Forecasting the Financial Times Stock Exchange 100 Index using Neural Networks

Raymond McBride
MSc Computing Science project report
School of Computer Science and Information Systems,
Birkbeck College,
University of London
2004

Abstract

The ability of an investor to accurately forecast price movements in shares and share indices would potentially enable them to realise enormous wealth. Many analytical techniques exist to try to identify trends within these movements. Neural Networks is an area renowned for its capacity to recognize complex patterns in data which cannot otherwise be easily identified.

This project focuses on the ability of the Multilayer Perceptron, the Time Delay Neural Network and the Recurrent Neural Network at forecasting end of day closing price of the FTSE 100 Index.

Many different network topologies were examined, with a variety of different parameters and training levels, before the best network of each type was compared with a Box Jenkins ARIMA model.

The best performance was achieved using a Recurrent Network with 5 inputs, 15 hidden units, 0.1 memory depth, 0.8 learning rate and 0.5 momentum, when trained with 10000 epochs. During testing and validation of the networks, better results were obtained from MLP networks than TDNN networks. However during the final forecast test this was reversed.

This research concludes that the selection of appropriate network parameters is vital to the networks prediction capability as is the optimal amount of training.

Table of Contents

List of Tables

List of Figures

Chapter 1 - Introduction

1.1 Overview

The ability to anticipate fluctuations in financial markets would allow any investor to realise enormous quantities of money. A large amount research of has therefore been undertaken, in a variety of different fields to try to come up with some way of accurately predicting these movements. Many statistical techniques have been formulated, which are employed by analysts to try to gain any competitive edge.

Neural Networks is an area that has long been heralded for its ability to identify patterns and trends in data that cannot easily be discerned by the human eye. For this reason, much research has and is being done in the application of Neural Networks to financial forecasting.

Current research includes work done by Kim *(Kim, 1998)* which uses a hybrid network called a Time Delay Recurrent Neural Network to forecast Korean stock market prices, research by Chenoweth and Obradovic *(Chenoweth, Obradovic, 1996)* which combines Neural Networks and Expert Systems to forecast the S & P 500 Index and are as diverse as research done by Shazly and Shazly *(Shazly, Shazly, 1999)* which combines Genetic Algorithms with Neural Networks to forecast currency prices.

1.2 Objectives

The aims of this project are to test and compare the ability of different types of neural networks at forecasting the end of day closing price of the Financial Times Stock Exchange 100 Index.

Three different types will be assessed. These are the Multilayered Perceptron, the Time Delay Neural Network and a version of the Recurrent Neural Network based on the Elman Network. The Time Delay and Recurrent Neural Networks have been included in this research because of their renowned capacity for tackling temporal problems.

This report will mainly focus of the design and implementation of the aforementioned neural networks together with analysis of their forecasting ability.

1.3 Organisation of this report

This report is divided into five additional chapters:

1) Chapter 2 - The FTSE 100 and financial time series. This chapter gives a brief description of what share indices are, followed by short history of the FTSE 100 Index. The remains of this chapter concentrate on how financial time series are analysed together with a description of a traditional method for forecasting them.

2) Chapter 3 – Artificial Neural Networks. This chapter looks at what Artificial Neural Networks are, and shows how artificial Neurons have been based on biological ones. It then goes on to discuss the Multilayer Perceptron and has a detailed look at the Back Propagation learning algorithm. Variations of the Multilayer Perceptron known as the Time Delay Neural Network and the Recurrent Neural Network are then described.

3) Chapter 4 – Design and Implementation. This chapter highlights a java implementation of the three networks mentioned above, together with testing parameters.

4) Chapter 5 – Presents the results together with analysis and discussion from testing the above implementation.

5) Chapter 6 – Discusses the successes and failures of this project.

Chapter 2 – The FTSE 100 and financial time series

2.1 What are Share Indices?

A Share Index is an average that lists the leading companies in a market based on their market capitalisation *(Valdez, 2003).* Market capitalisation is a calculation of the companies share price multiplied by the number of its shares. This calculation is used as a weighting system, whereby movements in the share price of larger companies will have a greater effect on the index than movements in smaller companies *(Valdez, 2003)*.

The popularity of investing directly in these indices has increased over recent. Kaufman suggests:

"Share Indices provide a useful means for allowing individuals or organisations to invest in the overall market movement rather than take the higher risk of selecting individual securities" *(Kaufman, 1998)*.

The companies that make up these indices come from all over the business spectrum. As a result of this diversity and also due to additional influences exerted by other markets, there are a great many factors that can affect their value. This makes the task of forecasting their movements an extremely difficult one. Even so, much time and effort is spent by institutions and investors analysing their behaviour.

2.2 The FTSE 100 index

The first index to be calculated on the market traded at the London Stock Exchange was known as the Financial Times Ordinary Share Index. This began in 1935 and was based on the top 30 companies *(Valdez, 2003)*.

As time went on, the 30 Ordinary Share Index became an inadequate measure, so the Financial Times Stock Exchange 100 Index was formed in 1984, increasing the number of companies listed to 100 *(Valdez, 2003)*. The index began at the level of 1000 and represent 77% of the capitalisation of the whole market. It is calculated every fifteen seconds from 8.30am to 4.30pm *(Valdez, 2003)*.

Figure 2.1 below shows how the index has fluctuated over the last three years.

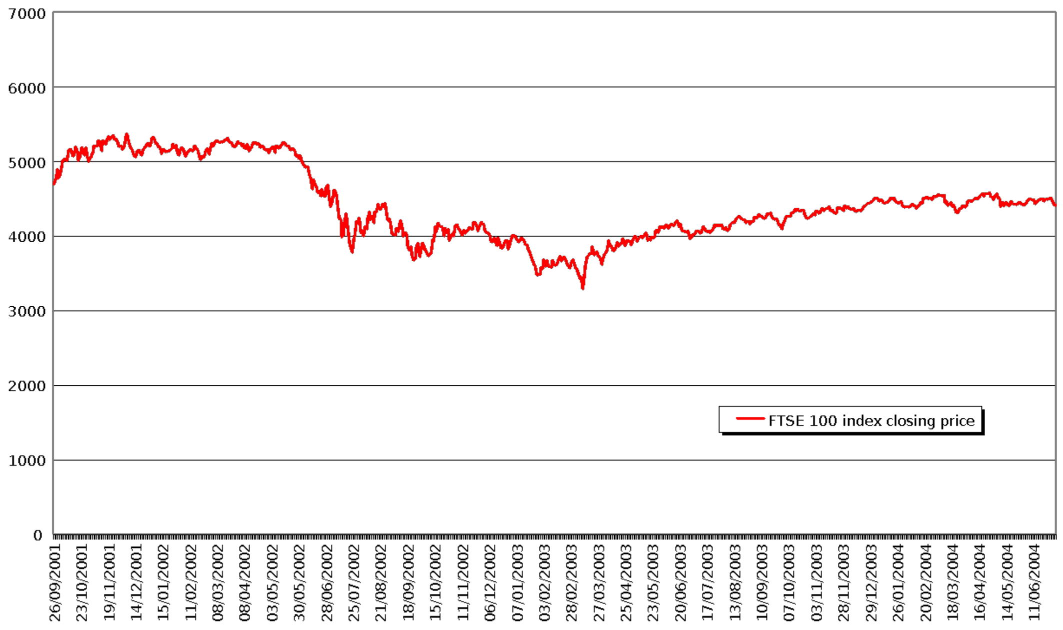


Figure 2.1: FTSE 100 index Closing price 26/09/2001 to 02/07/2004

2.3 Financial Time Series Analysis

As the value of Share Indices rise and fall over time, we can view this price change/time combination as a financial time series'. Chapman defines a time series as:

"A collection of observations made sequentially through time. It is said to be continuous when observations are made continuously through time and discrete when observations are taken at specific times, usually equally spaced." *(Chatfield, 2004)*

These definitions can be applied to the movements of the FTSE 100 closing price over time and available techniques for analysis and prediction are therefore applicable. Traditional methods of analysis are primarily concerned with decomposing the variations in the time series into components representing long term trends, seasonal and other cyclical variations *(Chatfield, 2004)*. See Figure 2.2 below.

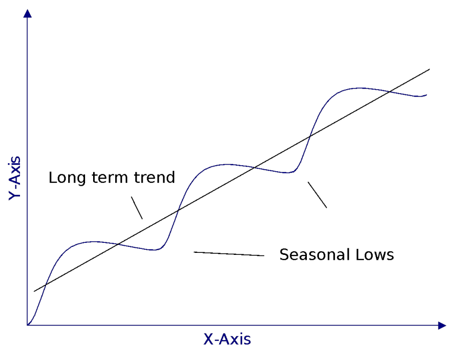


Figure 2.2 Seasonal and Long Term variations

After these variations have been removed, usually additional fluctuations remain. Whether or not these are random however is a contentious issue *(Chatfield, 2004).*

Fundamental Analysts study economic and political features, everything that make prices what they are. They believe that there are far too many random influencing factors so these fluctuations must be down to the price following a random walk *(Kahn, 1999)*. In other words, the price at time t equals the price at t-1 plus some random element.

Technical Analysts on the other hand are more concerned with the study of the market itself and trends in the price and traded volume *(Kahn, 1999).* They have techniques such as Autoregressive (AR) and Moving Averages (MA) modelling to try to find additional shorter trends in this remaining data *(Kaufman, 1998)*.

AR refers to using the past data to self predict and MA refers to concept of smoothing the data by using an average of the past n days *(Kaufman, 1998)*.

2.4 A Traditional Methodology for Forecasting Time Series

Box and Jenkins *(Box, Jenkins, and Reinsel, 1994)* developed a systematic approach for identifying forecasting models which incorporated both techniques of AR and MA. Known as Autoregressive Integrated Moving Average (ARIMA), it is made up of three stages, Model Identification, Estimation and Validation.

2.4.1 Model Identification

The first step in the process is to determine if the series is stationary and/or seasonal (*NIST/SEMATECH, 2004)*. A series is said to be stationary if the mean, variance and autocorrelation remain constant over time (*NIST/SEMATECH, 2004)*. This means that the series is flat and not trending. A common way to make a series stationary is to take differences *(Kaufman, 1998)*. Table 2.1 shows a small time series and its first difference.

Table 2.1: A Table showing a Time Series and its first difference

Time Series	First Difference
5837.8848	*
5842.3397	4.4549
5898.4163	56.0766
5860.7787	-37.6376
5947.2534	86.4747
6037.8071	90.5537
6019.8445	-17.9626
5987.3954	-32.4491

Figure 2.3 below shows a plot of the time series and its first difference

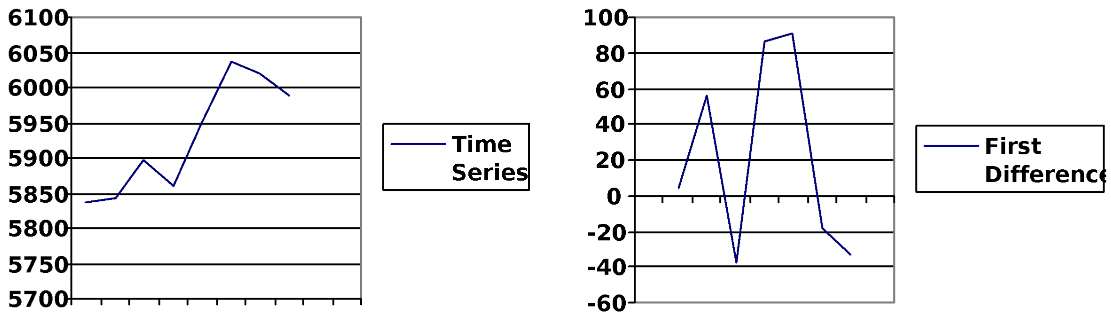


Figure 2.3: Plot of Time series and its first difference

Once it has been established that the series is stationary, the next step is to examine plots of its Autocorrelation (ACF) and Partial Autocorrelation (PACF) Functions *(MINITAB, 2004)*. These are measures of how related data values are to each other.

- An ACF with large spikes at initial lags that decay to zero or a PACF with a large spike at the first and possible at the second lag indicates an autoregressive process
- An ACF with a large spike at the first and possibly at the second lag and a PACF with large spikes at initial lags that decay to zero indicates a moving average process.
- The ACF and the PACF both exhibiting large spikes that gradually die out indicates that both autoregressive and moving averages processes are present *(taken from MINITAB, 2004).*

2.4.2 Estimation

ARIMA is usually shown in the format ARIMA(p, d, q), where p is the number of autoregressive terms, d is the number of differences and q is the number of moving average terms *(Kaufman, 1998)*. This next step involves the estimation of these coefficients. In general parameters are selected and then the validation step is performed. If the model is found to be unacceptable, new parameters are tested. Each parameter is a number between 0 and 5, with the sum of all 3 not exceeding 10 *(MINITAB, 2004).*

2.4.3 Validation

The final step is to examine the residuals, which is the data left over, and should be just noise. Plots of the ACF and PACF of the residuals are examined for any large spikes. If any appear, then new parameters should be selected *(MINITAB, 2004)*.

Once validation is found to be acceptable, the model is ready for forecasting. Appendix D shows forecasts made on the FTSE100 data used in this project, using the ARIMA process and Minitab 13

Chapter 3 – Artificial Neural Networks

3.1 What are Neural Networks?

Traditionally, the main focus of computing has been based on the creation of programmes that perform certain tasks to an exact specification and in a procedural way *(Schalkoff, 1997).* Computers have become extremely fast at tasks such as adding numbers and can complete these in a fraction of the time that it would take a human to *(Beale, Jackson, 2001).*

However, other tasks such as speech recognition and visual identification, which are performed easily by humans, are not executed as effectively by procedural computer programmes *(Haykin, 1999)*

To understand why humans are better than computers at these tasks, we need to have an understanding as to how computing is tackled in biological systems *(Beale, Jackson, 2001)*. The centre for computing in a human being is the brain. The architecture of the brain, rather than being serial like a computer, has a parallel design *(Beale, Jackson, 2001)*. This allows it to process many different pieces of information at the same time, which all need to interact to produce a solution *(Beale, Jackson, 2001)*.

In addition to its parallel nature, one of the most important features of the brain is its ability to learn. Rather than having specific instructions for each step, the brain can pick things up based on previous experience *(Beale, Jackson, 2001)*.

By using these principles we can design computer programmes known as Neural Networks which attempt to apply the brains solutions to these tasks.

3.2 The Biological Neuron

A human brain contains about 10^{10} processing units called neurons each of which is connected to about 10^{4} other neurons *(Beale, Jackson, 2001)*. Figure 3.1 below, shows two connected neurons.

The main components of a Neuron are as follows:

- The Soma – This is the body of the neuron and contains the cell nucleus.
- The Dendrites – These are fine nerves which receive all input into the neuron
- The Axon – This nerve is used for the output of the neuron
- The Synapses – These are special connectors that join the axon of one cell to the dendrite of another cell.

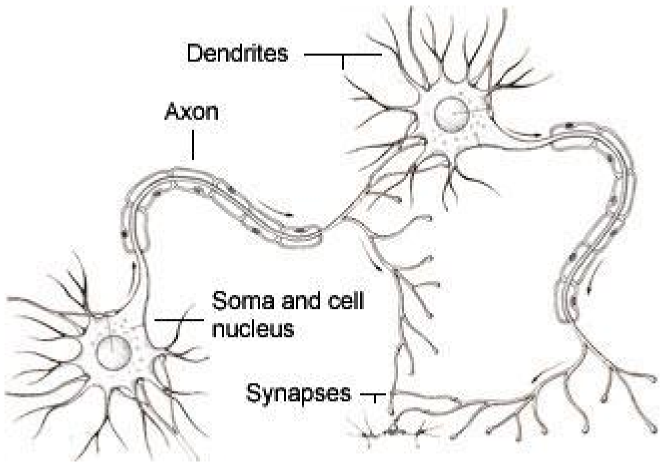


Figure 3.1: Two connected Neurons.

Neurons receive input from the synapses of other neurons in the form of chemicals know as Neurotransmitters. This causes an electrical charge to build up in the receiving neuron. If this charge exceeds a threshold limit, an electrical pulse will be sent down the axon to be transmitted to other neurons via the synapses *(Beale, Jackson, 2001)*.

Learning is thought to occur when modifications are made to the synaptic junctions allowing more or less neurotransmitters to be released *(Beale, Jackson, 2001).*

3.3 The Artificial Neuron

The Artificial Neuron was first proposed by Warren S McCulloch and Walter Pitts in 1943. The basic features of this model are:

- The neuron receives a series of weighted inputs
- The neuron calculates the summation of it's inputs
- The summation of the inputs is passed through an activation function which will output either a 0 or 1 based on whether the sum is above or below a certain threshold *(McCulloch, Pitts, 1943)*.

Figure 3.2 below shows a representation of an artificial neuron.

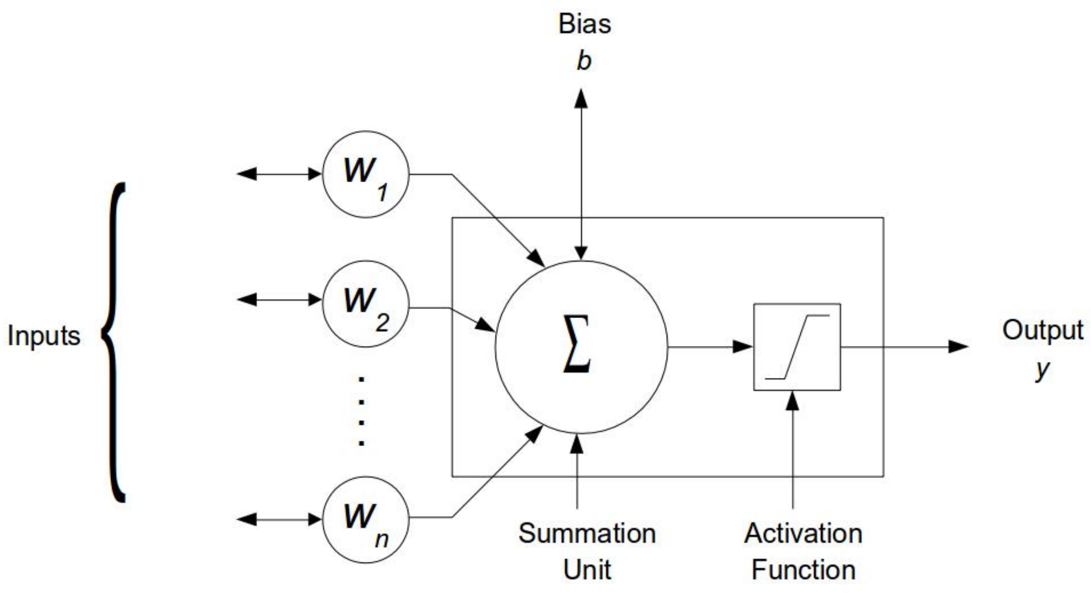


Figure 3.2: An Artificial Neuron

A bias can be added to the neuron which usually has the constant output of -1. This bias is added to push the activation towards 0 *(Beale, Jackson, 2001)*.

The activation function can take many forms. At its most simplest, a step or heavyside function can be used *(Beale, Jackson, 2001)*. Once the summation of the inputs reaches the threshold, the output becomes a one. An alternative to the step function is the sigmoid function. The sigmoid function has the following equation:

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

Where *e* is Euler's Number and *k* is the slope parameter. The slope parameter controls the gradient of the sigmoid and therefore controls its sensitivity *(Beale, Jackson, 2001)*. The lower the number, the more sensitive the function. The benefit of using the sigmoid function as opposed to the step function is that the sigmoid function doesn't produce such a dramatic effect on the output *(Beale, Jackson, 2001)*. This allows the network to accept large inputs and still remain sensitive to small changes *(Beale, Jackson, 2001)*.

Figure 3.3 shows the difference between the sigmoid and step functions.

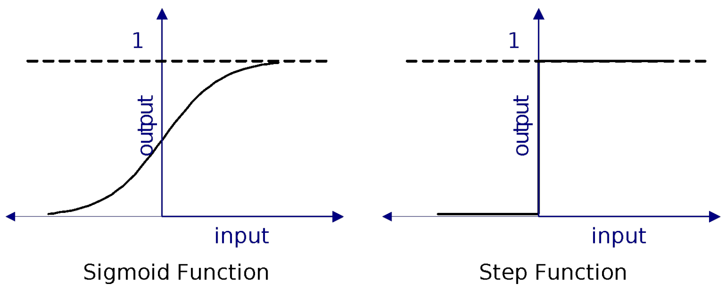


Figure 3.3: Sigmoid and Step Functions

Learning in single artificial neurons or Perceptrons as they are know, is achieved by altering the weights of the connections between neurons to increase or decrease their strength *(Beale, Jackson, 2001)*. There are several algorithms available setting out how weights should be adjusted, the most common of which is the Widrow-Hoff delta rule. This rule relies on being able to calculate the difference between the target output of the neuron and the actual output and then adjusting the weights in proportion to this error in order to minimise it *(Beale, Jackson, 2001)*.

3.4 The Multilayer Perceptron and the Back Propagation Learning Algorithm

The fundamental problem with single Perceptrons is their inability to solve simple linearly inseparable problems such as XOR *(Minsky, Papert, 1969)*. However, by arranging Perceptrons in layers it is possible to overcome this problem. This new model is known as the Multilayer Perceptron (MLP) *(Beale, Jackson, 2001)*, an example of which can be seen in figure 3.4 below. This MLP has 3 input units. The output of these input units is fed together with a bias to the 5 hidden units. These hidden units in turn, together with a bias feed the 2 output units.

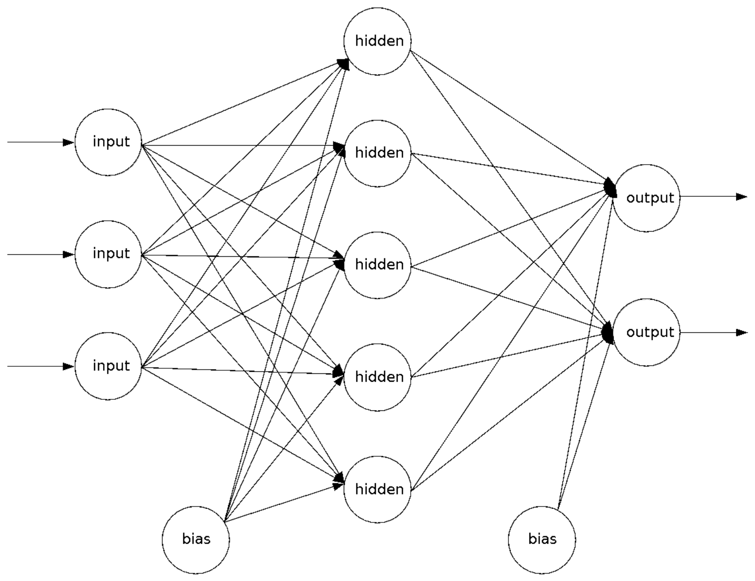


Figure 3.4: The Multilayer Perceptron

Unfortunately as we don't know the output of Perceptrons in the hidden layer of the MLP we cannot use the Widrow-Hoff learning algorithm.

A new learning algorithm is therefore required. Originally proposed by P Werbos *(Werbos, 1974)* and later 'rediscovered' by Rumelhart et al *(Rumelhart, Hinton, Williams, 1986),* the Generalised Delta Rule or Back-Propagation Rule tackles these problems by calculating the error for a particular input and then back-propagating it to the previous layers. The Back-Propagation algorithm consists of two phases know as the forward pass and the backward pass. At the outset, all the weights and thresholds of the MLP are set to small random values *(Beale, Jackson, 2001)*. During the forward pass, inputs are presented to the input neurons. These inputs are then transfer to each of the hidden neurons which computes the sum of the inputs and their weights. Using the following function *(Beale, Jackson, 2001)*:

$$y_{pj} = \sum_{i=0}^{n-1} w_i x_i$$

Where *w* is the weight of input *i* and *x* is the value of input *i*. This sum is then passed through the threshold function. The hidden neurons produce an output based on the threshold function which is then transferred to the output neurons. The output neurons do their own summation of inputs and weights and this is again passed through the threshold function. The output of the output neurons is calculated and this is compared to the target output, to produce an error. The error is computed as follows *(Beale, Jackson, 2001)*:

$$E_p = \tfrac{1}{2} \sum (t_{pj} - o_{pj})^2$$

Where *t* is the target output and *o* is the actual output. The next stage is the backward pass. This involves adjusting the weights of the connections between the hidden and output neurons and input and hidden neurons to minimise the error. The output of a neuron is based on its threshold function, so the derivative of its threshold function can be used to pass the error back to the previous layer. The function used to calculate the new weights is as follows *(Beale, Jackson, 2001)*:

$$w_{ij}(t + \Delta t) = w_{ij}(t) = \eta \delta_{pj} o_{pj} + \alpha(w_{ij}(t) - w_{ij}(t-1))$$

- o_{pj} – The output of the input Neuron
- δ_{pj} – The error term.

$\delta_{pj} = k o_{pj}(1 - o_{pj})(t_{pj} - o_{pj})$	For output units *(Beale, Jackson, 2001)* and
$\delta_{pj} = k o_{pj}(1 - o_{pj}) \sum_k \delta_{pn} w_{jn}$	For hidden units *(Tveter, 2001)* where the sum is over the n neurons in the layer above neuron j

- η - The Learning Rate. This is a number between 0 and 1 which controls the size of adjustments made to the weights. A small number means that the network could take a long time to learn. A large number will allow the network to learn faster, but could potentially push the network into a local minimum *(Beale, Jackson, 2001)*.
- α – Momentum. This is also a number between 0 and 1 which when used in conjunction with a small learning rate can help to speed up convergence *(Beale, Jackson, 2001)*

As the data set that we are using consists of a time series, we need some way of representing time across our network. We will pre-process time by performing a temporal to spatial transformation. The spatial information will be passed to the network which contains a representation of time *(Christodoulou, 2004).*

To achieve this we pass a 'sliding window' across our data. At any given time t, for *n* inputs and time step *τ*, the input values will be *t, t - τ, t - 2 τ …. t - n τ.* The target output will be *t + τ.*

At each time step the values of the nodes are shifted down and the one at the end drops out. A new value is then inserted at the first node. See Figure 3.5 below.

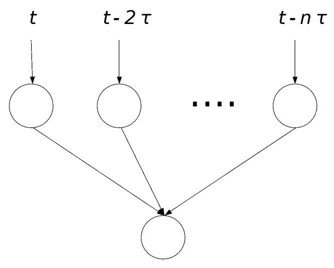


Figure 3.5: Time to Space Transformation

3.5 The Time Delay Neural Network

The main problem with representing time as discussed above is the inability of the network to recognise the same pattern shifted in time.

If we consider the binary pattern 0011100100 at time t, the corresponding pattern at time t + 2 *τ* might be 0000111001. The MLP with temporal to spatial transformation would see these as two completely different patterns *(Christodoulou, 2004)*.

The Time Delay Neural Network (TDNN) as described by Lang et al *(Lang, Waibel, Hinton, 1990)* can capture this translational invariance. Figure 3.6 below shows an example of this type of network.

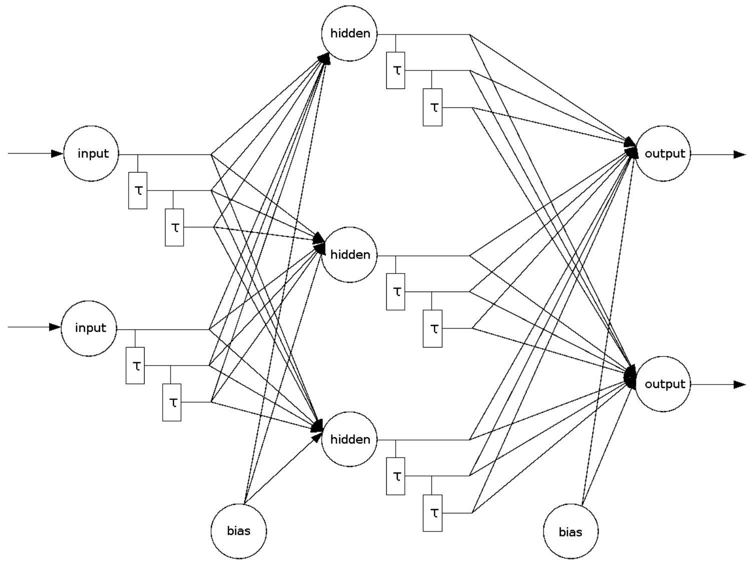


Figure 3.6: The Time Delay Neural Network

Each input and hidden node in the TDNN, stores a fixed amount of previous values, the amount of which depends on the delay length.

The TDNN uses the standard Back Propagation learning algorithm, with a slight variation:

- All values for each node are used to feed the nodes in the next layer.
- Weight changes are calculated for each value of each node and an average is taken.
- All weights for each node are then updated to the same value to achieve invariance *(Schalkoff, 1997).*

A ‘sliding window’ will also be used in conjunction with the network to feed in the data.

3.6 The Recurrent Neural Network

Using a fixed number of delays, the number of which is set in advance, can however impose limitation of the length of patterns that can be learnt *(Christodoulou, 2004)*. Additionally, each delay can be viewed to a certain extent, as an independent node. If large numbers of delays are used, this could dramatically increase the length of time required for training *(Christodoulou, 2004)*. The Recurrent Neural Network (RNN) is an alternative to the TDNN which works on the principle that time and memory are highly task dependent, so networks should be allowed to achieve their own representation of them, instead of having them fixed *(Elman, 1990).*

These networks have an extra set of nodes called Context Neurons. At each time step the current value of the hidden units is copied to corresponding context units *(Elman,* *1990)*. A proportion of the previous value of the context unit is also used to provide its new value. The size of this proportion is known as the Memory Depth. The following equation is used to calculate this new value *(Hertz, Krogh, Palmer, 1991):*

$$C_i(t+1) = \alpha C_i(t) + H_i(t)$$

Where *C* is the context unit, *H* is the hidden unit and *α* is the proportion. This value is then fed to the network at the next time step together with the new inputs from a sliding window. Figure 3.7 shows a similar layout to the RNN that is used in this project.

The back propagation learning algorithm can also be used for weight adjustment.

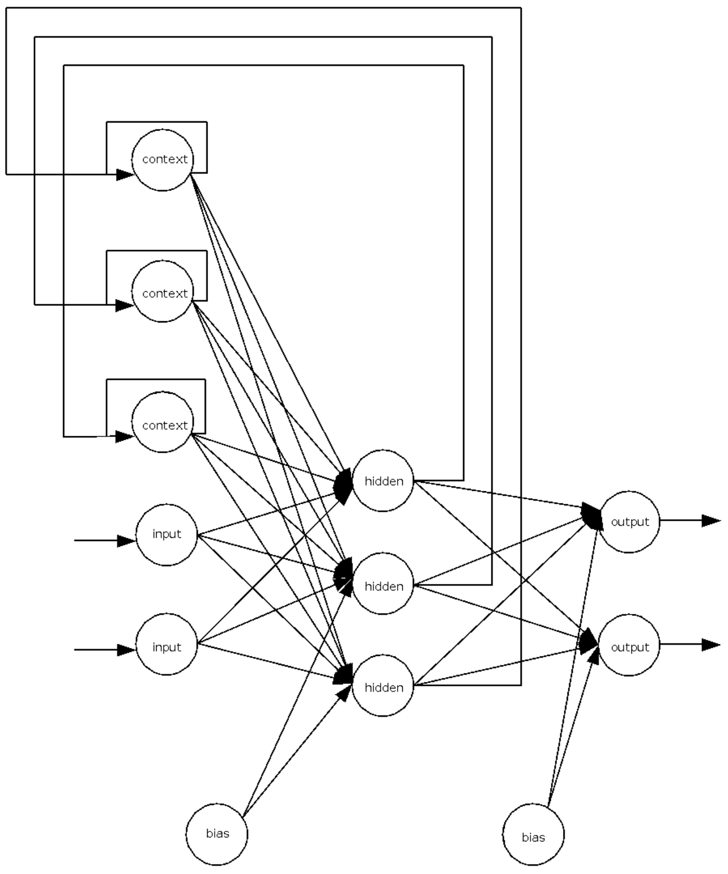


Figure 3.7: A Recurrent Neural Network

Chapter 4 – Design and Implementation

The usefulness of a neural network is measured on its ability to generalise. That is to successfully classify patterns that have not been previously presented *(Beale, Jackson, 2001)*. This generalisation ability is achieved through the combined effect of the network's parameters. However, there is no magic formula with respect to their selection to produce the ideal network topology *(Kaastra, Boyd, 1996).* Instead the process is pretty much trail and error. Many combinations must be tried out, for each type of network, the result of which can be exceptionally long training and testing periods.

Kaastra and Boyd *(Kaastra, Boyd, 1996)* highlight a number of distinct requirements that need to be considered.

4.1 Variable Selection

The aim of this project is to forecast the end of next day closing price of the FTSE 100 Index. There are many variables that can be used for this purpose, such as traded volume and moving averages etc. However for simplicity sake, I have only used historical closing price data.

4.2 Data Collection

Data was downloaded directly from the FTSE's website at www.ftse.com in CSV format, free of charge. Once downloaded the data was manually checked for error, before been converted into XML format and validated against the project schema which can be seen in Appendix B

4.3 Data Pre-Processing

There is a plethora of ways to transform the raw data in order to help the network identify trends and patterns, the usefulness of which should be compared and tested against each other. As a result of time requirements, I have only performed a simple scaling of the data between the values of 0 and 1.

4.4 Training, Testing and Validation Sets

There seems to be some confusion in current literature as to the correct sequence of names used for the different stages used in training and testing. Some refer to the process as being Train – Validate – Test *(Sarle, 2004)*, while others refer to it as Train – Test – Validate *(Kaastra, Boyd, 1996)*. The stages themselves however remain the same. For the purposes of this report, I will use the naming convention used by Kaastra and Boyd, which is the latter.

After the data was converted into XML format, it was divided up into the following sets:

- The Training Set – This data is used in the training process and is repeated shown to the network to try to get it to identify patterns. It is the largest of the three sets, comprising of 500 values dating from 26-09-2001 to 17-09-2003 inclusive.
- The Testing Set – This data is used after the network has been trained to the required level to test the networks ability to generalise. It consists of 100 values dating from 18-09-2003 to 09-02-2004
- The Validation Set – The purpose of this data is similar to that of the Testing Set, it is also used after the Training Set, but it is used to validate the testing, to prevent fluke results. It also consists of 100 values, dating from 10-02-2003 to 02-07-2004

4.5 Neural Network Parameters

There are a number of different parameters that fall under this heading. These include the number of input neurons, the number of hidden layers, the number of hidden neurons, the number of output neurons, the transfer function, the learning rate, the momentum, the number of time delays and the depth of memory.

4.5.1 Number of Input Neurons

In a general network that doesn't use the sliding window concept, it is usual to have one input for each input variable. However by using a sliding window we can increase the size of patterns that the network can see at each time step. Networks with 5, 10 and 15 input neurons have subsequently been tested.

4.5.2 Number of Hidden Layers

It is possible to create networks with multiple hidden layers. This project will only test networks with 1 hidden layer as Kolmogorov representation theorem proves that no more than 3 layers in total are needed to represent any function *(Beale, Jackson, 2001)*.

4.5.3 Number of Hidden Neurons

The number of hidden neurons controls the networks ability to generalise. Too few neurons and the network will not be able to learn, too many neurons and there is a danger that the network may learn too well and memorise individual points, losing its ability to generalise. For the purposes of these experiment, networks with 5, 10 and 15 hidden units have been tested.

4.5.4 Number of Output Neurons

The nature of the problem that is investigated in this project is one of trying to forecast a single specific future value. One way of tackling this could involve several output neurons, each representing a range of values. The approach taken in this project however is to have one output neuron, whose output value will be the forecasted value.

4.5.5 Transfer Function

For the purposes of this project, the sigmoid transfer function has been used.

4.5.6 Learning Rate

As described in the previous chapter, learning rate is used to control the size of adjustments made to the weights of the connections between neurons. This is a number between 0 and 1, the larger the number the faster the learning. Learning rates of 0.01, 0.2 and 0.8 have been chosen to cover a wide range of learning speeds.

4.5.7 Momentum

Also described in the previous chapter, momentum is used in conjunction with a low learning rate in order to speed up convergence. Values of 0.0, 0.1, 0.5 and 0.9 have been tested.

4.5.8 Number of Time Delays

Described in the TDNN section of the previous chapter, 1 and 2 delays have been tested.

4.5.9 Depth of Memory

Described in the RNN section of the previous chapter, depths of 0.1, 0.5, and 0.9 have been tested

4.6 Evaluation criteria

The performance of a neural network can be measured by the error produced between the target output and the actual output. There are many different error functions that can be used to compute this error. For the purposes of this project we will use the sum of the squared error *(Beale, Jackson, 2001)*.

$$E_p = \frac{1}{2}\sum(t_{pj} - o_{pj})^2$$

4.7 Number of Training Epochs

Networks need to be trained until they reach a point where there is little improvement in their error. This point is known as convergence. However if the network is trained for too long it can result in poor generalisation performance. This is known as overfitting *(Kaastra, Boyd, 1996)*. The networks in this report have been trained with 1000, 3000, 5000, 7000 and 10000 epochs, to test for this over training.

4.8 Implementation

The implementation language chosen for this project was Java, with processing data stored in XML format. These were chosen because of their ease of use and platform independence.

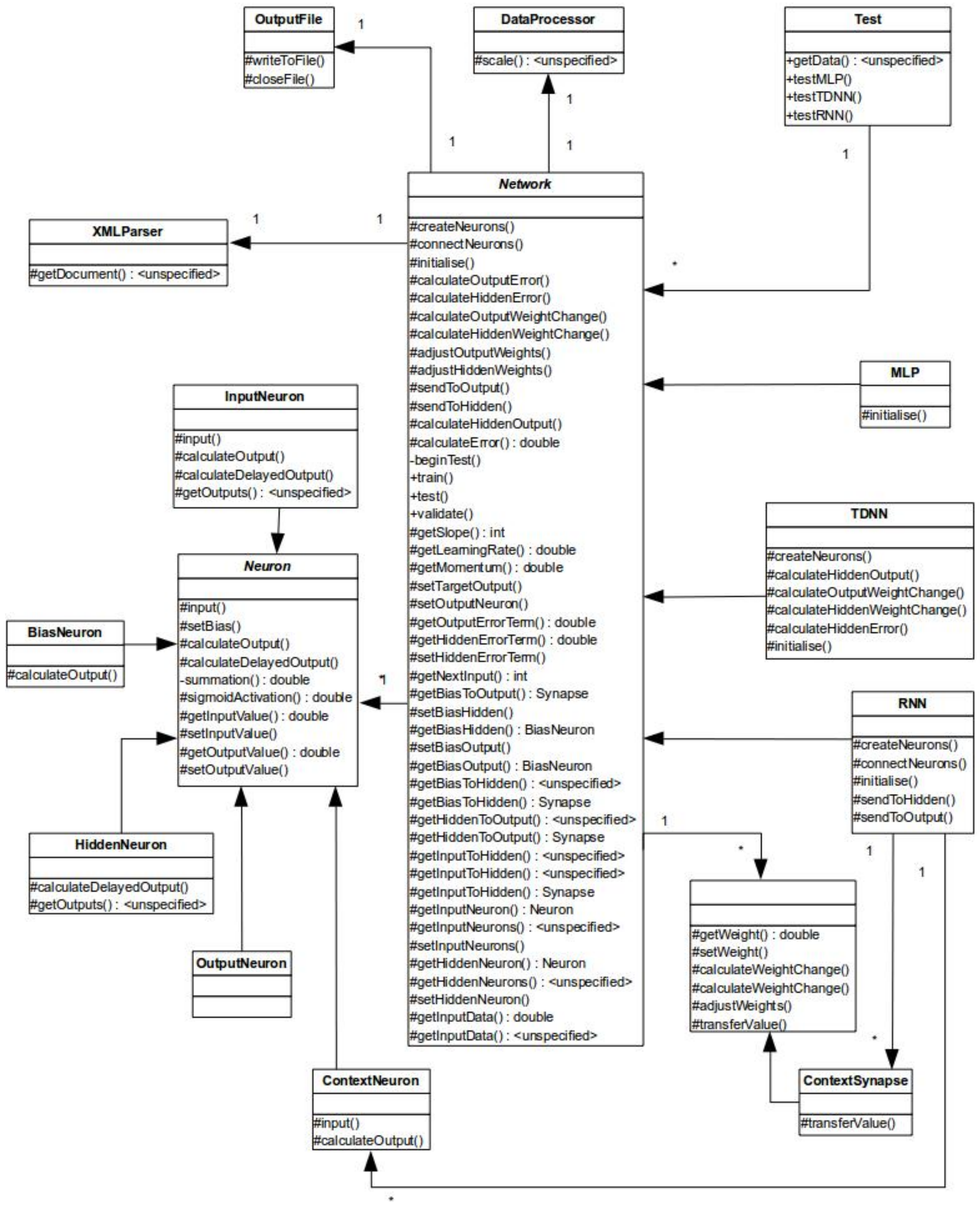
4.8.1 Requirements

The main focal point of this project is to test the ability of many different pre-selected MLP, TDNN and RNN topologies to forecast FTSE 100 closing prices. By the very nature of this problem, very little interaction is requirement between the user and the system. This requirements section will therefore focus on the requirements that the system has rather than the requirements that the user has.
These requirements are listed as follows:

- Parse XML data for training and testing or validating and store values in internal structures
- Normalise values to between 0 and 1
- Create network with specified parameters
- Train network to specified number of epochs
- Test or validate network
- Write results to log file

4.8.2 Structure

The software was designed using Object Oriented principles of Inheritance, Data Encapsulation and Polymorphism. Figure 4.1 below shows the UML Design Class Diagram which illustrating the software classes created.



```
#createNeurons()
#connectNeurons()
#initialise()
#calculateOutputError()
#calculateHiddenError()
#calculateOutputWeightChange()
#calculateHiddenWeightChange()
#adjustOutputWeights()
#adjustHiddenWeights()
#sendToOutput()
#sendToHidden()
#calculateHiddenOutput()
#calculateError() : double
-beginTest()
+train()
+test()
+validate()
#getSlope() : int
#getLearningRate() : double
#getMomentum() : double
#setTargetOutput()
#setOutputNeuron()
#getOutputErrorTerm() : double
#getHiddenErrorTerm() : double
#setHiddenErrorTerm()
#getNextInput() : int
#getBiasToOutput() : Synapse
#setBiasHidden()
#getBiasHidden() : BiasNeuron
#setBiasOutput()
#getBiasOutput() : BiasNeuron
#getBiasToHidden() : <unspecified>
#getBiasToHidden() : Synapse
#getHiddenToOutput() : <unspecified>
#getHiddenToOutput() : Synapse
#getInputToHidden() : <unspecified>
#getInputToHidden() : <unspecified>
#getInputToHidden() : Synapse
#getInputNeuron() : Neuron
#getInputNeurons() : <unspecified>
#setInputNeurons()
#getHiddenNeuron() : Neuron
#getHiddenNeurons() : <unspecified>
#setHiddenNeuron()
#getInputData() : double
#getInputData() : <unspecified>
```

Figure 4.1: UML Design Class Diagram

The design can be divided into five distinct sub-structures. These are as follows:

- The Test Class
- The Network Classes
- The Neuron Classes
- The Synapse Classes
- The Processing/IO Classes

4.8.2.1 The Test Class

This class is used to instantiate various network object with the correct parameters and train, test and validate them. This class is multi-threaded to take advantage of any dual processing capability of the execution environment.

4.8.2.2 The Network Classes

This group consists of the abstract base class Network and its derived classes MLP, TDNN and RNN. The base class contains the general methods for creating and connecting neurons, the standard implementation of the back propagation learning algorithm and methods for training, testing and validating. The derived classes contain overriding methods for network type specifics.

4.8.2.3 The Neuron Classes

This group comprises of the abstract base class Neuron and its derived classes InputNeuron, HiddenNeuron, OutputNeuron, BiasNeuron and ContextNeuron. The base class contains methods for receiving input, summation of the inputs, the sigmoid activation function, calculation of output and outputting. The derived classes contain overriding methods which deal with the use of time delays, recurrent memory and bias.

4.8.2.4 The Synapse Classes

There are two synapse classes. The base class is called Synapse and its methods deal with transferring values, calculating weight changes and adjusting the weight. The derived class is ContextSynapse and is specific to the RNN network. Its weight is not adjustable.

4.8.2.5 The Processing/I.O. Classes

This contains three different classes that are essential for the running of the project. These are the XMLParser class which is used to parse the XML document and return a W3 Consortium Document Object Model (DOM) object. The DataProcessor class, which is used to scale the data between 0 and 1. Finally the OutputFile class which is used to create log files in CSV format.

Full source code listings can be found in Appendix A

Chapter 5 – Results, Analysis and Findings

In total 648 network combinations began the training stage.

The project development took place on a PC with a 1GB AMD Athlon processor, running Windows XP professional. This machine however, was not sufficiently powerful enough for execution, so a dual 64bit SPARC machine running Sun Solaris OS was used.

Training times ranged from between 4 minutes to 30 minutes a network depending on the number of neurons and the number of training epochs used.

Throughout this chapter reference will be made to network topologies. This will take the following standard formats:

MLP:	input neurons-hidden neurons-learning rate-momentum-(epochs)
TDNN:	input neurons-hidden neurons-delays-learning rate-momentum-(epochs)
RNN:	input neurons-hidden neurons-memory depth-learning rate-momentum-(epochs)

5.1 Training Results and Analysis

The training stage involves repeatedly showing the networks the same set of data and adjusting the weights of the connections between neurons in order to reduce the output error. All networks were initially trained with 10000 epochs and their average error for each epoch was recorded in the log file. The data set used for this was from 09/2001 to 09/2003. It must be stated that a networks ability to learn training data bears little correlation to its ability to generalise with data that it hasn't seen before. However this research was carried out in order to eliminate networks that showed no sign of being able to learn or who have settled in high local minima. At this stage, networks were removed if they did not pass the following criteria:

- Achieve a final output error less than 1.0×10^{-3} and
- Showed smooth(ish) learning

1.0×10^{-3} was chosen as the acceptable error limit. On examination of training results it appeared to strike a balance between evicting networks that were obviously stuck in local minima and not evicting too many networks so that very few made it to the next stage.

Figure 5.1 below show the effects of training MLP networks with 5 inputs

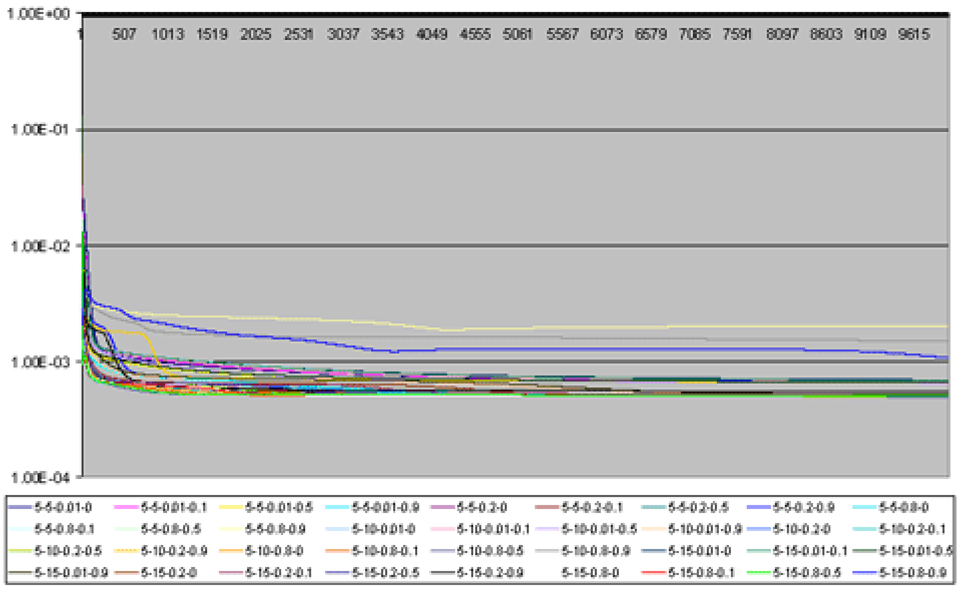


Figure 5.1: MLP networks with 5 inputs trained with 10000 epochs

It can be clearly seen in this graph, that 3 of the networks, shown as the yellow (5 hidden units), grey (10 hidden units) and blue lines (15 hidden units), did not achieve the output error of 1.0 x 10^{-3}. These 3 networks all have the same learning rate (0.8) and momentum (0.9). This would suggest that these networks have got stuck in local minima. Figure 5.2 below shows the effects of training MLP networks with 10 inputs.

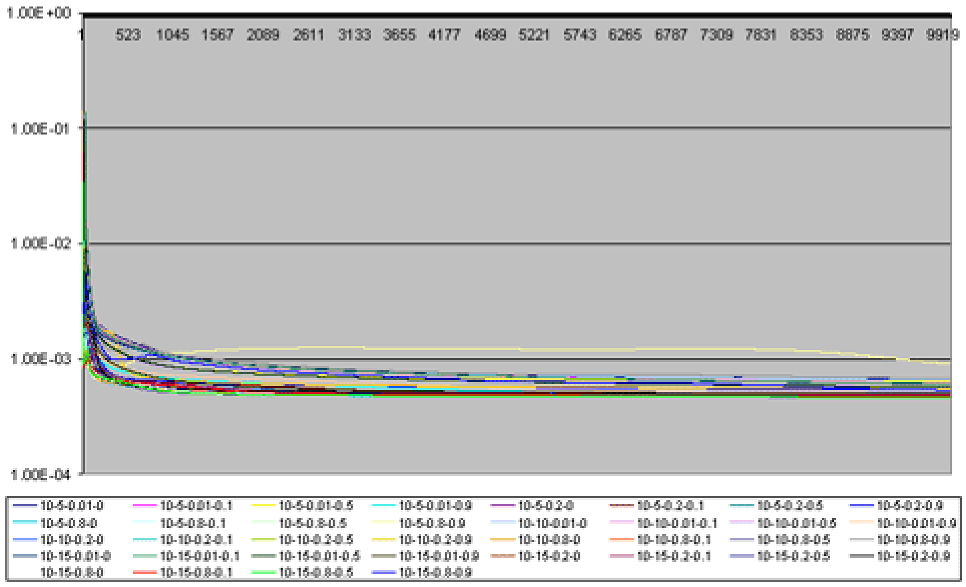


Figure 5.2: MLP networks with 10 inputs trained with 10000 epochs

All the MLP networks that had 10 input neurons managed to achieve an output error of at least 1.0 x 10^{-3}. The one network that causes a slight concern is the one shown in yellow (5 hidden units), which seems to get worse before getting better. Interesting, this network also had the same learning rate (0.8) and momentum (0.9) as the networks with 5 inputs units that failed this stage. This network however will continue to be included. Figures 5.3 below shows the effects of training MLP networks with 15 inputs

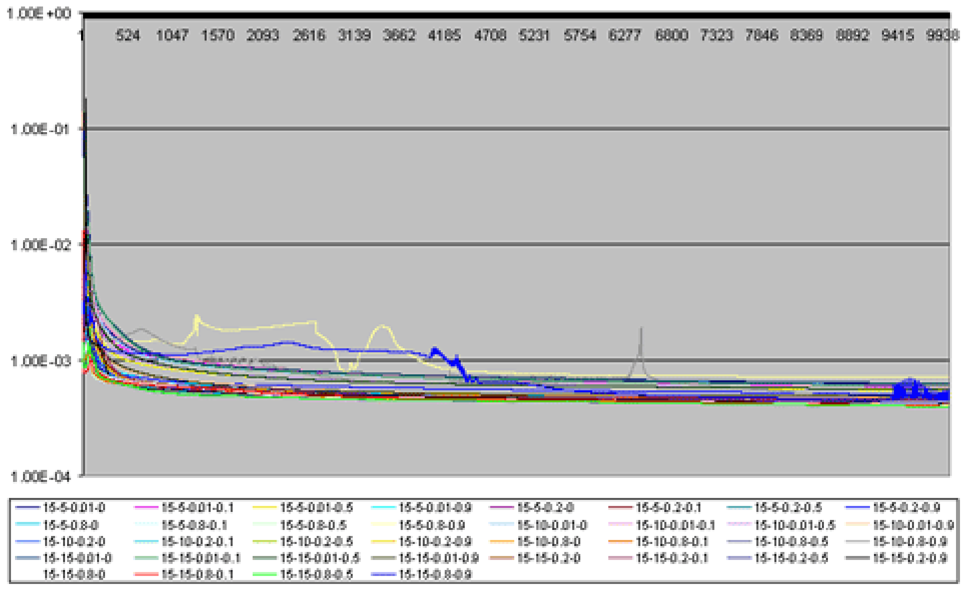


Figure 5.3: MLP networks with 15 inputs trained with 10000 epochs

Again as we have seen before, 3 networks shown in yellow (5 hidden units), grey (10 hidden units) and blue (15 hidden units) appear to have had problems. Not surprising, they too had a learning rate of 0.8 and a momentum of 0.9. This time though they did manage achieve a low enough final error, but their learning is erratic. This would suggest that these parameters had caused the networks to oscillate instead of finding their true minimum. Out of the 108 different MLP network combinations only 6 didn't pass. Table C.1 shows these topologies.

If we move on to the TDNN networks, Figures 5.4 below shows the effects of training TDNN networks with 5 inputs.

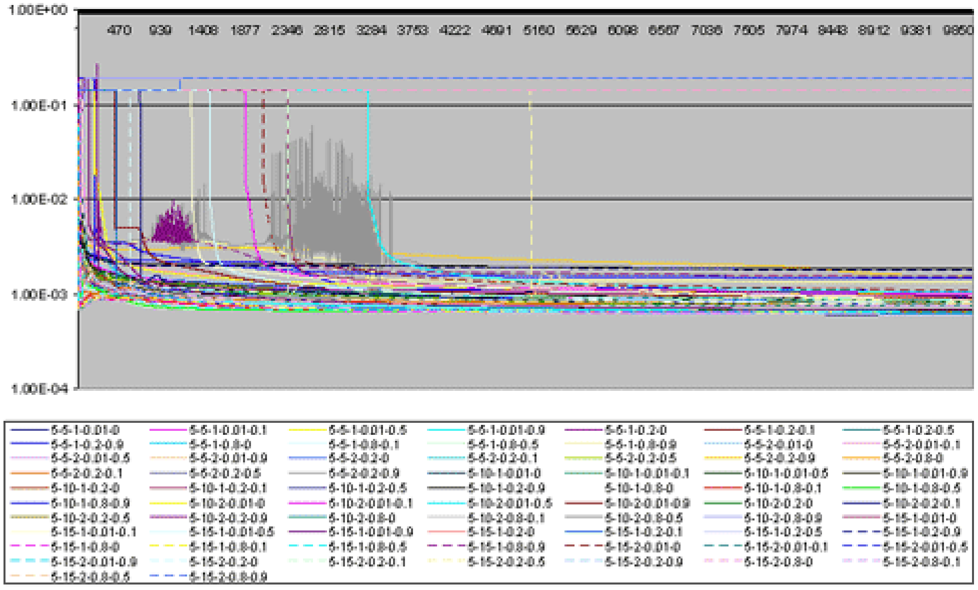


Figure 5.4: TDNN networks with 5 inputs trained with 10000 epochs

7 combinations showed no sign of learning and are shown under the broken red and blue lines. 6 of these had 15 hidden neurons and the other had 10. All had 2 delays. Several of the networks initially appeared not to learn, before their error rapidly dropped over a few epochs. Two networks shown as the grey (5-5-2-0.2-0.9) and mauve (5-15-1-0.8-0.9) show signs of oscillation about the error surface. Table C.2 shows the full listing of failed network combinations with 5 inputs. It can be seen that 21 out of these 28 networks had 2 delays. With 36 possible networks with 5 hidden neurons and 2 delays, this represents just over 58 percent of them. Figures 5.5 below shows the effects of training TDNN networks with 10 inputs

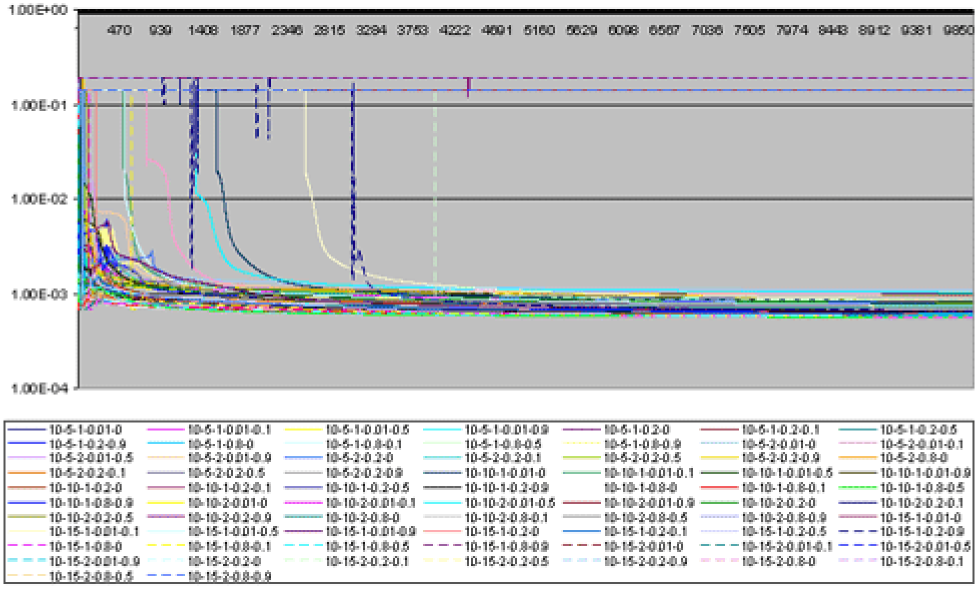


Figure 5.5: TDNN networks with 10 inputs trained with 10000 epochs

Out of the twelve networks that had 15 hidden units and a delay of 2, only 10-15-2-0.2-0.1 actually learnt anything. This network was also evicted though for not reaching a sufficiently low enough error. Nine others also didn't achieve any learning, and out of these, only 10-15-1-0.01-0 and 10-15-1-0.8-0.9 didn't have 2 delays. These two did however, have the largest and smallest learning rate/momentum combinations. Table C.3 shows the full listing of failed network combinations with 10 inputs. Again, only 7 out of 32 networks didn't have 2 delays. This time over 69 percent of the 2 delay networks failed.

Figures 5.6 below shows the effects of training TDNN networks with 15 inputs

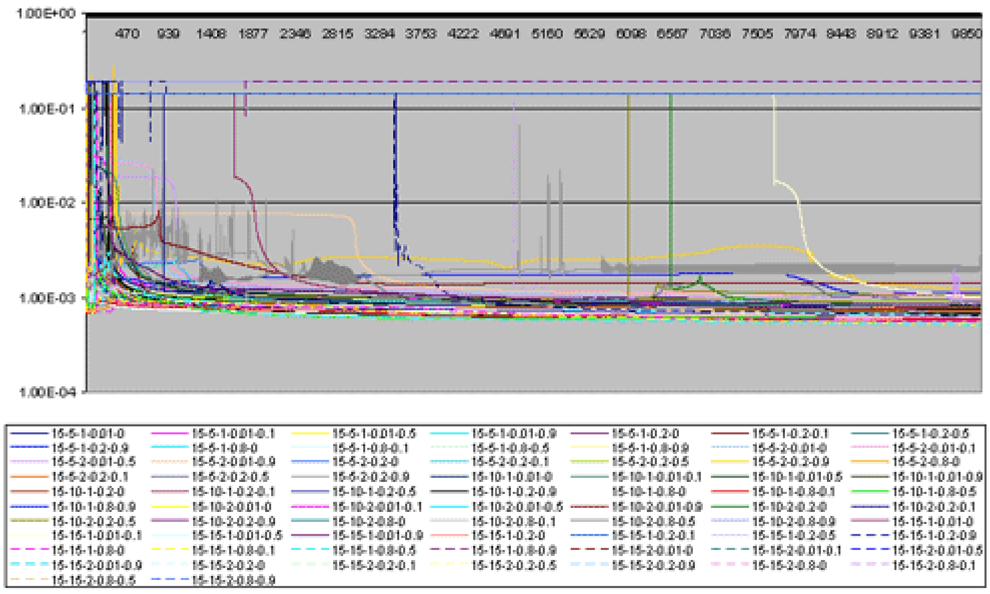


Figure 5.6: TDNN networks with 15 inputs trained with 10000 epochs

Again, only one network that had 15 hidden neurons and 2 delays showed learning ability. This was the 15-15-2-0.8-0.1 topology. However, what it did produce was not sufficient enough to get it through this stage. In addition, the 15-5-2-0.8-0.9 combination was the only 0.8 learning rate/0.9 momentum network that had any error reduction. Of the 216 different TDNN topologies trained, 101 showed poor learning or signs of being stuck in local minima.

Table C.4 shows the full listing of failed network combinations with 15 inputs. This time 13 from 41 failed networks had 1 delay, giving a 78 percent failure rate to the 2 delay networks. In addition, clustering can also be seen around the 15 hidden, 1 delay and 0.01 learning rate combination, with all of them failing.

As for the RNN networks, Figures 5.7 below shows the effects of training RNN networks with 5 inputs.

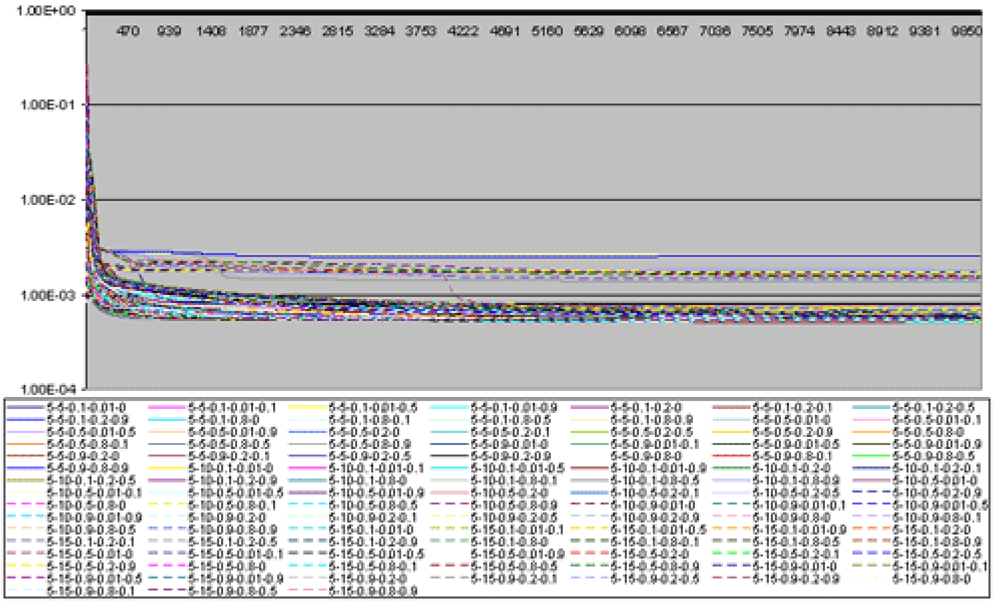


Figure 5.7: RNN networks with 5 inputs trained with 10000 epochs

RNN networks with 5 input neurons showed similar learning ability to MLP networks with 5 input neurons. As with the MLPs, the main problem faced by some of these networks was getting stuck in local minima. Not surprisingly all networks with the 0.8 learning rate/0.9 momentum suffered from this.

Figures 5.8 below shows the effects of training RNN networks with 10 inputs.

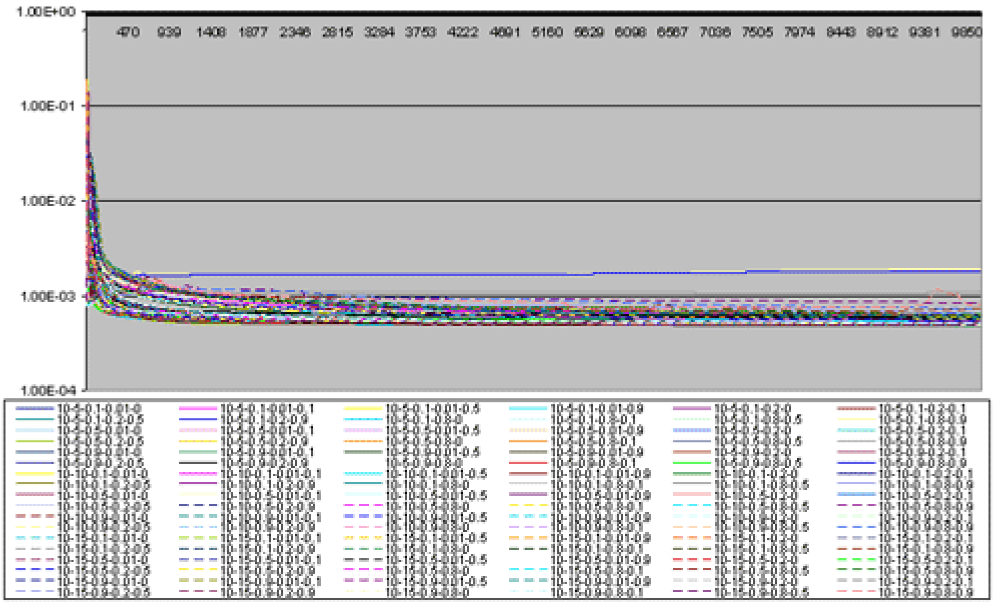


Figure 5.8: RNN networks with 10 inputs trained with 10000 epochs

RNN networks with 10 input neurons produced the best RNN results with only 5 out of a possible 108 failing. All of these had the 0.8 learning rate/0.9 momentum combination.

Figures 5.9 below shows the effects of training RNN networks with 15 inputs.

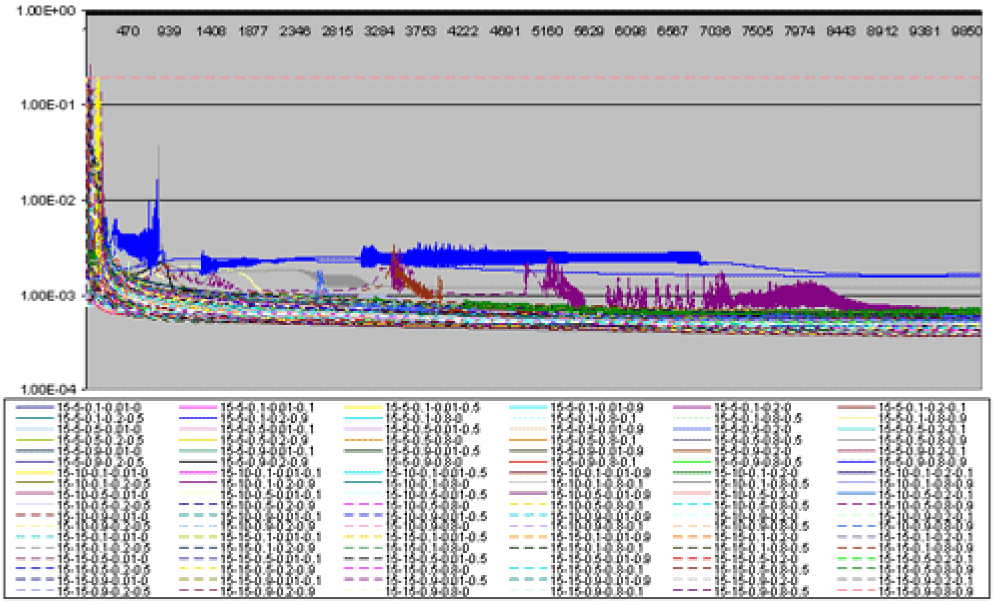


Figure 5.9: RNN networks with 15 inputs trained with 10000 epochs

The main cause of failure for RNNs with 15 units is the inability to converge. This can be seen in the blue (15-5-0.9-0.8-0.9) and maroon (15-10-0.5-0.8-0.9) lines. One network showed no learning. This is the 15-15-0.9-0.8-0.9 network and is shown as the broken red line. The vast majority of the RNN networks did pass the training phase though, with only 27 out of 324 being excluded from the next stage. These are listed in Table C.5.

5.2 Training Findings

What is obvious about the results of the MLP training is that all networks that were eliminated had learning rate of 0.8 and momentum of 0.9. There were initially nine networks that used these values, so this group represented two-thirds of them. The three networks with 5 inputs, showed smooth learning, but did not achieve a sufficiently low error. This would suggest that they were stuck in local minima. The three networks with 15 inputs achieved a low enough error, however they showed erratic learning. This would suggest that these parameters had caused the networks to oscillate instead of finding their true minimum.

The main cause of TDNN failure at this stage is the number of delays. Of the 101 networks that did not make it past this stage, 27 had one delay and 74 had two delays. The effect of the delay appears to outweigh more subtle effects caused by learning rate/momentum combinations. However, of the networks that had only one delay, clusters have emerged around larger network topology and certain learning rates. This is most evident with a learning rate of 0.01.

Training results of the RNN networks are similar to those of the MLP, where of the 27 failing networks, 23 had the 0.8 learning rate – 0.9 momentum combination. The remaining 4 networks also used a momentum value of 0.9. As with the TDNN networks, clusters of networks were eliminated which had similar topologies, but varying memory depths.

5.3 Testing and Validation Results and Analysis

The Testing and Validation Stages are where the real ability of the networks to generalise is discovered *(Kaastra, Boyd, 1996)*. The Test stage involves showing the networks a single pass through the testing data set and recording the errors produced. The main differences between this stage and the training are as follows:

- Data has been unseen by the network, as apposed to being repeatedly shown.
- The weights of the connections between neurons are not adjusted.

The test data set comprise of entries between 09/2003 to 02/2004. During the testing stage overfitting can be detected by training networks with different numbers of epochs and detecting deterioration in generalisation ability over the test data. For the purposes of this project, 1000, 3000, 5000, 7000 and 10000 epochs were tested. Figure 5.10 shows a cross-section of results for the MLP networks when trained with different training periods.

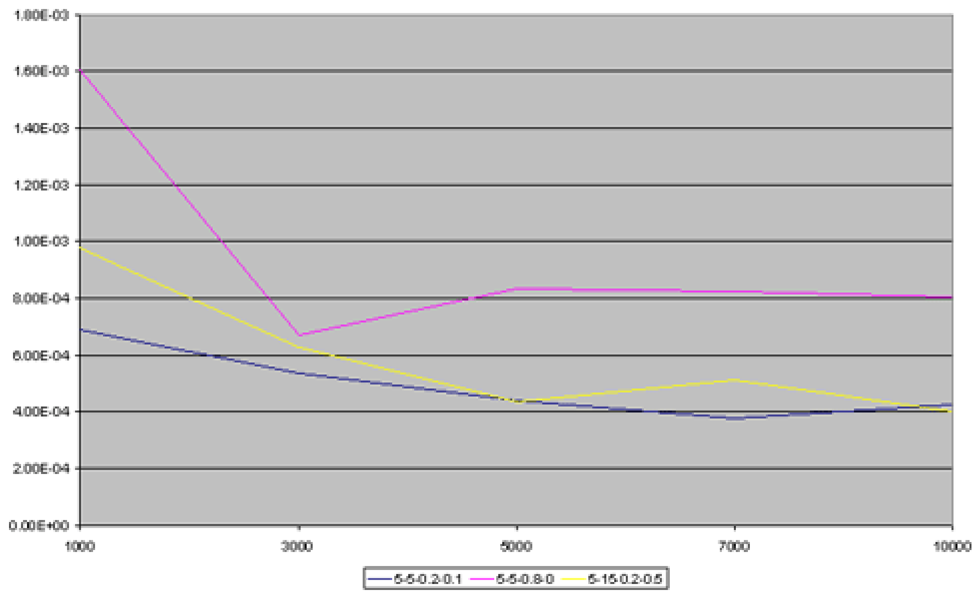


Figure 5.10: Errors of a cross-section of MLP networks when trained with different training periods showing optimal training length. Network topologies shown in the legend are represented

Figure 5.11 shows a cross-section of results for the TDNN networks when trained with different training periods.

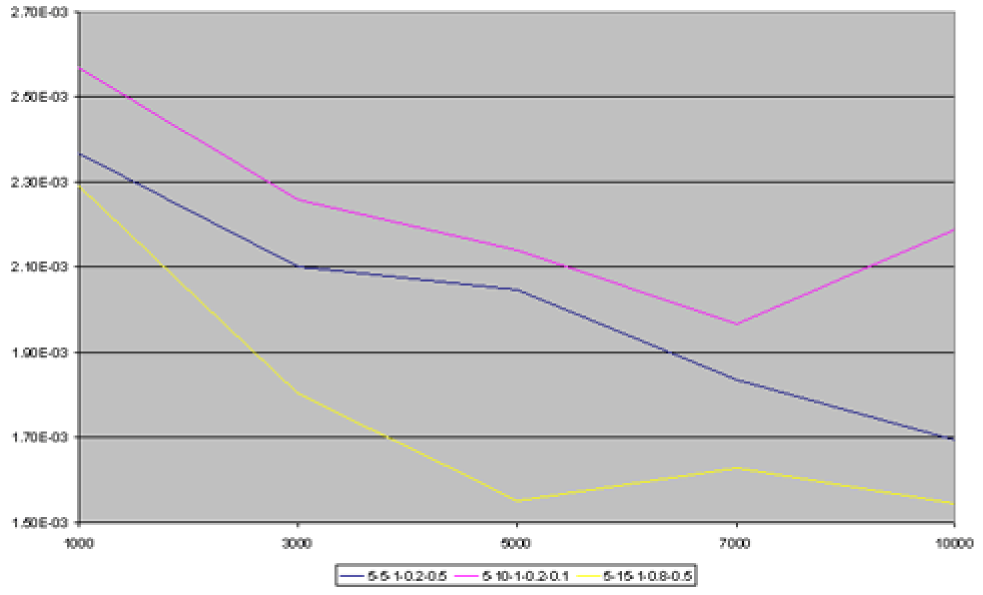


Figure 5.11: Errors of a cross-section of TDNN networks when trained with different training periods showing optimal training length. Network topologies shown in the legend are represented

Figure 5.12 shows a cross-section of results for the RNN networks when trained with different training periods.

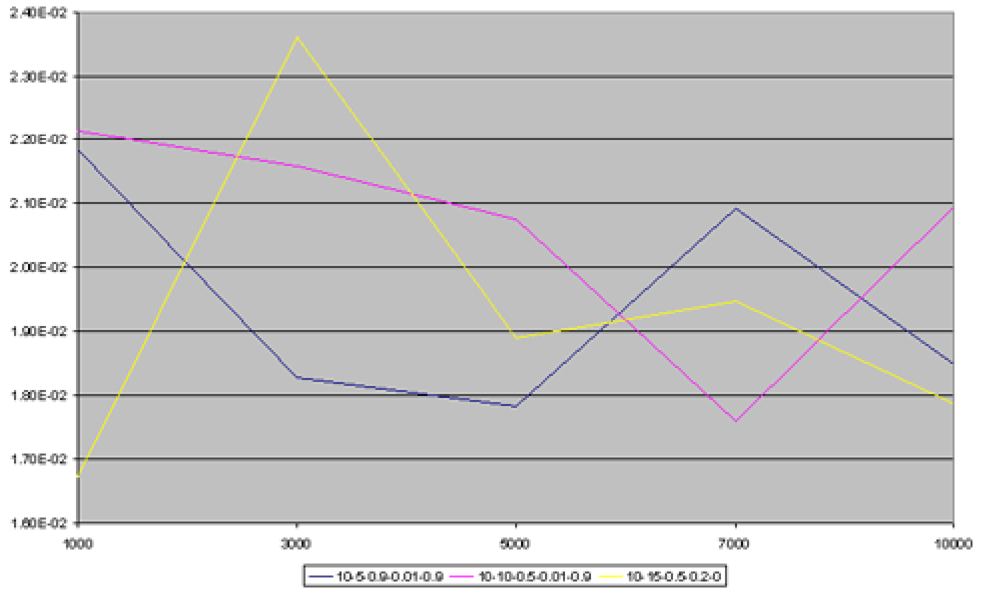


Figure 5.12: Errors of a cross-section of RNN networks when trained with different training periods showing optimal training length. Network topologies shown in the legend are represented

Initial analysis of the testing results showed a wide range of optimal training epochs. It was therefore decided that all networks would also be validated with each of the different number training epochs as well.

The validation stage is similar to the testing stage however a different data set is used. Generalisation ability is determined over performance on both sets. The data used for validation is from 02/2004 to 07/2004. Figures 5.13 shows the combined results of the test and validation, together with their mean, for the MLP networks trained with 1000 epochs.

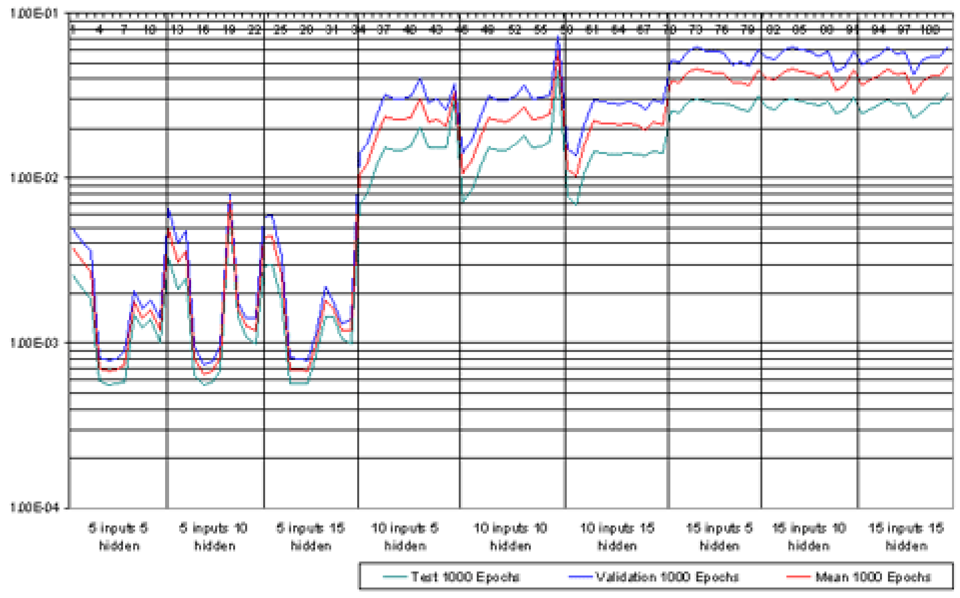


Figure 5.13: Log graph showing test and validation results together with the mean, for MLP networks trained with 1000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

Each individual network is represented by a point along each line. What is immediately obvious from this graph is that the 5 input networks have produced by far the lowest errors over both data sets. Three troughs can be seen in the 5 input section, one for 5 hidden units, one of 10 hidden units and one for 15 hidden units. These correspond to networks with learning rates of 0.2. It can also be seen that the 10 input networks have all produced better results than the 15 input networks, with one exception. The high peak that can be seen towards the end of the 10 input-10 hidden section represents network 10-10-0.8-0.9. An evenly poor performance can be seen throughout all 15 input networks. The test and validation lines follow similar paths, peaking and troughing at the same points. This similarity suggests generalisation on behalf of the networks involved. An interesting point to note is that all networks appear to have performed better over the test data than the validation data. This may be due to the fact that the test data was closest date wise to the training data.

Figure 5.14 shows the combined results of the test and validation, together with their mean, for the MLP networks trained with 3000 epochs.

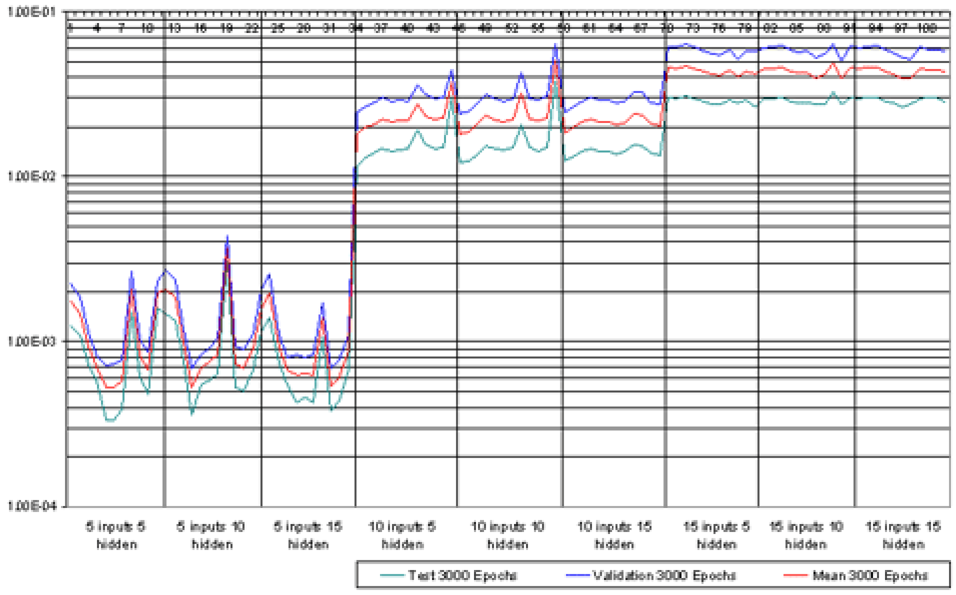


Figure 5.14: Log graph showing test and validation results together with the mean, for MLP networks trained with 3000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

The test and validation line for MLP networks trained with 3000 epochs again show that networks performed better with the test data than the validation data. Sections of the lines representing the 5 input networks are slightly lower than the corresponding sections for the 1000 epoch trained networks. This indicates they are producing better results. Little improvement however, can be seen with the 10 and 15 input networks.

Figure 5.15 shows the combined results of the test and validation, together with their mean, for the MLP networks trained with 5000 epochs.

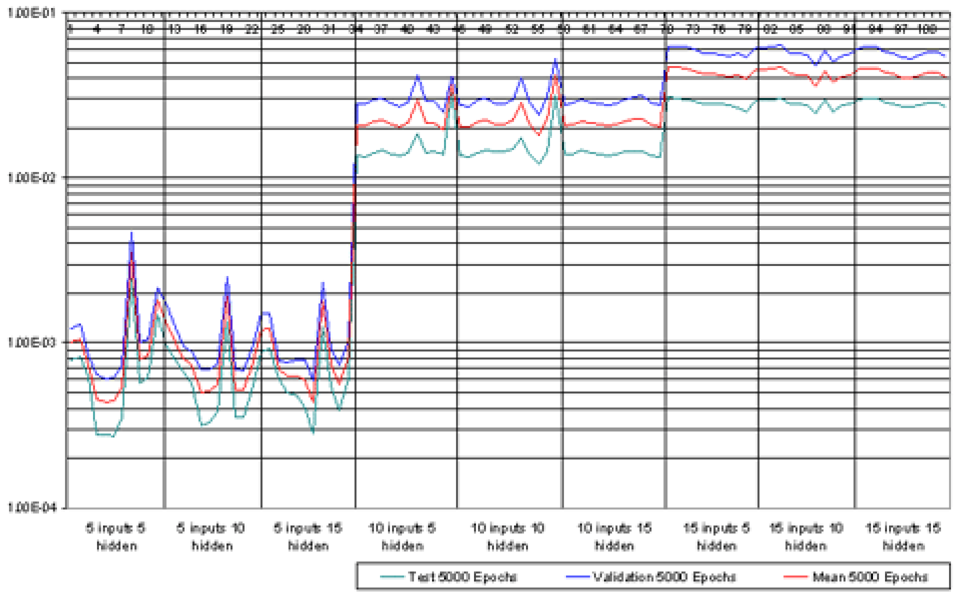


Figure 5.15: Log graph showing test and validation results together with the mean, for MLP networks trained with 5000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

Continued improvement can be seen with the 5 input networks and little, if any can be seen with the 10 and 15 input networks. Three peaks have also started to form in the 5 input section. These represent networks 5-5-0.2-0.9, 5-10-0.2-0.9 and 5-15-0.2-0.9 respectively.

Figures 5.16 shows the combined results of the test and validation, together with their mean, for the MLP networks trained with 7000 epochs.

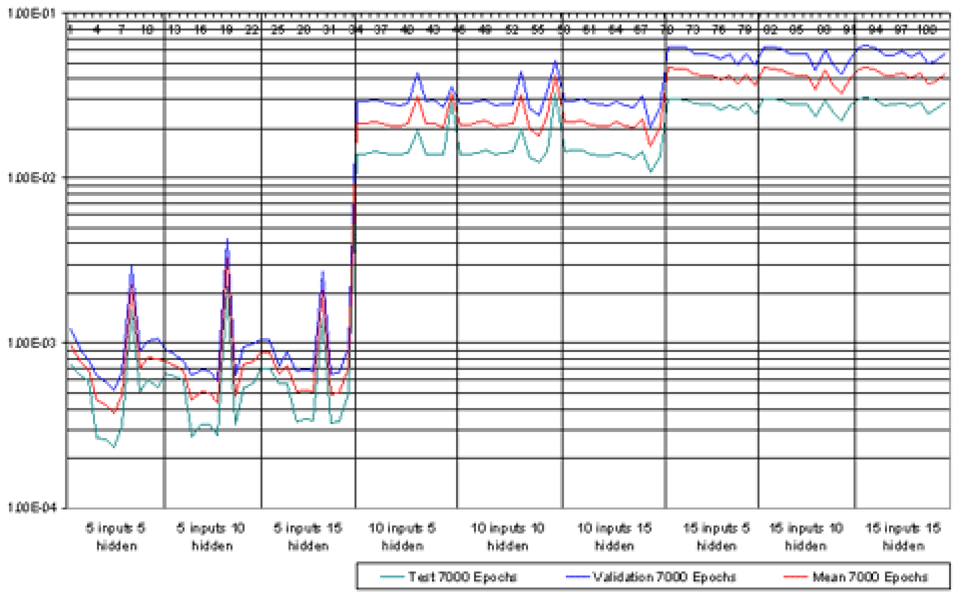


Figure 5.16: Log graph showing test and validation results together with the mean, for MLP networks trained with 7000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

Further improvement can be seen amongst the best performing 5 input networks, but deterioration can be seen in the networks representing the three peaks. Slight improvement can also be seen in parts of the 10 and 15 input sections, with the formation of several troughs.

Figure 5.17 shows the combined results of the test and validation, together with their mean, for the MLP networks trained with 10000 epochs.

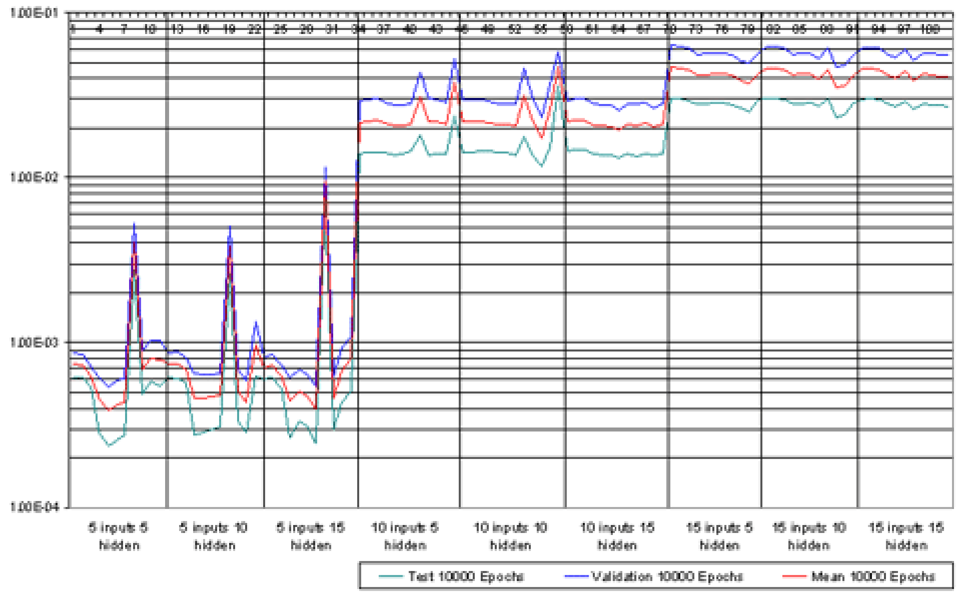


Figure 5.17: Log graph showing test and validation results together with the mean, for MLP networks trained with 10000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

Results for the 10000 epoch networks show a general increase in the number and magnitude of peaks. Very slight improvement can be seen in the best performing networks.

Moving on to the TDNN networks, figure 5.18 shows the combined results of the test and validation, together with their mean, for the TDNN networks trained with 1000 epochs.

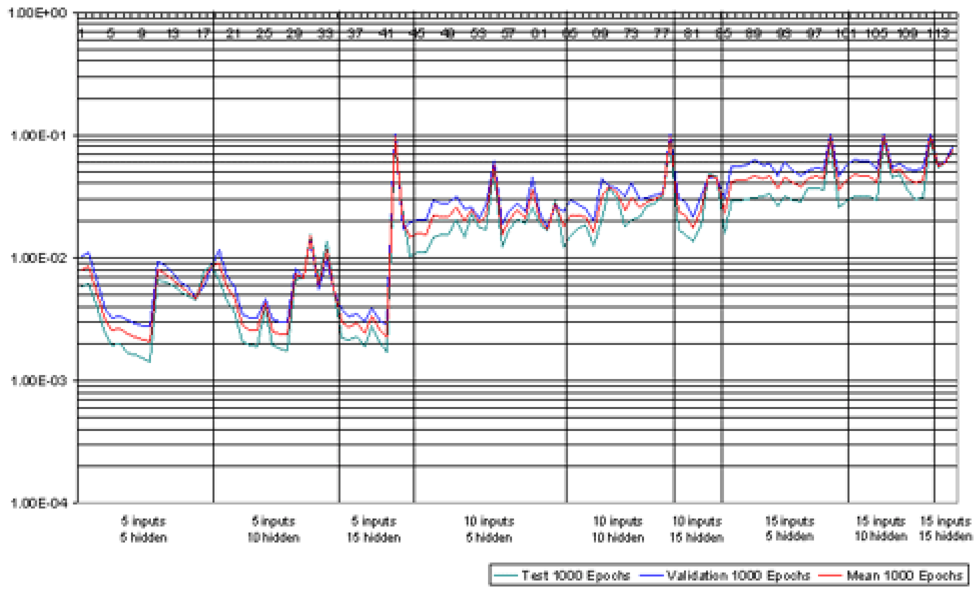


Figure 5.18: Log graph showing test and validation results together with the mean, for TDNN networks trained with 1000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

The first thing that can be noticed with these results compared to the preceding ones is that there isn't such a dramatic difference between the different numbers of inputs. 5 input networks show the best results, followed by 10 and then 15 inputs. One 5 input network appears to be standing out at this time, because of its poor performance when compared to other 5 input networks. This is the 5-15-2-0.2-0.1 network, and one of the few remaining 2 delay networks.

Figure 5.19 shows the combined results of the test and validation, together with their mean, for the TDNN networks trained with 3000 epochs.

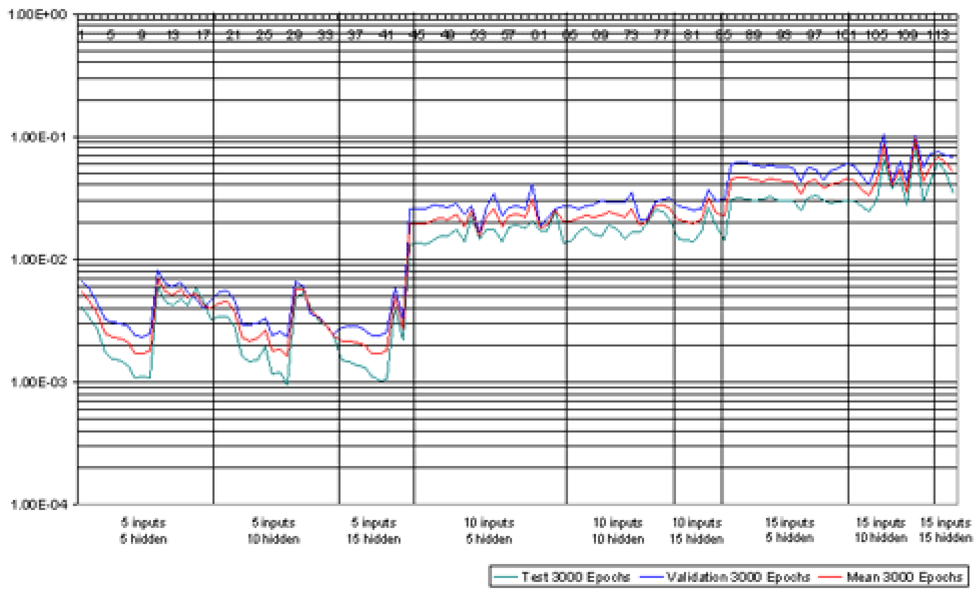


Figure 5.19: Log graph showing test and validation results together with the mean, for TDNN networks trained with 3000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

Results for the 3000 epoch trained networks show general improvement. This can be seen by the overall lowering and flattening of the lines.

Figure 5.20 shows the combined results of the test and validation, together with their mean, for the TDNN networks trained with 5000 epochs

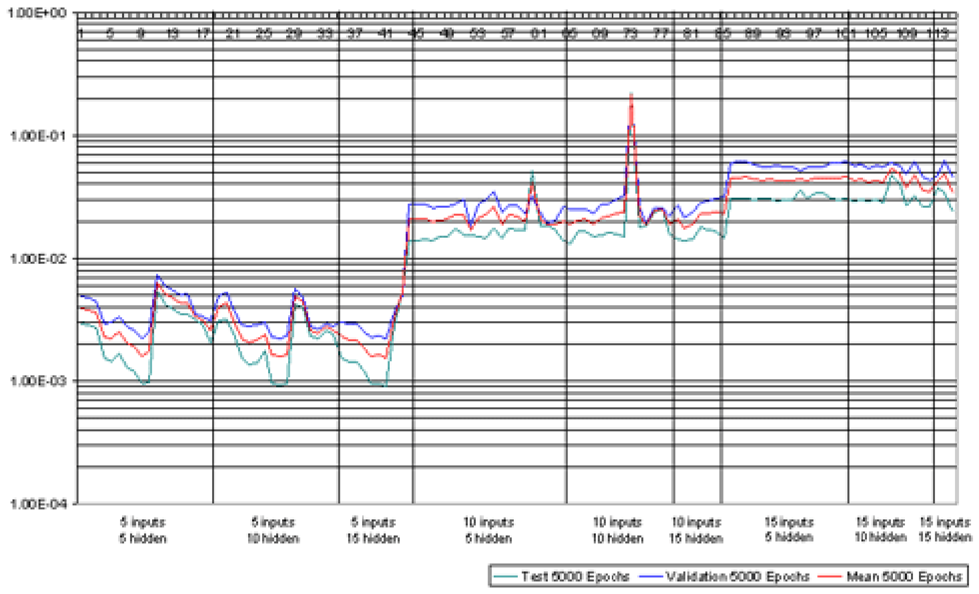


Figure 5.20: Log graph showing test and validation results together with the mean, for TDNN networks trained with 5000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

Continued improvement can be seen from the flattening of the lines. One network has performed quite badly though. This network had the 10-10-1-0.8-0.9 topology.

Figure 5.21 shows the combined results of the test and validation, together with their mean, for the TDNN networks trained with 7000 epochs

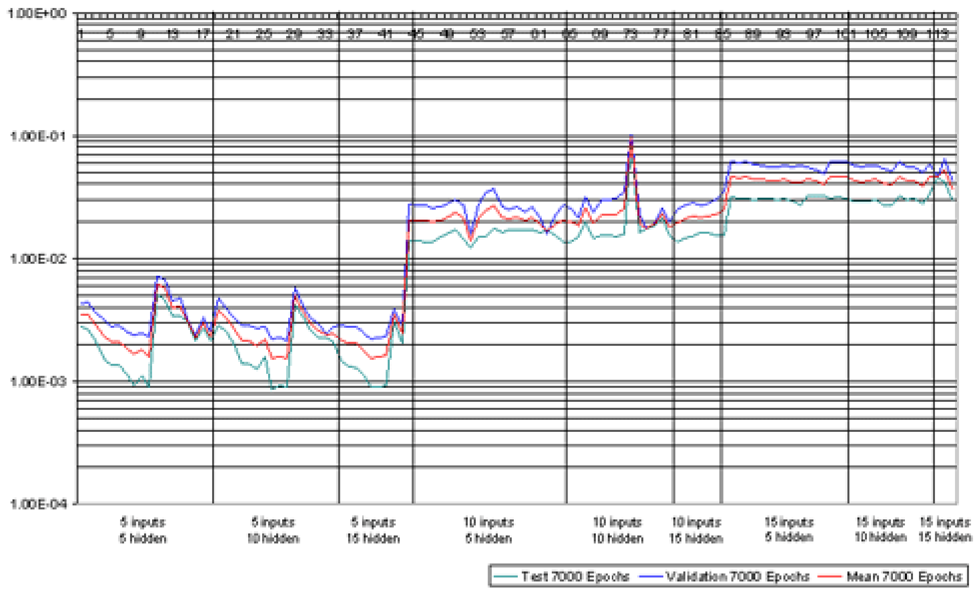


Figure 5.21: Log graph showing test and validation results together with the mean, for TDNN networks trained with 7000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

Slight improvement again can be seen throughout all networks, even the worst performing one. Quite an even performance can be seen throughout the 15 input networks.

Figure 5.22 shows the combined results of the test and validation, together with their mean, for the TDNN networks trained with 10000 epochs

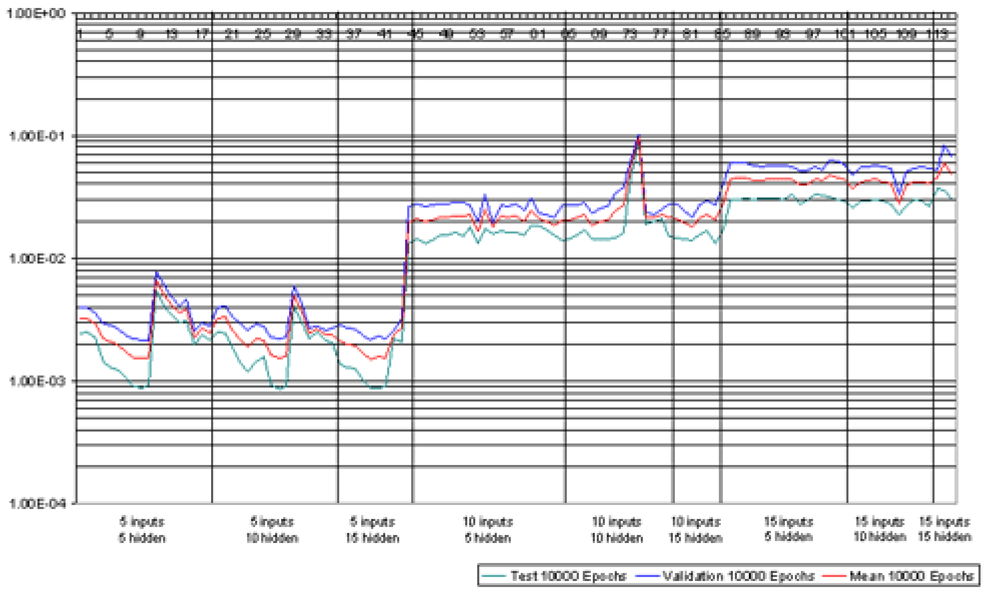


Figure 5.22: Log graph showing test and validation results together with the mean, for TDNN networks trained with 10000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

Very slight improvement can be seen in the 5 input networks. Greater improvement can be seen amongst the 10 and 15 input networks, with several troughs beginning to form.

Looking now at the RNN networks, figure 5.23 shows the combined results of the test and validation, together with their mean, for the RNN networks trained with 1000 epochs.

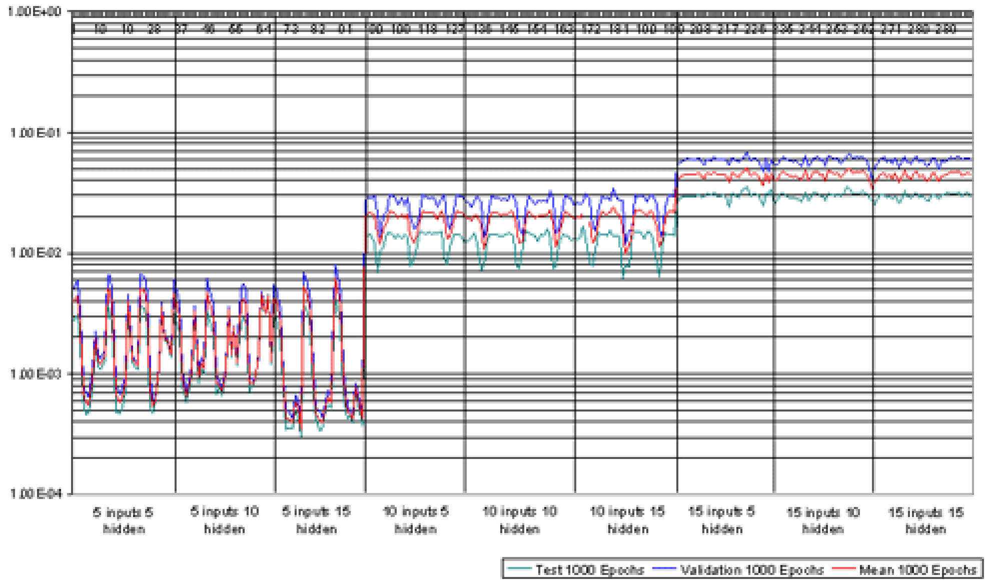


Figure 5.23: Log graph showing test and validation results together with the mean, for RNN networks trained with 1000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

First observations again show that the majority of networks have performed better with the test data than the validation data. Again, as with the MLP and TDNN, the 5 input networks have performed best, followed by the 10 and then the 15 input networks. Even performances can be witnessed amongst the best and worst performing networks. This is most evident with the 10 input networks, and to a lesser extent, the 5 input networks.

Figure 5.24 shows the combined results of the test and validation, together with their mean, for the RNN networks trained with 3000 epochs

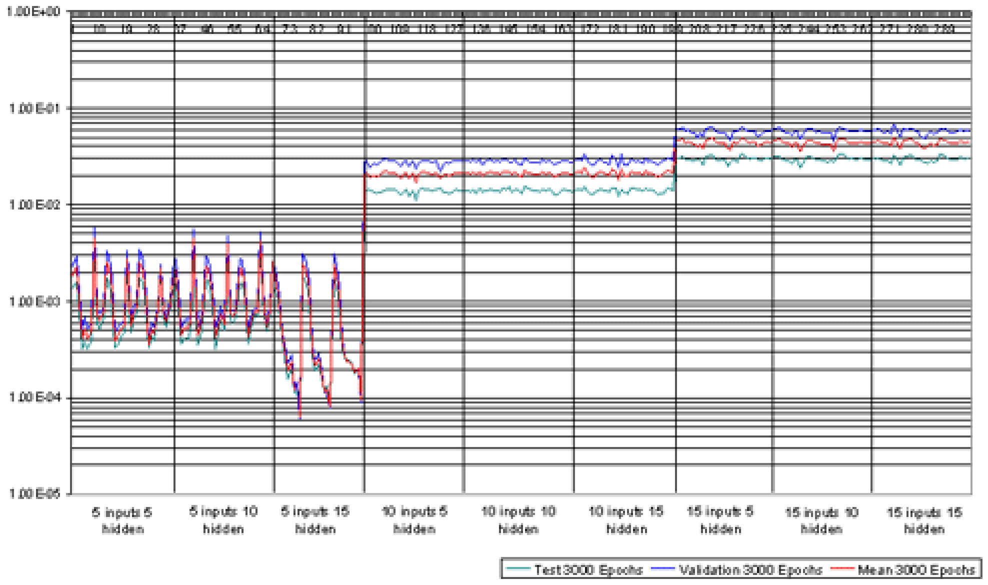


Figure 5.24: Log graph showing test and validation results together with the mean, for RNN networks trained with 3000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

Three deep troughs can be observed in the 5 input networks. These represent networks 5-15-0.1-0.8-0.5, 5-15-0.5-0.8-0.5 and 5-15-0.9-0.8-0.5 respectively. The remaining 5 input networks also show improvement. The results for the 10 input networks show a flattening out of the lines. In this case, this represents a general deterioration in results.

Figure 5.25 shows the combined results of the test and validation, together with their mean, for the RNN networks trained with 5000 epochs.

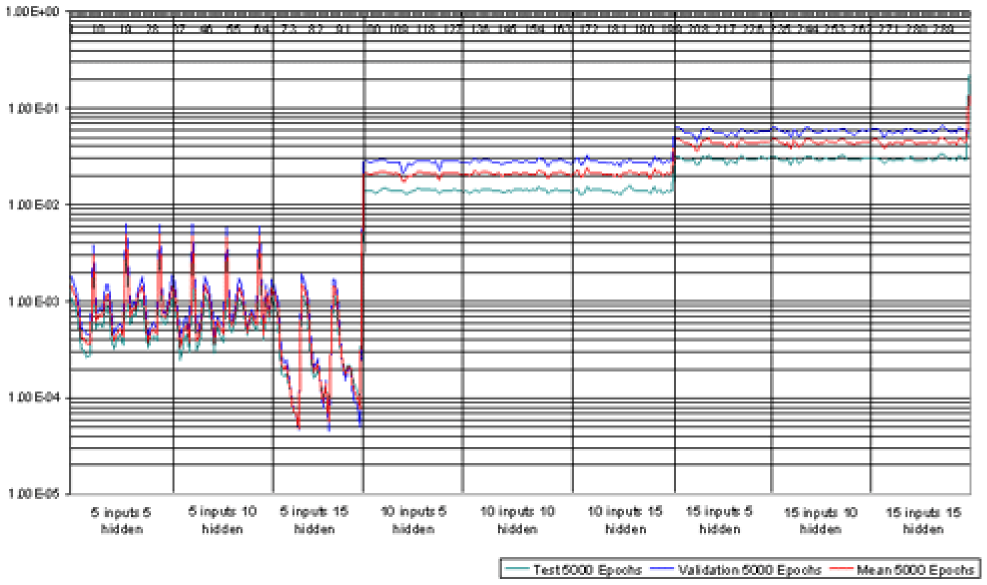


Figure 5.25: Log graph showing test and validation results together with the mean, for RNN networks trained with 5000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

Continued improvement can be seen in the best performing 5 input networks, but now deterioration in the worst performing 5 input networks has begun. The results for the 10 and 15 input networks are similar to the 3000 epoch results.

Figure 5.26 shows the combined results of the test and validation, together with their mean, for the RNN networks trained with 7000 epochs.

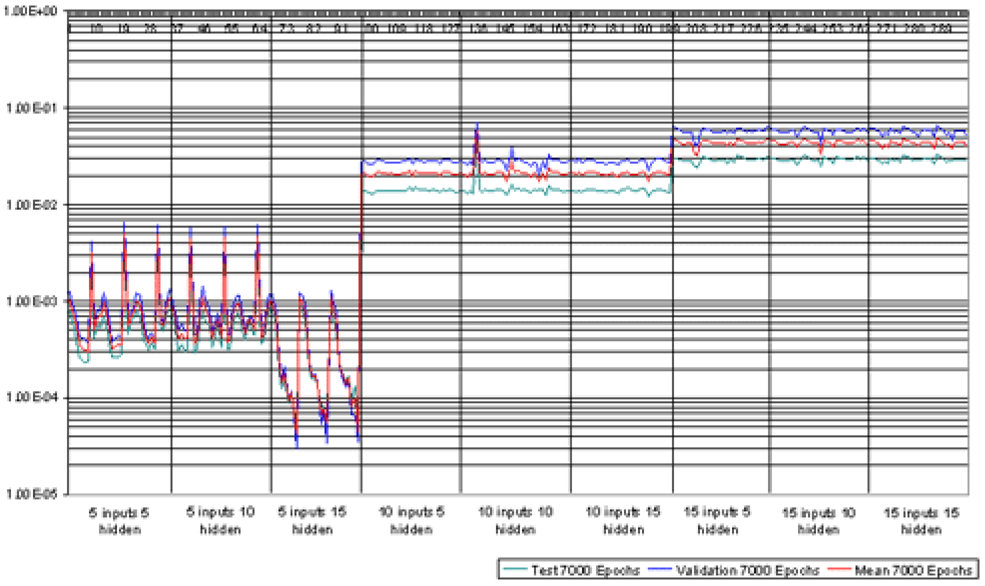


Figure 5.26: Log graph showing test and validation results together with the mean, for RNN networks trained with 7000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

Again, continued improvement can be seen in the best performing 5 input networks and continued deterioration can be seen in the worst performing 5 input networks. Results for the 10 and 15 input networks have remained constant.

Figure 5.27 shows the combined results of the test and validation, together with their mean, for the RNN networks trained with 10000 epochs.

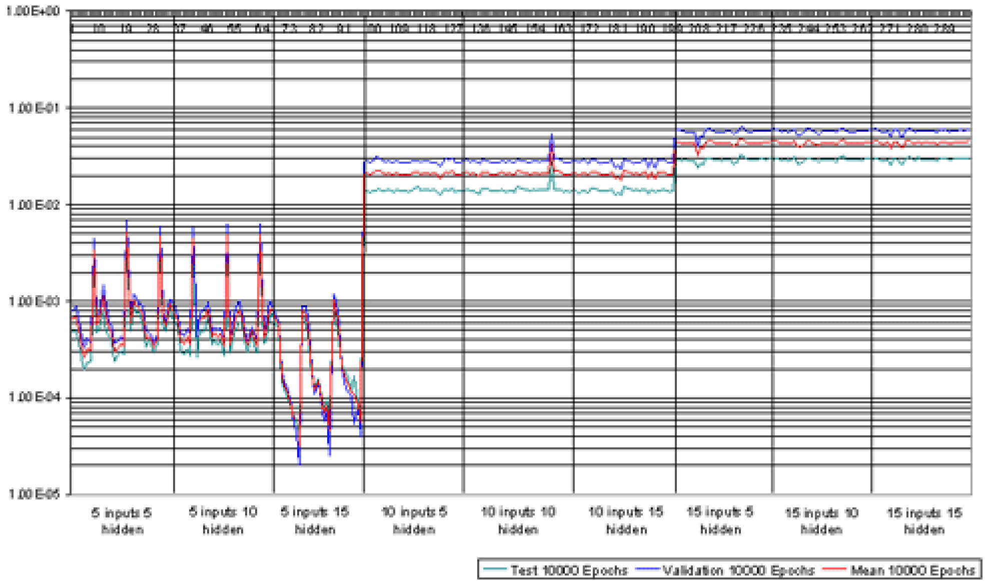


Figure 5.27: Log graph showing test and validation results together with the mean, for RNN networks trained with 10000 Epochs. Each individual network is represented by a point on each line. The scale across the top corresponds to the total number of networks. The scale across the bottom corresponds to the number of input and hidden units

And again, continued improvement can be seen in the best performing 5 input networks and continued deterioration can be seen in the worst performing 5 input networks. Again, results for the 10 and 15 input networks have remained constant.

Tables 5.1, 5.2 and 5.3 below shows the network topologies for the 10 lowest errors produced, for each of the MLP, TDNN and RNN networks respectively. This table is based on their mean error for the two stages.

Table 5.1: The 10 lowest error producing MLP network topologies

Input Neurons	Hidden Neurons	Learning Rate	Momentum	Training Epochs	Mean Error	Standard Deviation
5	5	0.2	0.1	7000	3.7702×10^{-04}	2.0410×10^{-04}
5	5	0.2	0	10000	3.8976×10^{-04}	2.1126×10^{-04}
5	10	0.2	0.5	10000	3.9954×10^{-04}	2.1623×10^{-04}
5	5	0.2	0	7000	4.2204×10^{-04}	2.2727×10^{-04}
5	5	0.2	0.1	10000	4.2345×10^{-04}	2.3550×10^{-04}
5	10	0.8	0	7000	4.3540×10^{-04}	2.2507×10^{-04}
5	10	0.2	0.5	5000	4.3662×10^{-04}	2.1735×10^{-04}
5	5	0.2	0.5	10000	4.4039×10^{-04}	2.2766×10^{-04}
5	10	0.2	0.1	10000	4.4094×10^{-04}	2.1690×10^{-04}
5	5	0.2	0	5000	4.4138×10^{-04}	2.2855×10^{-04}

Table 5.2: The 10 lowest error producing TDNN network topologies

Input Neurons	Hidden Neurons	Delays	Learning Rate	Momentum	Training Epochs	Mean Error	Standard Deviation
5	15	1	0.8	0	10000	1.5090×10^{-03}	8.9892×10^{-04}
5	5	1	0.8	0.1	10000	1.5242×10^{-03}	8.9611×10^{-04}
5	10	1	0.8	0.5	7000	1.5243×10^{-03}	8.6739×10^{-04}
5	10	1	0.8	0	7000	1.5288×10^{-03}	9.3511×10^{-04}
5	5	1	0.8	0.5	10000	1.5414×10^{-03}	8.5325×10^{-04}
5	10	1	0.8	0.1	10000	1.5443×10^{-03}	9.6858×10^{-04}
5	15	1	0.8	0.5	10000	1.5450×10^{-03}	9.1653×10^{-04}
5	15	1	0.8	0	7000	1.5468×10^{-03}	8.9529×10^{-04}
5	15	1	0.8	0.5	5000	1.5499×10^{-03}	8.9961×10^{-04}
5	5	1	0.8	0	10000	1.5506×10^{-03}	9.0834×10^{-04}

Table 5.3: The 10 lowest error producing RNN network topologies

Input Neurons	Hidden Neurons	Memory Depth	Learning Rate	Momentum	Training Epochs	Mean Error	Standard Deviation
5	15	0.1	0.8	0.5	10000	3.1274×10^{-05}	1.5598×10^{-05}
5	15	0.1	0.8	0.5	7000	4.3492×10^{-05}	1.9963×10^{-05}
5	15	0.9	0.8	0.5	7000	4.4383×10^{-05}	1.3158×10^{-05}
5	15	0.5	0.8	0.5	10000	4.6306×10^{-05}	2.9958×10^{-05}
5	15	0.1	0.8	0.5	5000	4.8895×10^{-05}	3.2985×10^{-06}
5	15	0.1	0.8	0.1	10000	5.0883×10^{-05}	9.6812×10^{-06}
5	15	0.5	0.8	0.5	7000	5.4801×10^{-05}	2.9280×10^{-05}
5	15	0.9	0.8	0.5	10000	5.5217×10^{-05}	2.0923×10^{-05}
5	15	0.5	0.8	0.5	5000	5.6708×10^{-05}	1.5012×10^{-05}
5	15	0.1	0.8	0	10000	5.8506×10^{-05}	6.8800×10^{-06}

5.4 Testing and Validation Findings

Probably the most significant indicator of networks ability to generalise is how low an error it produces when viewing unseen data. If this is taken as the basis for our analysis, then it can be seen from the above tables, that the RNNs have consistently produced lower mean errors than the MLPs and the TDNNs over the testing and validation sets, with the TDNNs unexpectedly producing the worst results of all.

Upon further analysis we can see the following for the MLP networks:

- The average error for the top 10 was 4.2065×10^{-4}
- The average standard deviation was 2.2098×10^{-4}
- all of the top 10 had 5 input neurons
- None had a learning rate 0.01 or a momentum of 0.9
- Network 5-5-0.2-0.1 appears at position 1 when trained with 7000 epochs and position 5 when trained with 10000 epochs. This would suggest that 10000 epochs causes this network to overfit the data.
- Network 5-5-0.2-0 appears at position 2 when trained with 10000 epochs, position 4 when trained with 7000 epochs and position 10 when trained with 5000 epochs. As this network has reached its lowest error with the maximum number of training epochs, it might improve with further training. It is interesting to note that the effect that momentum has had on the previous network which has allowing it to converge.
- Network 5-10-0.2-0.5 appears at position 3 when trained with 10000 epochs and position 7 when trained with 5000 epochs. As this network has also reached its lowest error with the maximum number of training epochs, it might improve with further training.

Figure 5.28 shows the performance of the top MLP network against the testing and validation data.

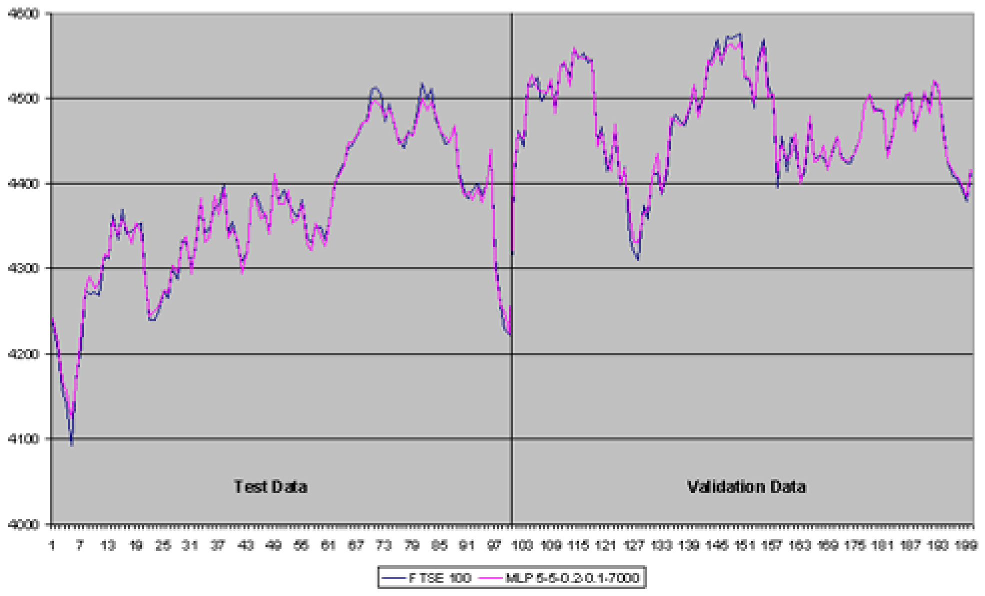


Figure 5.28: Performance of the top MLP network against the testing and validation data

This graph shows how close the 5-5-0.2-0.1 network was at predicting each individual values for both the test and validation data sets. It can be seen that while the network performed generally well against these data sets, slight fluctuations do appear.

Further analysis of the TDNN results reveal:

- The average error for the top 10 was 5.6968 x 10^{-3}
- The average standard deviation was 9.0391 x 10^{-4}
- All had 1 delay
- All had 5 inputs
- All had a learning rate of 0.8
- The network topology 5-15-1-0.8-0 appeared at position 1 when trained with 10000 epochs and 8 when trained with 7000 epochs. Having performed best at its maximum number of epochs, this network may perform better with further training.
- The network topology 5-15-1-0.8-0.5 appeared at position 7 when trained with 10000 epochs and 9 when trained with 7000 epochs. Having performed best at its maximum number of epochs, this network may perform better with further training.

Figure 5.29 shows the performance of the top TDNN network against the testing and validation data.

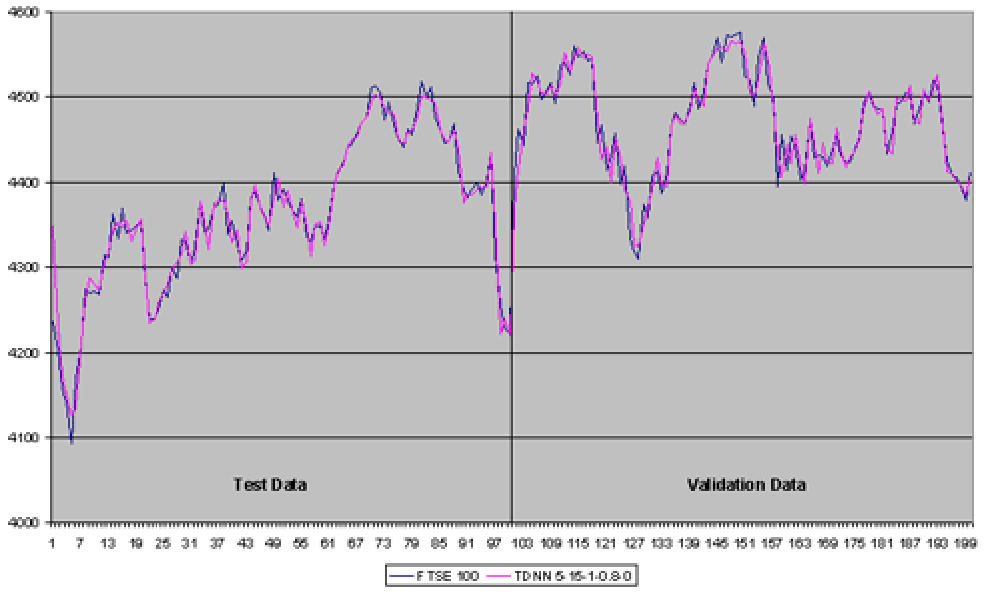


Figure 5.29: Performance of the top TDNN network against the testing and validation data

This graph shows how close the 5-15-1-0.8-0 network was at predicting each individual values for both the test and validation data sets. Similarly to the top MLP network, that while the network performed generally well against these data sets, slight fluctuations do appear.

Similar analysis of the RNN results highlight the following:

- The average error for the top 10 was 4.9047 x 10^{-5}
- The average standard deviation was 1.6375 x 10^{-5}
- All had 5 inputs
- All had 15 hidden units
- All had a learning rate of 0.8
- 8 had a momentum of 0.5
- The network topology 5-15-0.1-0.8-0.5 appeared at position 1 when trained with 10000 epochs, 2 when trained with 7000 epochs and 5 when trained with 5000 epochs. Having performed best at its maximum number of epochs, this network may perform better with further training.
- The network topology 5-15-0.9-0.8-0.1 appeared at position 3 when trained with 7000 epochs and 8 when trained with 10000 epochs. This would suggest that 10000 epochs overfit this network.
- The network topology 5-15-0.5-0.8-0.5 appeared at position 4 when trained with 10000 epochs, 7 when trained with 7000 epochs and 9 when trained with 5000 epochs. Having performed best at its maximum number of epochs, this network may perform better with further training.

Figure 5.30 shows the performance of the top RNN network against the testing and validation data.

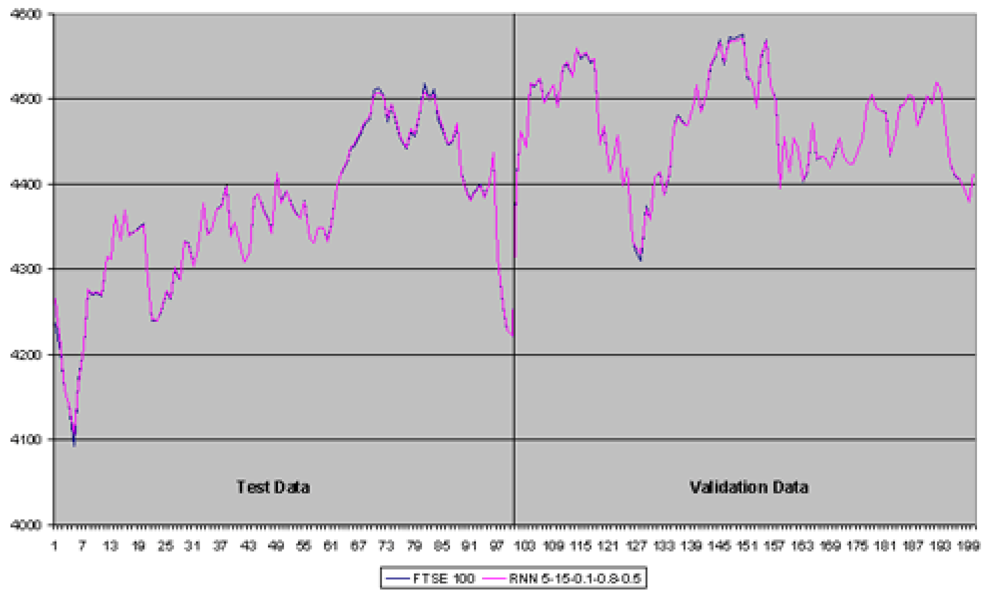


Figure 5.30: Performance of the top RNN network against the testing and validation data

This graph shows how close the 5-15-0.1-0.8-0.5 network was at predicting each individual values for both the test and validation data sets. It can be seen that the network has performed exceptionally well. The only slight discrepancies occur around the maximum and minimum values of each set. Please see Appendix E for a full listing of these results.

5.5 Forecasting

To further analyse the networks forecasting abilities, an additional experiment was conducted. This test was carried out only on the top network of each type, and each network was initially trained to its optimal level as identified earlier in this research. A comparison was then made with a forecast produced by an ARIMA model (see Appendix D). As a reminder, the network configurations where:

- MLP – 5-5-0.2-0.1-7000
- TDNN – 5-15-1-0.8-0-10000
- RNN – 5-15-0.1-0.8-0.5-10000

The forecasting data comprised of FTSE 100 closing price values dating from 05/07/2004 to 09/07/2004, which was held back from the validation data set.

The results of this test can be seen in table 5.4 below

Table 5.4: Forecasting ability comparison between top MLP, TDNN, RNN and ARIMA model

Actual value	MLP	TDNN	RNN	ARIMA(1, 1, 1)
4403.2998	4400.305	4401.8	4401.988	4383.90
4370.7002	4371.15	4371.201	4369.427	4372.02
4358.3999	4364.788	4361.637	4360.94	4361.06
4381.1001	4381.45	4382.397	4380.948	4375.08

Figure 5.31 below shows a plot of these results.

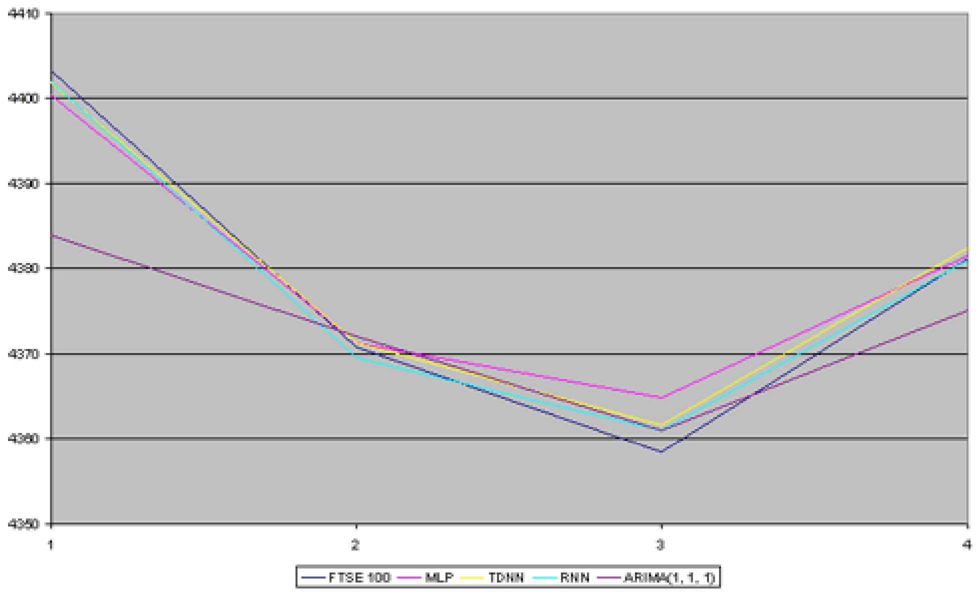


Figure 5.31: Forecasting ability comparison between top MLP, TDNN, RNN and ARIMA model

In this instance the ARIMA model has performed fairly badly. This is probably due to a poor selection of coefficients.

It does not come as any surprise that the RNN network has again come out as the top network. The RNN networks have consistently produced better results than the other two types of networks. These results are comparable to the results obtained by Kim *(Kim, 1998)* who in addition to the TDRNN, also compared RNN and TDNN networks. Kim also achieved a better performance from the RNNs than the TDNNs.

What is more of a surprise, with respect to the previous empirical results, documented in this report, is that the TDNN network has outperformed the MLP network on this occasion. This would make this result more in line with accepted theory.

Chapter 6 – Conclusions

6.1 Summary of Research

The purpose of this project was to test and compare the ability of three different types of neural networks at forecasting the end of day closing price of the Financial Times Stock Exchange 100 Index.

The networks chosen for this experiment where the standard Multilayer Perceptron, The Time Delay Neural Network and a version of the Recurrent Neural Network based on the Elman Network.

All networks used the back propagation learning algorithm, or in the case of the TDNN networks, a slight variation of it. All networks also used the Sigmoid transfer function with a fixed slope parameter of 1. Various different network topologies where examined. These where made up of either 5, 10 or 15 input units and 5, 10 or 15 hidden units. Several different values where used for the network parameters. These included 0.01, 0.2 and 0.8 for the learning rate and 0, 0.1, 0.5 and 0.9 for the momentum. Additionally, TDNN's were tested with 1 and 2 delays and RNN's were tested with memory depth of 0.1, 0.5 and 0.9. Data used in the experiments consisted of end of day closing price values of the FTSE. This was divided up into 3 sets, known as the training, testing and validation sets. By the very nature of the data used, values were fed to the networks using a sliding window.

The first stage in examining the networks, was training. This involved repeatedly showing the networks the training data, adjusting them and then recording how well they learnt. Networks that showed little ability to learn or learnt erratically were then eliminated.

The second stage was the testing stage. This concerned showing the networks new data only once and recording how well they predicted it. Various amounts of training were given to the networks to find the optimal amount.

The third stage was validation. This involved showing the networks that had been trained to their optimal point, another new set of data and recording their prediction. The networks that performed best over the testing and validation stages were then selected.

A final forecasting test was then undertaken on the best networks with data that had been held back from the validation set. The results of this test were then compared with a forecast made using an ARIMA model.

6.2 Conclusions

There are several conclusions that can be drawn from this research. Firstly, the performance of the best RNN network was far superior to that of the MLP and TDNN network. Over the test and validation stages the MLP performed better than the TDNN network, however on the final forecast test this was reversed. This would be more in line with proven theory.

The final mean error achieved by the top networks over the forecast data set was 3.0132 x 10^{-4} for the RNN, 3.7312 x 10^{-4} for the TDNN and 5.8141 x 10^{-4} for the MLP. This can be compared to the error obtained from the ARIMA model which was 1.6787 x 10^{-3}.

As a general rule, the combination of a high learning rate and a high momentum was a very bad one. Networks that used these as parameters tended to either get stuck in local minima or produced erratic results.

The number of neurons used also played an important fact in determining performance. Networks that had a high number of input neurons achieve exceptionally bad results. The desired number of hidden units varies for MLP and TDNN networks, but the best results obtained from RNN networks all used the largest amount of units.

While training is an important stage for assessing the ability of networks to learning, the true test of a networks generalisation ability can only be discovered during the testing and validation stages.

The optimal number of training epochs should be used to obtain the best results, but when testing and analysing a great many networks, there will also be many optimal values.

MLP networks produced the best results when they had a medium learning rate and a small amount of momentum.

TDNN networks produced best results when they had a high learning rate and a small to medium amount of momentum. The overriding factor that affected TDNN results was the number of delays. Any more than one delay appeared to prevent the networks from achieving any learning.

RNN networks produced best results when they had a high learning rate and a medium amount of momentum.

The high learning rate desired by TDNN and RNN networks coupled with the fact that the best results were obtained when the networks were trained with the maximum amount of training epoch suggests that they may have benefited from additional training. Unfortunately because of time limitation it was not possible to test this theory.

The overall success of this research has to be related to the ability of the best performing networks to forecast values from unseen data sets. From this perspective it would appear that this research has been successful, with the very best network not varying more than 16 points from the actual value for 200 different values.

6.3 Further Research

Probably the biggest guiding factor in the key decisions taken, relating to the course this research took, was the length of time required to train, test and validate all the different networks. As a result of this time limitation, it was decided not to test the effect that different size data sets would have on the results. Furthermore a cap was set on the maximum number of training epochs that was used. It is proposed therefore that further research could examine these important areas.

This research concentrated on the ability of networks to forecast closing price values for the FTSE 100 based solely on previous historical values. As discussed in earlier chapters, a great many factors are involves in making a price what it is. It is also proposed that some of these additional influences could also be tested as network inputs.

References

Beale R, Jackson T, *Neural Computing an introduction*, Institute of Physics Publishing, 2001

Box G, Jenkins G, and Reinsel G, *Time Series Analysis Forecasting and Control*, 3rd Ed, Prentice Hall, 1994

Chatfield C, *The Analysis of Time Series - An Introduction*, 6th Ed, Chapman & Hall/CRC, 2004

Chenoweth T, Obradovic Z, *A multi-component nonlinear prediction system for the S & P 500 Index,* Neurocomputing 10, 1996, pp275-290

Christodoulou C, *MSc Computing Science Course Notes*, Birkbeck College University of London, 2004

Elman JL, *Finding Structure in Time, Cognitive Science*, vol 14, 1990, pp 179-211

Haykin S, *Neural Networks A Comprehensive Foundation,* 2nd Ed, Prentice-Hall, 1999

Hertz J, Krogh A, Palmer RG, *Introduction to the Theory of Neural Computation*, Addison Wesley, 1991, pp 180-181

Kaastra I, Boyd M, Designing a neural network for forecasting financial and economic time series, Neurocomputing 10, 1996, pp 215-236

Kahn M, *Technical Analysis Plain & Simple*, Pearson Education Limited, 1999

Kaufman P, *Trading Systems and Methods*, 3rd Ed, Wiley Trading Advantage, 1998

Kim SS, *Time-delay recurrent neural network for temporal correlations and prediction*, Neurocomputing 20, 1998, 253-263

Lang KJ, Waibel AH, Hinton GE, *A Time_Delay Neural Network Architecture for Isolated Word Recognition, Neural Networks*, Vol 3 pp23-43, 1990

McCulloch WS and Pitts W, *A logical calculus of the ideas imminent in nervous activity. Bulletin of Mathematical Biophysics*, 5:115-133, 1943

MINITAB, *Time Series tutorial*, www.minitab.com, 2004

Minsky M & Papert S, Perceptrons MIT Press, 1969

NIST/SEMATECH *e-Handbook of Statistical Methods*, http://www.itl.nist.gov/div898/handbook/, date

Rumelhart D, Hinton G, Williams R, *Learning Representations by Back-Propagating Errors*, Nature, Vol 323, 1986, pp533-536

Sarle W, *comp.ai.neural-nets FAQ*, ftp://ftp.sas.com/pub/neural/FAQ.html, 2004

Schalkoff R, *Artificial Neural Networks*, McGraw-Hill, 1997

Shazly M, Shazly H, *Forecasting currency prices using a genetically evolved neural network architecture,* International Review of Financial Analysis 8:1, 1999, pp 67-82

Tveter D, *The Backprop Algorithm*, www.dontveter.com, 2001

Valdez S, *An Introduction to Global Financial Markets*, 4th Ed, Palgrave Macmillan, 2003

Werbos P, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences: PhD Thesis*, Harvard University, 1974

Appendices

Appendix A – Source Code

BiasNeuron.java

```
/**
 * The BiasNeuron class creates Bias Neurons which have a fixed value of -1
 * @author Raymond McBride
 */
public class BiasNeuron extends Neuron{

   /**
    * This constructor for the <code>BiasNeuron</code>initialises its input value to -1
    */
   public BiasNeuron(){
      super();
      setInputValue(-1.0);
   }

   /**
    * Calculates the output value
    */
   public void calculateOutput(){
      setOutputValue(getInputValue());
   }
}
```

ContextNeuron.java

```
/**
 * The ContextNeuron class creates Context Neuron which are used to provide recurrency
 * in the RNN network. They store the previous Hidden Neuron activated values and a variable
 * amount of their own previous values, which is fed back into the network
 * @see RNN
 * @author Raymond McBride
 */
public class ContextNeuron extends Neuron{

  private double memoryDepth;
  private double memoryContents;

  /**
   * This constructor for the <code>ContextNeuron</code> specifies the memory depth
   * and sets the input value to 0.5
   * @param memoryDepth The memory depth
   */
  public ContextNeuron(double memoryDepth){
    super();
    this.memoryDepth = memoryDepth;
    setInputValue(0.5);
    memoryContents = 0;
  }

  /**
   * Set the new input value and adds the old value to the memory
   * @param input The new input value
   */
  public void input(double input){
    memoryContents = getInputValue();
    setInputValue(input);
  }

  /**
   * Calculates the output value
   */
  public void calculateOutput(){
    setOutputValue(getInputValue() + memoryContents*memoryDepth);
  }

}
```

ContextSynapse.java

```java
/**
 * The ContextSynapse class is used to connect Context Neurons to Hidden Neurons. They
 * have a fixed weight of 1
 * @author Raymond McBride
 */
public class ContextSynapse extends Synapse{

   /**
    * This constructor for the <code>ContextSynapse</code> connects two Neurons together,
    * and sets it's weight to 1
    * @param inputNeuron The input Neuron
    * @param outputNeuron The output Neuron
    */
   public ContextSynapse(Neuron inputNeuron, Neuron outputNeuron){
      super(inputNeuron, outputNeuron);
      setWeight(1);
   }

   /**
    * Transfers a weighted value from the input Neuron to the output Neuron
    */
   public void transferValue(){
      super.transferValue();
   }
}
```

DataProcessor.java

```
/**
 * The DataProcessor class is used to scale the input data between 0 and 1
 * @author Raymond McBride
 */
public class DataProcessor{

   private double[] inputs;
   private double minimum;
   private double maximum;

   /**
    * This constructor for the <code>DataProcessor</code> sets its inputs
    * @param inputs An array of input values
    */
   public DataProcessor(double[] inputs){
      this.inputs = inputs;
   }

   /**
    * Scales its inputs array between 0 and 1
    * @return a double array of its scaled inputs
    */
   public double[] scale(){
      double[] temp = new double [inputs.length];
      minimum = inputs[0];
      maximum = inputs[0];
      for(int i = 0; i < inputs.length; i++){
         if (inputs[i] < minimum)
            minimum = inputs[i];
         if (inputs[i] > maximum)
            maximum = inputs[i];
      }
      for(int i = 0; i < inputs.length; i++)
         temp[i] = (inputs[i] - minimum) / (maximum - minimum);
      return temp;
   }
}
```

HiddenNeuron.java

```
/**
 * The HiddenNeuron class is used to create Hidden Neurons
 * @author Raymond McBride
 */
public class HiddenNeuron extends Neuron{

private double[] timeDelay;

    /**
     * This constructor for the <code>HiddenNeuron</code> sets the slope of its activation function
     * and the number of it's inputs
     * @param slope The slope of the activation function
     * @param numberOfInputs The number of inputs
     */
    public HiddenNeuron(int slope, int numberOfInputs){
        super(slope, numberOfInputs);
    }

    /**
     * This constructor for the <code>HiddenNeuron</code> sets the slope of its activation function,
     * the number of it's inputs and the number of delays
     * @param slope The slope of the activation function
     * @param numberOfInputs The number of inputs
     * @param delays The number of delays
     */
    public HiddenNeuron(int slope, int numberOfInputs, int delays){
        super(slope, numberOfInputs);
        timeDelay = new double[delays];
    }

    /**
     * Calculates the summation of the delayed output values
     */
    public void calculateDelayedOutput(){
        super.calculateOutput();
        for(int i = timeDelay.length - 1; i > 0; i--)
            timeDelay[i] = timeDelay[i-1];
        timeDelay[0] = getOutputValue();
        double tempOutput = 0;
        for(int i = 0; i < timeDelay.length; i++){
            tempOutput += timeDelay[i];
        }
        setOutputValue(tempOutput);
    }

    /**
     * gets the delayed output values
     * @return a double array of delayed output values
     */
    public double[] getOutputs(){
        return timeDelay;
    }
}
```

InputNeuron.java

```
/**
 * InputNeuron class is used to create Input Neurons
 * @author Raymond McBride
 */
public class InputNeuron extends Neuron{

   private double[] timeDelay;

   /**
    * This constructor for the <code>InputNeuron</code> sets the slope of its activation function
    * @param slope The slope of the activation function
    */
   public InputNeuron(int slope){
      super(slope);
   }

   /**
    * This constructor for the <code>InputNeuron</code> sets the slope of its activation function
    * and the number of delays
    * @param slope The slope of the activation function
    * @param delays The number of delays
    */
   public InputNeuron(int slope, int delays){
      super(slope);
      timeDelay = new double[delays];
   }

   /**
    * Sets the new input Value
    * @param input The new input value
    */
   public void input(double input){
      setInputValue(input);
   }

   /**
    * Calculates the output value
    */
   public void calculateOutput(){
      setOutputValue(getInputValue());
   }

   /**
    * Calculates the summation of the delayed output values
    */
   public void calculateDelayedOutput(){
      for(int i = timeDelay.length - 1; i > 0; i--)
         timeDelay[i] = timeDelay[i-1];
      timeDelay[0] = getInputValue();
      double tempOutput = 0;
      for(int i = 0; i < timeDelay.length; i++){
         tempOutput += timeDelay[i];
      }
      setOutputValue(tempOutput);
   }
```

```
    /**
     * gets the delayed output values
     * @return a double array of delayed output values
     */
    public double[] getOutputs(){
        return timeDelay;
    }
}
```

MLP.java

```
/**
 * The MLP class is used to create Multilayer Perceptron networks
 * @author Raymond McBride
 */
public class MLP extends Network{

   /**
    * This constructor for the <code>MLP</code> specifies the number of Input Neurons and Hidden
Neurons
    * the slope of their activation functions, the learning rate, the momentum, the number of training
    * epochs and set a network topology id for use with log files.
    * @param inputs The number of Input Neurons
    * @param hiddens The number of Hidden Neurons
    * @param slope The slope of their activation functions
    * @param learningRate The learning rate
    * @param momentum The momentum
    * @param totalEpochs The number of training epochs
    * @param fileID The network topology id
    */
   public MLP(int inputs, int hiddens, int slope, double learningRate, double momentum, int totalEpochs,
String fileID){
      super(inputs, hiddens, slope, learningRate, momentum, totalEpochs, fileID);
      createNeurons();
       connectNeurons();
   }

   /**
    * Initialises the network with data
    */
   protected void initialise(){
      for(int i = 0; i < getInputNeurons().length; i++){
         getInputNeuron(i).input(getInputData((getNextInput() + i +
getInputData().length)%getInputData().length));
         getInputNeuron(i).calculateOutput();
      }
      getBiasHidden().calculateOutput();
      getBiasOutput().calculateOutput();
      setTargetOutput(getInputData((getNextInput() + getInputNeurons().length -
1)%getInputData().length));
   }
}
```

Network.java

```
import java.text.*;
import java.util.*;
import java.io.*;

/**
 * The Network class is the abstract base class for all networks
 * @see MLP
 * @see TDNN
 * @see RNN
 * @author Raymond McBride
 */
abstract class Network{

    private Neuron[] inputNeurons;
    private Neuron[] hiddenNeurons;
    private Neuron outputNeuron;
    private double learningRate;
    private double momentum;
    private int epoch;
    private int slope;
    private double targetOutput;
    private double outputErrorTerm;
    private double[] hiddenErrorTerm;
    private double[] inputData;
    private BiasNeuron biasHidden;
    private BiasNeuron biasOutput;
    private Synapse[][] inputToHidden;
    private Synapse[] hiddenToOutput;
    private Synapse[] biasToHidden;
    private Synapse biasToOutput;
    private int nextInput;
    private double totalNetworkError;
    private OutputFile detailFile;
    private int totalEpochs;
    private String fileID;

    /**
     * This constructor for the <code>MLP</code> specifies the number of Input Neurons and Hidden
     * Neurons
     * the slope of their activation functions, the learning rate, the momentum, the number of training
     * epochs and set a network topology id for use with the log file.
     * @param inputs The number of Input Neurons
     * @param hiddens The number of Hidden Neurons
     * @param slope The slope of their activation functions
     * @param learningRate The learning rate
     * @param momentum The momentum
     * @param totalEpochs The number of training epochs
     * @param fileID The network topology id
     */
    public Network(int inputs, int hiddens, int slope, double learningRate, double momentum, int
totalEpochs, String fileID){
         inputNeurons = new Neuron[inputs];
        hiddenNeurons = new Neuron[hiddens];
        inputToHidden = new Synapse[inputs][hiddens];
        hiddenToOutput = new Synapse[hiddens];
        hiddenErrorTerm = new double[hiddens];
        biasToHidden = new Synapse[hiddens];
        this.slope = slope;
        this.learningRate = learningRate;
        this.momentum = momentum;
         nextInput = 0;
         totalNetworkError = 0;
         this.totalEpochs = totalEpochs;
         this.fileID = fileID;
    }

    /**
     * Creates the Neurons
     */
    protected void createNeurons(){
        for(int i = 0; i < inputNeurons.length; i++)
            inputNeurons[i] = new InputNeuron(slope);
        for(int i = 0; i < hiddenNeurons.length; i++)
            hiddenNeurons[i] = new HiddenNeuron(slope, inputNeurons.length);
        outputNeuron = new OutputNeuron(slope, hiddenNeurons.length);
        biasHidden = new BiasNeuron();
        biasOutput = new BiasNeuron();
    }

    /**
     * Connects the Neurons
     */
    protected void connectNeurons(){
        for(int i = 0; i < inputNeurons.length; i++){
            for(int j = 0; j < hiddenNeurons.length; j++){
                inputToHidden[i][j] = new Synapse(inputNeurons[i], hiddenNeurons[j]);
            }
        }
        for(int i = 0; i < hiddenNeurons.length; i++){
            hiddenToOutput[i] = new Synapse(hiddenNeurons[i], outputNeuron);
            biasToHidden[i] = new Synapse(biasHidden, hiddenNeurons[i]);
        }
        biasToOutput = new Synapse(biasOutput, outputNeuron);
    }

    /**
     * Abstract method to initialise the network
     */
    protected abstract void initialise();

    /**
     * Calculates the output error term
     */
    protected void calculateOutputError(){
        outputErrorTerm =
slope*outputNeuron.getOutputValue()*(1-outputNeuron.getOutputValue())*(targetOutput -
outputNeuron.getOutputValue());
    }

    /**
     * Calculates the hidden error term
     */
    protected void calculateHiddenError(){
        for(int i = 0; i < hiddenNeurons.length; i++){
```

```
        hiddenErrorTerm[i] = slope * hiddenNeurons[i].getOutputValue() * (1 -
hiddenNeurons[i].getOutputValue()) * outputErrorTerm * hiddenToOutput[i].getWeight();
      }
    }

    /**
     * Calculates the weight change to be made to the Synapses between the Output and Hidden Neurons
     * and Bias Neurons
     */
    protected void calculateOutputWeightChange(){
      for (int i = 0; i < hiddenToOutput.length; i++){
        hiddenToOutput[i].calculateWeightChange(learningRate, momentum, outputErrorTerm);
      }
      biasToOutput.calculateWeightChange(learningRate, momentum, outputErrorTerm);
    }

    /**
     * Adjusts the weights of the Synapses between the Output and Hidden Neurons and Bias Neurons
     */
    protected void adjustOutputWeights(){
      for (int i = 0; i < hiddenToOutput.length; i++){
        hiddenToOutput[i].adjustWeight();
      }
      biasToOutput.adjustWeight();
    }

    /**
     * Calculates the weight change to be made to the Synapses between the Hidden and Input Neurons
     * and Bias Neurons
     */
    protected void calculateHiddenWeightChange(){
      for (int i = 0; i < inputToHidden.length; i++){
        for(int j = 0; j < inputToHidden[i].length; j++){
          inputToHidden[i][j].calculateWeightChange(learningRate, momentum, hiddenErrorTerm[j]);
        }
      }
      for (int i = 0; i < biasToHidden.length; i++){
      biasToHidden[i].calculateWeightChange(learningRate, momentum, hiddenErrorTerm[i]);
      }
    }

    /**
     * Adjusts the weights of the Synapses between the Hidden and Input Neurons and Bias Neurons
     */
    protected void adjustHiddenWeights(){
      for (int i = 0; i < inputToHidden.length; i++){
        for(int j = 0; j < inputToHidden[i].length; j++){
          inputToHidden[i][j].adjustWeight();
        }
      }
      for (int i = 0; i < biasToHidden.length; i++){
        biasToHidden[i].adjustWeight();
      }
    }

    /**
     * Transfers the weighted values to the HiddenNeurons
     */
```

```
protected void sendToHidden(){
   for(int i = 0; i < inputToHidden.length; i++){
       for(int j = 0; j < inputToHidden[i].length; j++)
           inputToHidden[i][j].transferValue();
   }
  for (int i = 0; i < biasToHidden.length; i++)
       biasToHidden[i].transferValue();
}

/**
 * Calculates the output of the HiddenNeurons
 */
protected void calculateHiddenOutput(){
   for(int i = 0; i < hiddenNeurons.length; i++){
       hiddenNeurons[i].calculateOutput();
   }
}

/**
 * Calculates the output error
 */
protected double calculateError(){
   double error = targetOutput - outputNeuron.getOutputValue();
   return 0.5 * error * error;
}

/**
 * Transfers the weighted values to the OutputNeuron
 */
protected void sendToOutput(){
   for(int i = 0; i < hiddenToOutput.length; i++)
      hiddenToOutput[i].transferValue();
   biasToOutput.transferValue();
}

/**
 * Trains the network
 */
public void train(double[] data){
   inputData = data;
   epoch = 0;
   double error;
   while((epoch < totalEpochs)){
      initialise();
      sendToHidden();
      calculateHiddenOutput();
      sendToOutput();
      outputNeuron.calculateOutput();
      totalNetworkError += calculateError();
      calculateOutputError();
      calculateOutputWeightChange();
      calculateHiddenError();
      calculateHiddenWeightChange();
      adjustOutputWeights();
      adjustHiddenWeights();
      if (nextInput >= (inputData.length - 1)){
         nextInput = 0;
         totalNetworkError = 0;
          epoch++;
```

```
            }
            else nextInput++;
        }
    }

    /**
     * Tests the network with the required test type
     */
    private void beginTest(double[] data, String testType){
        detailFile = new OutputFile("../output/"+ fileID + testType + "_" +"details.csv");
        inputData = data;
        totalEpochs = 1;
        nextInput = 0;
        while(nextInput <= (inputData.length - 1)){
            initialise();
            sendToHidden();
            calculateHiddenOutput();
            sendToOutput();
            outputNeuron.calculateOutput();
            totalNetworkError = calculateError();
            detailFile.writeToFile("Error" + "," + totalNetworkError);
            nextInput++;
        }
        detailFile.closeFile();
    }

    /**
     * Starts testing the network
     */
     public void test(double[] data){
        beginTest(data, "T");
    }

    /**
     * Starts validating the network
     */
    public void validate(double[] data){
        beginTest(data, "V");
    }

    /**
     * Gets the slope
     * @return slope
     */
    protected int getSlope(){
        return slope;
    }

    /**
     * Gets the learning rate
     * @return learningRate
     */
    protected double getLearningRate(){
        return learningRate;
    }

    /**
     * Gets the momentum
```

```
 * @return momentum
 */
protected double getMomentum(){
   return momentum;
}

/**
 * Sets the target output
 * @param targetOutput the target output
 */
protected void setTargetOutput(double targetOutput){
   this.targetOutput = targetOutput;
}

/**
 * Sets the OutputNeuron
 * @param outputNeuron the OutputNeuron
 */
protected void setOutputNeuron(OutputNeuron outputNeuron){
   this.outputNeuron = outputNeuron;
}

/**
 * Gets the OutputErrorTerm
 * @return outputErrorTerm
 */
protected double getOutputErrorTerm(){
   return outputErrorTerm;
}

/**
 * Gets a hiddenErrorTerm
 * @param i its position
 * @return hiddenErrorTerm an array of error terms
 */
protected double getHiddenErrorTerm(int i){
   return hiddenErrorTerm[i];
}

/**
 * Sets a hiddenErrorTerm
 * @param errorTerm the new error term
 * @param i its position
 */
protected void setHiddenErrorTerm(double errorTerm, int i){
   hiddenErrorTerm[i] = errorTerm;
}

/**
 * Gets the next input position
 * @return nextInput
 */
protected int getNextInput(){
   return nextInput;
}


/**
```

```
 * Gets the Synapse connecting the BiasNeuron to the OutputNeuron
 * @return biasToOutput
 */
protected Synapse getBiasToOutput(){
   return biasToOutput;
}

/**
 * Sets the BiasNeuron to the HiddenNeurons
 * @param biasHidden the BiasNeuron to the HiddenNeurons
 */
protected void setBiasHidden(BiasNeuron biasHidden){
   this.biasHidden = biasHidden;
}

/**
 * Gets the BiasNeuron to the HiddenNeurons
 * @return biasHidden the BiasNeuron to the HiddenNeurons
 */
protected BiasNeuron getBiasHidden(){
   return biasHidden;
}

/**
 * Sets the BiasNeuron to the OutputNeuron
 * @param biasOutput the BiasNeuron to the OutputNeuron
 */
protected void setBiasOutput(BiasNeuron biasOutput){
   this.biasOutput = biasOutput;
}

/**
 * Gets the BiasNeuron to the OutputNeuron
 * @return biasOutput the BiasNeuron to the OutputNeuron
 */
protected BiasNeuron getBiasOutput(){
   return biasOutput;
}

/**
 * Gets the Synapses between the BiasNeuron and the HiddenNeurons
 * @return biasToHidden the array of Synapses
 */
protected Synapse[] getBiasToHidden(){
   return biasToHidden;
}

/**
 * Gets a Synapse between the BiasNeuron and a HiddenNeuron
 * @param i its position
 * @return the Synapse at that position
 */
protected Synapse getBiasToHidden(int i){
   return biasToHidden[i];
}


/**
```

```
 * Gets a Synapse between a HiddenNeuron and the OutputNeuron
 * @param i its position
 * @return the Synapse at the position
 */
protected Synapse getHiddenToOutput(int i){
   return hiddenToOutput[i];
}

/**
 * Gets the Synapses between the HiddenNeurons and the OutputNeuron
 * @return hiddenToOutput the array of Synapses
 */
protected Synapse[] getHiddenToOutput(){
   return hiddenToOutput;
}

/**
 * Gets all the Synapses between the InputNeurons and the HiddenNeurons
 * @return inputToHidden the 2D array of Synapses
 */
protected Synapse[][] getInputToHidden(){
   return inputToHidden;
}

/**
 * Gets the Synapses between an InputNeuron and the HiddenNeurons
 * @param i its position
 * @return the array of Synapses at that position
 */
protected Synapse[] getInputToHidden(int i){
   return inputToHidden[i];
}

/**
 * Gets the Synapse between an InputNeuron and a HiddenNeuron
 * @param i its x position
 * @param j its y position
 * @return the Synapse at that position
 */
protected Synapse getInputToHidden(int i, int j){
  return inputToHidden[i][j];
}

/**
 * Gets an InputNeuron
 * @param i its position
 * @return the InputNeuron
 */
protected Neuron getInputNeuron(int i){
   return inputNeurons[i];
}

/**
 * Gets the InputNeurons
 * @return inputNeurons the array of InputNeurons
 */
protected Neuron[] getInputNeurons(){
   return inputNeurons;
```

```
    }

    /**
     * Sets an InputNeuron
     * @param inputNeuron the new InputNeuron
     * @param i its position
     */
    protected void setInputNeuron(InputNeuron inputNeuron, int i){
        inputNeurons[i] = inputNeuron;
    }

    /**
     * Gets a HiddenNeuron
     * @param i its position
     * @return the HiddenNeuron
     */
    protected Neuron getHiddenNeuron(int i){
        return hiddenNeurons[i];
    }

    /**
     * Gets the HiddenNeurons
     * @return hiddenNeurons the array of HiddenNeurons
     */
    protected Neuron[] getHiddenNeurons(){
        return hiddenNeurons;
    }

    /**
     * Sets a HiddenNeuron
     * @param hiddenNeuron the new HiddenNeuron
     * @param i its position
     */
    protected void setHiddenNeuron(HiddenNeuron hiddenNeuron, int i){
        hiddenNeurons[i] = hiddenNeuron;
    }

    /**
     * Gets input data at a given position
     * @param i its position
     * @return the double at that position
     */
    protected double getInputData(int i){
        return inputData[i];
    }

    /**
     * Gets the input data
     * @return inputData
     */
    protected double[] getInputData(){
        return inputData;
    }
}
```

Neuron.java

```
/**
 * The Neuron class is the abstract base class for all Neurons
 * @see InputNeuron
 * @see HiddenNeuron
 * @see OutputNeuron
 * @see BiasNeuron
 * @see ContextNeuron
 * @author Raymond McBride
 */
abstract class Neuron{

   private double inputValue;
   private double outputValue;
   private double[] inputValues;
   private double bias;
   private int slope;
   private int nextFree;

   /**
    * This constructor for the <code>Neuron</code> sets the slope of its activation function to 1
    */
   public Neuron(){
      this(1);
   }

   /**
    * This constructor for the <code>Neuron</code> sets the slope of its activation function
    * @param slope The slope of the activation function
    */
   public Neuron(int slope){
      this.slope = slope;
   }

   /**
    * This constructor for the <code>Neuron</code> sets the slope of its activation function
    * and the number of it's inputs
    * @param slope The slope of the activation function
    * @param numberOfInputs The number of inputs
    */
   public Neuron(int slope, int numberOfInputs){
       this(slope);
       inputValues = new double[numberOfInputs];
       bias = 0;
       nextFree = 0;
   }

   /**
    * Adds a new input value
    * @param input The new input value
    */
   public void input(double input){
      inputValues[nextFree] = input;
      nextFree = (nextFree + 1)%inputValues.length;
   }
```

```
/**
 * Sets the biased input
 * @param input The new input value
 */
public void setBias(double input){
    bias = input;
}

/**
 * Calculates the output
 */
public void calculateOutput(){
   outputValue = sigmoidActivation(this.summation());
}

/**
 * Calculates the summation of the delayed output values
 */
public void calculateDelayedOutput(){}

/**
 * Calculates the summation of the inputs
 */
private double summation(){
    double temp = 0;
    for(int i = 0; i < inputValues.length; i++)
        temp += inputValues[i];
    return temp + bias;

}

/**
 * The Sigmoid Activation Function
 * @return the activated value
 */
protected double sigmoidActivation(double summation){
    return 1/(1 + Math.exp(-slope*summation));
}

/**
 * Gets the input value
 * @return the input value
 */
protected double getInputValue(){
   return inputValue;
}

/**
 * Sets the input value
 * @param inputValue the input value
 */
protected void setInputValue(double inputValue){
   this.inputValue = inputValue;
}

/**
 * Gets the output value
 * @return the output value
```

```
     */
    protected double getOutputValue(){
        return outputValue;
    }

    /**
     * Sets the output value
     * @param outputValue the output value
     */
    protected void setOutputValue(double outputValue){
        this.outputValue = outputValue;
    }
}
```

OutputFile.java

```
import java.io.*;
import java.util.*;

/**
 * The OutputFile class creates log files
 * @author Raymond McBride
 */
public class OutputFile{

   private PrintWriter printWriter;

   /**
    * Constructor for the <code>OutputFile</code>
    * @throws IOException e
    */
   public OutputFile(String fileName){
      try{
         printWriter = new PrintWriter(new BufferedWriter(new FileWriter(fileName)));
      }
      catch(IOException e){
         System.out.println(e.toString());
      }
   }

   /**
    * Writes data to the file
    * @param data The data to be written
    */
   public void writeToFile(String data){
      printWriter.println(data);
   }

   /**
    * Closes the file
    */
   public void closeFile(){
      printWriter.close();
   }
}
```

OutputNeuron.java

```
/**
 * The OutputNeuron class creates Output Neurons
 * @author Raymond McBride
 */
public class OutputNeuron extends Neuron{

    /**
     * This constructor for the <code>OutputNeuron</code> sets the slope of its activation function
     * and the number of it's inputs
     * @param slope The slope of the activation function
     * @param numberOfInputs The number of inputs
     */
    public OutputNeuron(int slope, int numberOfInputs){
        super(slope, numberOfInputs);
    }
}
```

RNN.java

```
/**
 * The RNN class is based on the Elman Network, which has recurrency at the Hidden Layer.
 * In addition, there is a self loop at the context layer
 * @author Raymond McBride
 */
public class RNN extends Network{

   private Neuron[] contextNeurons;
   private ContextSynapse[] contextToHidden;
   private ContextSynapse[] hiddenToContext;

   /**
    * This constructor for the <code>RNN</code> specifies the number of Input Neurons and Hidden
    * Neurons,
    * the memory depth, the slope of their activation functions, the learning rate, the momentum,
    * the number of training, epochs and set a network topology id for use with log files.
    * @param inputs The number of Input Neurons
    * @param hiddens The number of Hidden Neurons
    * @param memoryDepth The memory depth
    * @param slope The slope of their activation functions
    * @param learningRate The learning rate
    * @param momentum The momentum
    * @param totalEpochs The number of training epochs
    * @param fileID The network topology id
    */
   public RNN(int inputs, int hiddens, double memoryDepth, int slope, double learningRate, double
momentum, int totalEpochs, String fileID){
      super(inputs, hiddens, slope, learningRate, momentum, totalEpochs, fileID);
      contextNeurons = new Neuron[hiddens];
      contextToHidden = new ContextSynapse[hiddens];
      hiddenToContext = new ContextSynapse[hiddens];
      createNeurons(memoryDepth);
       connectNeurons();
   }

   /**
    * Creates the Neurons
    * @param memoryDepth Their memory depth
    */
   protected void createNeurons(double memoryDepth){
      super.createNeurons();
      for(int i = 0; i < contextNeurons.length; i++)
         contextNeurons[i] = new ContextNeuron(memoryDepth);
   }

   /**
    * Connects the Neurons
    */
   protected void connectNeurons(){
      super.connectNeurons();
      for(int i = 0; i < contextNeurons.length; i++){
         contextToHidden[i] = new ContextSynapse(contextNeurons[i], getHiddenNeuron(i));
         hiddenToContext[i] = new ContextSynapse(getHiddenNeuron(i), contextNeurons[i]);
      }
   }
   /**
    * Transfers the weighted values to the Hidden Neurons
    */
   protected void sendToHidden(){
      super.sendToHidden();
      for(int i = 0; i < contextToHidden.length; i++)
         contextToHidden[i].transferValue();
   }

   /**
    * Transfers the weighted values to the Output Neuron
    */
   protected void sendToOutput(){
      super.sendToOutput();
      for(int i = 0; i < hiddenToContext.length; i++)
         hiddenToContext[i].transferValue();
   }

   /**
    * Initialises the network with data
    */
   protected void initialise(){
      for(int i = 0; i < getInputNeurons().length; i++){
         getInputNeuron(i).input(getInputData((getNextInput() + i +
getInputData().length)%getInputData().length));
         getInputNeuron(i).calculateOutput();
      }
      getBiasHidden().calculateOutput();
      getBiasOutput().calculateOutput();
      for(int i = 0; i < contextNeurons.length; i++){
         contextNeurons[i].calculateOutput();
      }
      setTargetOutput(getInputData((getNextInput() + getInputNeurons().length -
1)%getInputData().length));
   }
}
```

Synapse.java

```
/**
 * The Synapse class is used to connect Neurons together. Synapses are created with
 * adjustable random weights.
 * @see ContextSynapse
 * @author Raymond McBride
 */
public class Synapse{


    private double weight;
    private Neuron inputNeuron;
    private Neuron outputNeuron;
    private double weightChange;


    /**
     * This constructor for the <code>Synapse</code> connects two Neurons together,
     * and sets it's weight to a random value
     * @param inputNeuron The input <code>Neuron</code>
     * @param outputNeuron The output <code>Neuron</code>
     */
    public Synapse(Neuron inputNeuron, Neuron outputNeuron){
        weightChange = 0;
        weight = Math.random();
        this.inputNeuron = inputNeuron;
        this.outputNeuron = outputNeuron;
    }

    /**
     * Gets the Synapse weight
     * @return The weight
     */
    public double getWeight(){
       return weight;
    }

    /**
     * Sets the Synapse weight
     * @param newWeight The new weight
     */
    protected void setWeight(int newWeight){
       weight = newWeight;
    }

    /**
     * Calculates the weight change
     */
    public void calculateWeightChange(double learningRate, double momentum, double errorTerm){
       weightChange = (learningRate * errorTerm * inputNeuron.getOutputValue()) + (momentum *
weightChange);
    }

    /**
     * Calculates the average weight change for use with time delays
     */
    public void calculateWeightChange(double learningRate, double momentum, double errorTerm, int
delays){
       double[] delayedOutputs;
```

```
        if(inputNeuron instanceof InputNeuron){
            delayedOutputs = ((InputNeuron)inputNeuron).getOutputs();
        }
        else{
            delayedOutputs = ((HiddenNeuron)inputNeuron).getOutputs();
        }
        double previousWeight = weightChange;
        weightChange = 0;
        for (int i = 0; i < delays; i++)
            weightChange += (learningRate * errorTerm * delayedOutputs[i]) + (momentum *
previousWeight);
        weightChange = weightChange/delays;
    }

    /**
     * Adjusts the weight
     */
    public void adjustWeight(){
        weight = weight + weightChange;
    }

    /**
     * Transfers a weighted value from the input Neuron to the output Neuron
     */
    public void transferValue(){
         if(inputNeuron instanceof BiasNeuron){
            outputNeuron.setBias(inputNeuron.getOutputValue() * weight);
        }
         else{
            outputNeuron.input(inputNeuron.getOutputValue() * weight);
         }
    }
}
```

TDNN.java

```
/**
 * The TDNN class is used to create Time Delay Neural Networks
 * @author Raymond McBride
 */
public class TDNN extends Network{

  private int delays;

  /**
   * This constructor for the <code>TDNN</code> specifies the number of Input Neurons and Hidden
Neurons,
   * the number of delays, the slope of their activation functions, the learning rate, the momentum,
   * the number of training, epochs and set a network topology id for use with log files.
   * @param inputs The number of Input Neurons
   * @param hiddens The number of Hidden Neurons
   * @param delays The number of delays
   * @param slope The slope of their activation functions
   * @param learningRate The learning rate
   * @param momentum The momentum
   * @param totalEpochs The number of training epochs
   * @param fileID The network topology id
   */
  public TDNN(int inputs, int hiddens, int delays, int slope, double learningRate, double momentum, int
totalEpochs, String fileID){
    super(inputs, hiddens, slope, learningRate, momentum, totalEpochs, fileID);
    this.delays = delays;
    createNeurons(delays);
     connectNeurons();
  }

  /**
   * Creates the Neurons
   * @param delays The number of delays
   */
  protected void createNeurons(int delays){
    for(int i = 0; i < getInputNeurons().length; i++)
      setInputNeuron((new InputNeuron(getSlope(), delays)), i);
    for(int i = 0; i < getHiddenNeurons().length; i++)
      setHiddenNeuron((new HiddenNeuron(getSlope(), getInputNeurons().length, delays)), i);
    setOutputNeuron(new OutputNeuron(getSlope(), getHiddenNeurons().length));
    setBiasHidden(new BiasNeuron());
    setBiasOutput(new BiasNeuron());
  }

  /**
   * Calculates the output of the Hidden Neurons
   */
  protected void calculateHiddenOutput(){
    for(int i = 0; i < getHiddenNeurons().length; i++){
      getHiddenNeuron(i).calculateDelayedOutput();
    }
  }

  /**
   * Calculates the weight change to be made to the Synapses between the Output and Hidden Neurons
   * and Bias Neurons
```

```
     */
    protected void calculateOutputWeightChange(){
       for (int i = 0; i < (getHiddenToOutput()).length; i++){
          getHiddenToOutput(i).calculateWeightChange(getLearningRate(), getMomentum(),
getOutputErrorTerm(), delays);
       }
       getBiasToOutput().calculateWeightChange(getLearningRate(), getMomentum(),
getOutputErrorTerm());
    }

    /**
     * Calculates the weight change to be made to the Synapses between the Hidden and Input Neurons
     * and Bias Neurons
     */
    protected void calculateHiddenWeightChange(){
       for (int i = 0; i < (getInputToHidden()).length; i++){
          for(int j = 0; j < (getInputToHidden(i)).length; j++){
             getInputToHidden(i, j).calculateWeightChange(getLearningRate(), getMomentum(),
getHiddenErrorTerm(j), delays);
          }
       }
       for (int i = 0; i < (getBiasToHidden()).length; i++){
          getBiasToHidden(i).calculateWeightChange(getLearningRate(), getMomentum(),
getHiddenErrorTerm(i));
       }
    }

    /**
     * Calculates the hidden error term
     */
    protected void calculateHiddenError(){
       for(int i = 0; i < getHiddenNeurons().length; i++){
          double errorTerm = 0;
          double[] temp = ((HiddenNeuron)getHiddenNeuron(i)).getOutputs();
          for(int j = 0; j < delays; j++)
             errorTerm += getSlope() * temp[j] * (1 - temp[j]) * getOutputErrorTerm() *
(getHiddenToOutput(i)).getWeight();
          setHiddenErrorTerm(errorTerm/delays, i);
       }
    }

    /**
     * Initialises the network with data
     */
    protected void initialise(){
       int n = 0;
       for(int i = 0; i < getInputNeurons().length; i++){
          getInputNeuron(i).input(getInputData((getNextInput() + i +
getInputData().length)%getInputData().length));
          getInputNeuron(i).calculateDelayedOutput();
       }
       getBiasHidden().calculateOutput();
       getBiasOutput().calculateOutput();
       setTargetOutput(getInputData((getNextInput() + getInputNeurons().length -
1)%getInputData().length));
    }
}
```

Test.java

```
import java.util.*;
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

/**
 * The Test class is the main class. It creates, trains, tests and validates MLP, TDNN and
 * RNN networks with the specified parameters
 * @author Raymond McBride
 */
public class Test{

    private final static double[] MOMENTUM = {0.0, 0.1, 0.5, 0.9};
   private final static double[] LEARNING_RATE = {0.01, 0.2, 0.8};
   private final static double[] MEMORY_DEPTH = {0.1, 0.5, 0.9};
   private final static int[] DELAYS = {1, 2, 3};
   private final static int[] EPOCHS = {1000, 3000, 5000, 7000, 10000};
   private final static int[] INPUTS = {5, 10, 15};
    private final static int[] HIDDENS = {5, 10, 15};
    private double[] trainingData;
    private double[] testingData;
    private double[] validatingData;
    private Thread mlpThread;
   private Thread tdnnThread;
   private Thread rnnThread;

    /**
     * This constructor for the <code>Test</code> creates training, testing and validating data sets
     * @param trainPath The location of the training data
     * @param trainField The XML node tag containing the training data
     * @param testingPath The location of the testing
     * @param testingField The XML node tag containing the testing data
     * @param validatingPath The location of the validating data
     * @param validatingField The XML node tag containing the validating data
     */
   public Test(String trainPath, String trainField, String testingPath, String testingField, String
validatingPath, String validatingField){
      trainingData = getData(trainPath, trainField);
      testingData = getData(testingPath, testingField);
      validatingData = getData(validatingPath, validatingField);
   }

   /**
     * Gets the data from the <code>Document</code>
     * @return a double array containing the data
     */
   public double[] getData(String path, String field){
      XMLParser parser = new XMLParser(path);
      Document xmlDoc = parser.getDocument();
      NodeList elementNodes = xmlDoc.getElementsByTagName(field);
      ArrayList arrayList = new ArrayList();
      for (int i = 0; i < elementNodes.getLength(); i++) {
         NodeList textNodes = elementNodes.item(i).getChildNodes();
         for (int j = 0; j < textNodes.getLength(); j++) {
            Node node = textNodes.item(j);
            arrayList.add(node.getNodeValue());
            }
        }
        double[] data = new double[arrayList.size()];
        for(int i = 0; i < arrayList.size(); i++)
            data[i] = Double.parseDouble((String)arrayList.get(i));
        DataProcessor dataProcessor = new DataProcessor(data);
         data = dataProcessor.scale();
         return data;
    }

    /**
     * Trains, tests and validates the MLP networks
     */
    public void testMLP(){
        mlpThread = new Thread(){
            public void run(){
                for(int i = 0; i < INPUTS.length; i++){
                    for(int j = 0; j < HIDDENS.length; j++){
                        for(int k = 0; k < LEARNING_RATE.length; k++){
                            for(int m = 0; m < MOMENTUM.length; m++){
                                for(int n = 0; n < EPOCHS.length; n++){
                                    MLP mlp = new MLP(INPUTS[i], HIDDENS[j], 1, LEARNING_RATE[k],
MOMENTUM[m], EPOCHS[n], "MLP_" + i + "_" + j + "_" + k + "_" + m + "_" + n + "_");
                                    mlp.train(trainingData);
                                    mlp.test(testingData);
                                    mlp.validate(validatingData);
                                }
                            }
                        }
                    }
                }
            }
        };
        mlpThread.start();
    }

    /**
     * Trains, tests and validates the TDNN networks
     */
    public void testTDNN(){
        tdnnThread = new Thread(){
            public void run(){
                for(int i = 0; i < INPUTS.length; i++){
                    for(int j = 0; j < HIDDENS.length; j++){
                        for(int k = 0; k < DELAYS.length; k++){
                            for(int m = 0; m < LEARNING_RATE.length; m++){
                                for(int n = 0; n < MOMENTUM.length; n++){
                                    for(int p = 0; p < EPOCHS.length; p++){
                                        TDNN tdnn = new TDNN(INPUTS[i], HIDDENS[j], DELAYS[k], 1,
LEARNING_RATE[m], MOMENTUM[n], EPOCHS[p], "TDNN_" + i + "_" + j + "_" + k + "_" + m + "_" + n + "_"
+ p + "_");
                                        tdnn.train(trainingData);
                                        tdnn.test(testingData);
                                        tdnn.validate(validatingData);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        };
        tdnnThread.start();
    }

    /**
     * Trains, tests and validates the RNN networks
     */
    public void testRNN(){
        rnnThread = new Thread(){
            public void run(){
                for(int i = 0; i < INPUTS.length; i++){
                    for(int j = 0; j < HIDDENS.length; j++){
                        for(int k = 0; k < MEMORY_DEPTH.length; k++){
                            for(int m = 0; m < LEARNING_RATE.length; m++){
                                for(int n = 0; n < MOMENTUM.length; n++){
                                    for(int p = 0; p < EPOCHS.length; p++){
                                        RNN rnn = new RNN(INPUTS[i], HIDDENS[j], MEMORY_DEPTH[k], 1,
LEARNING_RATE[m], MOMENTUM[n], EPOCHS[p], "RNN_" + i + "_" + j + "_" + k + "_" + m + "_" + n + "_" +
p + "_");
                                        rnn.train(trainingData);
                                        rnn.test(testingData);
                                        rnn.validate(validatingData);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        };
        rnnThread.start();
    }

    public static void main(String[] args){
        Test test = new Test("../data/Train500.xml", "indexValue", "../data/Test100.xml", "indexValue",
"../data/Validate100.xml", "indexValue");
        test.testMLP();
        test.testTDNN();
        test.testRNN();
    }
}
```

XMLParser.java

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.*;

/**
 * The XMLParser class is used to parse XML documents and create W3C DOM objects
 * @author Raymond McBride
 */
public class XMLParser{

   private DocumentBuilderFactory factory;
   private DocumentBuilder documentBuilder;
   private Document document;

   /**
    * This constructor for the <code>XMLParser</code>builds a new <code>Document</code> from
    * an XML file
    * @param location The location of the XML file
    */
   public XMLParser(String location){

      try{
         factory = DocumentBuilderFactory.newInstance();
         documentBuilder = factory.newDocumentBuilder();
         document = documentBuilder.parse(new File(location));
      }
      catch(Exception e){
         System.out.println(e.toString());
      }
   }

   /**
    * Gets the Document
    * @return a Document
    */
   public Document getDocument(){
      return document;
   }
}
```

Appendix B – Project Schema and XSL Stylesheet

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="project">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element ref="ftse"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="ftse">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="date" minOccurs="0" type="xsd:timeInstant"/>
        <xsd:element name="indexValue" minOccurs="0" type="xsd:double"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
 </xsd:schema>
```

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <title>Raymond McBride - MSc Computing Science Project 2004</title>
      <body>
        <h2 align="center">Financial Time Stock Exchange 100 Index</h2>
        <p align="center">
        <table border="1" cellpadding="5">
          <tr>
            <th>Date</th>
            <th>Index Value</th>
          </tr>
          <xsl:for-each select="project/ftse">
            <tr>
              <td><xsl:value-of select="date" /></td>
              <td><xsl:value-of select="indexValue" /></td>
            </tr>
          </xsl:for-each>
        </table>
        </p>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Appendix C – Networks that failed the training stage

Table C.1: MLP network topologies that failed the training stage

Input Neurons	Hidden Neurons	Learning Rate	Momentum
5	5	0.8	0.9
5	10	0.8	0.9
5	15	0.8	0.9
15	5	0.8	0.9
15	10	0.8	0.9
15	15	0.8	0.9

Table C.2: TDNN network (5 inputs) topologies that failed the training stage

Input Neurons	Hidden Neurons	Delays	Learning Rate	Momentum
5	5	1	0.2	0.8
5	5	1	0.8	0.9
5	5	2	0.01	0
5	5	2	0.01	0.1
5	5	2	0.2	0.9
5	5	2	0.8	0.9
5	10	1	0.2	0.9
5	10	1	0.8	0.9
5	10	2	0.01	0
5	10	2	0.01	0.1
5	10	2	0.01	0.5
5	10	2	0.2	0
5	10	2	0.2	0.1
5	10	2	0.8	0.9
5	15	1	0.01	0.1
5	15	1	0.2	0.9
5	15	1	0.8	0.9
5	15	2	0.01	0
5	15	2	0.01	0.1
5	15	2	0.01	0.5
5	15	2	0.01	0.9
5	15	2	0.2	0
5	15	2	0.2	0.1
5	15	2	0.2	0.5
5	15	2	0.2	0.9
5	15	2	0.8	0
5	15	2	0.8	0.1
5	15	2	0.8	0.9

Table C.3: TDNN network (10 inputs) topologies that failed the training stage

Input Neurons	Hidden Neurons	Delays	Learning Rate	Momentum
10	5	2	0.01	0
10	5	2	0.01	0.1
10	5	2	0.01	0.5
10	5	2	0.8	0.9
10	10	1	0.8	0.9
10	10	2	0.01	0
10	10	2	0.01	0.1
10	10	2	0.01	0.5
10	10	2	0.01	0.9
10	10	2	0.2	0.1
10	10	2	0.2	0.5
10	10	2	0.2	0.9
10	10	2	0.8	0
10	10	2	0.8	0.9
10	15	1	0.01	0
10	15	1	0.01	0.1
10	15	1	0.01	0.5
10	15	1	0.2	0.1
10	15	1	0.2	0.9
10	15	1	0.8	0.9
10	15	2	0.01	0
10	15	2	0.01	0.1
10	15	2	0.01	0.5
10	15	2	0.01	0.9
10	15	2	0.2	0
10	15	2	0.2	0.1
10	15	2	0.2	0.5
10	15	2	0.2	0.9
10	15	2	0.8	0
10	15	2	0.8	0.1
10	15	2	0.8	0.5
10	15	2	0.8	0.9

Table C.4: TDNN network (15 inputs) topologies that failed the training stage

Input Neurons	Hidden Neurons	Delays	Learning Rate	Momentum
15	5	1	0.2	0.9
15	5	1	0.8	0.9
15	5	2	0.01	0
15	5	2	0.01	0.1
15	5	2	0.2	0.9
15	5	2	0.8	0.5
15	5	2	0.8	0.9
15	10	1	0.2	0.9
15	10	1	0.8	0.9
15	10	2	0.01	0
15	10	2	0.01	0.1
15	10	2	0.01	0.5
15	10	2	0.01	0.9
15	10	2	0.2	0
15	10	2	0.2	0.5
15	10	2	0.2	0.9
15	10	2	0.8	0
15	10	2	0.8	0.1
15	10	2	0.8	0.5
15	10	2	0.8	0.9
15	15	1	0.01	0
15	15	1	0.01	0.1
15	15	1	0.01	0.5
15	15	1	0.01	0.9
15	15	1	0.2	0.1
15	15	1	0.2	0.5
15	15	1	0.2	0.9
15	15	1	0.8	0.1
15	15	1	0.8	0.9
15	15	2	0.01	0
15	15	2	0.01	0.1
15	15	2	0.01	0.5
15	15	2	0.01	0.9
15	15	2	0.2	0
15	15	2	0.2	0.1
15	15	2	0.2	0.5
15	15	2	0.2	0.9
15	15	2	0.8	0
15	15	2	0.8	0.1
15	15	2	0.8	0.5
15	15	2	0.8	0.9

Table C.5: RNN network topologies that failed the training stage

Input Neurons	Hidden Neurons	Memory Depth	Learning Rate	Momentum
5	5	0.1	0.8	0.9
5	5	0.5	0.8	0.9
5	5	0.9	0.8	0.9
5	10	0.1	0.8	0.9
5	10	0.5	0.8	0.9
5	10	0.9	0.8	0.9
5	15	0.1	0.2	0.9
5	15	0.1	0.8	0.9
5	15	0.5	0.2	0.9
5	15	0.5	0.8	0.9
5	15	0.9	0.2	0.9
5	15	0.9	0.8	0.9
10	5	0.1	0.8	0.9
10	5	0.5	0.8	0.9
10	5	0.9	0.8	0.9
10	15	0.5	0.8	0.9
10	15	0.9	0.8	0.9
15	5	0.1	0.2	0.9
15	5	0.1	0.8	0.9
15	5	0.5	0.8	0.9
15	5	0.9	0.8	0.9
15	10	0.1	0.8	0.9
15	10	0.5	0.8	0.9
15	10	0.9	0.8	0.9
15	15	0.1	0.8	0.9
15	15	0.5	0.8	0.9
15	15	0.9	0.8	0.9

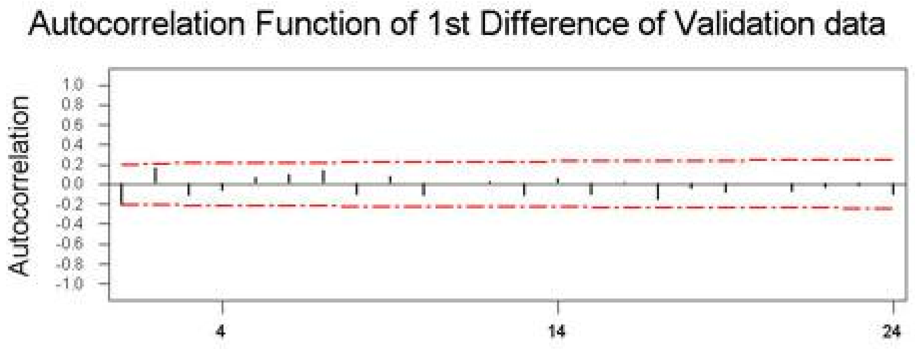
Appendix D – ARIMA forecasts of project FTSE 100 data

The first step in perform an ARIMA forecast is to ensure that the data you are using is stationary. This usually involves taking its first difference. The table below shows the validation data set's first difference.

Table D.1 validation data used in this project, together with its first differences.

Date	Index Value	1st Difference	Date	Index Value	1st Difference
10/02/2004	4404.95	*	22/04/2004	4571.83	31.963
11/02/2004	4396.05	-8.900	23/04/2004	4569.95	-1.876
12/02/2004	4377.73	-18.319	26/04/2004	4571.85	1.891
13/02/2004	4412.01	34.283	27/04/2004	4575.68	3.838
16/02/2004	4408.12	-3.894	28/04/2004	4524.48	-51.204
17/02/2004	4461.49	53.375	29/04/2004	4519.53	-4.947
18/02/2004	4442.90	-18.591	30/04/2004	4489.69	-29.844
19/02/2004	4515.57	72.663	04/05/2004	4547.23	57.545
20/02/2004	4515.04	-0.523	05/05/2004	4569.53	22.298
23/02/2004	4524.31	9.271	06/05/2004	4516.17	-53.362
24/02/2004	4496.76	-27.551	07/05/2004	4498.37	-17.798
25/02/2004	4507.55	10.783	10/05/2004	4395.16	-103.210
26/02/2004	4515.89	8.341	11/05/2004	4454.72	59.564
27/02/2004	4492.21	-23.672	12/05/2004	4412.93	-41.794
01/03/2004	4537.00	44.789	13/05/2004	4453.81	40.880
02/03/2004	4540.11	3.111	14/05/2004	4441.79	-12.019
03/03/2004	4525.13	-14.982	17/05/2004	4403.02	-38.771
04/03/2004	4559.07	33.939	18/05/2004	4414.41	11.391
05/03/2004	4547.08	-11.990	19/05/2004	4471.80	57.394
08/03/2004	4553.75	6.670	20/05/2004	4428.71	-43.094
09/03/2004	4542.01	-11.744	21/05/2004	4431.43	2.724
10/03/2004	4545.33	3.327	24/05/2004	4428.87	-2.561
11/03/2004	4445.22	-100.115	25/05/2004	4418.00	-10.872
12/03/2004	4467.35	22.130	26/05/2004	4438.29	20.283
15/03/2004	4412.93	-54.418	27/05/2004	4453.62	15.334
16/03/2004	4428.90	15.964	28/05/2004	4430.69	-22.931
17/03/2004	4456.80	27.903	01/06/2004	4422.68	-8.009
18/03/2004	4397.87	-58.931	02/06/2004	4422.80	0.118
19/03/2004	4417.74	19.867	03/06/2004	4435.41	12.609
22/03/2004	4333.77	-83.966	04/06/2004	4454.45	19.042
23/03/2004	4318.51	-15.259	07/06/2004	4491.60	37.150
24/03/2004	4309.45	-9.065	08/06/2004	4504.83	13.227
25/03/2004	4373.63	64.188	09/06/2004	4489.47	-15.352
26/03/2004	4357.53	-16.104	10/06/2004	4486.10	-3.369
29/03/2004	4406.73	49.200	11/06/2004	4483.96	-2.150
30/03/2004	4412.82	6.094	14/06/2004	4433.17	-50.782
31/03/2004	4385.67	-27.150	15/06/2004	4458.61	25.441
01/04/2004	4410.71	25.040	16/06/2004	4491.13	32.515
02/04/2004	4465.61	54.894	17/06/2004	4493.29	2.162
05/04/2004	4480.70	15.090	18/06/2004	4505.81	12.515
06/04/2004	4472.82	-7.879	21/06/2004	4502.18	-3.629
07/04/2004	4468.69	-4.130	22/06/2004	4468.49	-33.682
08/04/2004	4489.67	20.983	23/06/2004	4486.73	18.232
13/04/2004	4515.78	26.108	24/06/2004	4503.19	16.460
14/04/2004	4485.42	-30.357	25/06/2004	4494.05	-9.136
15/04/2004	4505.50	20.078	28/06/2004	4518.68	24.631
16/04/2004	4537.28	31.775	29/06/2004	4512.41	-6.270
19/04/2004	4546.22	8.940	30/06/2004	4464.07	-48.343
20/04/2004	4569.02	22.803	01/07/2004	4424.72	-39.345
21/04/2004	4539.87	-29.152	02/07/2004	4407.40	-17.322

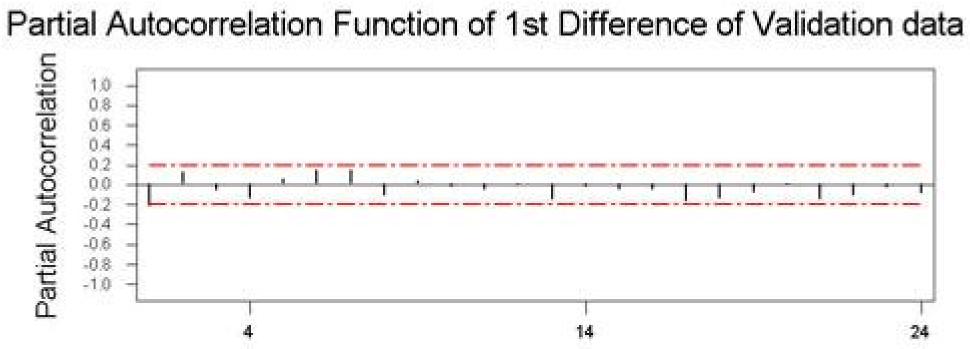
Once you are sure that your data is stationary, you need to identify which model (AR, MA or both) your data fits. This is done by examining the ACF & PACF plots of the differenced data. Figures D.1 and D.2 below, show ACF and PACF plots for the first differences of the validation data set respectively.



```
Lag  Corr     T    LBQ   Lag  Corr     T    LBQ   Lag  Corr     T    LBQ   Lag  Corr     T    LBQ

  1 -0.22 -2.18   4.90     8 -0.12 -1.04  14.54    15 -0.12 -1.00  20.77    22 -0.05 -0.37  26.68
  2  0.17  1.58   7.75     9  0.07  0.65  15.15    16  0.01  0.11  20.79    23 -0.03 -0.21  26.77
  3 -0.12 -1.11   9.26    10 -0.13 -1.10  16.94    17 -0.17 -1.41  24.20    24 -0.11 -0.93  28.51
  4 -0.08 -0.71   9.89    11  0.01  0.05  16.94    18 -0.05 -0.42  24.52
  5  0.06  0.58  10.33    12  0.03  0.23  17.02    19 -0.09 -0.75  25.55
  6  0.09  0.82  11.21    13 -0.12 -1.08  18.83    20  0.01  0.06  25.56
  7  0.13  1.17  13.04    14  0.05  0.45  19.15    21 -0.08 -0.67  26.41
```

Figure D.1: Autocorrelation Function of first difference of validation data



```
Lag   PAC      T   Lag   PAC      T   Lag   PAC      T   Lag   PAC      T

  1 -0.22  -2.18     8 -0.11  -1.12    15 -0.06  -0.59    22 -0.11  -1.09
  2  0.12   1.24     9  0.03   0.31    16 -0.05  -0.50    23 -0.04  -0.38
  3 -0.06  -0.65    10 -0.02  -0.24    17 -0.17  -1.69    24 -0.09  -0.94
  4 -0.14  -1.41    11 -0.05  -0.55    18 -0.15  -1.46
  5  0.05   0.53    12 -0.01  -0.10    19 -0.09  -0.85
  6  0.15   1.47    13 -0.15  -1.53    20 -0.01  -0.08
  7  0.15   1.47    14 -0.02  -0.21    21 -0.16  -1.56
```

Figure D.2: Partial Autocorrelation Function of first difference of validation data

The validation data shows small spikes throughout the plot. The shortness of these spikes suggests a randomness about the data. This makes the identification of a model quite difficult. However, as both plots are very similar this would suggest that both AR and MA elements exist. Through trial and error, the model ARIMA(1, 1, 1) has been chosen.

Figures D.3 and D.4 below show the ACF and PACF plots of the residuals of the validation data.

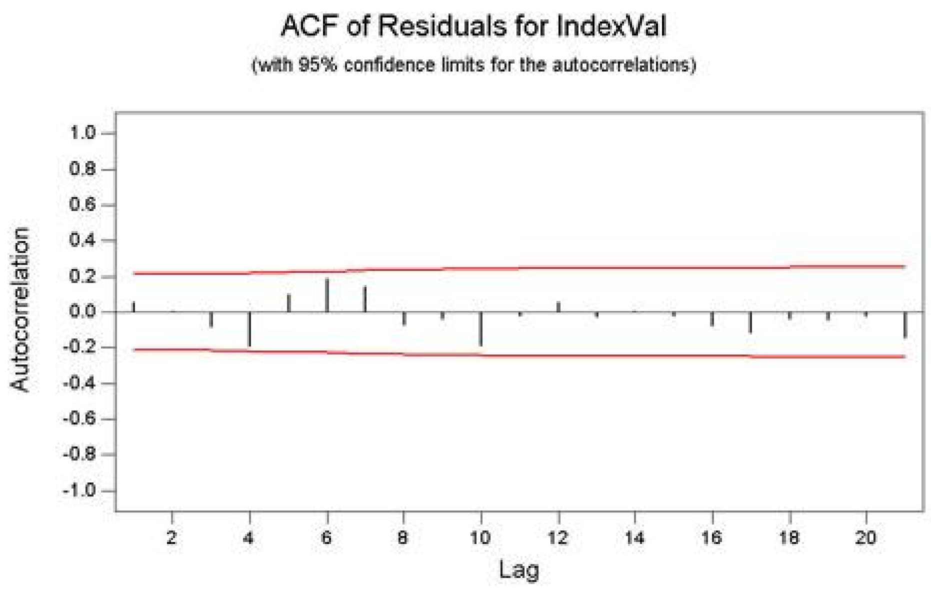


Figure D.3: Autocorrelation Function of residuals of validation data

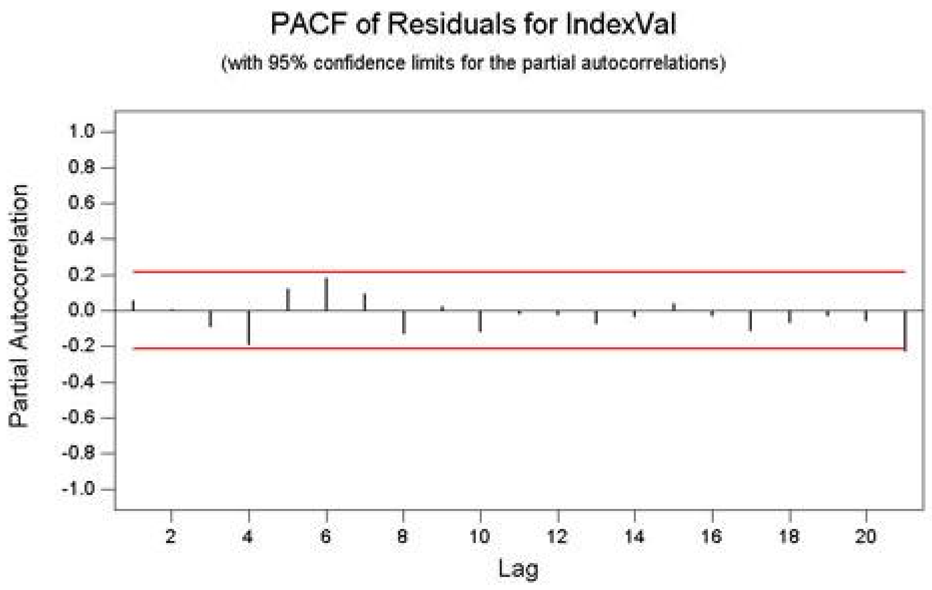


Figure D.4: Partial Autocorrelation Function of residuals of validation data

Once it is established that the remaining spikes are due to noise, the model can be used to forecast. Figure D.5 below is a time series plot of the validation data which shows a 5 day forecast. The forecast values are displayed as red triangles and the upper and lower 95% confidence limits are displayed as blue triangles.

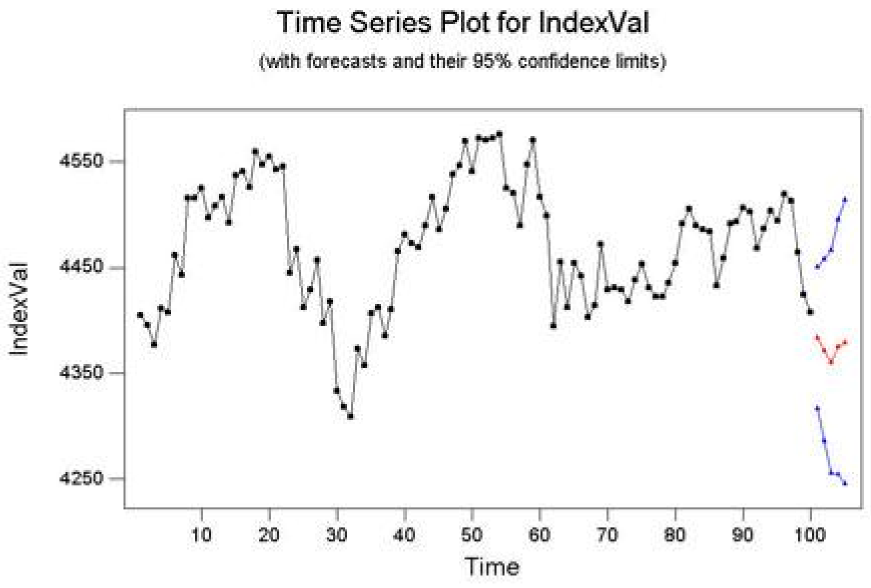


Figure D.5: Time series plot together with 5 day forecast for validation data

Listed below in figure D.6 is the printout from Minitab 13 showing various key statistics together with the forecasted values.

```
ARIMA model for IndexValue

Estimates at each iteration
Iteration       SSE     Parameters
    0        224259    0.100    0.100    0.100    0.100   -1.223
    1        167665    0.038   -0.050    0.162    0.249   -1.404
    2        158166   -0.112    0.029    0.020    0.385   -1.488
    3        148504   -0.262    0.086   -0.123    0.508   -1.569
    4        137685   -0.412    0.123   -0.267    0.632   -1.658
    5        123858   -0.562    0.116   -0.406    0.766   -1.812
    6        110027   -0.631   -0.034   -0.446    0.837   -2.092
    7        105695   -0.586   -0.184   -0.378    0.846   -1.983
    8        105283   -0.594   -0.229   -0.398    0.850   -1.898
    9        105243   -0.569   -0.242   -0.373    0.851   -1.823
   10         105239   -0.583   -0.246   -0.391    0.852   -1.832
   11         105239   -0.569   -0.247   -0.374    0.852   -1.808
   12         105238   -0.580   -0.248   -0.388    0.852   -1.823
   13         105238   -0.570   -0.247   -0.376    0.852   -1.809
   14         105238   -0.579   -0.248   -0.386    0.852   -1.821
   15         105238   -0.571   -0.248   -0.377    0.852   -1.810
   16         105238   -0.578   -0.248   -0.385    0.852   -1.819
   17         105238   -0.572   -0.248   -0.378    0.852   -1.811
   18         105238   -0.577   -0.248   -0.384    0.852   -1.818
   19         105238   -0.573   -0.248   -0.379    0.852   -1.812
   20         105238   -0.576   -0.248   -0.384    0.852   -1.817
   21         105238   -0.573   -0.248   -0.380    0.852   -1.813
   22         105238   -0.576   -0.248   -0.383    0.852   -1.817
   23         105238   -0.574   -0.248   -0.380    0.852   -1.814
   24         105238   -0.575   -0.248   -0.382    0.852   -1.816
   25         105238   -0.574   -0.248   -0.381    0.852   -1.814

Final Estimates of Parameters
Type         Coef    SE Coef       T       P
AR   1     -0.5744     0.3866    -1.49   0.141
SAR 13      -0.2477     0.1260    -1.97   0.053
MA   1     -0.3812     0.4293    -0.89   0.377
SMA 13      0.8522     0.1109     7.69   0.000
Constant   -1.8145     0.9366    -1.94   0.056

Differencing: 1 regular, 1 seasonal of order 13
Number of observations:  Original series 100, after differencing 86
Residuals:    SS =  93938.9  (backforecasts excluded)
           MS =  1159.7  DF = 81

Modified Box-Pierce (Ljung-Box) Chi-Square statistic
Lag              12       24       36       48
Chi-Square     15.7     22.4     50.6     60.7
DF                7       19       31       43
P-Value       0.028    0.266    0.015    0.039

Forecasts from period 100
                        95 Percent Limits
Period     Forecast       Lower       Upper      Actual
 101       4383.90     4317.14     4450.66
 102       4372.02     4286.23     4457.80
 103       4361.06     4255.64     4466.47
 104       4375.08     4255.23     4494.93
 105       4379.39     4245.60     4513.17
```

Figure D.6: Minitab printout of forecast for validation data together with key statistics

Appendix E: Test and validation results for RNN – 5-15-0.1-0.8-0.5-10000

Table E.1 Test and validation results for RNN – 5-15-0.1-0.8-0.5-10000

Actual Test Value	Predicted Test Value	Actual Validation Value	Predicted Validation Value
4236.39	4265.181	4408.119	4406.415
4202.181	4208.949	4461.494	4461.213
4157.129	4157.912	4442.903	4442.558
4142.725	4142.685	4515.567	4517.578
4091.305	4106.473	4515.044	4515.818
4169.19	4164.496	4524.314	4524.765
4209.055	4209.194	4496.763	4495.647
4274.036	4275.805	4507.546	4507.344
4270.103	4271.01	4515.887	4516.557
4271.96	4272.845	4492.214	4491.34
4268.594	4269.318	4537.003	4539.318
4313.882	4314.114	4540.114	4542.977
4311.024	4310.858	4525.132	4525.688
4362.34	4361.885	4559.072	4559.347
4334.086	4333.073	4547.082	4550.135
4368.816	4368.12	4553.752	4554.986
4339.675	4338.738	4542.008	4544.414
4343.975	4343.482	4545.335	4547.193
4347.549	4346.945	4445.22	4445.309
4352.313	4351.627	4467.35	4467.942
4285.634	4286.285	4412.932	4413.808
4240.219	4241.061	4428.896	4429.856
4238.996	4239.813	4456.799	4456.974
4251.313	4252.413	4397.868	4398.4
4272.927	4273.979	4417.735	4418.443
4265.652	4266.619	4333.77	4334.43
4300.929	4301.426	4318.511	4323.46
4287.594	4287.999	4309.446	4317.925
4332.574	4332.494	4373.634	4371.241
4330.25	4329.546	4357.53	4357.037
4303.433	4303.601	4406.73	4407.957
4324.195	4324.173	4412.824	4413.436
4376.873	4376.741	4385.674	4386.252
4341.764	4340.659	4410.714	4411.347
4345.07	4344.418	4465.608	4465.707
4371.246	4370.624	4480.698	4480.056
4373.041	4371.996	4472.819	4471.932
4396.992	4396.51	4468.689	4468.022
4338.881	4337.913	4489.672	4489.331
4354.705	4354.295	4515.78	4516.79
4327.353	4327.204	4485.424	4484.308
4307.977	4308.501	4505.501	4505.327
4319.025	4319.317	4537.276	4540.12

4382.354	4382.697	4546.216	4548.647
4388.745	4387.876	4569.019	4566.201
4370.346	4368.777	4539.867	4542.998
4361.09	4359.962	4571.83	4567.363
4342.602	4342.04	4569.954	4567.893
4410.029	4411.492	4571.846	4568.398
4378.941	4377.582	4575.684	4570.199
4392.024	4391.061	4524.48	4526.899
4378.213	4377.032	4519.533	4519.785
4367.029	4366.022	4489.688	4489.295
4359.842	4359.076	4547.233	4549.915
4379.583	4379.063	4569.531	4567.419
4335.385	4334.806	4516.169	4516.952
4331.268	4331.245	4498.371	4497.425
4347.643	4347.363	4395.16	4395.134
4347.993	4347.389	4454.724	4455.944
4332.96	4332.537	4412.93	4413.657
4354.228	4353.701	4453.81	4454.539
4397.257	4397.479	4441.791	4441.805
4412.279	4412.446	4403.019	4403.655
4423.985	4424.392	4414.411	4415
4440.871	4442.708	4471.805	4472.124
4444.7	4446.747	4428.711	4428.992
4457.489	4460.266	4431.435	4431.966
4470.383	4473.687	4428.874	4429.238
4476.866	4479.996	4418.003	4418.637
4510.178	4504.565	4438.286	4438.691
4513.252	4507.271	4453.62	4453.522
4505.217	4503.297	4430.689	4431.018
4472.965	4478.022	4422.68	4423.211
4494.168	4494.4	4422.798	4423.321
4466.292	4471.081	4435.407	4435.821
4449.611	4451.779	4454.449	4454.404
4440.143	4441.843	4491.599	4491.564
4461.391	4464.634	4504.825	4504.705
4456.081	4459.351	4489.474	4488.335
4487.877	4489.223	4486.105	4485.269
4518.144	4508.673	4483.955	4483.36
4499.27	4500.16	4433.173	4433.623
4511.181	4505.651	4458.614	4459.004
4476.793	4482.279	4491.129	4491.397
4460.81	4464.012	4493.291	4492.704
4445.483	4447.767	4505.806	4505.523
4447.001	4449.286	4502.177	4501.579
4468.116	4471.562	4468.495	4467.826
4411.533	4411.057	4486.727	4486.521
4390.676	4389.52	4503.187	4503.446
4381.366	4380.835	4494.051	4493.316
4390.581	4390.297	4518.682	4519.444

4398.465	4398.141	4512.411	4512.478
4384.353	4383.221	4464.069	4463.503
4402.723	4402.43	4424.724	4425.358
4434.423	4436.595	4407.402	4408.062
4314.696	4314.474	4404.95	4405.626
4257.01	4257.953	4396.05	4396.636
4228.152	4228.659	4377.73	4377.874
4221.707	4222.325	4412.013	4412.726

Appendix F: Read Me

This project requires extensive processing capability. It is inadvisable to run this program from disk. Instead, please copy all files to disk. This project uses Java and XML and therefore requires the Java API for XML Processing to be installed on the execution environment

There are two options to run this project

Navigate to /project/lib/ and execute project.jar

Navigate to /project/classes/ and execute java Test from a command prompt

The source code for this project is available here:

- BiasNeuron.java
- ContextNeuron.java
- ContextSynapse.java
- DataProcessor.java
- HiddenNeuron.java
- InputNeuron.java
- MLP.java
- Network.java
- Neuron.java
- OutputFile.java
- OutputNeuron.java
- RNN.java
- Synapse.java
- TDNN.java
- Test.java
- XMLParser.java

There are several other additional files also included. These are:

- The Javadoc
- The data files Train500.xml, Test100.xml, Validate100.xml and Forecast5.xml
- The project schema and XSL stylesheet
- The ant build.xml used for compilation