other, it happens sometimes to have a return of 0%.

For this reason, and for the scope of this simulation, if this situation happens to be true, it is necessary to make the algorithm behave as if it opened a position and made 0 profit. Hence why I wrote this other code:

```
'''newindicator is an adjustment to the indicator list created before,
in order to insert a 1 everytime I find a 0 in the returns list, so that I can
    ignore it
and act as if the algorithm doesn't open any position'''

newindicator = []
for i in range(0,len(indicator)-1):

    if returns[i] == 0:
        newindicator.append(1)
    elif returns[i] != 0:
        newindicator.append(indicator[i])
```

Here, the algorithm confronts the *indicator* list described before with the *return* list presented right above, and it changes whatever value in *indicator* to 1 every time there is a 0 in

return. In this way I obtain the equivalent of not opening any position if the return is 0. If I would have forced a -1 instead of a 1 in the *indicator* list, the code would have given me unrealistic data due to its inabilities to process this situation with a negative sign.

Once the code has all the indication it needs to run an investment simulation, I defined the its core as follow:

```
1  '''Calculates the profit and therefore the total profit supposing an
       investment of USD 1000 for every operation'''
2
3  x = 1000
4
5  z = next(i for i, v in enumerate(returns[1:len(returns)]) if v != 0)
6  #print(z)
7
8  a = [i for i in range(0,len(newindicator)-1)]
9
10 for j in range(z,len(newindicator)-1):
11     a[j]=((x*newindicator[j+1]*(returns[j+1])))
12
13 #print(a)
14 profit = np.sum(a)
15 #print(z)
16 #print(profit)
17 for i in range(0,z+1):
18     a.insert(i,x)
19     #print(a[0])
20 for i in range(z+1,len(a)):
21     a[i] = a[i] + a[i-1]
22
23 totalprofit = ((x+profit)/x)-1
24 tot_profit = 'Total profit: {percent:.4%}'.format(percent=totalprofit)
25 print(tot_profit)
```

Here, I define an investment of x amount of USD to invest at each operation (as we will see in the results, this has been set to USD 1000). Therefore I defined another list, called a, which will includes all the profits and losses of the various operations. To fill this list, I calculated the profit of each operation as written in line 11 of the code shown above. In other words, the

profit of each operation is:

$$profit = x * I * r, \tag{4.1}$$

where x is the amount invested, I is the indicator 1 or -1 from the list *newindicator*, and r is the return calculated on the real data.

It is important to notice that, as for this simulation, I have considered every operation as independent, and I have considered as well that there are no budgetary constraints, therefore the algorithm has access to unlimited liquidity.

Having the list of all the profits, it then calculates the cumulative profit for the whole period, which is used to assess if this algorithm allows to obtain an overall profit or loss.

In order to have more information about the accuracy of the prediction, I also calculated the time in which the forecasted data show the same movements of the real data.

```
'''Calcluates the number of times the price movement prediction was right'''

up_down = [1 for x in range(0,l-2) if indicator[x] == indicator_real[x]]

k = up_down.count(1)
#print(up_down)
precision = k/len(indicator_real)
price_movement = 'Price movement prediction accuracy: {percent:.2%}'.format(
    percent=precision)
print(price_movement)
```

I created a list *up_down*, in which there is a 1 every time that an element of the list *indicator* is equal to the same element of the list *indicator_real*. In this way I could record every time that the price movement of the forecasted price was equal to the price movement of the real data (for price movement is meant if the price increases or decreases). In fact, counting the element of the newly create *up_down* list, I had a total of how many time the forecasted price movement was right, and therefore I could calculated the percentage accuracy.

To visualise the results, I plotted the cumulative return with respect of the time steps (10 minutes each) for the whole time period considered, writing the following lines of code:

```
'''Plots the profit graph'''

plt.figure(dpi=1200)
plt.plot(a, 'g')
```

```
plt.ylabel('Profit')
plt.xlabel('Time Steps')
plt.annotate(tot_profit+' - '+price_movement, (0,0), (0, -35), xycoords='axes fraction', textcoords='offset points', va='top')
#plt.savefig('Simulation_LowLow.png')
plt.show()
```

4.2 Investment with a Threshold

To mimic the transaction costs, I had to introduce a threshold in my algorithm when building the *indicator* list described in the previous section. To do so I added the following line of code to the algorithm shown before:

```
'''Define the threshold p and how it should be used to adjust the indicator array'''
p = 0.0010
indicator1 = []
beg = forecast[0]

for i in range(1,l-1):
    kk = (forecast[i]/beg)-1
    #print(kk)
    if kk >= p or kk<= -p:
        beg = forecast[i]
        if kk >= p:
            indicator1.append(1)
        elif kk <= -p:
            indicator1.append(-1)
    else:
        indicator1.append(0)

indicator1.insert(0,1)
```

The threshold is defined as p, and limits the number of position opened. In fact, the new list, *indicator1*, is now composed of 1, -1 and 0, with the latter being there everytime that the difference between two data points is less than the threshold p.

The remaining part of the algorithm is the same as the one described before, except that I also decided to calculate the total number of positions opened by counting the subtracting the number of time a 0 appears in the list *indicator1* length of the length of the same list, as shown in the following lines of code:

```
"""Calcluate the number of orders placed corresponding to how many time the
    forecasted return exceed the threshold"""

k = 'Total Orders '+str(len(indicator1) - indicator1.count(0))
print(k)
limit = 'Investment Limit: {percent:.2%}'.format(percent=p)
print(limit)
```

Results

Chapter 5

Efficency of the Neural Network Forecasting Capability

To run the algorithm I used the 10-minute exchange rate data for the USD/GBP. The analysis has been carried in two different periods characterised respectively by low volatility and high volatility.

Thanks to the code explained before, I calculated the 80-day volatility for the whole dataset (from January 1st 2016 to December 31st 2016, and selected the period with the lowest and the highest volatility.

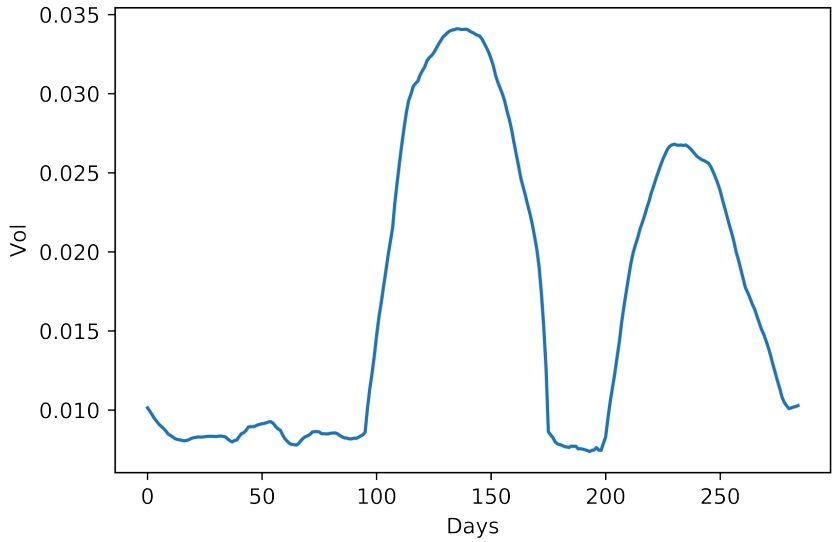


Figure 5.1: 80-day volatility in 2016

As can be seen from figure 5.1, the period with lower volatility is the period which goes

from the beginning of the year to the beginning of April, while the period with the highest volatility is the one going the beginning of April to the beginning of July.

After this preliminary analysis, I could choose the periods in which test my algorithm:

- low volatility: from 10/01/2016 to 31/03/2016;
- high volatility: from the 15/04/2016 to the 06/07/2016.

Once I defined the periods of interest, I set the parameters to use in the neural network as follow:

```

'''Build the Neural Network using Keras API and use it to evaluate the model and make the prediction'''

randomnormal = keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=1000)
model = Sequential()
model.add(Dense(10, kernel_initializer=randomnormal, input_dim = t, activation = 'sigmoid'))
model.add(Dense(1,  kernel_initializer=randomnormal, activation = 'sigmoid'))

sgd = optimizers.SGD(lr=0.01, decay=0.0001, momentum=0.9)
model.compile(loss='mean_squared_error', optimizer=sgd, metrics=['mae'])
```

As we can see from this code, the network is made of 2 layers, characterised by 17 input nodes (t in the code, corresponding to the window used for the predictions), 10 nodes in the hidden layer and 1 node in the output layer. The optimisation function is the Stochastic Gradient Descent, with learning rate of 0.01, learning rate decay of 0.0001 and momentum of 0.9. To train the network, it iterates over the train dataset for 500 times (*epochs*) and it updated the weights every times it 32 data points of the train set (*epochs*).

In the following sections I am going to show the results of different trials I made to see how this algorithm reacts to different situations. First, we are going to see how the efficiency of the neural networks is related to the volatility of the period examined. Secondly, how it differs if test set is part of the train set, or if the two are completely separated. Finally, I will show how the algorithm can reacts when forecasting data at different time horizons, starting to forecast a data point 10 minutes after, to 24 hours after.

5.1 Efficiency Related to the Volatility of the Period.

In this part, I emphasised on how the prediction is affected by the choice of the period for the train and the test set with different volatility. For this purpose, I did the following analysis:

- both the train and the test set are characterised by low volatility;
- both the train and the test set are characterised by high volatility;
- the train set is taken from the low volatility period, while the test set from the high volatility period;
- the train set is taken from the high volatility period, while the test set from the low volatility period.

For the low volatility period, I defined:

- train set from 10/01/2016 to 10/03/2016;
- the test set goes from 11/03/2016 to 31/03/2016.

For the high volatility period:

- train set from 15/04/2016 to 15/06/2016;
- the test set goes from 16/06/2016 to 06/07/2016.

In all these analysis, the algorithm used a window of 17 consecutive data to predict the next data point, n=18. In real time data, it means that the algorithm utilises the data of the previous 2 hours and 50 minutes to predict the value in the after 10 minutes. For both the train and the test set, the 17 input data are given by the real dataset.

5.1.1 Low Volatility Data for Train and Test

In figure 5.2, it is shown in blue the real data, in green the forecasted train data and in red the forecasted test data. As it can be clearly seen, the neural network was able to predict accurately both the train and the test set, with the forecasted data almost perfectly overlapping with the real data. This can be seen also from the Mean Squared Error and the Mean Absolute Error. This are calculated only for the test set, as the train data is used only to *train* the algorithm, and I decided to give more relevance to the precision in forecasting the test set.

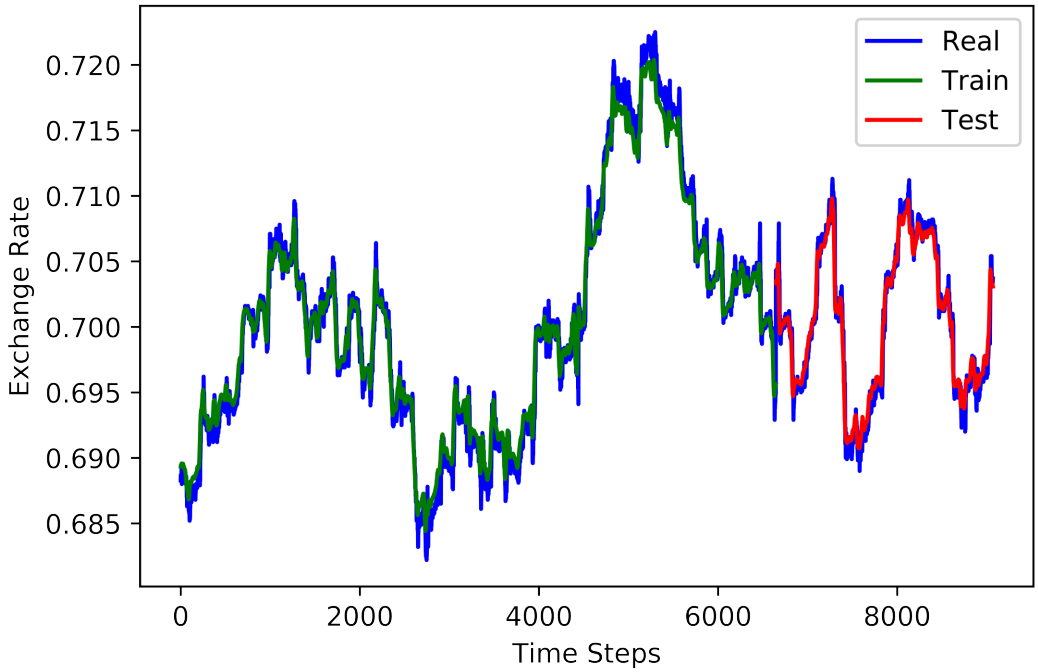
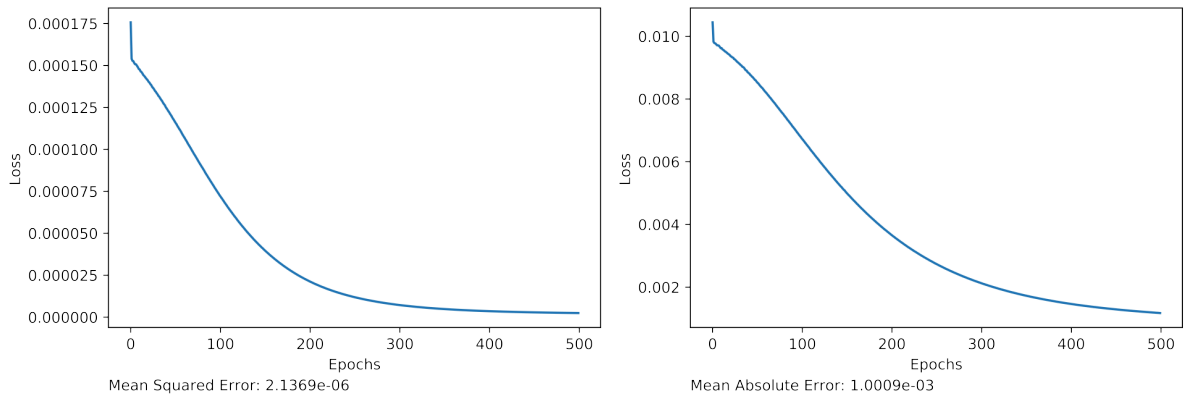


Figure 5.2: Train and Test forecast data as compared to the real data



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.3: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

As we can see from figure 5.3, both the errors drop radically in the first 200/300 iterations to plateau afterwards. This means that the network wasn't able to reduce the error any more and therefore reached the optimal configuration. We can also notice that the mean squared error is $2.1369x10^{-6}$ while the mean absolute error is $1.0009x10^{-3}$. Both are really small, meaning that the neural network was able to go really close to the real value of the data presented.

5.1.2 High Volatility Data for Train and Test

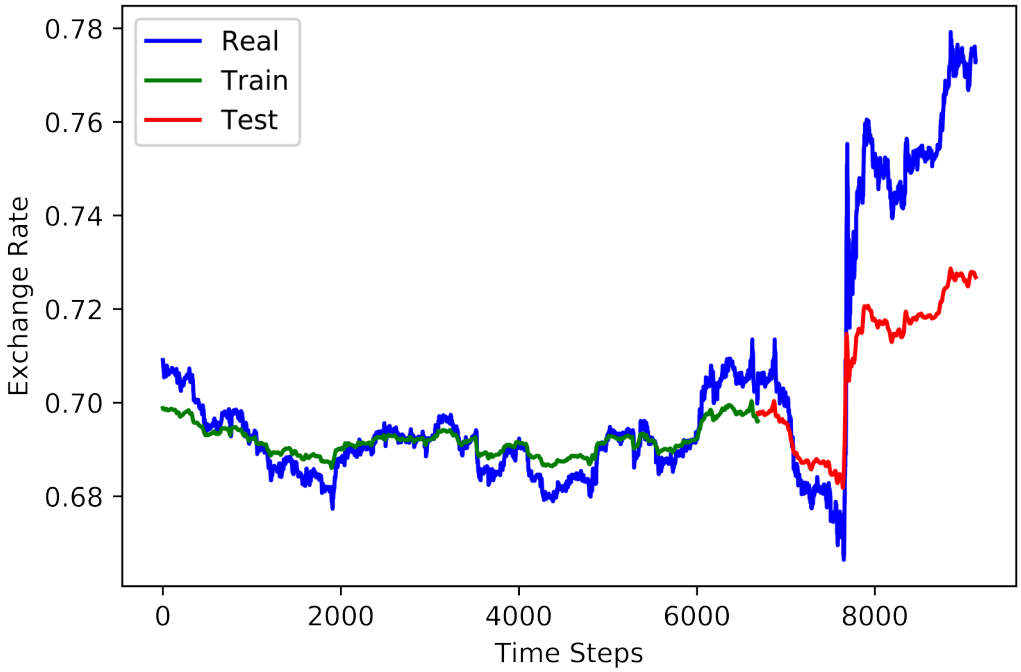
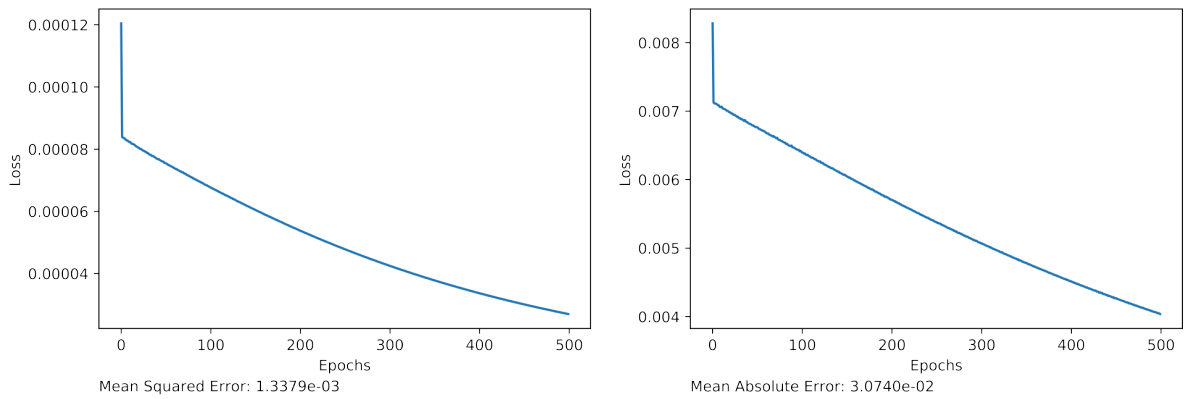


Figure 5.4: Train and Test forecast data as compared to the real data

In figure 5.4, it is shown in blue the real data, in green the forecasted train data and in red the forecasted test data. As it can be clearly seen, the neural network, contrary to before, wasn't able to predict accurately both the train and the test set. In fact, it can be noticed how the algorithm was able to catch the trend of the data, but wasn't able to predict accurately the value. We notice how the forecasted data, this time, don't overlap with the real data. This can be seen also from the Mean Squared Error and the Mean Absolute Error.

This time, contrary to the previous example, the errors don't plateau even at the end of the 500 iterations, as we can see from figure 5.5. This means that the network wasn't able to optimise the weights related to the various nodes in the iteration given and it would have needed many more to be able to minimise the error. We can also notice that the mean squared error is $1.3379x10^{-3}$ while the mean absolute error is $3.0740x10^{-2}$. They are respectively 3 and 2 orders of magnitude higher than the forecast of low volatile data.



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.5: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

5.1.3 Low Volatility Data for Train and High Volatility Data for Test

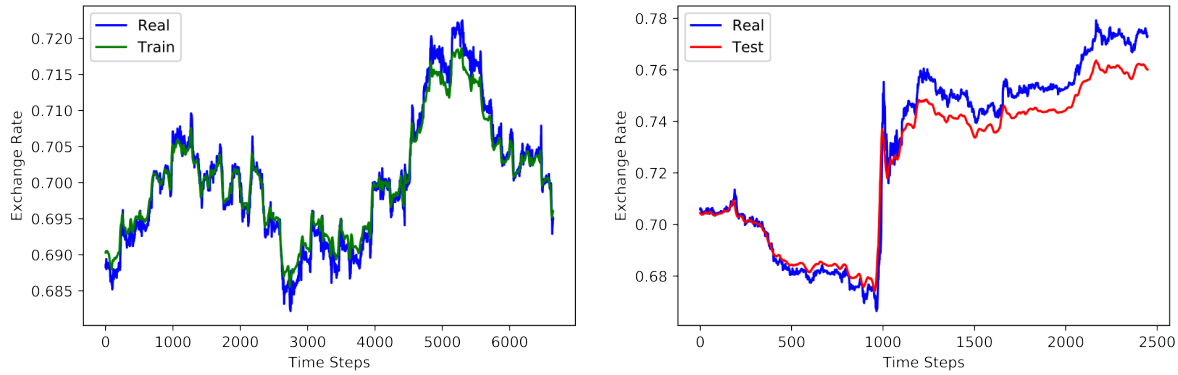
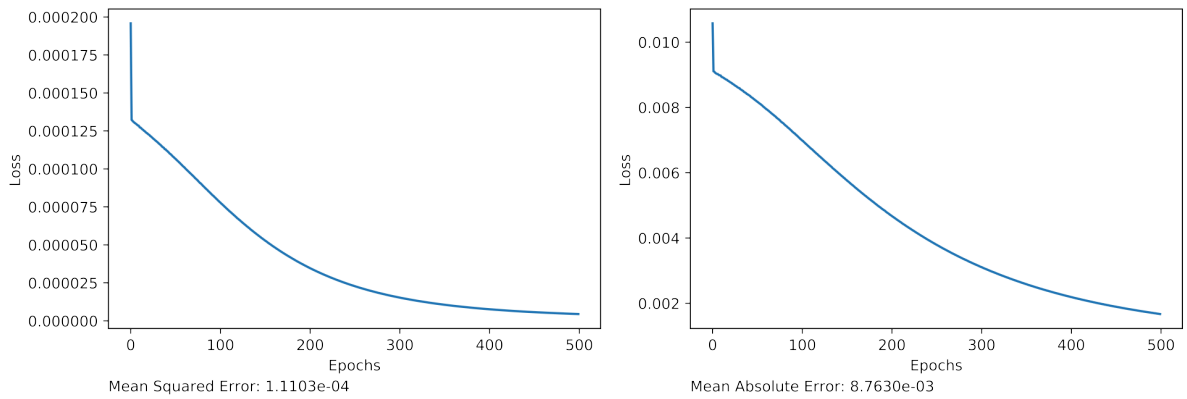


Figure 5.6: Train and Test forecast data as compared to the real data

In figure 5.6, it is shown in blue the real data, in green the forecasted train data and in red the forecasted test data. Utilising the low volatility data for the train set and the high volatility data for the test set, it was able to predict more accurately both the train and the test set. This time, it can be noticed that the algorithm was able to catch the trend of the data, as well as being able to predict accurately the value. Again, we can notice this can be seen also from the Mean Squared Error and the Mean Absolute Error.



(a) Mean Squared Error **(b)** Mean Absolute Error

Figure 5.7: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

This time again, the errors do plateau after some iteration (300/400), as we can see from figure 5.7. This means that the network was able to optimise the weights of the nodes, even though it wasn't able to do it as accurately as in the first case seen. We can also notice how the errors again are smaller, but still some orders of magnitude greater than in the low volatility case for both train and test set. In fact we have that the mean squared error is $1.1103x10^{-4}$ while the mean absolute error is $8.7630x10^{-3}$.

5.1.4 High Volatility Data for Train and Low Volatility Data for Test

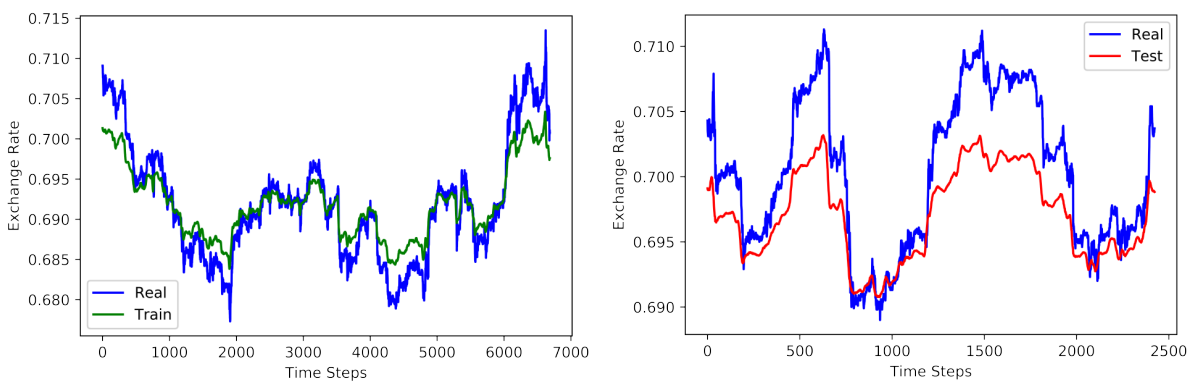
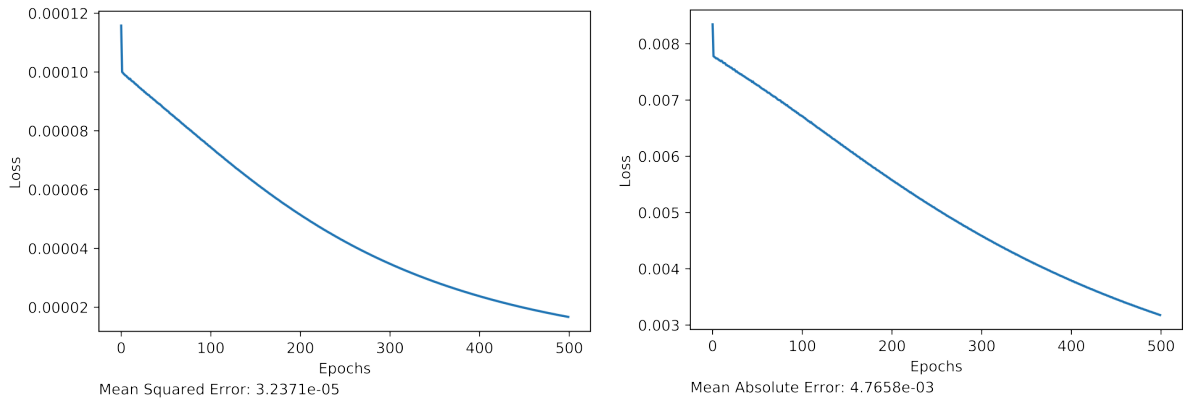


Figure 5.8: Train and Test forecast data as compared to the real data

Contrary to the previous example, and in line with what expected, as it can be seen from figure 5.6, which shows in blue the real data, in green the forecasted train data and in red the forecasted test data, when we use the high volatility data to train the model, the prediction

aren't accurate as when we use the low volatility data. In fact again we have that the algorithm was able to predict the trend of the data, but wasn't able to catch the enough features to allow it to give a realistic forecast of the value of the data. This is clear looking at the Mean Squared Error and the Mean Absolute Error.



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.9: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

Like in the case of high volatility data for both the train and the test set, the neural network wasn't able to optimise the weight of the nodes enough to minimise the error. As we can see from figure 5.9, the network didn't collect enough features in the given iterations, and therefore it was able only to represent the trend of the data but not their values.In this case, the mean squared error is $3.2371x10^{-5}$ while the mean absolute error is $4.7658x10^{-3}$. Although from the plots we can certainly say that the prediction isn't accurate, looking at the value of the errors, we notice that still it is better than the high volatility case for both train and test set. This is due to the low volatility of the test set, which presents easier-to-replicate features compared to the high volatility test set.

Following the results of this preliminary analysis, we can see how the best performance are achieved in the following scenarios:

- for low volatility, when both the train and the test set belong to low volatile data;
- for high volatility, when the train is carried on low volatility data, while the test on high volatile data.

5.2 Efficienty Related to the Whether the Test Set is Part of the Train Set

After seeing what happens if we consider periods with different volatility and use different periods for the train and the test set, I decided to show also an unrealistic situation: the test set is part of the train set. This is unrealistic as it is impossible in real life to already know what will happen for sure.

For this analysis I considered only the period of low volatility and high volatility for the train sets, which at this time comprise the whole periods described at the beginning of this chapter.

Given that the neural network already *sees* the data on which it will later test, the results will be much more accurate than the one we saw in the previous section. I decided to use try this scenario, even though unrealistic, only to show how easily the prediction can be affected by a wrongful choice of train and test set.

5.2.1 Low Volatility Period

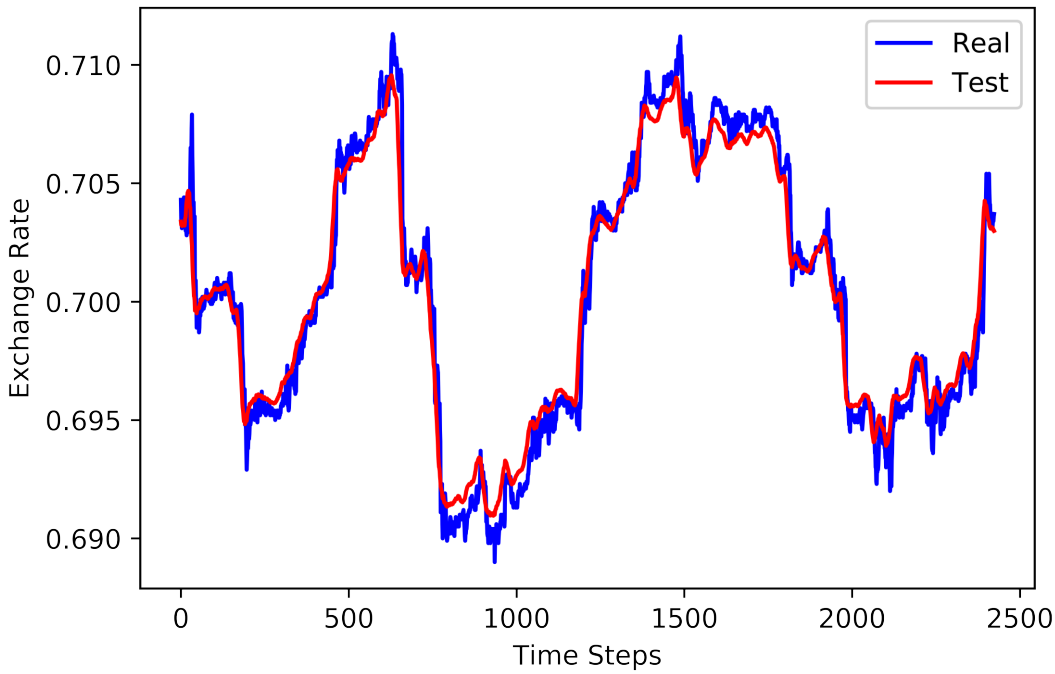
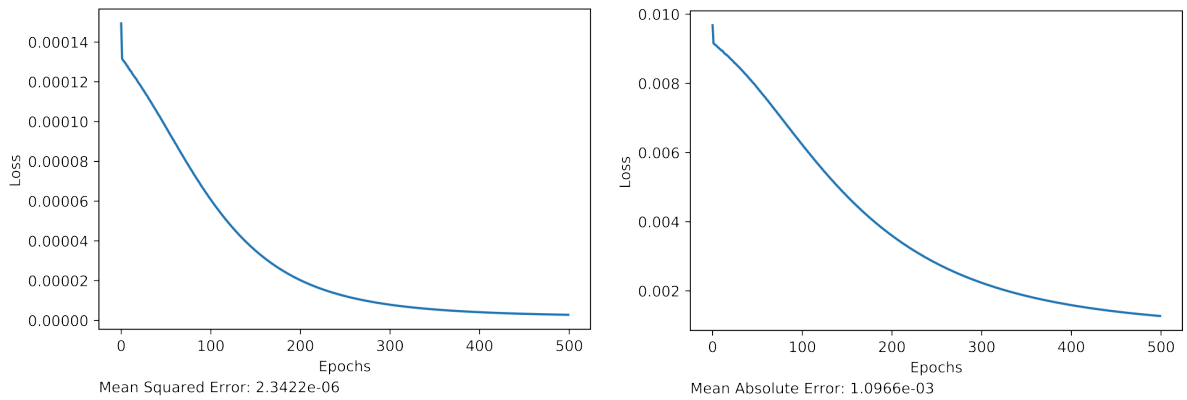


Figure 5.10: Test forecasted data as compared to the real data

In figure 5.10, it is shown in blue the real data and in red the forecasted test data (the train set would have been equal to the one shown before). As it can be clearly seen, and as we could expect given the results shown in the previous section, the neural network was able to predict accurately the test set, with the forecasted data almost perfectly overlapping with the real data. This can be seen also from the Mean Squared Error and the Mean Absolute Error.



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.11: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

As we can see from figure 5.11, both the errors drop radically in the first 200/300 iterations to plateau afterwards. As explained before, this means that the error was reduced to a point in which the neural network couldn't have done more. We can also notice that the mean squared error is $2.3422x10^{-6}$ while the mean absolute error is $1.0966x10^{-3}$. Both are really small, and very close to the one calculated in the previous section.

5.2.2 High Volatility Period

In figure 5.4, it is shown in blue the real data and in red the forecasted test data. As it can be clearly seen, the neural network, contrary to before, was able to predict accurately the test set. In fact, it can be noticed how the algorithm was able to both catch the trend of the data and to predict accurately the value. This is the confirmation that using a test set which is part of the train set can give a biased idea on the performance of a neural network. This can be seen also from the Mean Squared Error and the Mean Absolute Error.

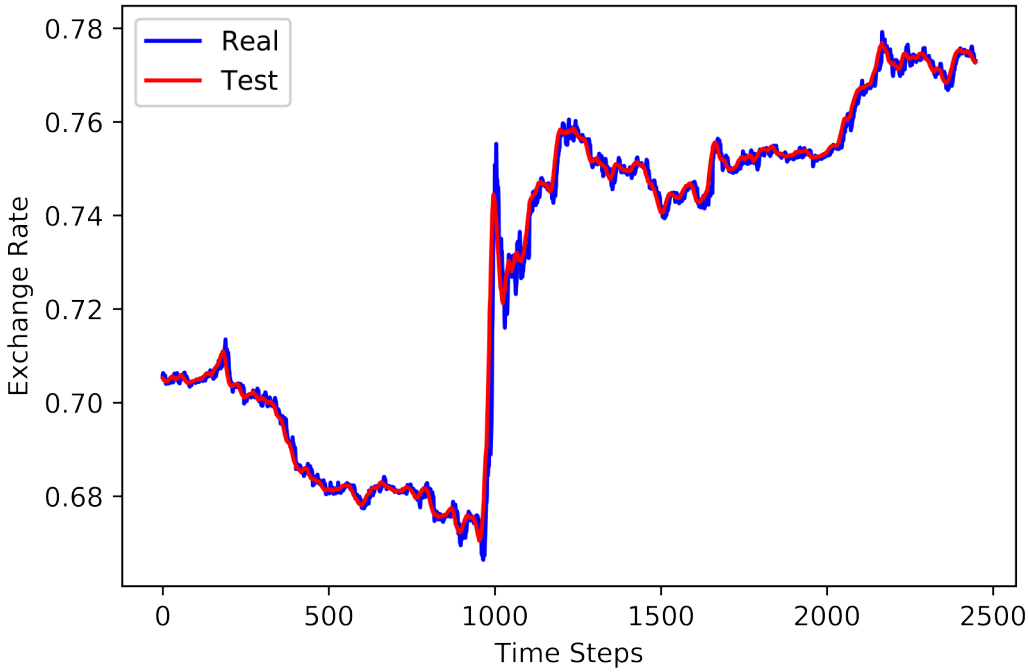
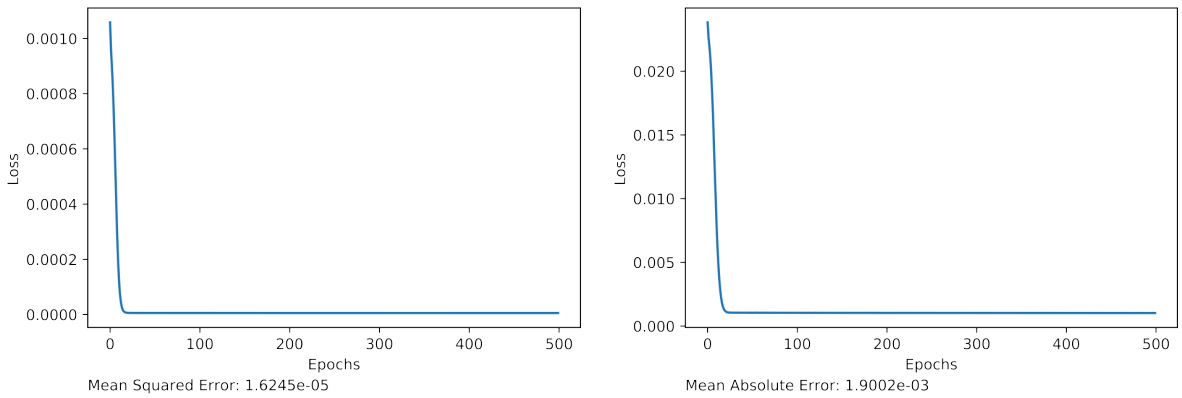


Figure 5.12: Test forecasted data as compared to the real data



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.13: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

This time, contrary to the previous example, the errors plateau very early after less than 100 iterations, as we can see from figure 5.13. This means that the network was able to optimise the weights very early and causes a problem of overfitting. In this case, the mean squared error is $1.6245x10^{-5}$ while the mean absolute error is $1.9002x10^{-3}$. These are much higher than the scenario seen in the previous section.

5.3 Efficiency Related to Different Time Horizon

In this section, I will show how the performance of a neural network is affected by the time horizon of the forecast. Keeping in mind the results of the first section, I used the low volatility train set to forecast both the low and high volatility test set.

To show the neural network performance, I forecasted the data point n=30, n=48, n=78 and n=144, corresponding respectively to 2 hours, 5 hours, 10 hours and 21 hours after the data used for forecasting.

5.3.1 Low Volatility Period

For the low volatility period, I defined:

- train set from 10/01/2016 to 10/03/2016;
- the test set goes from 11/03/2016 to 31/03/2016.

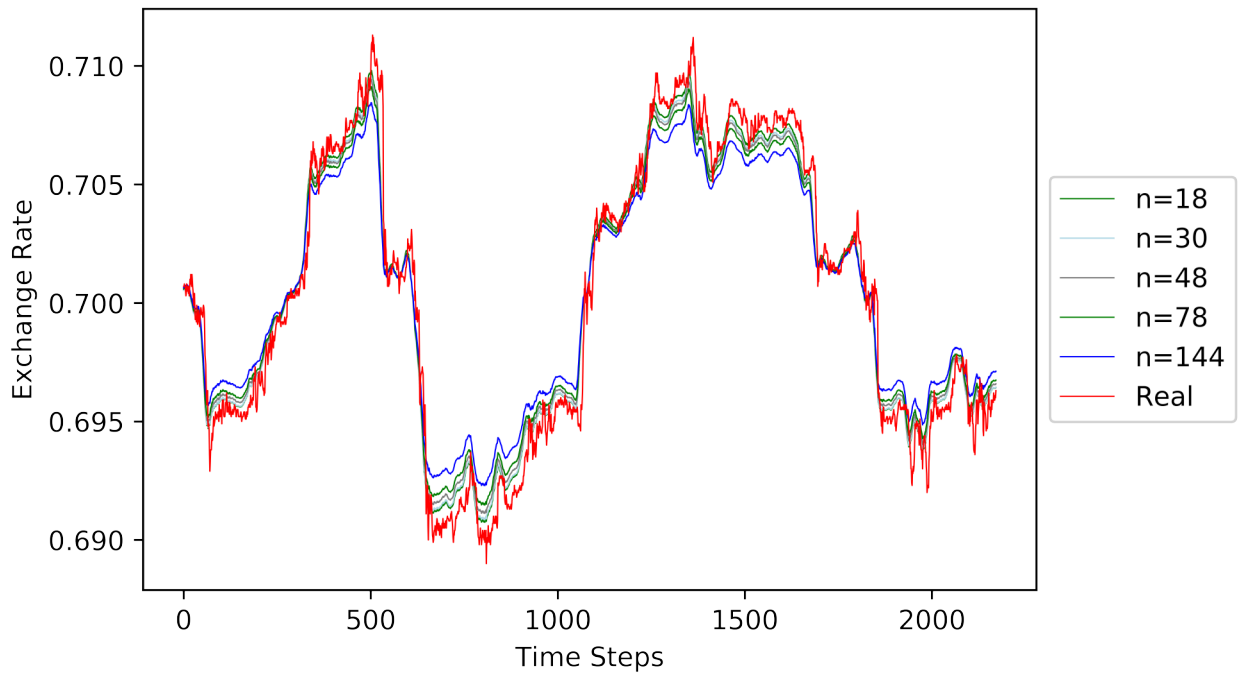


Figure 5.14: Comparison of the forecasted data with respect to the real data for the low volatility test set

In figure 5.14, we can see how the predicted data varies when changing the temporal horizon of the prediction. We can see that for short term prediction, the forecast is close to

the real data, both in term of trend and of value. As we try to make longer term prediction, the results tend to loose their precision, maintaining some of the features of the real data. In particular we can notice how the curve of the forecasted data is smoother for longer term as compared to lower term: this indicates that the neural network is less affected by the random movements characterising the high frequency data, but rather focuses on the trend of the data.

In the following paragraphs it will be shown the details of the various prediction at the different time horizons.

Forecast of the n = 30 data point

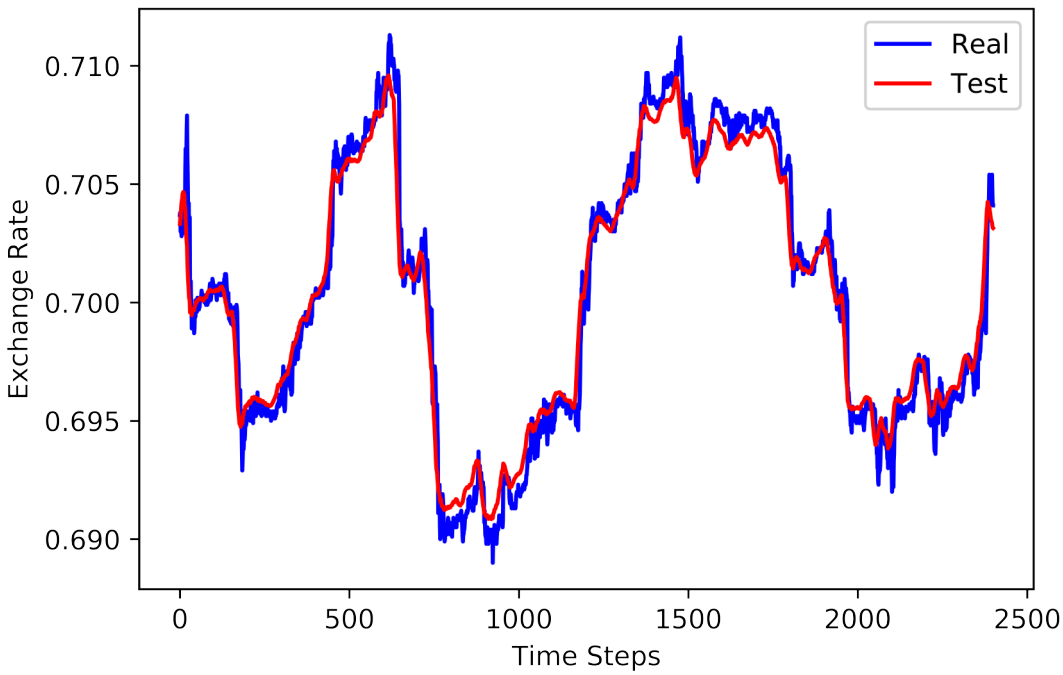
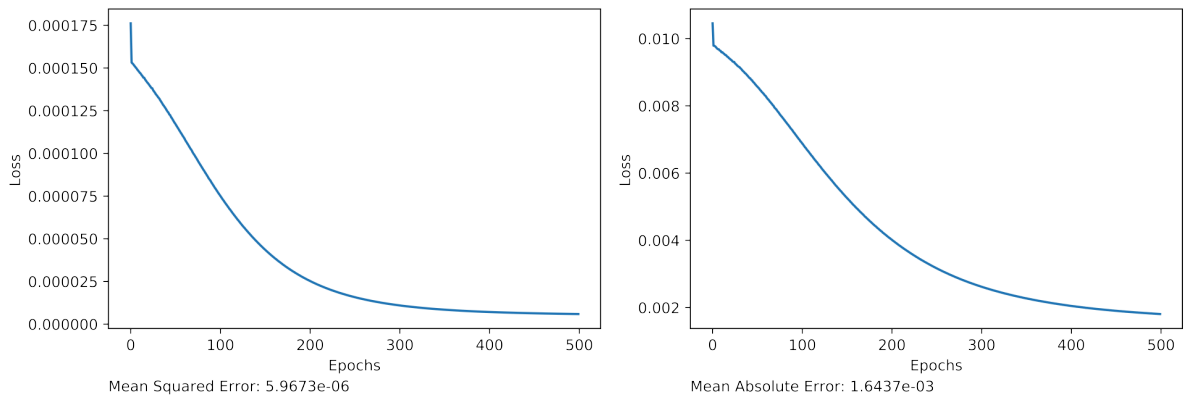


Figure 5.15: Test forecasted data as compared to the real data

In figure 5.15, it is shown in blue the real data and in red the forecasted test data. As it can be seen, the neural network was able to predict accurately the test set, with the forecasted data almost perfectly overlapping with the real data. We can see how this forecast is very similar to the one shown befor for $n = 18$, but it is less precise and smoother. We can also look at the Mean Squared Error and the Mean Absolute Error, to see how precisely the neural network could catch the dataset features.



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.16: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

As we can see from figure 5.16, both the errors drop radically in the first 200/300 iterations to plateau afterwards. This means that the error was reduced to a point in which the neural network couldn't adjust the weighs anymore, and therefore couldn't obtain a smaller error. We can also notice that the mean squared error is $5.9673x10^{-6}$ while the mean absolute error is $1.6437x10^{-3}$. Both are really small, and very close to the ones calculated for the data point $n = 18$.

Forecast of the n = 48 data point

In figure 5.17, it is shown in blue the real data and in red the forecasted test data. As it can be seen, the neural network was able to predict accurately the test set, with the forecasted data almost perfectly overlapping with the real data. It appears more obious how the forecast is starting to loose precision, and therefore smoothing all the fast movements which are present in the real data. It is still able to predict pretty well the trend of the real data. We can also look at the Mean Squared Error and the Mean Absolute Error, to see how precisely the neural network could catch the dataset features.

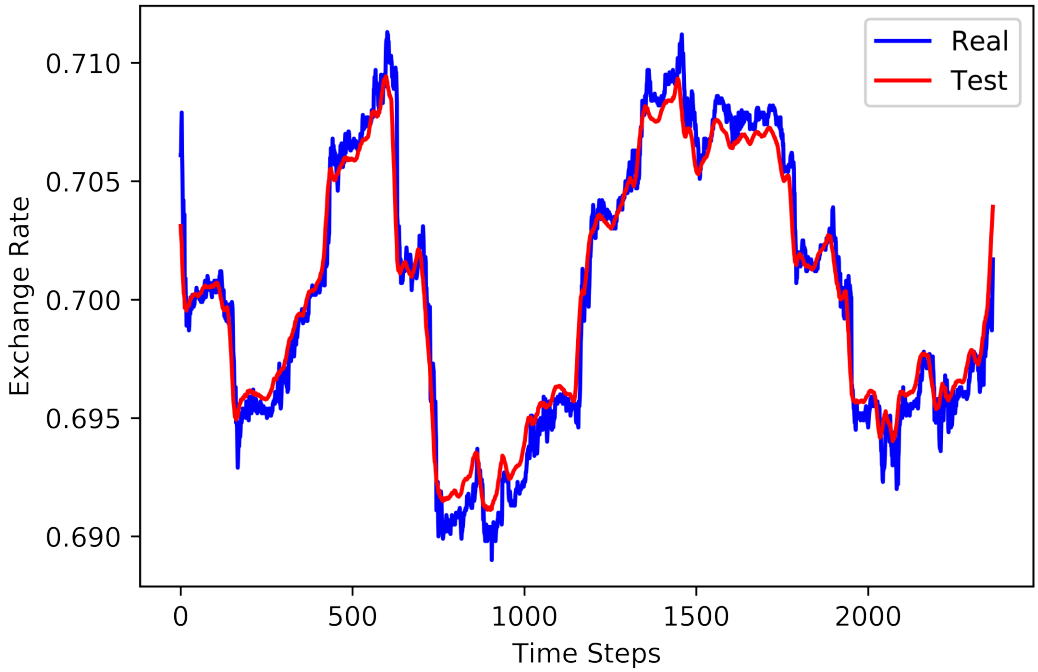
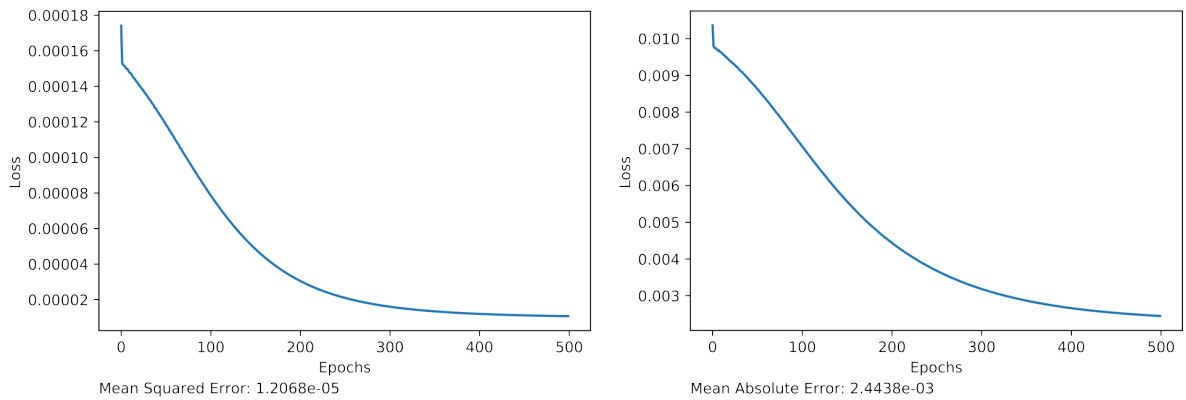


Figure 5.17: Test forecasted data as compared to the real data



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.18: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

As we can see from figure 5.18, both the errors drop radically in the first 200/300 iterations to plateau afterwards. This means that the error was reduced to a point in which the neural network couldn't adjust the weighs anymore, and therefore couldn't obtain a smaller error. We can also notice that the mean squared error is $1.2068x10^{-5}$ while the mean absolute error is $2.4438x10^{-3}$. Both are still very really small, and very close to the one calculated for the

data point n=18.

Forecast of the n = 78 data point

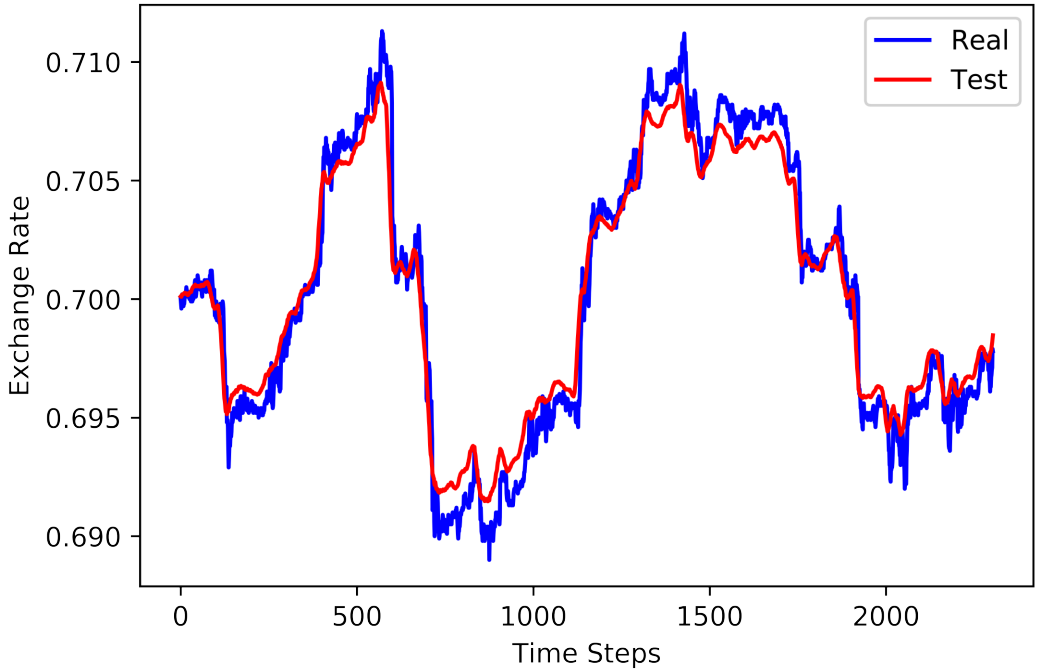
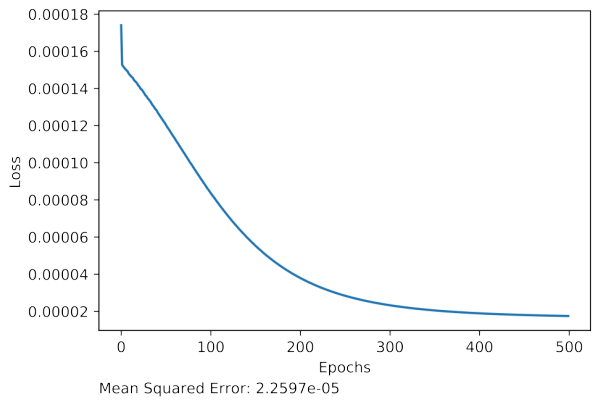
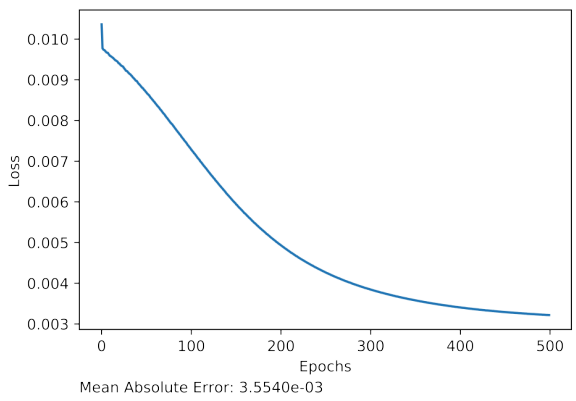


Figure 5.19: Test forecasted data as compared to the real data

In figure 5.19, it is shown in blue the real data and in red the forecasted test data. As it can be seen, the neural network was able to predict accurately the test set, with the forecasted data almost perfectly overlapping with the real data. It can be seen in a much clear way now how the forecast is starting to loose precision, something that we can see specially near the peaks of the graph. It is still able to predict pretty well the trend of the real data. We can also look at the Mean Squared Error and the Mean Absolute Error, to see how precisely the neural network could catch the dataset features.



(a) Mean Squared Error



(b) Mean Absolute Error

Figure 5.20: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

As we can see from figure 5.20, both the errors drop radically in the first 200/300 iterations to plateau afterwards. This means that the error was reduced to a point in which the neural network couldn't adjust the weighs anymore, and therefore couldn't obtain a smaller error. We can also notice that the mean squared error is $2.2597x10^{-5}$ while the mean absolute error is $3.5540x10^{-3}$. It continue to be very small and in line with the values find before.

Forecast of the n = 144 data point

In figure 5.21, it is shown in blue the real data and in red the forecasted test data. As it can be seen, the neural network was able to predict accurately the test set, with the forecasted data overlapping in the periods of less volatility, but it completely loose precision, specially near the peaks of the graph, where teh network was able to only predict the trend. We can also look at the Mean Squared Error and the Mean Absolute Error, to see how precisely the neural network could catch the dataset features.

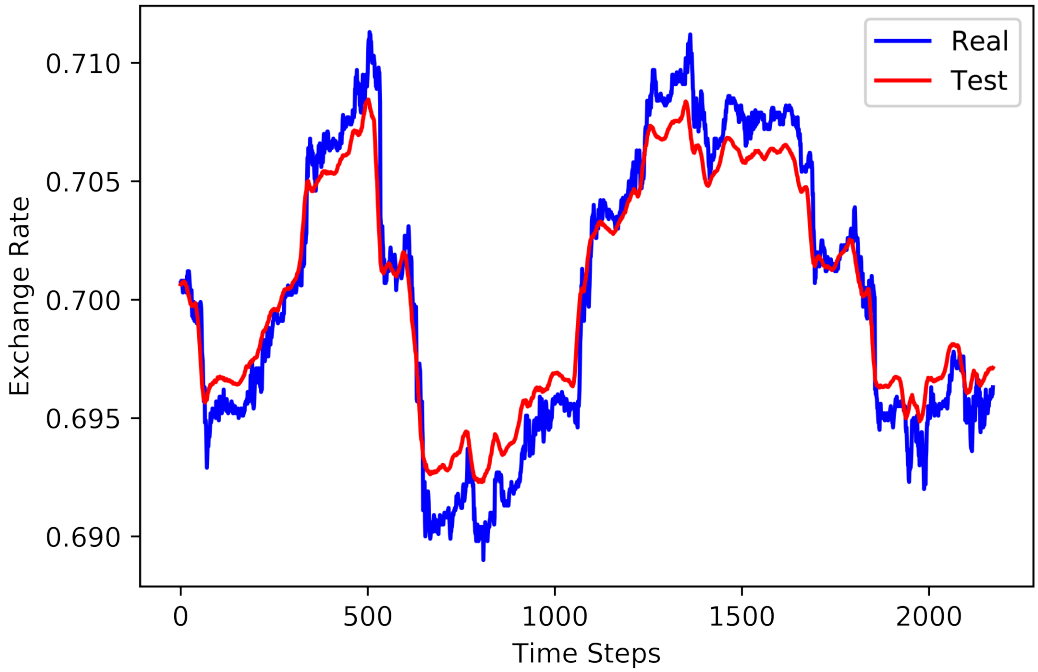
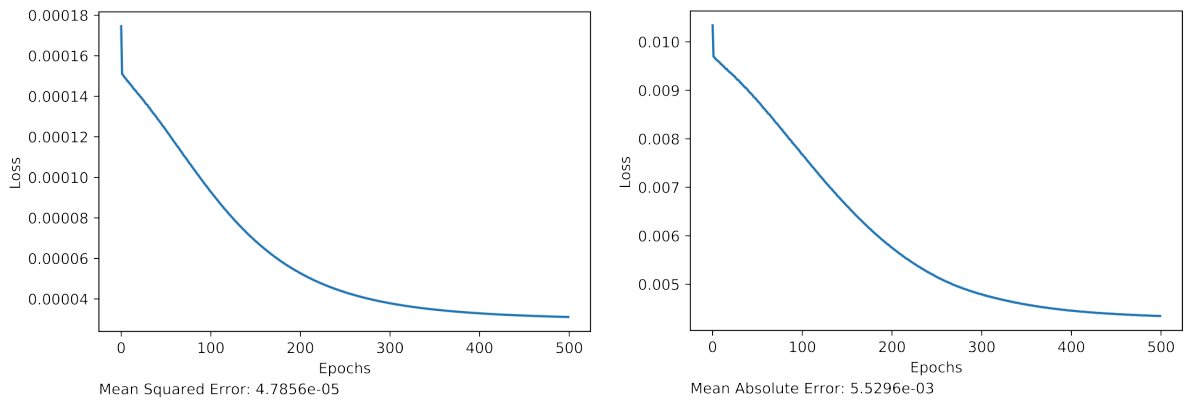


Figure 5.21: Test forecasted data as compared to the real data



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.22: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

As we can see from figure 5.22, both the errors drop radically in the first 200/300 iterations to plateau afterwards. This means that the error was reduced to a point in which the neural network couldn't adjust the weighs anymore, and therefore couldn't obtain a smaller error. We can also notice that the mean squared error is $4.7856x10^{-5}$ while the mean absolute error is $5.5296x10^{-3}$. It still is very small and in line with the values find before.

5.3.2 High Volatility Period

For the low volatility period, I defined:

- train set from 10/01/2016 to 10/03/2016;
- the test set goes from 16/06/2016 to 06/07/2016.

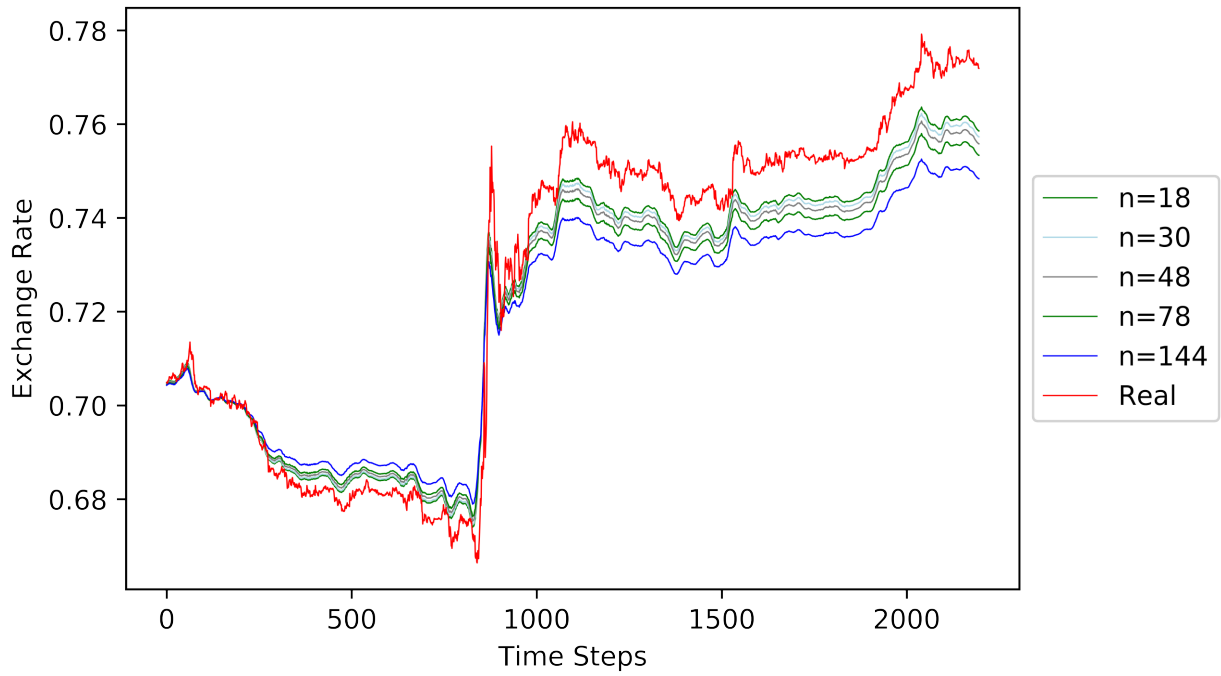


Figure 5.23: Comparison of the forecasted data with respect to the real data for the high volatility test set

In figure 5.23, we can notice the same characteristics shown in figure 5.14. Even in this case the short term forecast is more precise than the long term, with the latter shifted towards right. As we try to make longer term prediction, the results tend to loose their precision, maintaining some of the features of the real data, with the curve of the forecasted data is smoother for longer term as compared to lower term. We can also notice how the amplitude of the movements reduces, when forecasting longer terms.

Following, the details of the various prediction at the different time horizons.

Forecast of the n = 30 data point

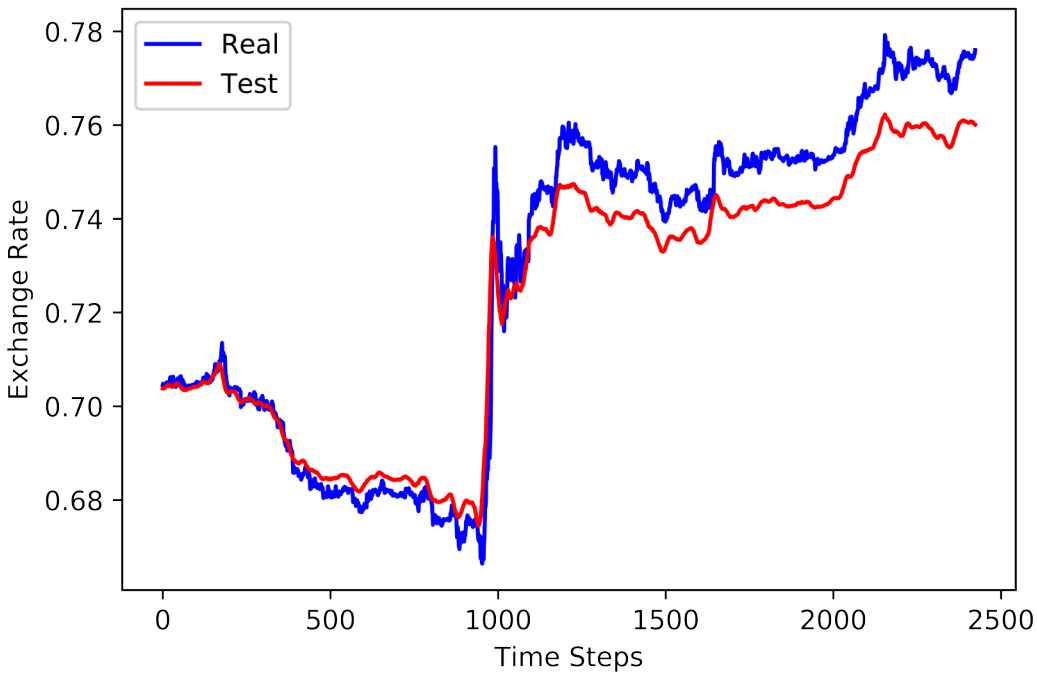
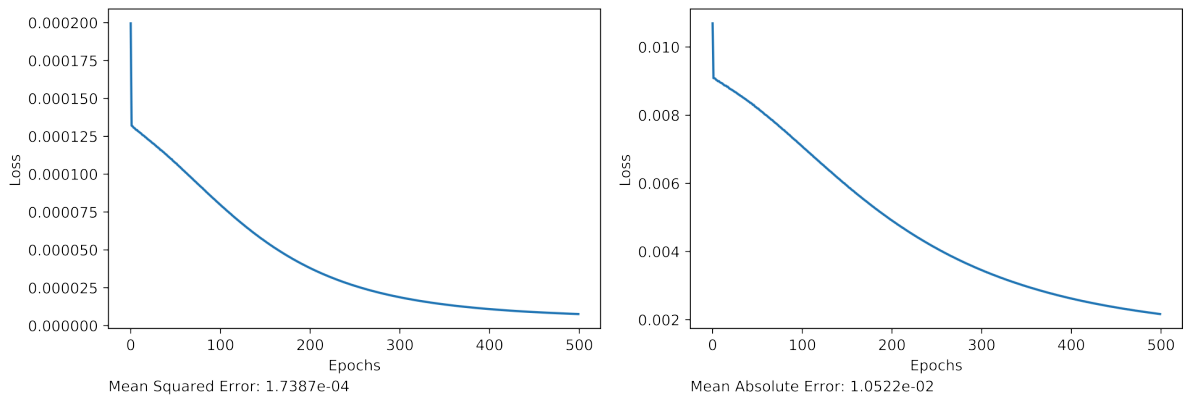


Figure 5.24: Test forecasted data as compared to the real data

In figure 5.24, it is shown in blue the real data and in red the forecasted test data. As it can be seen, the neural network was able to predict accurately the test set, with the forecasted data almost perfectly overlapping with the real data. We can see how this forecast is very similar to the one shown befor for $n = 18$, but it is less precise and smoother. We can also look at the Mean Squared Error and the Mean Absolute Error, to see how precisely the neural network could catch the dataset features.



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.25: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

As we can see from figure 5.25, both the errors drop radically in the first 300/400 iterations to plateau afterwards. This means that the error was reduced to a point in which the neural network couldn't adjust the weighs anymore, and therefore couldn't obtain a smaller error. We can also notice that the mean squared error is $1.7387x10^{-4}$ while the mean absolute error is $1.0522x10^{-2}$. Both are really small, and very close to the ones calculated for the data point $n = 18$.

Forecast of the n = 48 data point

In figure 5.26, it is shown in blue the real data and in red the forecasted test data. As it can be seen, the neural network was able to predict accurately the test set, with the forecasted data almost perfectly overlapping with the real data. It appears more obious how the forecast is starting to loose precision, and therefore smoothing all the fast movements which are present in the real data. It is still able to predict pretty well the trend of the real data. We can also look at the Mean Squared Error and the Mean Absolute Error, to see how precisely the neural network could catch the dataset features.

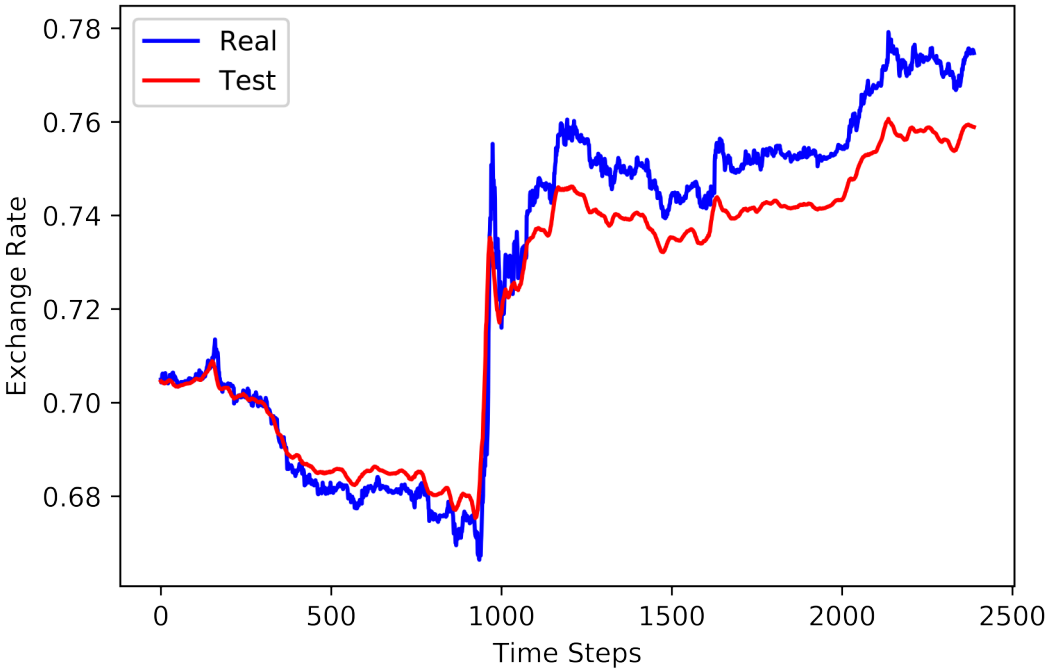
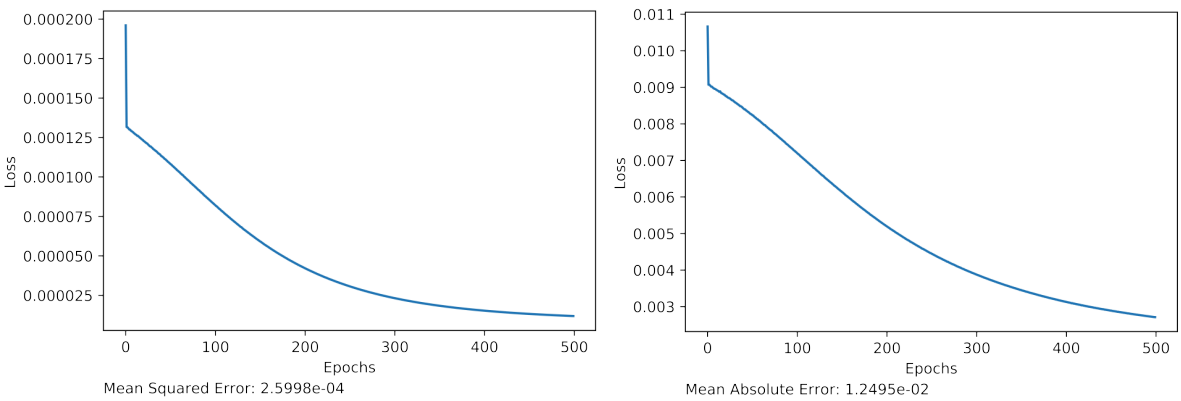


Figure 5.26: Test forecasted data as compared to the real data



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.27: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

As we can see from figure 5.27, both the errors drop radically in the first 300/400 iterations to plateau afterwards. This means that the error was reduced to a point in which the neural network couldn't adjust the weighs anymore, and therefore couldn't obtain a smaller error. We can also notice that the mean squared error is $2.5998x10^{-4}$ while the mean absolute error is $1.2495x10^{-2}$. Both are still very really small, and very close to the one calculated for the

data point n=18.

Forecast of the n = 78 data point

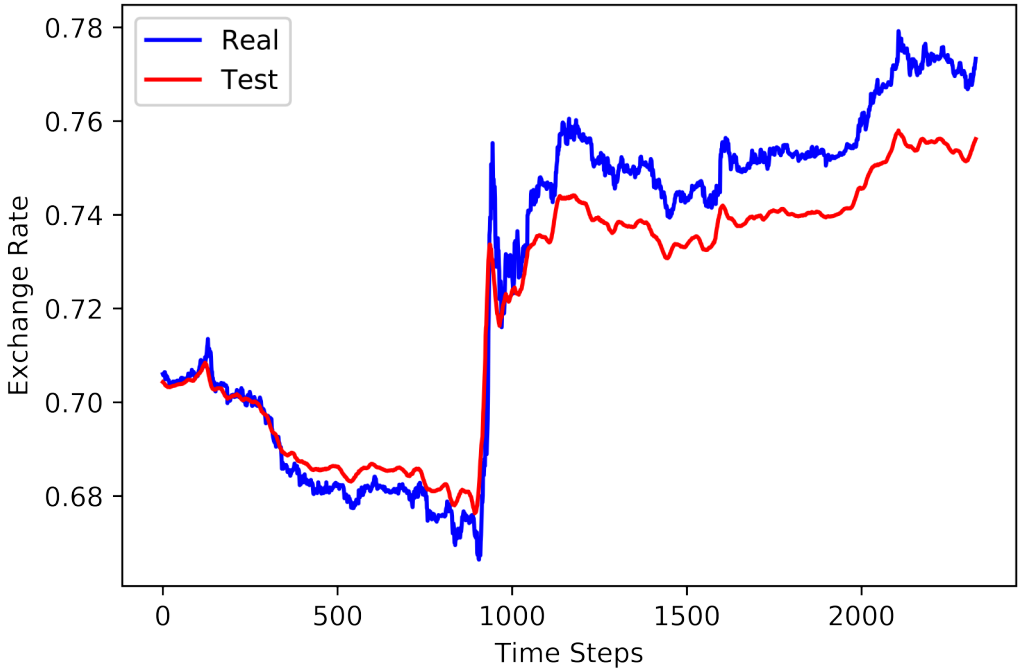
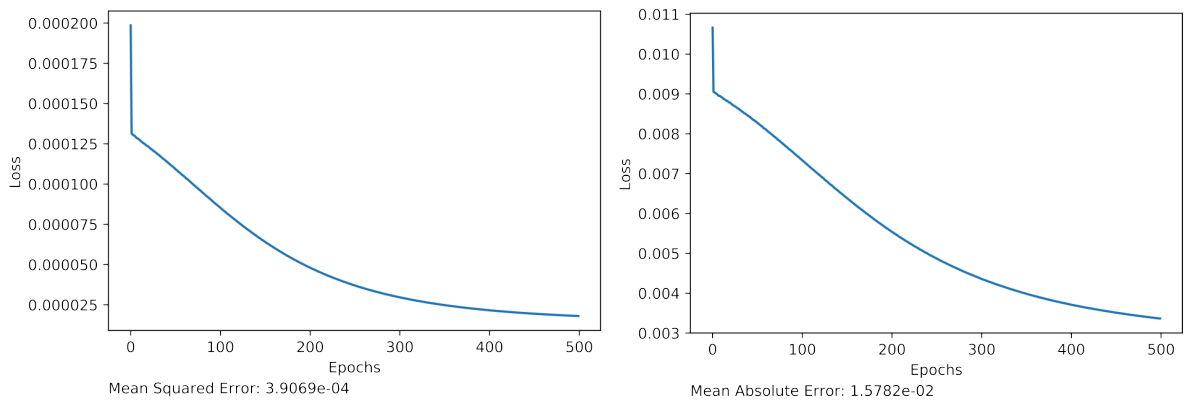


Figure 5.28: Test forecasted data as compared to the real data

In figure 5.28, it is shown in blue the real data and in red the forecasted test data. As it can be seen, the neural network was able to predict accurately the test set, with the forecasted data almost perfectly overlapping with the real data. It can be seen in a much clear way now how the forecast is starting to loose precision, something that we can see specially near the peaks of the graph. It is still able to predict pretty well the trend of the real data. We can also look at the Mean Squared Error and the Mean Absolute Error, to see how precisely the neural network could catch the dataset features.



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.29: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

As we can see from figure 5.29, both the errors drop radically in the first 300/400 iterations to plateau afterwards. This means that the error was reduced to a point in which the neural network couldn't adjust the weighs anymore, and therefore couldn't obtain a smaller error. We can also notice that the mean squared error is $3.9069x10^{-4}$ while the mean absolute error is $1.5782x10^{-2}$. It continue to be very small and in line with the values find before.

Forecast of the n = 144 data point

In figure 5.30, it is shown in blue the real data and in red the forecasted test data (the train set would have been equal to the one shown before). As it can be seen, the neural network was able to predict accurately the test set, with the forecasted data overlapping in the periods of less volatility, but it completely loose precision, specially near the peaks of the graph, where the network was able to only predict the trend. We can also look at the Mean Squared Error and the Mean Absolute Error, to see how precisely the neural network could catch the dataset features.

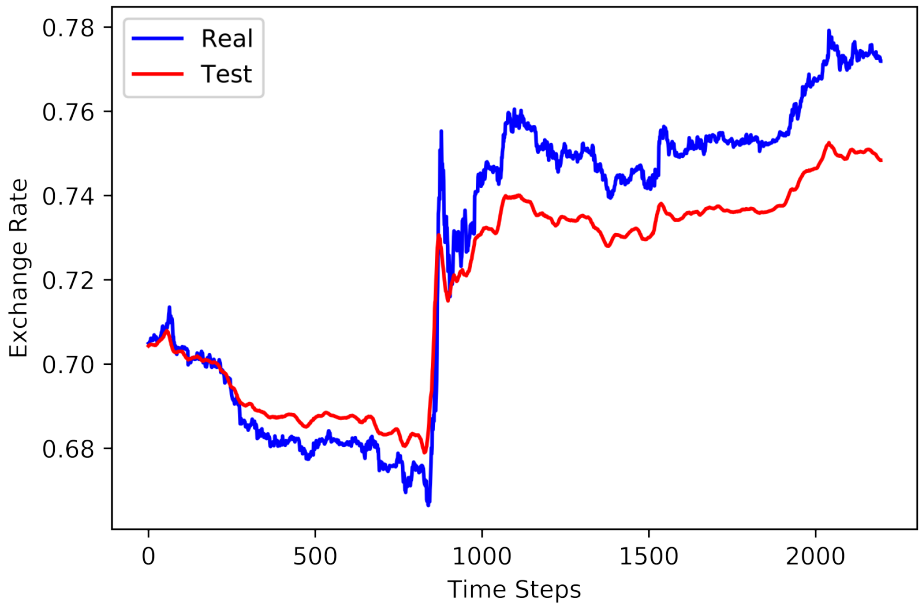
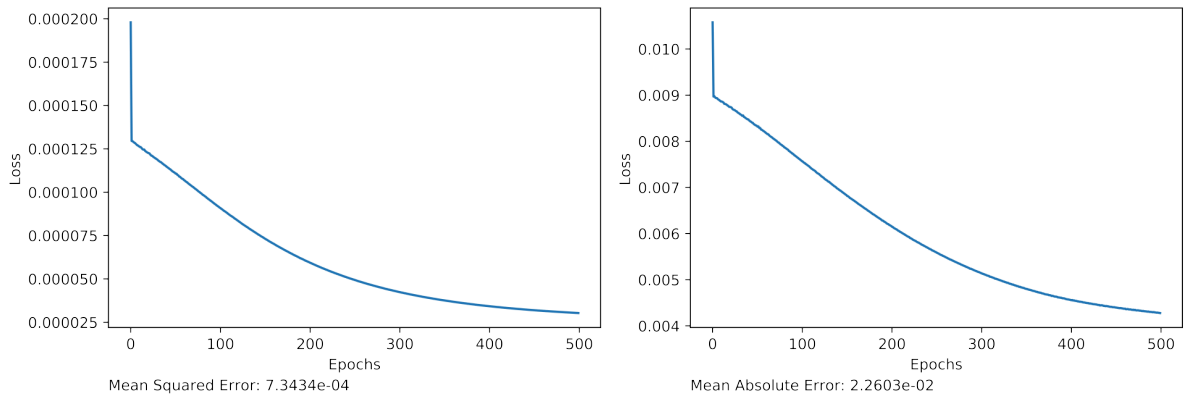


Figure 5.30: Test forecasted data as compared to the real data



(a) Mean Squared Error

(b) Mean Absolute Error

Figure 5.31: The Mean Squared Error and the Mean Absolute Error are plotted with respect to the iteration of the algorithm

As we can see from figure 5.31, both the errors drop radically in the first 300/400 iterations to plateau afterwards. This means that the error was reduced to a point in which the neural network couldn't adjust the weighs anymore, and therefore couldn't obtain a smaller error. We can also notice that the mean squared error is $7.3434x10^{-4}$ while the mean absolute error is $2.2603x10^{-2}$. It still is very small and in line with the values find before.

Chapter 6

Investment Simulation

Following the results obtained in the previous chapter, I decided to test them in a real life situation, simulating an investment utilising the forecasted data as indicator of whether to invest or not. What I was particularly interested in, was to see how an investment strategy based on the results of the last part of the previous chapter would perform. In fact, we saw that the prediction accuracy changes when varying the time horizon of the forecast, and it is clear that the farther in time we go, the less accurate will be the prediction. At the same time though, it loses the randomness of the very short term prediction while maintaining the trend of the real data.

To perform my analysis I assumed two different strategy:

- *continuous investment*: the algorithm opens a position at every data point, when the percentage difference with the previous is different from 0;
- *investment limited by a threshold*: the algorithm opens a position only when a certain threshold is exceeded in term of percentage difference between one data point and the previous.

For simplicity, there are no transaction costs (these are partially emulated with the threshold), short selling is allowed and there are no budget constraints (I assumed to have unlimited availability of liquidity).

The investment decision is made upon the signals given by the forecast of the test set. I used the test set for both the low and high volatility periods. To obtain the forecasts of the periods, the neural network has been trained on low volatility data, as it was shown before to be better performing.

The algorithm opens a positions of USD 1000 on the USD/GBP exchange rate, and the profit is calulated at the time of the investment: the operations are independent from each

other.

6.1 Continuous Investment

In this section we'll see how if an investment strategy which opens a position every time a data point of the forecasted test set is greater than or less than the previous data point.

To run the simulation I used both the low and high volatility period dataset, and for each I used all the time horizon seen in the previous chapter: n=18, n=30, n=48, n=78 and n=144, corresponding respectively to 10 minutes, 2 hours, 5 hours, 10 hours and 21 hours after the data used for forecasting.

6.1.1 Low Volatility Investment Simulation

Simulation utilising the forecast of n = 18.

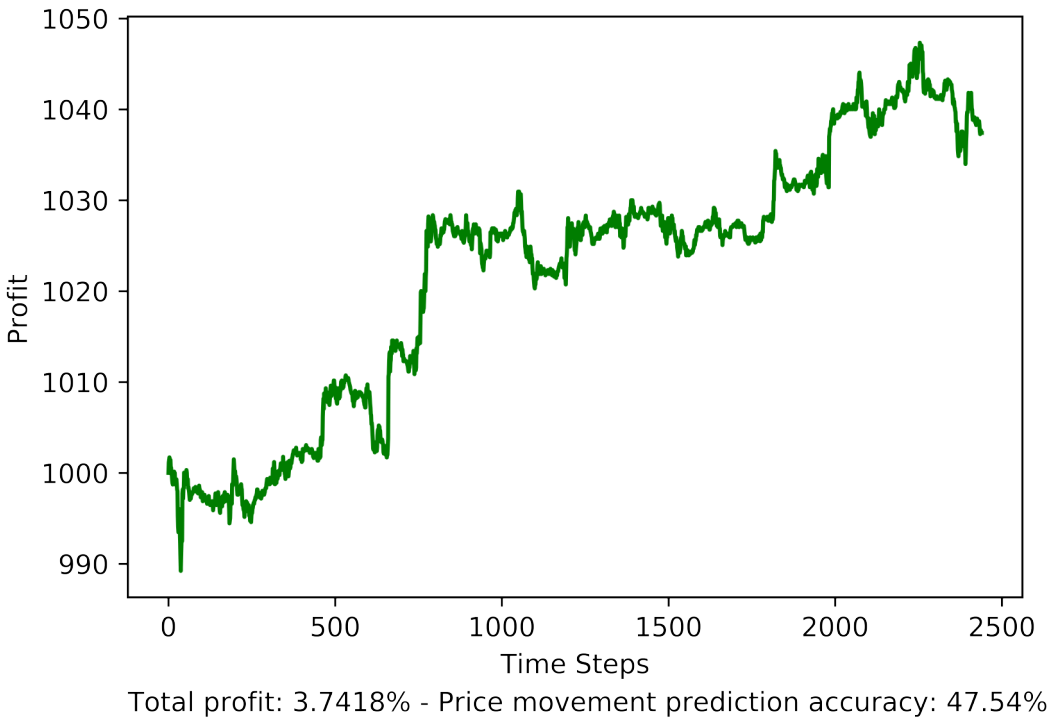


Figure 6.1: Cumulative profit using the forecast at 10 minutes

In figure 6.1, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 18$. The algorithm got a profit of 3.7418%, something that we were expecting

due to the high quality of the prediction with this time horizon seen in the previous chapter. What was less expected is the price movement prediction accuracy: confronting the price movement of the data forecasted by the neural network with the price movement of the real data when predicting 10 minutes ahead in time, the accuracy is only 47.54%. This is due to high level of randomness of the high frequency of the data. Predicting the next data point is subject to a lot of noise due to the lack of structure of the time series: to avoid this, the best approximation of the next data point, given the ones before, would be the previous itself.

Simulation utilising the forecast of n = 30.

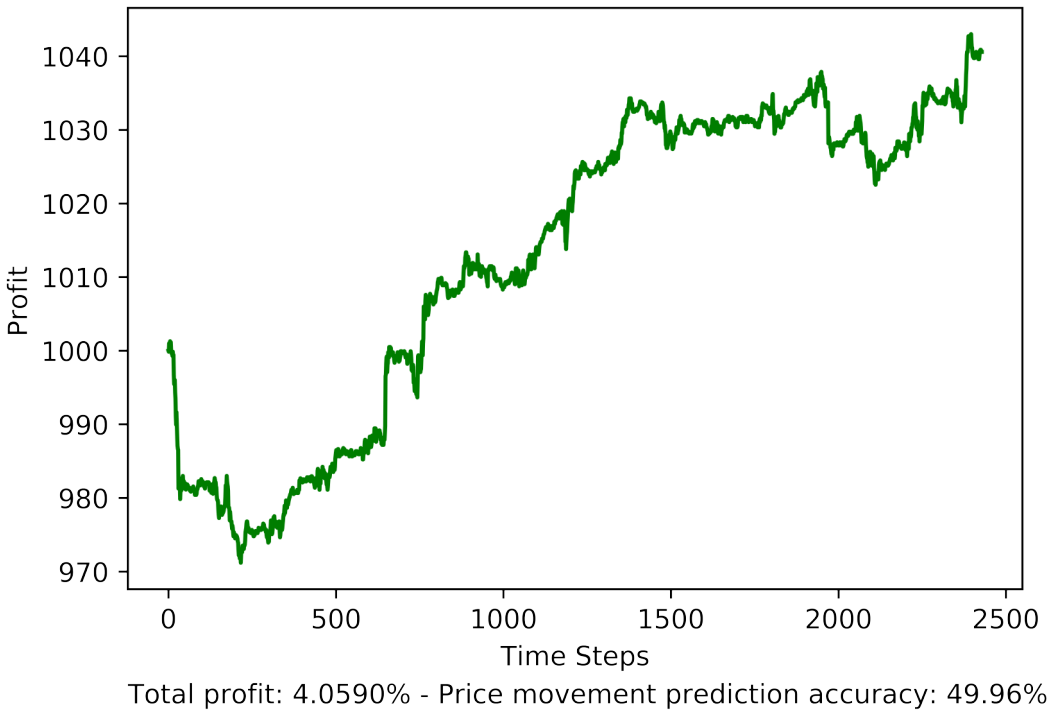


Figure 6.2: Cumulative profit using the forecast at 2 hours

In figure 6.2, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 30$. The algorithm got a profit of 4.0590%, something that we were expecting due to the high quality of the prediction with this time horizon seen in the previous chapter. Looking at the price movement prediction accuracy the accuracy of the 2 hours prediction is 49.96%.

Simulation utilising the forecast of n = 48.

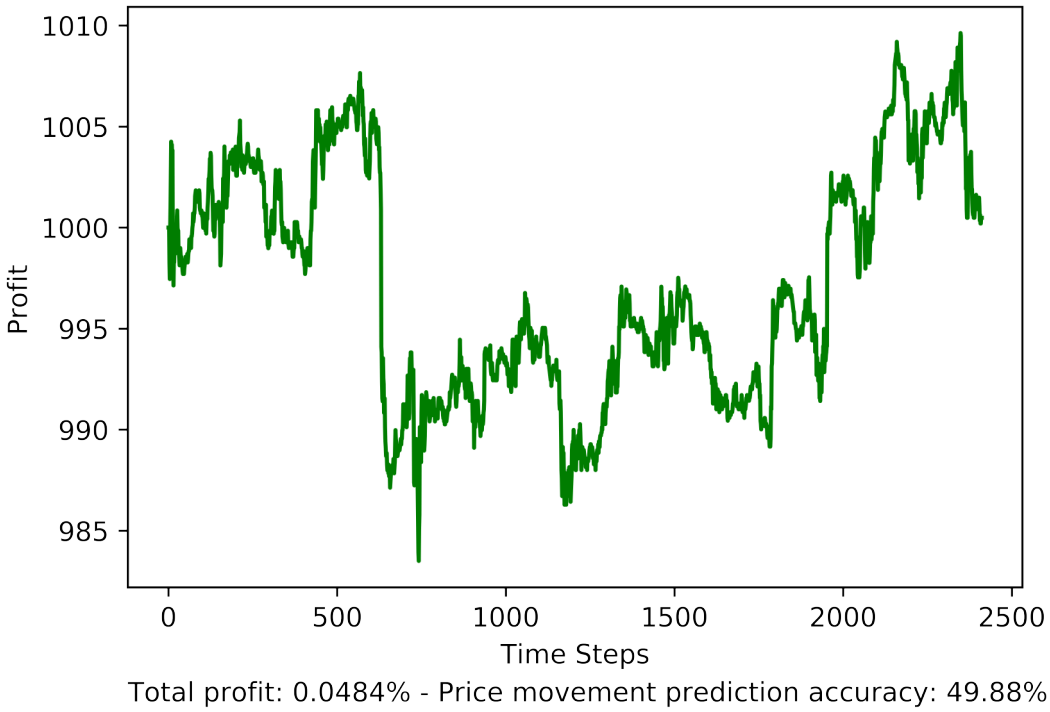


Figure 6.3: Cumulative profit using the forecast at 5 hours

In figure 6.3, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 48$. The algorithm got a profit of 0.0484%, something that we were expecting due to the high quality of the prediction with this time horizon seen in the previous chapter. Looking at the price movement prediction accuracy the accuracy of the 5 hours prediction is 49.88%.

Simulation utilising the forecast of n = 78.

In figure 6.4, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 78$. The algorithm got a profit of 3.4811%, something not expected due to the lack of precision of the prediction with this time horizon seen in the previous chapter. Intresting also in this case is the price movement prediction accuracy: confronting the price movement of the data forecasted by the neural network with the price movement of the real data when predicting 10 hours ahead in time, the accuracy is 49.92%.

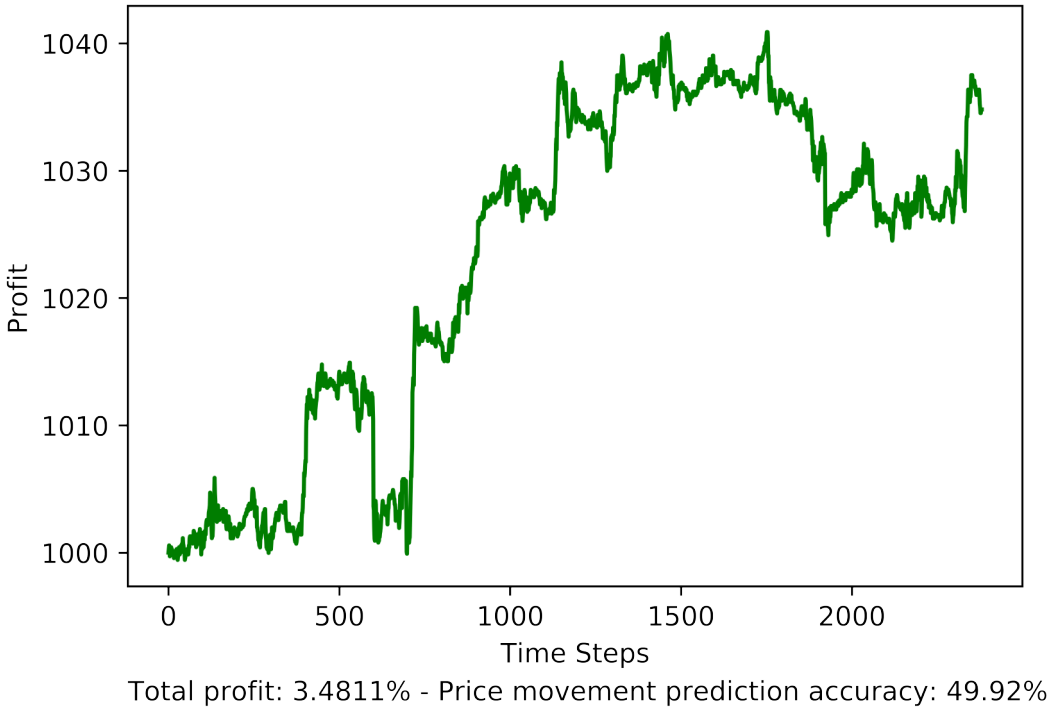


Figure 6.4: Cumulative profit using the forecast at 10 hours

Simulation utilising the forecast of n = 144.

In figure 6.5, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 144$. The algorithm got a profit of 3.2668%, something not expected due to the lack of precision of the prediction with this time horizon seen in the previous chapter. Besides the profit, what is really interesting is the price movement prediction accuracy: confronting the price movement of the data forecasted by the neural network with the price movement of the real data when predicting 21 hours ahead in time, the accuracy is 50.22%. This is the highest accuracy seen for this for the investment simulation in low volatile periods. Both the accuracy and the fact that we have a profit is related, as the longer term forecast loses the randomnes which characterise lower terms, and focuses on the general trend of the data.

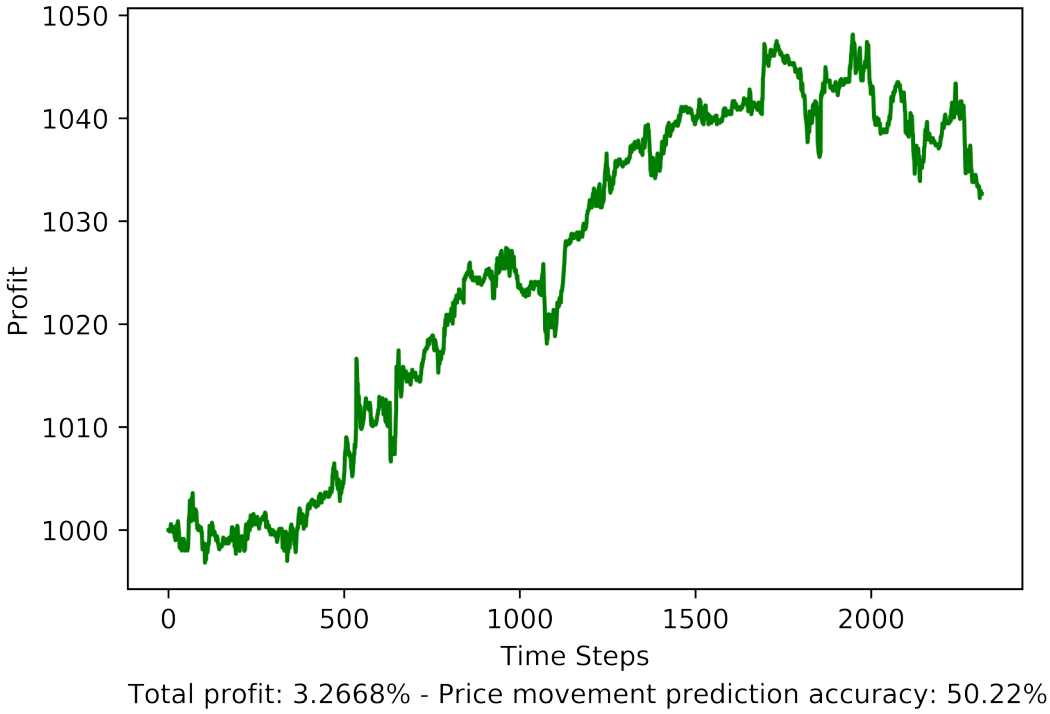


Figure 6.5: Cumulative profit using the forecast at 21 hours

6.1.2 High Volatility Investment Simulation

Simulation utilising the forecast of n = 18.

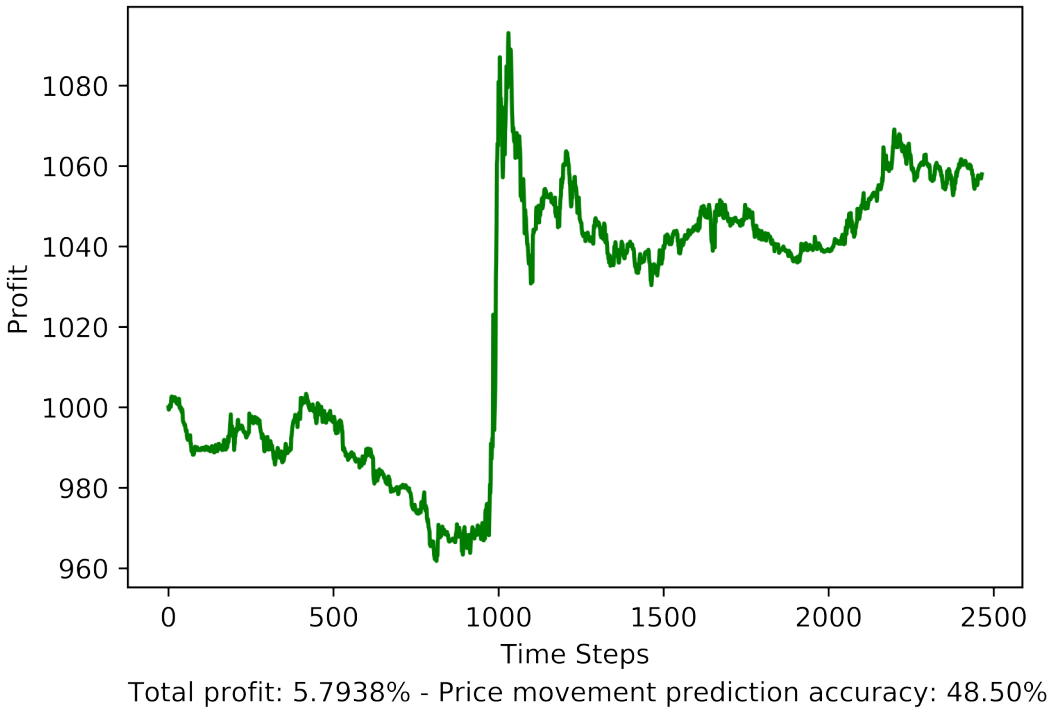


Figure 6.6: Cumulative profit using the forecast at 10 minutes

In figure 6.6, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 18$. The algorithm got a profit of 5.7938%, something that we were expecting due to the high quality of the prediction with this time horizon seen in the previous chapter. What was less expected is the price movement prediction accuracy: confronting the price movement of the data forecasted by the neural network with the price movement of the real data when predicting 10 minutes ahead in time, the accuracy is only 48.50%. This is due, as in the previous case, to high level of randomness of the high frequency of the data. Predicting the next data point is subject to a lot of noise due to the lack of structure of the time series: to avoid this, the best approximation of the next data point, given the ones before, would be the previous itself.

Simulation utilising the forecast of n = 30.

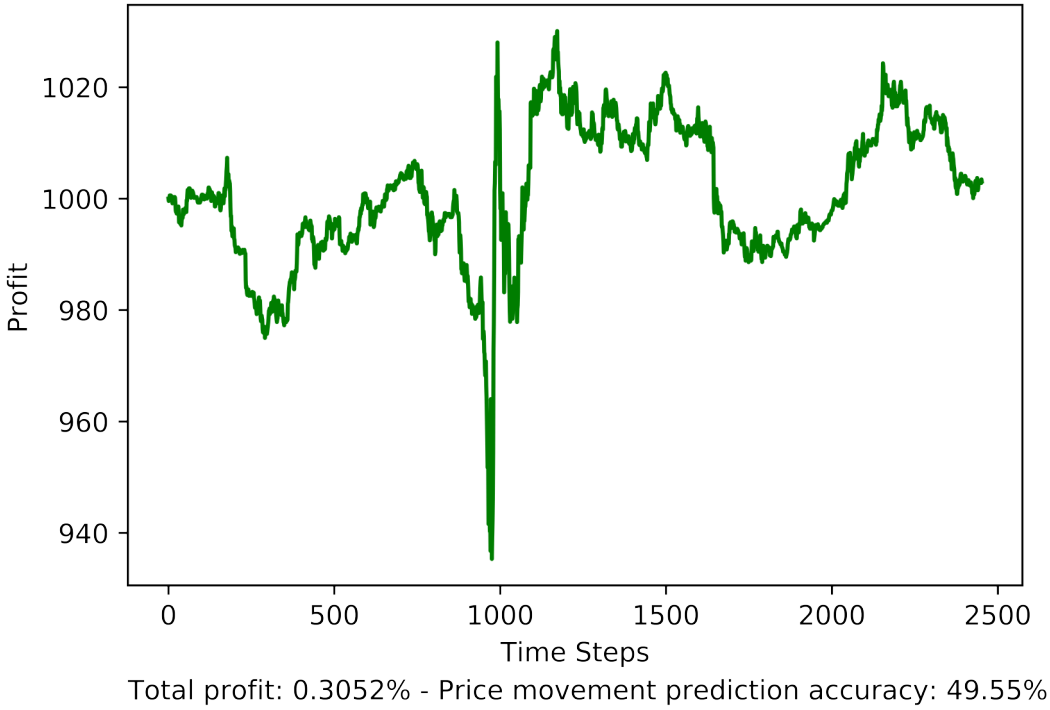


Figure 6.7: Cumulative profit using the forecast at 2 hours

In figure 6.7, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 30$. The algorithm got a profit of 0.3052%, something that we were expecting due to the high quality of the prediction with this time horizon seen in the previous chapter. Looking at the price movement prediction accuracy the accuracy of the 2 hours prediction is 49.55%. It is interesting to see how the wrong movement prediction, around the spike in the centre of the plot, almost caused a loss of more than 6%, but was later compensated by a move in the opposite direction.

Simulation utilising the forecast of n = 48.

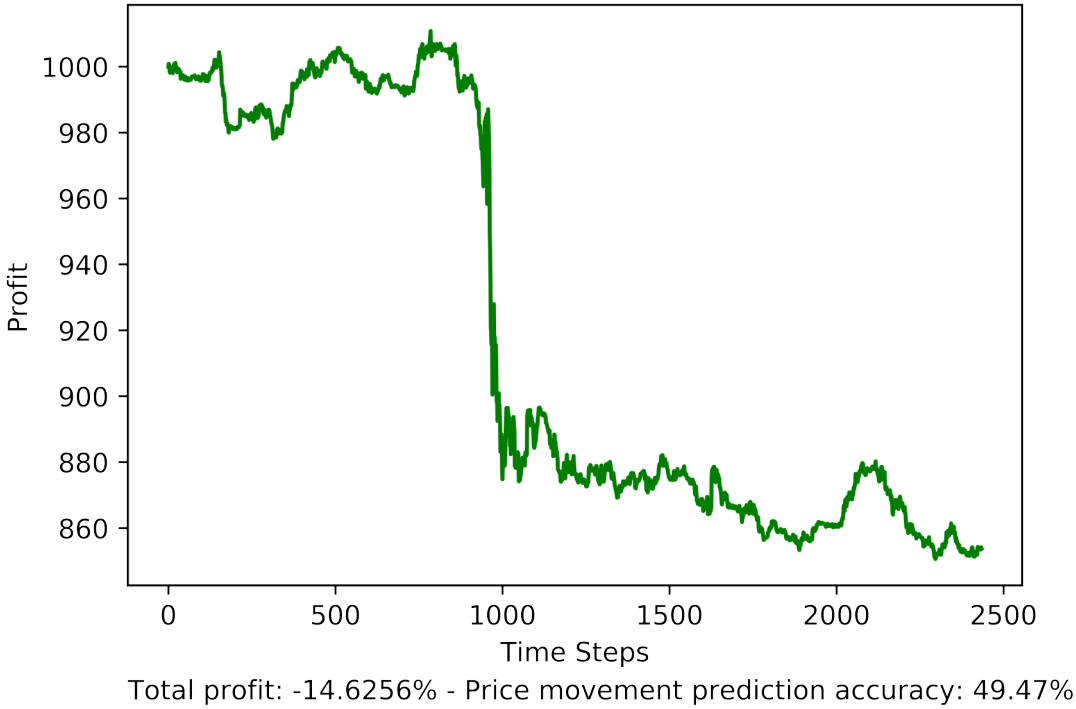


Figure 6.8: Cumulative profit using the forecast at 5 hours

In figure 6.8, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 48$. The algorithm got a profit of -14.6256%, something that we were expecting due to the high quality of the prediction with this time horizon seen in the previous chapter. Looking at the price movement prediction accuracy the accuracy of the 5 hours prediction is 49.47%. This is the first big loss we encounter in the simulation: although the accuracy of the price movement prediction is as high as before, due to the intrinsic nature of this dataset (can be seen in all the simulation a spike in the plot, corresponding to the Brexit days), it is very possible to have very high loss, as well as very high profit.

Simulation utilising the forecast of n = 78.

In figure 6.9, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 78$. The algorithm got a profit of 15.0559%, something not expected due to the lack of precision of the prediction with this time horizon seen in the previous chapter. Intresting also in this case is the price movement prediction accuracy: confronting the price

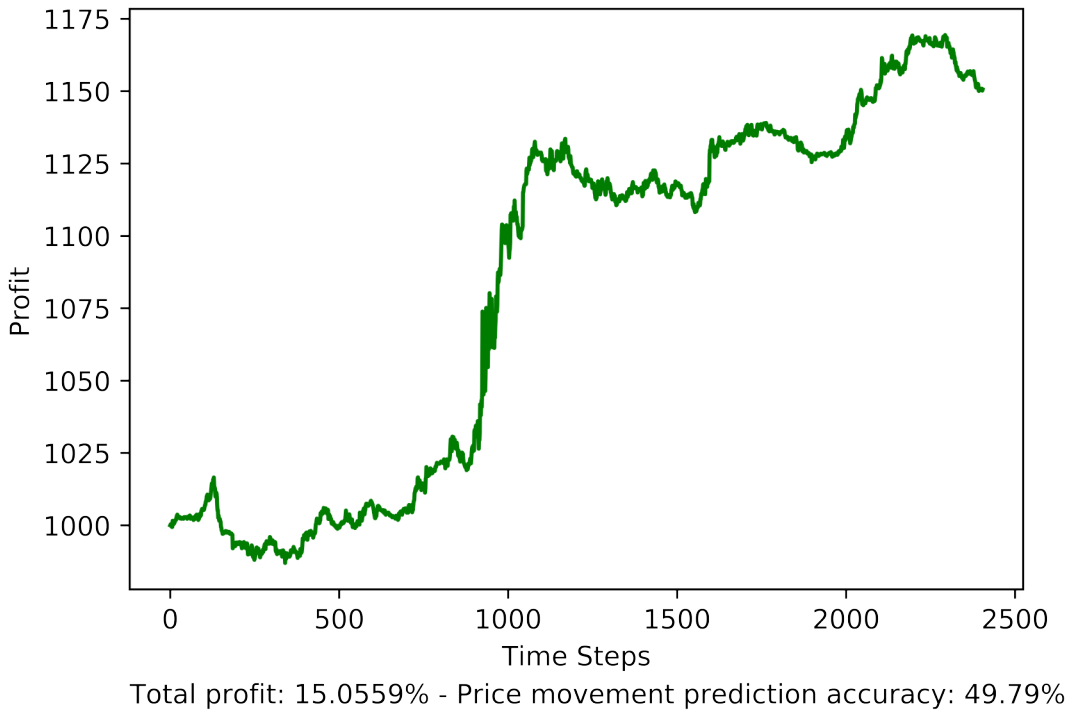


Figure 6.9: Cumulative profit using the forecast at 10 hours

movement of the data forecasted by the neural network with the price movement of the real data when predicting 10 hours ahead in time, the accuracy is 49.79%. Contrary to the previous case, we got a very big profit, the biggest in the continutou investing simulation. Again the accuracy is not that different from the one found for the low volatility period, but due to the nature of the dataset, as we can have huge losses, we can also have huge profits.

Simulation utilising the forecast of n = 144.

In figure 6.10, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 144$. The algorithm got a profit of 1.3679%, something not expected due to the lack of precision of the prediction with this time horizon seen in the previous chapter. Besides the profit, what is really interesting is the price movement prediction accuracy: confronting the price movement of the data forecasted by the neural network with the price movement of the real data when predicting 21 hours ahead in time, the accuracy is 50.26%. Also in this case, this is the highest accuracy seen for this for the investment simulation in high volatile periods. Both the accuracy and the fact that we have a profit is related, as the longer term forecast loses the randomnes which characterise lower terms, and focuses on the general

trend of the data.

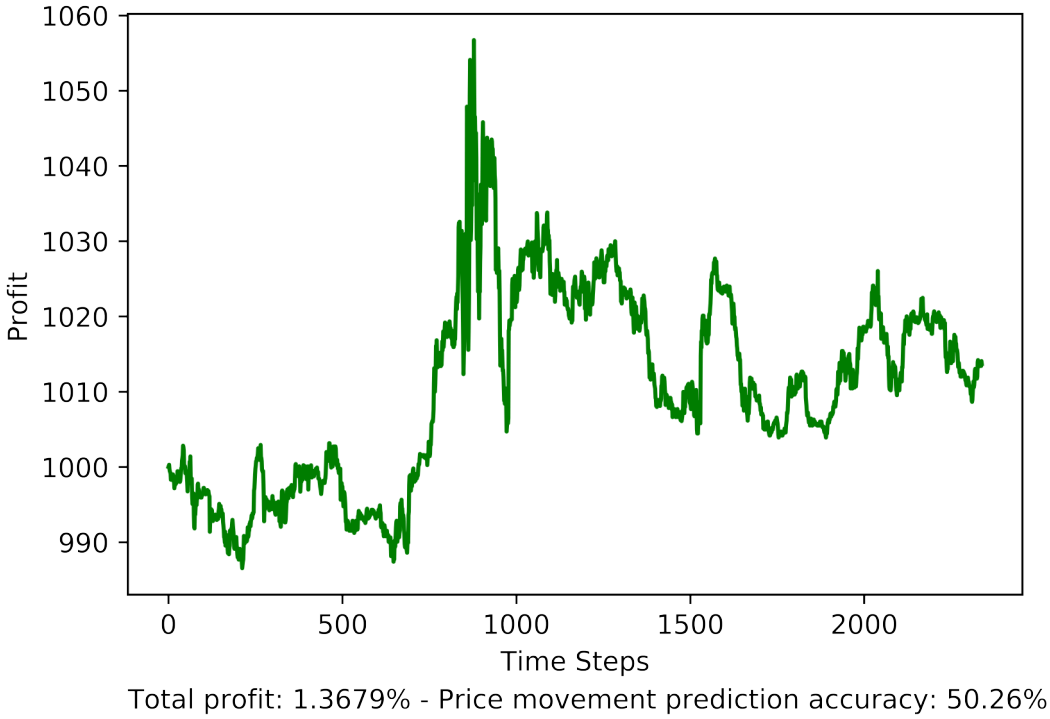


Figure 6.10: Cumulative profit using the forecast at 21 hours

6.2 Investment Limited by a Threshold

In this section we'll see how if an investment strategy which opens a position only when a certain threshold is exceeded. The threshold has been set at 0.10%. This value was found through trial and error: as the data point are so close to each other, it was difficult to find big differences between them. After different trials, the value which better performed for the simulation purpouse was 0.10%.

To implement the threshold, I had to slightly modify the code as follow:

```
p = 0.0010
indicator1 = []
beg = forecast[0]

for i in range(1,l-1):
    kk = (forecast[i]/beg)-1
    #print(kk)
    if kk >= p or kk<= -p:
        beg = forecast[i]
        if kk >= p:
            indicator1.append(1)
        elif kk <= -p:
            indicator1.append(-1)
    else:
        indicator1.append(0)

indicator1.insert(0,1)
```

In this part of the code, as explained in chapter 4, p is the threshold, and then a control is performed to check the percentage difference between two consecutive data: if it is greater than p, the data point is considered for the investment, else is discarded and the same comparison is done with the next one.

To run the simulation, as before, I used both the low and high volatility period dataset, and for each I used all the time horizon seen in the previous chapter: n=18, n=30, n=48, n=78 and n=144, corresponding respectively to 10 minutes, 2 hours, 5 hours, 10 hours and 21 hours after the data used for forecasting.

6.2.1 Low Volatility Investment Simulation

Simulation utilising the forecast of n = 18.

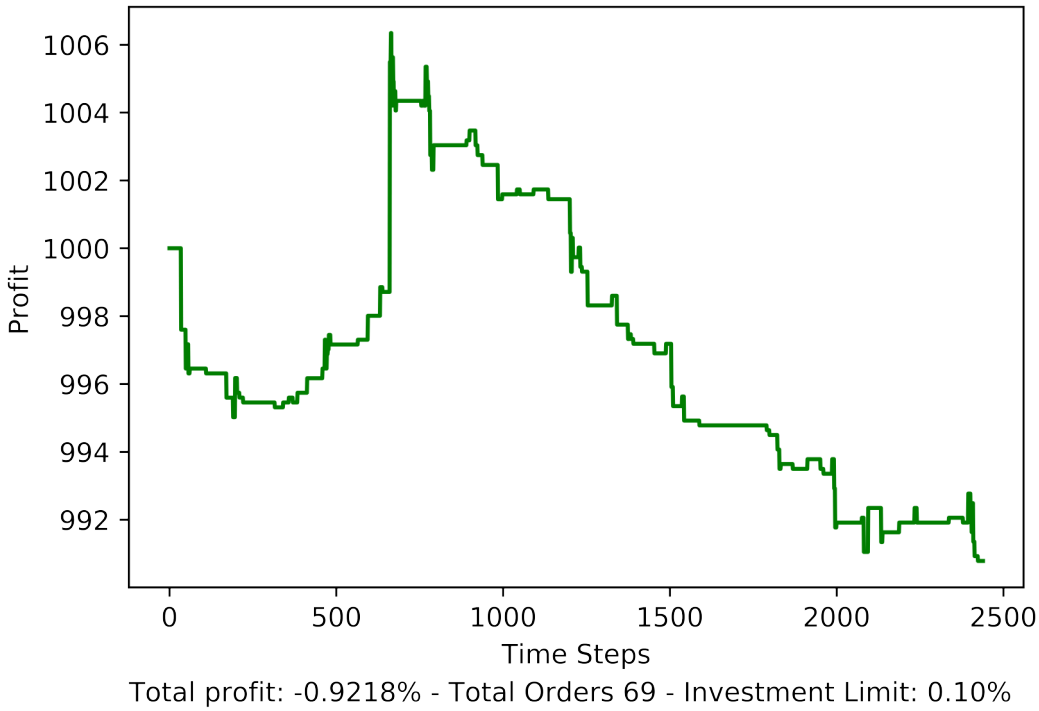


Figure 6.11: Cumulative profit using the forecast at 10 minutes

In figure 6.11, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 18$. The algorithm got a loss of 0.9218%, something different from what found in the simulation without threshold, but similar to the reality (in which we have transaction costs). We can see that the simulation only opened 69 positions, very less if compared with before, in which for every step a position was opened.

Simulation utilising the forecast of n = 30.

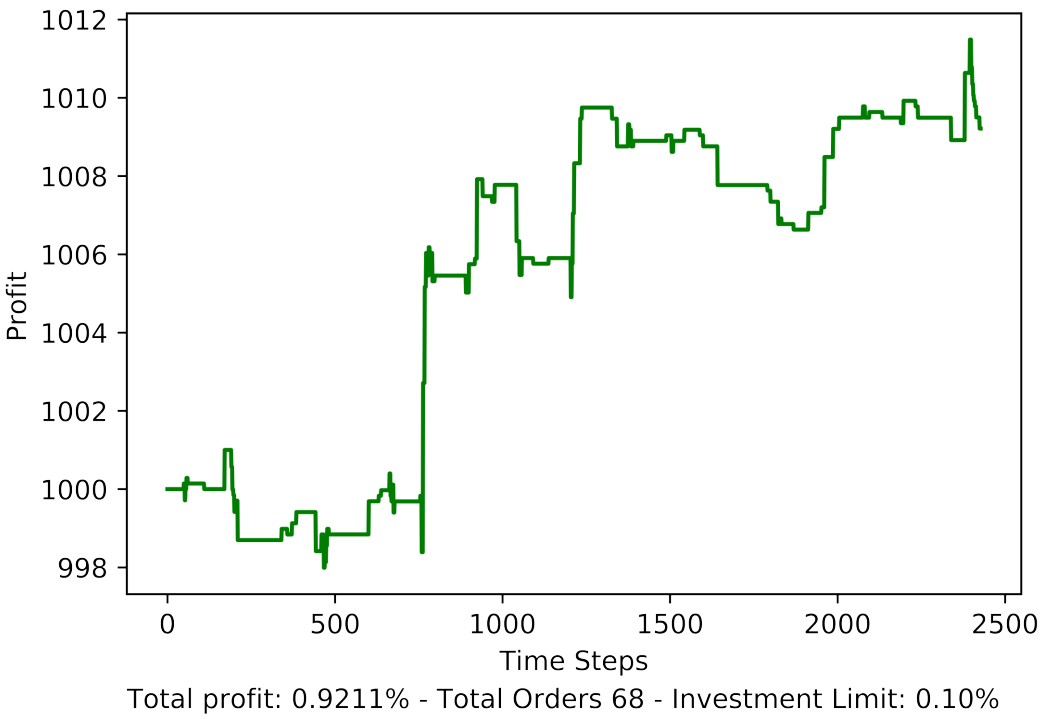


Figure 6.12: Cumulative profit using the forecast at 2 hours

In figure 6.12, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 30$. The algorithm got a profit of 0.9211%, different from the previous simulation, but again more realistic. It is important to notice as well the number of positions opened: also in this case it is a very low value, 68.

Simulation utilising the forecast of n = 48.

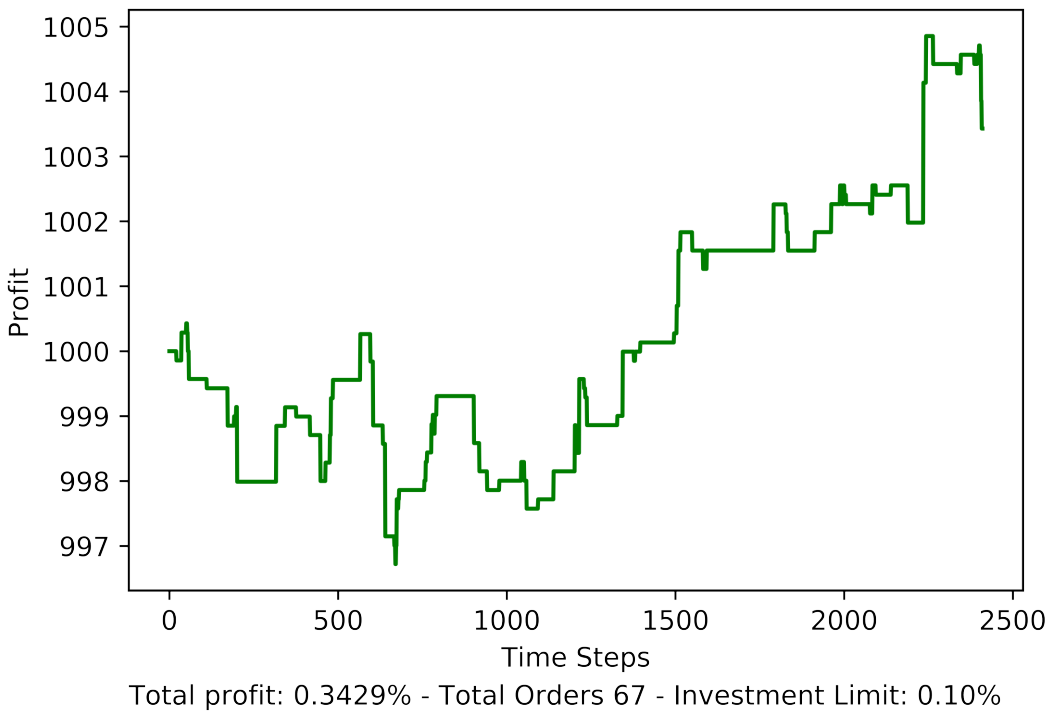


Figure 6.13: Cumulative profit using the forecast at 5 hours

In figure 6.13, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 48$. The algorithm got a profit of 0.3429%, this time closer to the one obtained before, and even greater. The positions opened are 67.

Simulation utilising the forecast of n = 78.

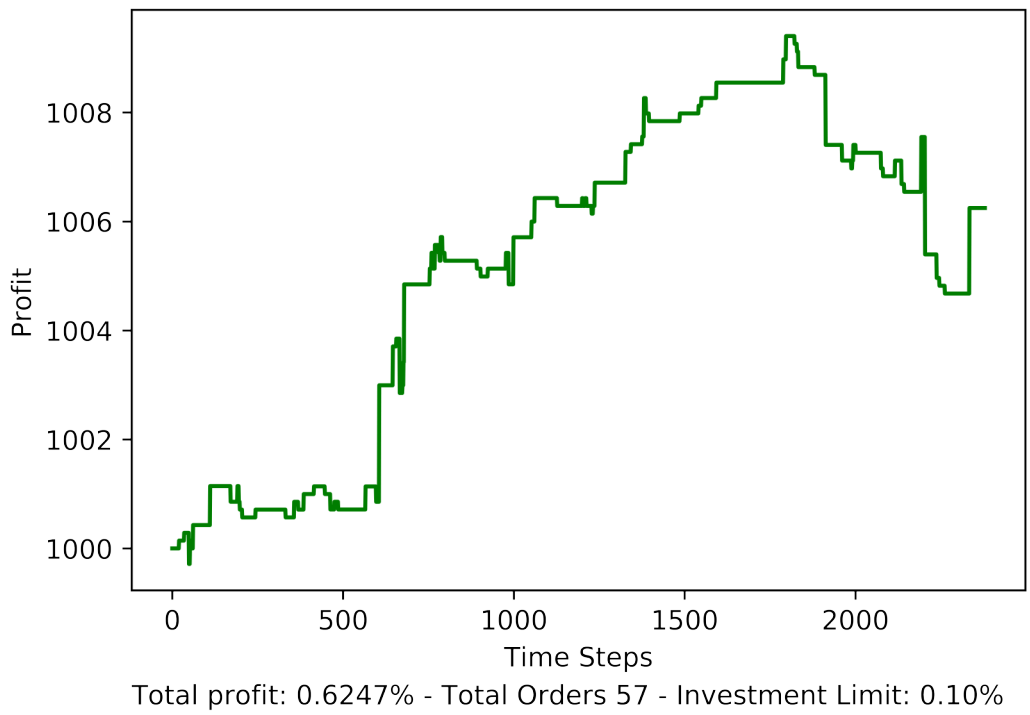


Figure 6.14: Cumulative profit using the forecast at 10 hours

In figure 6.14, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 78$. The algorithm got a profit of 0.6247%, less than the previous simulation, but more in line with the expectations, seen the low precision of the forecast. The position opened are 57.

Simulation utilising the forecast of n = 144.

In figure 6.15, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 144$. The algorithm got a profit of 0.1447%, less than before and in line with my expectation, seen the lack of precision of the prediction with this time horizon. The positions opened are even lower than the other cases, just 50.

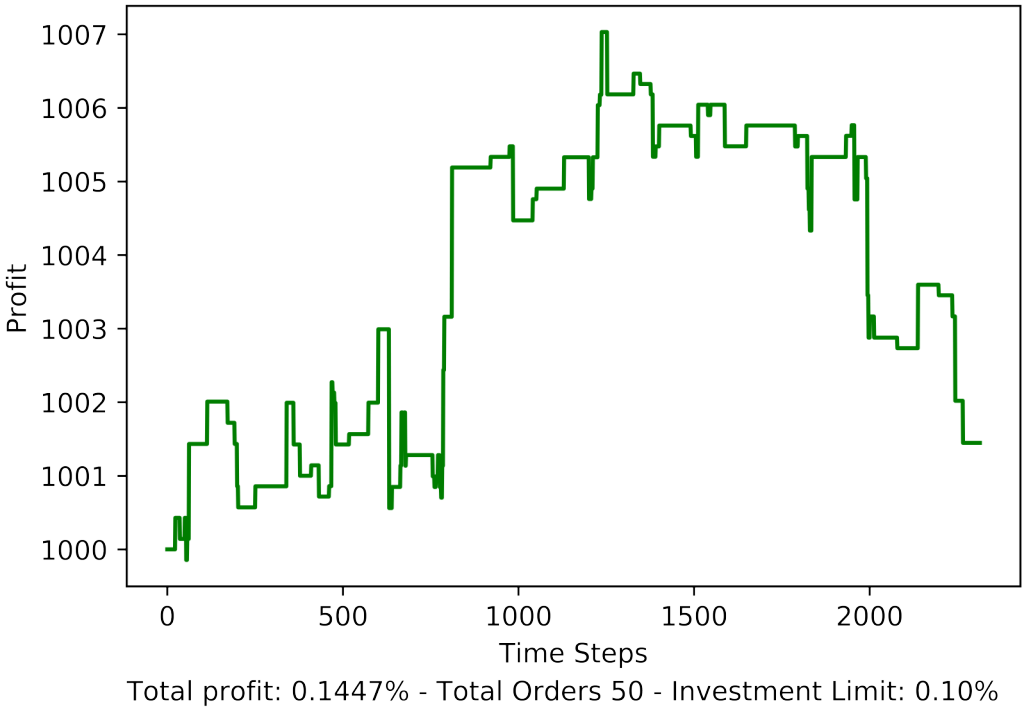


Figure 6.15: Cumulative profit using the forecast at 21 hours

6.2.2 High Volatility Investment Simulation

Simulation utilising the forecast of n = 18.

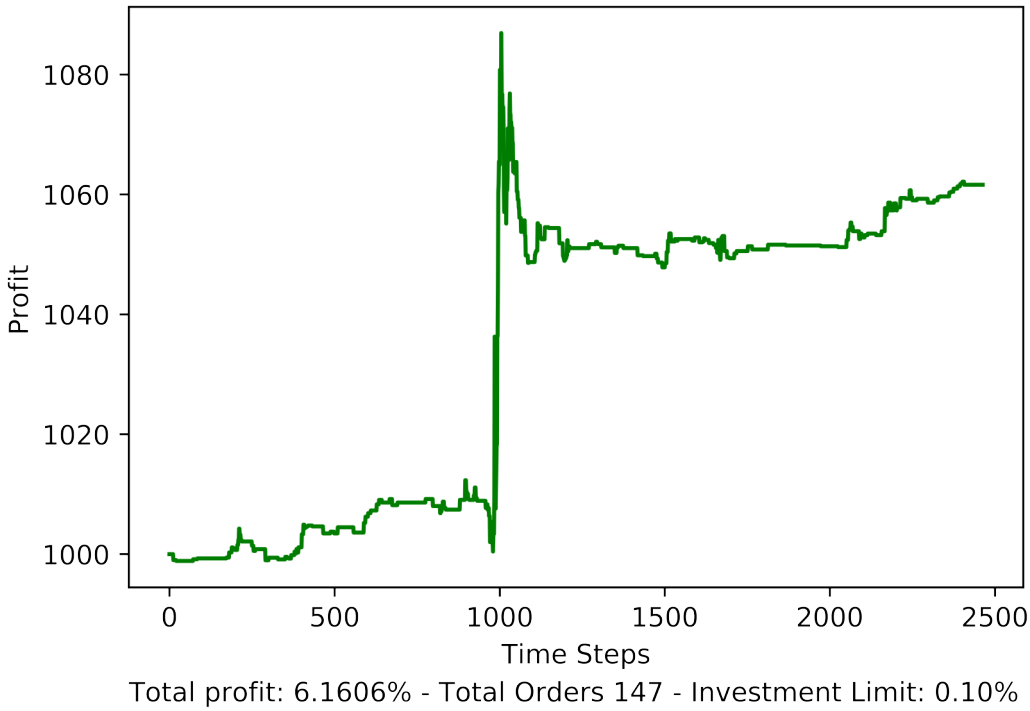


Figure 6.16: Cumulative profit using the forecast at 10 minutes

In figure 6.16, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 18$. The algorithm got a profit of 6.1606%, much higher than the case of low volatility. The volatility of the period has an impact also on the number of position opened, which are, in this case, 147, more than twice the ones seen for the low volatility period. In fact, being the data more volatile, it is more likely to have big differences between one data point and the other.

Simulation utilising the forecast of n = 30.

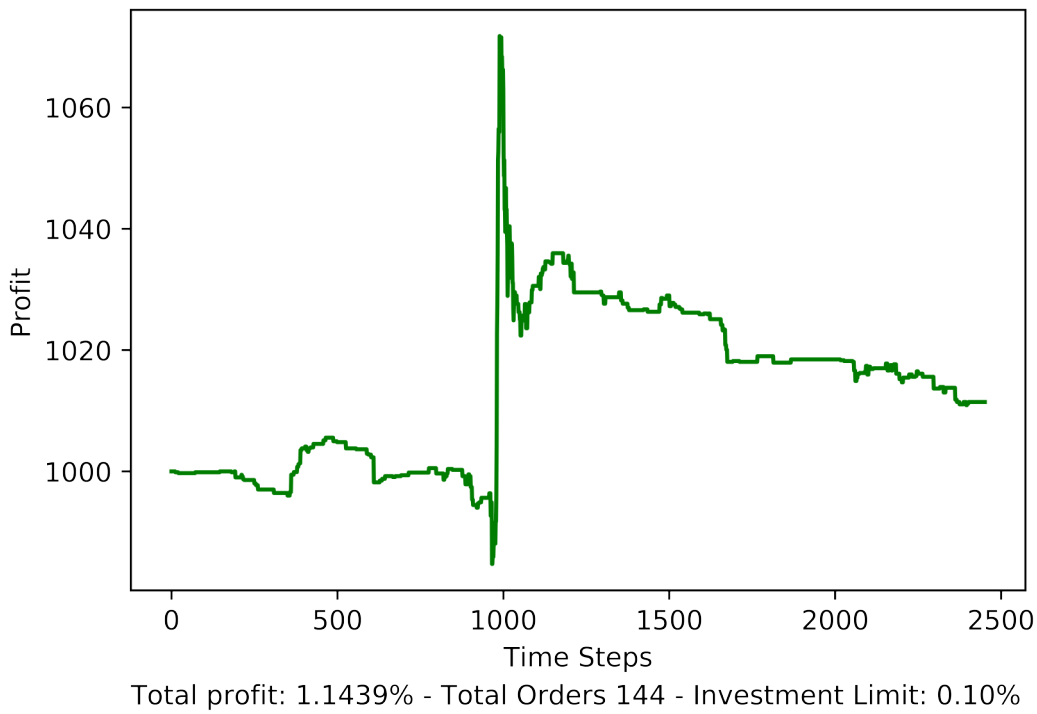


Figure 6.17: Cumulative profit using the forecast at 2 hours

In figure 6.17, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 30$. The algorithm got a profit of 1.1439%, again higher than the case of low volatility. Also in this case, the volatility of the period has an impact also on the number of position opened, which are 144.

Simulation utilising the forecast of n = 48.

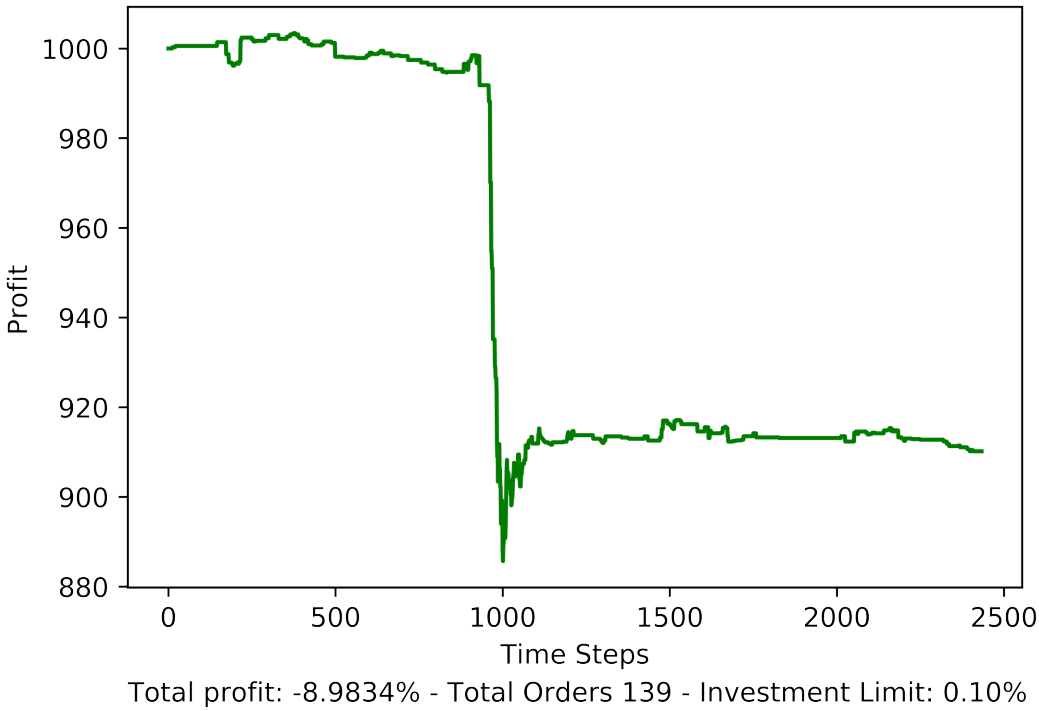


Figure 6.18: Cumulative profit using the forecast at 5 hours

In figure 6.18, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 48$. The algorithm got a loss of -8.9834%. This is consistent with the value found in the simulation without threshold, but it is also clear that controlling more the algorithm, can limit the losses. The number of position opened is 139.

Simulation utilising the forecast of n = 78.

In figure 6.19, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 78$. The algorithm got a profit of 6.1380%. Also in this case, this is consistent with the value found in the simulation without threshold, having obtained a high profit, but as controlling more the algorithm can limit the losses, it can limit the profit as well. The number of position opened is 129.

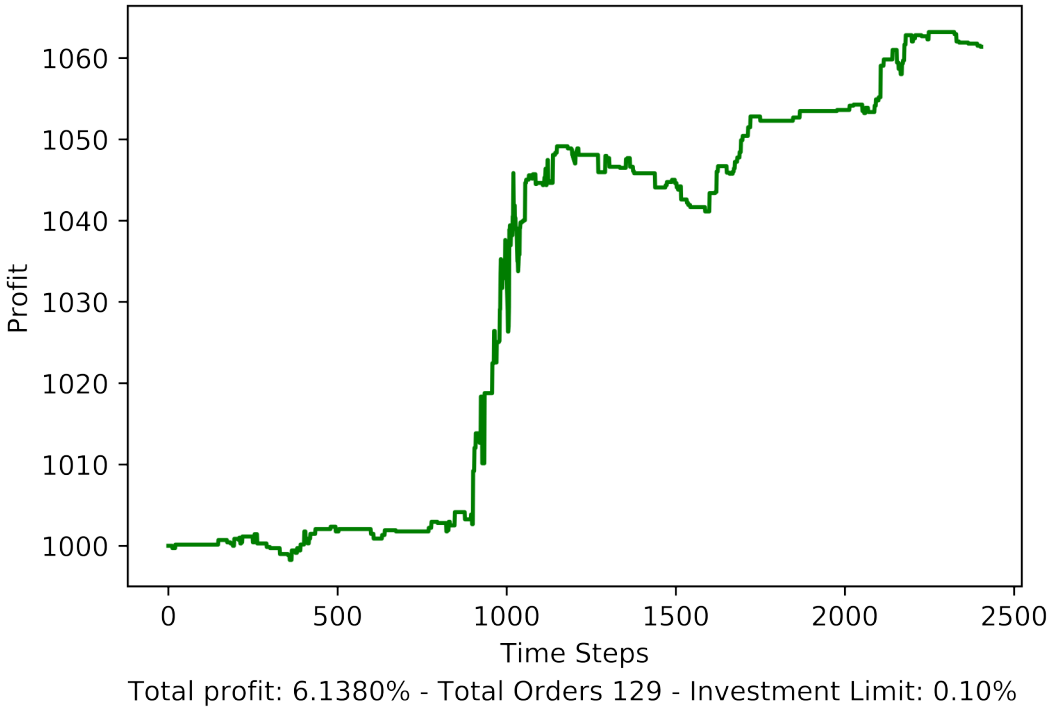


Figure 6.19: Cumulative profit using the forecast at 10 hours

Simulation utilising the forecast of n = 144.

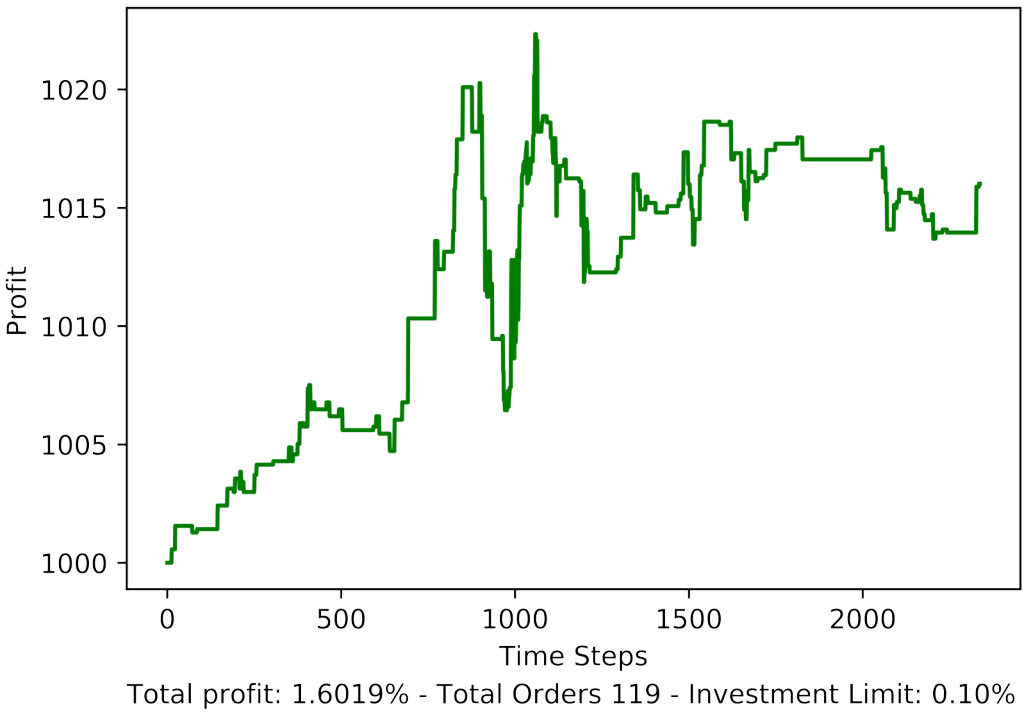


Figure 6.20: Cumulative profit using the forecast at 21 hours

In figure 6.20, we can see the cumulative profit when utilising the forecasted data relative to the data $n = 144$. The algorithm got a profit of 1.6019%, more than previous case, and in line with the results obtained for the other time horizons. The positions opened are 119.

Conclusions

The objective of my thesis was to find whether a trading algorithm based on neural network would be profitable if used on low or high volatility data.

For this purpose, we have seen in *Chapter 1* the main used algorithm in financial markets. In *Chapter 2*, I have described the neural networks, focusing on the one used for my algorithm, a *multilayer perceptron*, a particular class of feedforward neural network. It has been shown its architecture as well as its functioning, describing the concepts of *activation function*, *supervised* and *unsupervised learning*, as well as the *backp-ropagation* algorithm used to train the network.

This has been further explained in *Chapter 3*, where I described the algorithm I built, and how I implemented a 2-layer feedforward neural network, showing the data and the parameters used.

Finally, in *Chapter 4*, we have seen how I built the investment simulation, and how it has been implemented in the model I used.

Main Findings

Following are the findings related to the results described in *Chapter 5* and *Chapter 6*.

In *Chapter 5* we have seen the description of the forecasts obtained running the neural network model.

In the first part we have seen how the predictions were affected by the choice of the train and test set related to the volatility of the period, obtaining the following results:

- when both the train set and the test set are built utilising low volatility data, we have seen that the neural network was able to produce high quality prediction, with a mean squared error of the prediction with respect to the real data of $2.1369x10^{-6}$;

- when both the train set and the test set are built utilising high volatility data, we have seen instead that the neural network struggled to produce quality prediction, and obtained a much higher mean squared error of $1.3379x10^{-3}$;
- combining the train and the test set in order to have the first built on low volatility data and the latter on high volatility data, gave a much higher quality prediction forecasting high volatility data, with a mean squared error of $1.1103x10^{-4}$;
- doing the opposite, we obtained again a low quality prediction of the low volatility data (compared to the one obtained using the low volatility train set), with a mean squared error of $1.3271x10^{-5}$.

Following this results, in order to build my investment simulation, I decided to utilise the low volatility train set for the forecasting of both the low and high volatility test set, as they obtain predictions with higher accuracy.

In the second part of *Chapter 5*, I showed the results obtained when the test set is part of the train set, both for the low and high volatility data. This is an unrealistic situation, given that the algorithm already knows the data it has to forecast, and therefore it learns accordingly. As a matter of fact, for both the configurations, the algorithm gave high quality predictions.

Finally, in the last part of this chapter, utilising the low volatility train set, I described the results obtained when forecasting at different time horizons. This means that I tried to forecast not the first data point after the ones used as inputs, used to obtained the results described before (10-minutes prediction), but the data points corresponding to 2, 5, 10 and 21 hours after the ones used as the inputs of the model. Following is a brief explanation of these results.

When forecasting the low volatility test set, I obtained the following results:

- 2 hours prediction: the network was able to produce high quality forecast, with a mean squared error of the output obtained with respect to the real data of $5.9673x10^{-6}$;
- 5 hours prediction: even in this case the network did a good job, obtaining a good forecast of the real data, with a mean squared error of $1.2068x10^{-5}$;
- 10 hours prediction: although the forecast is still acceptable, the forecasted data start to lose the precision characterising the previous cases, while they still maintain a good approximation of the trend of the real data; the mean squared error is of $2.2597x10^{-5}$;

- 21 hours prediction: the forecasted data curve appears much smoother, meaning that at this distance the network was able to predict only the general trend of the data, losing the randomness characterising the shorter time horizons. For this reason this forecast is meaningful only regarding the general trend of the data, but as a mean to obtain a define value. The mean squared error for this case is $4.7856x10^{-5}$.

When forecasting the high volatility test set, I obtained the same results just described, meaning that the forecast accuracy in both cases has the same trend: the farther in time the data point we are trying to forecast, the more the general trend will prevail over the accuracy of the value forecast. In details:

- 2 hours prediction: the network was able to produce a good quality forecast, with a mean squared error of the output obtained with respect to the real data of $1.7387x10^{-4}$;
- 5 hours prediction: again the forecast is of relative good quality, and comparable with the 10 minute forecast shown before, with a mean squared error of $2.5998x10^{-4}$;
- 10 hours prediction: the forecasted data start to lose the precision characterising the previous cases, while they still maintain a good approximation of the trend of the real data; the mean squared error is of $3.9069x10^{-4}$;
- 21 hours prediction: the network was able to predict only the general trend of the data, losing precision in the forecast of the actual value of the data point, is characterised by a mean squared error of $7.3434x10^{-4}$.

Given the results obtained and described in *Chapter 5*, in *Chapter 6* I described the results of the investment simulation performed using the forecasted data for both the low and high volatility period, for all the time horizons described before. The two different simulations have been performed: the first, supposing that the algorithm opens a position (buy or sell) for every data points, investing always the USD 1000. In the second simulation, the position is opened only when the difference between two data point of the forecasted data is greater than a certain threshold, define as 0.1%. In both cases I have assumed no budgetary constraints as well as no transaction costs (partially approximated by the threshold set), and the simulation has been made on the data relative to the test set, corresponding to a time period of 20 days.

For each simulation I have calculated the profit (or loss), the accuracy of the model in forecasting the price movement and the number of position opened (for the simulation with the threshold).

In the first case there has never been a loss when running the investment simulation on low volatility data, but we got some losses when the simulation run on the high volatility data set. In the second case, which is closer to reality as the threshold imitates the transaction costs, we saw the following results:

- simulation on low volatility data:
  - 10 minutes forecast data: loss of −0.9218%, prediction accuracy of 47.54% and 69 positions opened;
  - 2 hours forecast data: profit of 0.9211%, prediction accuracy of 49.96% and 68 positions opened;
  - 5 hours forecast data: profit of 0.3429%, prediction accuracy of 49.88% and 67 positions opened;
  - 10 hours forecast data: profit of 0.6247%, prediction accuracy of 49.92% and 57 positions opened;
  - 21 hours forecast data: profit of 0.1447%, prediction accuracy of 50.22% and 50 positions opened;
- simulation on high volatility data:
  - 10 minutes forecast data: profit of 6.1606%, prediction accuracy of 48.50% and 147 positions opened;
  - 2 hours forecast data: profit of 1.1439%, prediction accuracy of 49.55% and 144 positions opened;
  - 5 hours forecast data: loss of −8.9834%, prediction accuracy of 49.47% and 139 positions opened;
  - 10 hours forecast data: profit of 6.1380%, prediction accuracy of 49.79% and 129 positions opened;
  - 21 hours forecast data: profit of 1.6019%, prediction accuracy of 50.26% and 119 positions opened.

From this results we can deduce some important conclusions. The first is that it appears that this kind of algorithm is affected in the same way by the volatility of the humans traders. In fact we can notice how when running the simulation on low volatility data, we obtain low profit, and low losses (only one). If we run the same algorithm, again we get only one loss, but both the profit and loss can be very high as well as very low.

Secondly, we can see that in both situation, there is always a profit when using data forecasted at 10 hours and 21 hours distance. This is due to the same reason described when discussing the results of *Chapter 5*. When forecasting data at longer time horizons, the algorithm is less affected by the randomness of high frequency data, while it retain its ability to predict the trend of the data. This is also visible from the price movement prediction accuracy: while it always is around 50%, we can see that the only times it is higher than this value is when utilising the forecast at 21 hours, while the lowest price movement accuracy is found for the 10 minutes time horizon.

Finally, given the accuracy just explained, we can see that this simple algorithm isn't ready to go in action in the real world. Having a 50% prediction accuracy of the price movement, is equivalent to take decisions based on the toss of a coin.

In conclusion, as we could see from the results just explained, we can safely say that this kind of algorithm can be used equally in low volatility as well as high volatility period, given that the neural network is properly train. Of course it needs to be improved, but from this very basic model we can already understand the potential of these algorithms.

FURTHER WORKS

Given the results obtained using this model, we can see that there is a range of possible improvement to be done, as well as things to investigate.

First of all, as in this model I utilised real data as the input to predict the test set, it would be interesting to use the forecasted data as input to generate the subsequent forecasts.

Moreover, it would be interesting to introduce more features in the data, like the bid/ask spread, to investigate how the neural network manages to produce forecast with more complex data.

Finally, the neural network used, a *Feedforward*, although it is one of the most common and easy to use, it is not the best choice for these problems. One of the most recent neural network, which has been found very accurate in forecasting financial data, as well as in a range of other application. This is the *LSTM*, *Long Short Term Memory* network. This has proved to be highly efficient in analysing huge amount of data, being capable of extracting a major number of features without compromising their ability to learn long-term dependencies. For this reason it would be possible to adapt the algorithm described in this work to utilise a *LSTM* network.

Appendix A: The Code

In this appendix can be found the code used to model the data, build the neural network, forecast and perform the analysis as well as the simulations.

```
import tensorflow as tf
import keras
from keras import optimizers
from keras.models import Sequential
from keras.layers import Dense, Activation
import timeit
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt

'''Starts the Timer'''
start = timeit.default_timer()

'''Read and Reshape the Data;
Load the data using pandas, remove the possible duplicates (checking the 'Date' column),
convert the data in the 'Date' column as datetime format "YYYY-MM-DD hh:mm:ss'''

raw_data = pd.read_csv('./Data.csv')
clear = raw_data.drop_duplicates(['date'])
clear.to_csv('./Data_clean.csv', index = False)
data = pd.read_csv('./Data_clean.csv')
data['date'] = pd.to_datetime(data['date'])
```

```
print(data.shape)

'''Define the periods considered to Train and Test the Neural Network'''

training_period = ('2016-01-09 00:00:00','2016-03-11 23:50:00')
mask_train = (data['date'] >= training_period[0]) & (data['date'] <=
    training_period[1])
train_set = data.loc[mask_train]
train_set = np.array(train_set['last_price'])

print(train_set.shape)

test_period = ('2016-03-10 00:00:00','2016-04-01 23:50:00')
mask_test = (data['date'] >= test_period[0]) & (data['date'] <= test_period
    [1])
test_set = data.loc[mask_test]
test_set = np.array(test_set['last_price'])

print(test_set.shape)

'''Define t as the number of inputs and n as the data point to predict'''

n = 18
t = 17
poi = np.concatenate((train_set[n:-n],test_set[n:-n]))

'''Reshape the dataset in order to contain n column'''

new_train=[]
for x in range(0, len(train_set)-n):
    new_train.append(train_set[x:x+n])
train = np.array(new_train)

new_test=[]
for x in range(0, len(test_set)-n):
    new_test.append(test_set[x:x+n])
```

```
test = np.array(new_test)

'''Scale the data subtracting the minimum of the series and
dividing by the maximum so to have values betweem 0 and 1'''

a = poi.min()
print(a)
b = poi.max()
print(b)
train = train - a
train = train / b

test = test - a
test = test / b

'''Define the inputs (x) and the targets (y) for both the train and the test
    set'''

x_train = np.array(train[:,:t])
x_test = np.array(test[:,:t])

print('X Train - shape: ' +str(x_train.shape))
print('X Test - shape: ' +str(x_test.shape))

y_train = np.array(train[:,n-1:n])
y_test = np.array(test[:,n-1:n])

print('Y Train - shape: ' +str(y_train.shape))
print('Y Test - shape: ' +str(y_test.shape))

'''Build the Neural Network using Keras API and use it to evaluate the model
    and make the prediction'''

randomnormal = keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed
    =1000)
model = Sequential()
```

```
model.add(Dense(10, kernel_initializer=randomnormal, input_dim = t, activation
    = 'sigmoid'))
model.add(Dense(1,  kernel_initializer=randomnormal, activation = 'sigmoid'))

sgd = optimizers.SGD(lr=0.01, decay=0.0001, momentum=0.9)
model.compile(loss='mean_squared_error', optimizer=sgd, metrics=['mae'])

fit = model.fit(x_train, y_train, epochs=500, batch_size=32)

train_score = model.evaluate(x_train, y_train)
test_score = model.evaluate(x_test, y_test)

train_forecast = model.predict(x_train)
test_forecast = model.predict(x_test)

'''Unscale and save a .csv of the data for visualisation purposes'''

train_forecast = train_forecast*b
train_forecast = train_forecast+a
test_forecast = test_forecast*b
test_forecast = test_forecast+a

traincsv = pd.DataFrame(train_forecast)
traincsv.to_csv('./train_LowLow_vol.csv', index = False)

testcsv = pd.DataFrame(test_forecast)
testcsv.to_csv('./test_LowLow_vol.csv', index = False)

'''Show Summary of the neural network model'''

print(train_score)
print(test_score)

model.summary()
from keras.utils.visualize_util import plot
plot(model, to_file='model.png')

```

```
'''Build plots to visualise the real data considered, the train and test set as compared to the real data,
the Mean Squared Error and the Mean Absolute Error evolution over the iterations'''

plt.figure(dpi=1200)
plt.plot(poi, 'b', label = 'Real')
plt.ylabel('Exchange Rate')
plt.xlabel('Time Steps')
plt.legend()
plt.show()

print(poi.shape)
print(train_forecast.shape)
print(test_forecast.shape)

shift_array = [[None]*(len(poi)-len(test_forecast)+n)]
shift_array = np.array(shift_array)
shift_array = shift_array.T
test_shift = np.concatenate((shift_array,test_forecast[n:]))

plt.figure(dpi=1200)
plt.plot(poi, 'b', label = 'Real')
plt.plot(train_forecast[n:], 'g', label = 'Train')
plt.plot(test_shift, 'r', label = 'Test')
plt.ylabel('Exchange Rate')
plt.xlabel('Time Steps')
plt.legend()
#plt.savefig('forecast-LowLow.png')
plt.show()

plt.figure(dpi=1200)
plt.plot(train_set[n:-n], 'b', label = 'Real')
plt.plot(train_forecast[n:], 'g', label = 'Train')
plt.ylabel('Exchange Rate')
plt.xlabel('Time Steps')
plt.legend()
```

```
#plt.savefig('forecast_train-LowLow.png')
plt.show()

plt.figure(dpi=1200)
plt.plot(test_set[n:-n], 'b', label = 'Real')
plt.plot(test_forecast[n:], 'r', label = 'Test')
plt.ylabel('Exchange Rate')
plt.xlabel('Time Steps')
plt.legend()
#plt.savefig('forecast_test-LowLow.png')
plt.show()

plt.figure(dpi=1200)
plt.plot(fit.history['loss'])
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.annotate('Mean Squared Error: '+"{:.4e}".format(test_score[0]), (0,0), (0,
    -35), xycoords='axes fraction', textcoords='offset points', va='top')
#plt.savefig('mean_squared_error-LowLow.png')
plt.show()

plt.figure(dpi=1200)
plt.plot(fit.history['mean_absolute_error'])
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.annotate('Mean Absolute Error: '+"{:.4e}".format(test_score[1]), (0,0),
    (0, -35), xycoords='axes fraction', textcoords='offset points', va='top')
#plt.savefig('mean_absolute_error-LowLow.png')
plt.show()

'''Investment Simulation without Threshold'''

'''Define the real and forecast set'''

real = test_set[n:]
forecast = test_forecast
l = len(real)
```

```python

print(len(real))
print(len(forecast))

'''Creates a list of 1 and -1 according to the price movement of the forecasted data:
if at t+1 the value is > than the one at t, I write 1, and vice versa'''

indicator = []

for j in range(0,l-1):
    if forecast[j+1]-forecast[j]>0:
        indicator.append(1)
    elif forecast[j+1]-forecast[j]<=0:
        indicator.append(-1)

indicator.insert(0,1)
#print(len(indicator))

'''I do the same on the real data set, in order to check the performance
of the algorithm related to the price movement (% of times it got the right movement)'''

indicator_real = []

for j in range(0,l-1):
    if real[j+1]-real[j]>0:
        indicator_real.append(1)
    elif real[j+1]-real[j]<=0:
        indicator_real.append(-1)

indicator_real.insert(0,1)
#print(len(indicator_real))

'''Creates a list containing the % difference between two consecutive values in the real data'''

```

```
returns = []

for j in range(0,l-1):
        returns.append(((real[j+1])/real[j])-1)

returns.insert(0,1)
print(len(returns))
#print(returns)

'''newindicator is an adjustment to the indicator list created before,
in order to insert a 1 everytime I find a 0 in the returns list, so that I can ignore it
and act as if the algorithm doesn't open any position'''

newindicator = []
for i in range(0,len(indicator)-1):

    if returns[i] == 0:
        newindicator.append(1)
    elif returns[i] != 0:
        newindicator.append(indicator[i])
#print(len(newindicator))
#print(newindicator[1])

'''Calculates the profit and therefore the total profit supposing an investment of USD 1000 for every operation'''

x = 1000

z = next(i for i, v in enumerate(returns[1:len(returns)]) if v != 0)
#print(z)

a = [i for i in range(0,len(newindicator)-1)]

for j in range(z,len(newindicator)-1):
    a[j]=((x*newindicator[j+1]*(returns[j+1])))

```

```
#print(a)
profit = np.sum(a)
#print(z)
#print(profit)
for i in range(0,z+1):
    a.insert(i,x)
    #print(a[0])
for i in range(z+1,len(a)):
    a[i] = a[i] + a[i-1]

totalprofit = ((x+profit)/x)-1
tot_profit = 'Total profit: {percent:.4%}'.format(percent=totalprofit)
print(tot_profit)

'''Calcluates the number of times the price movement prediction was right'''

up_down = [1 for x in range(0,l-2) if indicator[x] == indicator_real[x]]

k = up_down.count(1)
#print(up_down)
precision = k/len(indicator_real)
price_movement = 'Price movement prediction accuracy: {percent:.2%}'.format(
    percent=precision)
print(price_movement)

'''Plots the profit graph'''

plt.figure(dpi=1200)
plt.plot(a, 'g')
plt.ylabel('Profit')
plt.xlabel('Time Steps')
plt.annotate(tot_profit+' - '+price_movement, (0,0), (0, -35), xycoords='axes
    fraction', textcoords='offset points', va='top')
#plt.savefig('Simulation_LowLow.png')
plt.show()

'''Investment Simulation with Threshold'''
```

```

'''Define the threshold p and how it should be used to adjust the indicator array'''
p = 0.0010
indicator1 = []
beg = forecast[0]

for i in range(1,l-1):
    kk = (forecast[i]/beg)-1
    #print(kk)
    if kk >= p or kk<= -p:
        beg = forecast[i]
        if kk >= p:
            indicator1.append(1)
        elif kk <= -p:
            indicator1.append(-1)
    else:
        indicator1.append(0)

indicator1.insert(0,1)
#print((indicator1))

'''Creates a list containing the % difference between two consecutive values in the real data'''

returns1 = []

for j in range(0,l-1):
        returns1.append(((real[j+1])/real[j])-1)

returns1.insert(0,1)
print(len(returns1))
#print(returns1)

'''newindicator1 is an adjustment to the indicator1 list created before,
in order to insert a 1 everytime I find a 0 in the returns1 list, so that I can ignore it
```

```
and act as if the algorithm doesn't open any position'''

newindicator1 = []
for i in range(0,len(indicator1)-1):
    #for j in range(0,len(returns)-1):
    if returns1[i] == 0:
        newindicator1.append(1)
    elif returns1[i] != 0:
        newindicator1.append(indicator1[i])
#print(len(newindicator1))
#print(newindicator1[1])

'''Calculates the profit and therefore the total profit supposing an
    investment of USD 1000 for every operation'''

x = 1000 #initial investment

z = next(i for i, v in enumerate(returns1[1:len(returns1)]) if v != 0)
#print(z)

a = [i for i in range(0,len(newindicator1)-1)]

a[z] = x*newindicator1[z+1]*(returns1[z+1])
#print(a[z])

for j in range(z+1,len(newindicator1)-1):
    a[j]=((x*newindicator1[j]*(returns1[j])))

#print(a)
profit=np.sum(a)
#print(z)
#print(profit)
for i in range(0,z+1):
    a.insert(i,x)
for i in range(z+1,len(a)):
    a[i] = a[i] + a[i-1]

```

```
#print(a)
totalprofit = ((x+profit)/x)-1
tot_profit = 'Total profit: {percent:.4%}'.format(percent=totalprofit)
print(tot_profit)

"""Calcluate the number of orders placed corresponding to how many time the
    forecasted return exceed the threshold"""

k = 'Total Orders '+str(len(indicator1) - indicator1.count(0))
print(k)
limit = 'Investment Limit: {percent:.2%}'.format(percent=p)
print(limit)

'''Plots the profit graph'''

plt.figure(dpi=1200)
plt.plot(a, 'g')
plt.ylabel('Profit')
plt.xlabel('Time Steps')
plt.annotate(tot_profit+' - '+k+' - '+limit, (0,0), (0, -35), xycoords='axes
    fraction', textcoords='offset points', va='top')
#plt.savefig('./Report/Profit_Limit_March_'+'(n='+str(n)+').png')
plt.show()

'''stops the timer'''
stop = timeit.default_timer()
print('Run time: ' + str(stop-start))
```

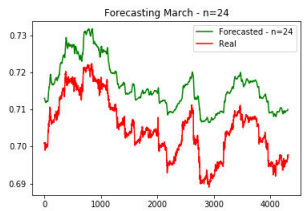
Appendix B: Previous Analysis

Before working on this thesis, I built a different model, utilising the same approach and the same data, but with different algorithm. I built my own neural network, defining all its parameters, and forecasted the data relative to the month of March and June 2016. Of course it was less precise than the one utilised for this work, but it gave me the idea of improving it and investigate other configurations.

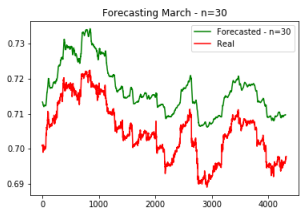
Following are the results obtained in my previous analysis.

Different Time Horizon Forecasting

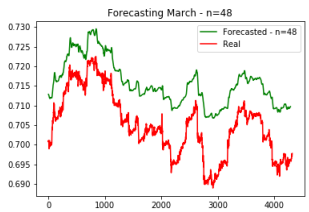
March



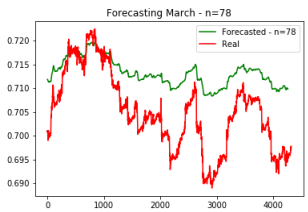
(a) n=24 - Data point 1h10m later



(b) n=30 - Data point 2h10m later



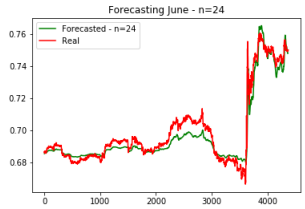
(c) n=48 - Data point 5h10m later



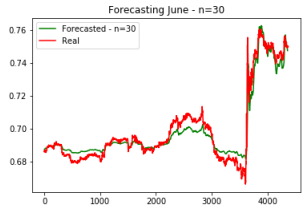
(d) n=78 - Data point 10h10m later

Figure 6.21: Comparison between the real and the forecasted data, when trying to predict at different time horizons - Period of interest: March

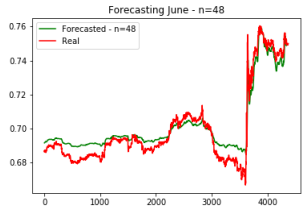
June



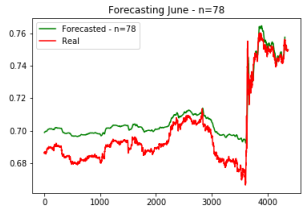
(a) n=24 - Data point 1h10m later



(b) n=30 - Data point 2h10m later



(c) n=48 - Data point 5h10m later

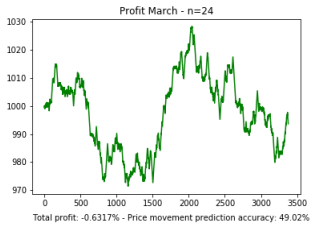


(d) n=78 - Data point 10h10m later

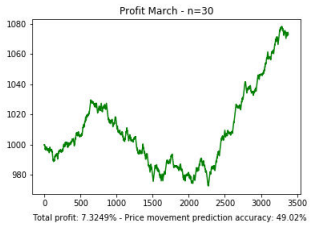
Figure 6.22: Comparison between the real and the forecasted data, when trying to predict at different time horizons - Period of interest: June

Investment Simulation

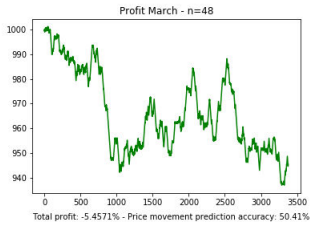
March



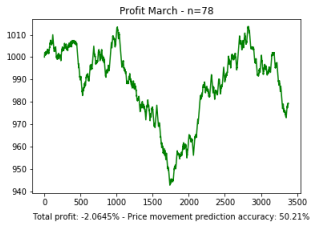
(a) n=24 - Data point 1h10m later



(b) n=30 - Data point 2h10m later



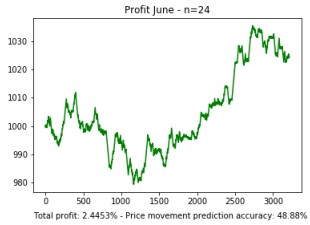
(c) n=48 - Data point 5h10m later



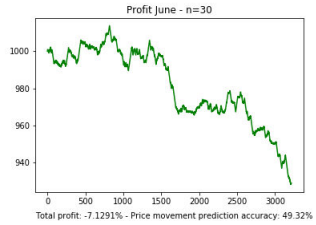
(d) n=78 - Data point 10h10m later

Figure 6.23: Investment simulation - Period of interest: March

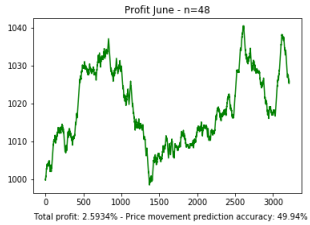
June



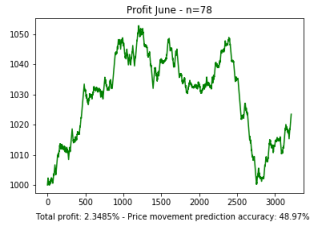
(a) n=24 - Data point 1h10m later



(b) n=30 - Data point 2h10m later



(c) n=48 - Data point 5h10m later



(d) n=78 - Data point 10h10m later

Figure 6.24: Investment simulation - Period of interest: June

Bibliography

[1] http://www.investopedia.com/articles/optioninvestor/10/trend-following-indicators.asp.

[2] http://www.nasdaq.com/article/dont-be-fooled-by-the-fancy-name-statistical-arbitrage-is-a-simple-way-to-profit-cm254669.

[3] http://www.investopedia.com/articles/investing/091615/world-high-frequency-algorithmic-trading.asp.

[4] Simon Haykin. *Neural Networks and Learning Machines.* Pearson, 2009.

[5] https://upload.wikimedia.org/wikipedia/commons/6/6d/error_surface_of_a_linear_neuron_with_two_input_weights.png.

[6] J. Levendovszky and F. Kia. Prediction based - high frequency trading on financial time series. *Per. Pol. Elec. Eng. and Comp. Sci.*, 56(1):29–34, 2012.

[7] A. Arévalo, J. Ni no, G. Hernández, and J. Sandoval. High-Frequency Trading Strategy Based on Deep Neural Networks. *Intelligent Computing Methodologies. ICIC 2016*, 9973.

[8] E. Guresen, G. Kayakutlu, and T.U. Daim. Using artificial neural network models in stock market index prediction. *Expert Systems with Applications*, 38(8):10389–10397, 2011.

[9] https://www.theguardian.com/business/2016/jun/23/british-pound-given- boost-by-projected-remain-win-in-eu-referendum.

[10] E. Budish, P. Cramton, and J. Shim. The high-frequency trading arms race: frequent batch auctions as a market design response. *The Quarterly Journal of Economics*, 130(4):1547–1621, 2015.

[11] http://www.reuters.com/article/us-europe-stocks-algos-insight/rocky-markets-test-the-rise-of-amateur-algo-traders-iduskcn0v61t6.

[12] J. Hasbrouk and G. Saar. Low-latency trading. *Journal of Financial Markets*, 16:647–679, 2013.

[13] E. F. Fama. Efficient Capital Markets: A Review of Theory and Empirical Work. *Journal of Finance*, 25(2):383–417, 1969.

[14] W.S. McCulloch and W.H. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1949.

[15] D.O. Hebb. *The Organization of Behavior*. Wiley, 1949.

[16] B.G. Farley and W.A. Clark. Simulation of Self-Organizing Systems by Digital Computer. *IRE Transactions on Information Theory*, 4(4):76–84, 1954.

[17] F. Rosenblatt. The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain. *Psychological Review*, 65(6):386–408, 1958.

[18] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[19] Paul Werbos. *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. PhD Thesis. Harvard University, 1974.

[20] D.E. Rumelhart and James McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, 1986.

[21] G.I. Webb C. Sammut. *Encyclopedia of Machine Learning*. Springer, 2011.