Generating high frequency trading strategies with arti cial neural networks and testing them through simulation

Thesis · September 2010

1 author:

Nicolas Dickreuter
University of St.Gallen

Generating high frequency trading strategies with artificial neural networks and testing them through simulation

September 24, 2010

A comparison of second order training methods for time series modelling

Nicolas Dickreuter - University of London, Goldsmiths College

(dickreuter@yahoo.com)

MSc in Cognitive Computing and Artificial Intelligence

Abstract

Various backpropagation algorithms are used to test whether a weak or a semi-strong form of the efficient market hypothesis can be rejected in trying to predict stock market returns with neural networks. First training algorithms such as Levengerg-Marquardt, Bayesian Regularization, BFGS Quasi-Newton and various (scaled) conjugate gradient gradient methods are used to train a network. In a second step the result is simulated to see whether the neural network can outperform a simple buy-and-hold strategy. Three examples are presented. In the first two, past returns of the Dow Jones Industrial average are used to predict future returns; in the first case with a 3-day and in the second case with a 1-day time horizon. In the third example input neurons are fed financial data of individual stocks, such as p/e, p/b and dividend payout ratios. The neural network then optimizes how a constant amount of money is allocated between stocks of the S&P100. While the different training methods converge to an optimum at different speeds and to different degrees, some of them can be described as being more efficient than others. However, in all the examples and all the methods, a simple, passive buy-and-hold strategy

has not been outperformed by a neural network over the entire period. Much more imporant than the backpropagation training method of the neural network is the actual model specification. The aim of this thesis is much more to lay the grounds of a theoretical framework how potential strategies need to be tested before they can be confirmed to be working, rather than finding actual working strategies.

Contents

1 Introduction

When forecasting the future of markets we can broadly distinguish between two categories: a) strategies which rely on historical data such as prices, volume of trading or volatility and b) fundamental analysis, which is based on external information from the economic system, such as company financial, interest rates, profitability ratios of the companies and other micro- and macroeconomic data. The use of technical analysis is mostly frowned upon in academic circles. The efficient markets hypothesis (EMH) asserts that the price of an asset reflects all of the information that can be obtained from past prices of the asset. Any opportunity for a profit will be exploited immediately and hence disappear. Despite the controversial reputation of technical analysis, it still finds respectable application among amateur traders, and professionals in the equity world often base their trading decisions on historical prices, among other information (otherwise, why would anybody look at price charts?). If there are some patterns which lead indeed to an outperformance, they should be detectable with the help of neural networks. It is the goal of this thesis to describe how such a trading strategy would have to be developed and tested for validity.

The first part (Chapter 2) describes the theoretical financial framework which is used as a basis to forecast prices with the help of neural networks. In Chapter 3 general properties of neural networks are described and in Chapter 4 the different training algorithms which are used in the analysis are described, and how generally neural networks and the simulator are programmed on Chapter 5.

I then show empirical results in Chapter 6 how the neural networks can be used to test the strong form of the efficient market hypothesis in trying to make predictions of the Dow Jones Industrial when only taking price as an input and empirically test the application of neural networks to trading strategies. Two such examples are presented. In a third example the input neurons are no longer past prices, but rather financial factors of individual stocks. The neural network should then decide how a fixed amount of money should be allocated among stocks of the S&P100.

First a network is trained to recognize buying and selling signals. This is done with a variety of different training algorithms that are described in more detail. The trained neural network is then implemented into a simulator which calculates trading results with various parameters in the threshold function, which activates buying and selling signals. The calculated optimal parameters which are extracted through numerical approximation are then used as a trading strategy. In order to claim that a trading strategy is potentially working in a real-life environment, it needs to be tested for statistical significance in an out-of-sample test.

Finally the conclusion (Chapter 7) ends the thesis with a summary of the findings.

2 Theoretical Framework

2.1 Efficient market hypothesis (EMH) - expectations based on other research

The Efficient market hypothesis assumes that financial markets are informationally efficient and that excess market return cannot be generated consistently through any strategy.

There are three forms of the EMH: The weak, the semi-strong and the strong form. In the weak form, future prices can't be predicted simply by analyzing past prices and no excess returns can be achieved in the long run using historical price data only. In other words technical analysis won't be able to produce abnormal returns when the weak form of the EMH is assumed to be holding but other forms such as insider trading and fundamental analysis may still be applied successfully.

The semi-strong form of the EMH implies that all publicly available information, meaning not just historical prices but also company or securities specific information is included in the current price, making it impossible to outperform the market not just by technical analysis, but also by fundamental analysis. Whenever a new piece of information about a security or any of its underlyings becomes available, this would be incorporated into the price instantly.

To test for strong-form efficiency, a market needs to exist where investors cannot consistently earn excess returns over a long period of time. Even if some money managers are consistently observed to beat the market, no refutation even of strong-form efficiency follows: with hundreds of thousands of fund managers worldwide, even a normal distribution of returns (as efficiency predicts) should be expected to produce a few dozen "star" performers.

Various research papers have disputed the efficient-market hypothesis, both on empirical and theoretical basis. For example "Behavioral economists" claim that factors such as overconfidence, overreaction, representative bias, information bias, and various other predictable human errors can cause distortions which could potentially be exploited from investors recognizing them. Empirical evidence has been mixed, but has generally not supported strong forms of the efficient-market hypothesis. (Nicholson [Jan/Feb 1968], Basu [1977], Rosenberg B [1985], Fama E [1992], Michaely R [1993])

Other papers have shown that P/E ratios can have influence on stock returns N. and A. [1992] and other company specific (or "fundamental" factors) have been identified to have potentially predictive power on stock returns. While only a fraction of academic research in the field has been able to produce abnormal return in empirical application, any such market inefficiency would automatically vanish once the volume of the profit generating trades is high enough. That's why it is very likely that if any neural network could find a successful strategy from the past, it may potentially only last for a short period of time because market participants quickly exploit it.

In this thesis I aim to show how a weak and semi-strong form of the EMH could be rejected or accepted with the help of neural networks. In the first case I only use past prices of the Dow Jones Industrial as input neurons. If the neural network would find any strategy which could outperform the market, the weak form of the EMH would be rejected. Given the large amount of research which does have confirmed the weak form of EMH, this would be unexpected when based purely on plain vanilla equity instruments.

When taking fundamental factors (company specific financial information) into account the neural networks are set up to reject or accept the semi-strong EMH. As in this analysis only a fraction of the available information about the companies' financials are used in the model, it won't be possible to draw any definitive conclusions from the results. Moreover, the forthcoming analyses should give a framework under which potential market inefficiencies can be tested for possible exploitation. Research in this domain is mixed and opinions are polarized very strongly. Advocates of an inefficient market often point to start fund managers and skeptics often try to highlight the fact that ex-post academic analyses are often flawed and biased. When looking at the performance of professionally managed funds, where many of them take fundamental factors into consideration, it is in my view very unlikely that a semi-strong form of the EMH can be rejected. The reason behind this is based on two factors:

1. Market complexity is generally underestimated. To be able to predict the outcome of a stock price based on a small number of factors is intuitively impossible when assuming the following: When hitting a billiard ball on a table and trying to predict the second or third bounce is relatively simple in reality and can be replicated in a laboratory. However, the ninth bounce, an equation that includes every single person around is needed, because the gravitational pull of a man will impact the trajectory. The equation will get significantly more complex. To predict the 53rd bounce of the billiard ball on the table every single elementary particle in the universe

needs to be part of the equation. There is no question that to predict the future price of a stock is more complicated that predicting the 53rd bounce of a billiard ball (Taleb [2007]).

2. Large, unpredictable (exceptional) events (black swans) have a significant influence on returns. "Based on data for the Dow Jones Industrial Average over the last 107 years, is unequivocal: outliers have a massive impact on long-term performance. Missing the best 10 days resulted in portfolios 65% less valuable than a passive investment, whereas avoiding the worst 10 days resulted in portfolios 206% more valuable than a passive investment."Estrada [2009]. It appears that the outliers are the decisive events rather than the small fluctuations. Those outliers are per definition unexpected and would not be predicted by any related variables, even when statistical significance is given. That's why any investment approach based on a neural network (or probably worse: based on normal distribution and linear regression) would most likely fail in the long run.

Nevertheless, it is possible that a neural network may spot some market inefficiencies that could reject the weak form of the EMH for a short period of time. It would be virtually impossible to transform this into a profit in a real-world environment, but I'd like to outline the theoretical framework that could be applied to identify such inefficiencies and then test them on statistical significance. While passing the tests would still be no guarantee of potential success in a real environment, failing them would have to lead to an exclusion of the active strategy.

2.2 Development of trading strategies

In order to test whether the neural network can generate a trading strategy, we need to give some framework under which the strategy is developed. Since the neural network can only act within its own model specifications, it helps when some economic intuition is behind the data that is fed to the input neurons. As outlined above, historical prices are only one factor that is potentially of value. Intuitively (although not necessarily so in reality) additional factors should improve the performance of the neural network.

In this paper the following strategies are being testes:

1. Buying and selling signal given through a 3-day forward looking neural network, taking into consideration the returns that will be earned over

the next 3 days as output neuron, and the last 5 days as input neurons. This strategy is based on the Dow Jones Industrial Index.

2. Buying and selling signal given through a 1-day forward looking neural network, considering the next day's return as output neuron and the last 10 days as input neurons. This strategy is based on the Dow Jones Industrial Index.

3. Buying and selling signal given through a network which takes 3 financial factors of different companies into consideration. The output neuron is connected to the next day's return of the respective stocks. This strategy is based on individual stocks of the S&P100.

2.3 Out of sample testing

Once the network has been trained with a subset of the dataset, the neural network needs to be tested with an out-of-sample application. It is generally agreed that for forecasting methods out-of sample tests rather than goodness of fit to past data (in-sample tests) should be used. The performance of a model on data outside of that used in its construction and training remains the ultimate touchstone which determines its validity. There are two arguments as shown by[Tashman, 2001]:

For a given forecasting model, in-sample errors are likely to understate forecasting errors. Estimation and method selection are designed to calibrate a forecasting procedure. This leads to the problem that in many cases the small nuances of past history are not repeated in the future and other occurrences which may occur in the future may not be part of the training sample. Needless to say, this can have a strong impact on the performance of the model.

In addition common extrapolative forecasting methods, such as exponential smoothing or also neural networks, are based on updating procedures, in which one makes each forecast as if one were standing in the immediately prior period. For updating methods, the traditional measurement of goodness-of-fit is based on one step-ahead errors - errors made in estimating the next time period from the current time period. Real time assessment has practical limitations for forecasting practitioners, since a long wait may be necessary before a reliable picture of a forecasting track record will materialize. As a result, tests based on holdout samples have become commonplace. If the forecaster withholds all data about events occurring after the end of the fit period, the forecast-accuracy evaluation is structurally identical to the real-world-forecasting environment, in which we stand in the present and forecast the future. However, taking already into consideration the held-out data while selecting the forecasting method pollutes the evaluation environment.

2.4 Procedure of the simulation

2.4.1 Simulation with different parameters after training of the network

Once the neural network has been trained, its effectiveness needs to be tested with a trading simulation. It is not enough to just look at the mean square error of the trained neural network but an actual simulation is imperative. In our case the testing is done with a custom built simulator in Matlab where different strategies can be easily defined in rewriting the respective class.

While the total trading performance is the most important factor to look at, it is crucial to see it in the context with what the strategy actually does, such as the amount of total trades, % of winning trades and % of days long. As a benchmark we always take a simple buy-and-hold strategy (passive investment strategy) which can always be achieved at much lower cost as there is only minimal infrastructure necessary. In the active strategy the model assumes a 2% risk free rate, meaning that when the index (or security) is sold, a 2% p.a. return can be achieved.

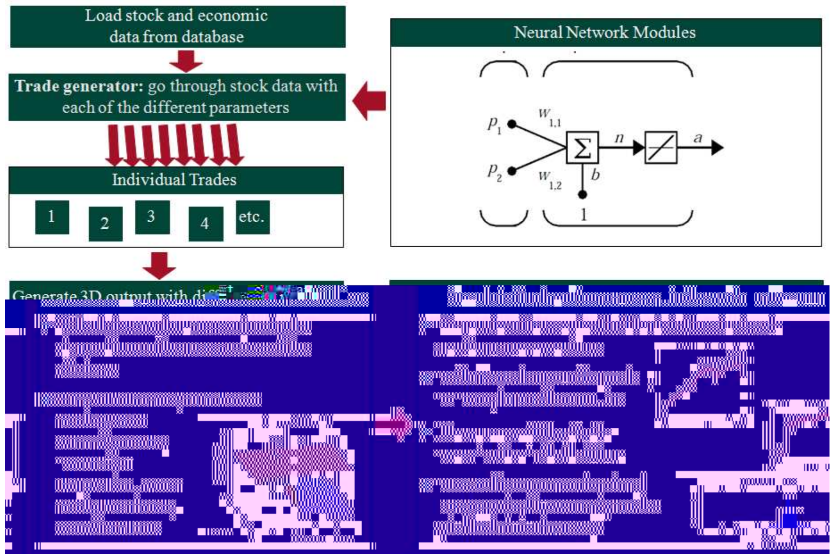


Figure 2.1: Procedure of simulation

There are 3 steps in the actual simulation:

1. The simulation takes the trained neural network and varies the different threshold parameters for the buying and selling decision individually (see Fig. 3.3 on page 19 for examples of threshold functions).

2. An optimal strategy will evolve. In Fig 2.1 we can see at the bottom left how the total annual return of the strategy will peek for some parameters. This strategy will need to be further investigated as it is the most promising one.

3. The optimal strategy is tested for statistical significant outperformance versus a buy-and-hold strategy in an out of sample test.

2.4.2 Calculation of returns

There are two ways to calculate the return between t and $t+1$. For the purposes to calculate cumulative abnormal return, the compounded return (i.e. logarithmic) presents the distinctive advantage that different time periods of return can be added up in order to calculate the return of a cumulative period. Unless stated otherwise I use for all calculations in this paper compounded returns:

$$R := LN\left(\frac{S_t}{S_{t+x}}\right) = LN(S_t) - LN(X_{t+x})$$

S_t is the stock price (or index price) at day t. With the given calculation each strategy is compared with a buy-and-hold approach of the same securities and a number of different metrics are calculated as outlined below. Returns are calculated in the TimeSeries class right after the excel files are loaded.

2.4.3 Calculation of other metrics

- The total performance of a given trading strategy is calculated in summing up the individual logarithmic returns of the generated trades, such as $R^s = (\sum_{i=1}^{n} R_i) + R_x^f$ where R_i depicts the return of each individual trades from $1 \dots n$ and R_x^f the risk free return of the period where no trade was open.

- The total strategy performance p.a. is the annualized version of the above: $R^{sp} = \frac{R^s}{totaldays} \cdot 365$

- The total risk free return R^f for period $x$is calculated as $R_x^f = R^f \cdot x$ where x constitutes the amount of idle days where no trade is open.

- The number of idle days is calculated as $TS_n - \sum T_i^d$ where $TS_n$is the total amount of returns of the time series and $T_i^d$the duration of trade i.

- The buy-and-hold performance is simply calculated in summing up all the individual returns of the time series or $R^b = \frac{Log(TS_{stop})}{Log(TS_{start})}$

- The buy-and-hold performance p.a. is calculated as $R^{bp} = \frac{R^b}{totaldays} \cdot 365$

- The out-performance p.a. R^{op} is calculated as $R^{op} = R^{sp} - R^{pb}$

The calculations above are taking place in the Trade Analyzer class and is listed in Chapter D.2.4 on page 91.

2.4.4 Necessity of statistical testing

In order to determine whether the neural network's generated trading performance is better than a random strategy, we need to test the result for statistical significance.

Hypothesis testing is defined by the following general procedure:

1. State the relevant null hypothesis. In our case this is $\mu_r = \mu_n$where as $\mu_r$is the result of a random investment strategy and the $\mu_n$the average return of the strategy developed by the neural network.

2. Statistical assumptions need to be tested. For our purposes what matters is whether returns are normally distributed. "The normal distribution is a poor fit to the daily percentage returns of the S&P 500. The log-normal distribution is a poor fit to single period continuously compounded returns for the S&P 500, which means that future prices are not log-normally distributed. However, sums of continuously compounded returns are much more normal in their distribution, as would be expected based on the central limit theorem. The t-distribution with location/scale parameters is shown to be an excellent fit to the daily percentage returns of the S&P 500 Index." Egan [2007]

3. Decision of which statistical test is applied: Due to the inconclusive results I use a non-parametric test which is much more lax in terms of statistical assumptions: The Wilcoxon signed-rank test can be used as alternative to t-tests when the sample cannot be assumed to be normally distributed.

4. Next the values need to be computed.

5. Decision whether the null hypothesis can be rejected and the returns are statistically significantly different to 0. Alternatively the null hypothesis cannot be rejected and the Neural network does not outperform the random strategy

2.4.5 Non parametric test

The Wilcoxon signed-rank test is a statistical hypothesis test for the case of two related samples or repeated measurements on a single sample. It can be used as an alternative to the paired Student's t-test when the population cannot be assumed to be normally distributed. Similar to the paired or related sample t-test, the Wilcoxon test involves comparisons of differences between measurements. The advantage is that it doesn't require assumptions about the form of the distribution.

The Wilcoxon test transforming each instance of $X_A - X_B$ into its absolute value, and removes all the positive and negative signs. In most applications of the Wilcoxon procedure, the cases in which there is zero difference between X_A

and X_B are at this point eliminated from consideration, since they provide no useful information, and the remaining absolute differences are then ranked from lowest to highest, with tied ranks included where appropriate.

Suppose we collect $2n$ observations, two observations of each of the n subjects. Let i denote the particular subject that is being referred to and the first observation measured on subject i be denoted by x_i and second observation be y_i. For each i in the observations, x_i and y_i should be paired together. The null hypothesis tested is $H_0 : \theta = 0$. The Wilcoxon signed rank statistic W_+ is computed by ordering the absolute values $|Z_1| \ldots, |Z_n|$ the rank of each ordered Z_i is given a rank of R_i. Denote the positive Z_i values with $\varphi_i = I(Z_i > 0)$ where $I(.)$ is an indicator function. The Wilcoxon signed ranked statistic W_+ is defined as

$$W_+ = \sum_{i=1} \phi_i R_i$$

Tied scores are assigned a mean rank. The sums for the ranks of scores with positive and negative deviations from the central point are then calculated separately. A value S is defined as the smaller of these two rank sums. S is then compared to a table of all possible distributions of ranks to calculate p, the statistical probability of attaining$S$ from a population of scores that is symmetrically distributed around the central point. The test statistic is analyzed using a table of critical values. If the test statistic is less than or equal to the critical value based on the number of observations n, then the null hypothesis is rejected in favor of the alternative hypothesis. Otherwise, the null cannot be rejected. Lowry [2010],Wikipedia [2010]

3 Neural networks

3.1 Structure and properties of neural networks

3.1.1 Neural networks vs. other statistical methods

Neural networks have a similar approach to other statistical methods used for data modeling[1]. In all cases it is the goal to limit the number of assumptions and to use the method to optimize a procedure to find the best parameters to reproduce the results. As described in Carverhill and Cheuk [2003] already countless research has been made in the financial industry where neural networks

[1]See Riplay [1996] for a general di cu ion of Neural Network and it relation to non-parametric e timation method

have been used to forecast option prices (as substitute method for Black-scholes) or Kwon et al. [2010] used a neural network in an attempt to forecast stock prices. While it may be possible that neural networks can indeed predict prices more accurately than other models, research which is actually testing the success of the neural network in a statistically rigorous manner is very rare.

3.1.2 Structure

The neural networks in this thesis are build on three layers. An input layer, one hidden layer and one output layer. It would be possible to use additional hidden layers as shown in Fig. 3.1, but for simplicity only one hidden layer has been used here. The amount of input neurons varies for the examples, but is generally between 3 and 5. Hidden neurons are between 5 and 10 and in all cases there is one output neuron which is used to predict the time series (i.e. the returns).

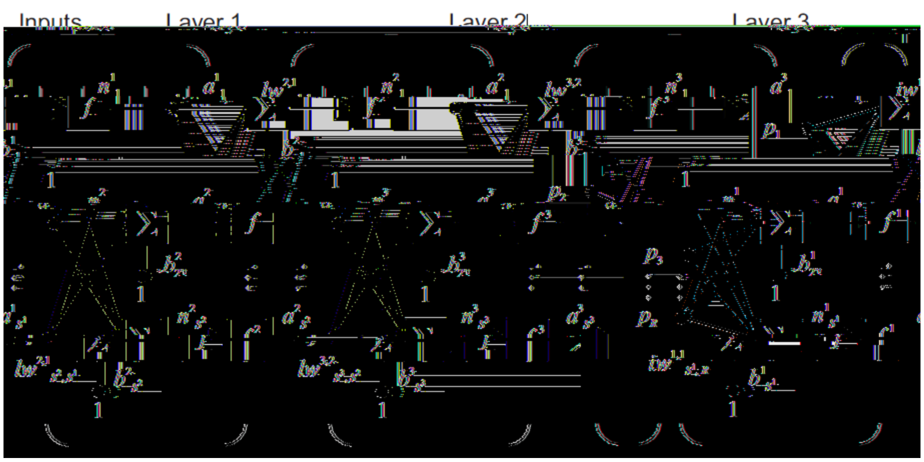


Figure 3.1: Neural network structure with multiple layers as described in Mark Hudson Beale [2010]

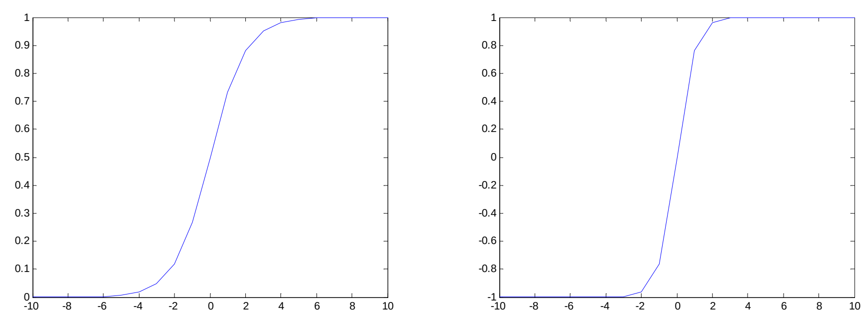
The input vector elements enter the network through the weight matrix **W**.

$$\mathbf{w} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,r} \\ w_{2,1} & w_{2,2} & w_{2,r} \\ \vdots & \vdots & \vdots \\ w_{s,1} & w_{s,2} & w_{s,r} \end{bmatrix}$$

The row indices on the elements of matrix **W** indicate the destination neuron of the weight, and the column indices indicate which source is the input for that weight. Before training of the network can start the weights will need to be initialized with random values.

3.1.3 Activation function

Hidden layers are using the sigmoid activation function which is defined as $P(x) = \frac{1}{1+e^{-x}}$ Haykin [2009]. Experiments have also been made with the TanH function, but no significant difference could be found in the results. The comparison of the two functions is shown in Fig. 3.2. The output neuron is using a linear activation function which is sometimes activating the signal through a threshold as is further explained in Chapter 3.1.6.



(a) Sigmoid function (b) TANH function

Figure 3.2: Activation functions

3.1.4 Momentum

Momentum is used to stabilize the weight change by making nonradical revisions using a combination of the gradient decreasing term with a fraction of the previous weight change: This gives the system a certain amount of inertia since the weight vector will tend to continue moving in the same direction unless opposed by the gradient term. The momentum smooths the weight changes and suppresses cross-stitching, that is cancels side-to-side-oscillations across the error valley. When all weight changes are all in the same direction the momentum amplifies the learning rate causing a faster convergence. Momentum also enables to escape from small local minima on the error surface. We can define momentum as follows:

$$w_{ij} = w'_{ij} + (1 - M) \cdot LR \cdot e_j \cdot X_j + M \cdot (w'_{ij} - w''_{ij})$$

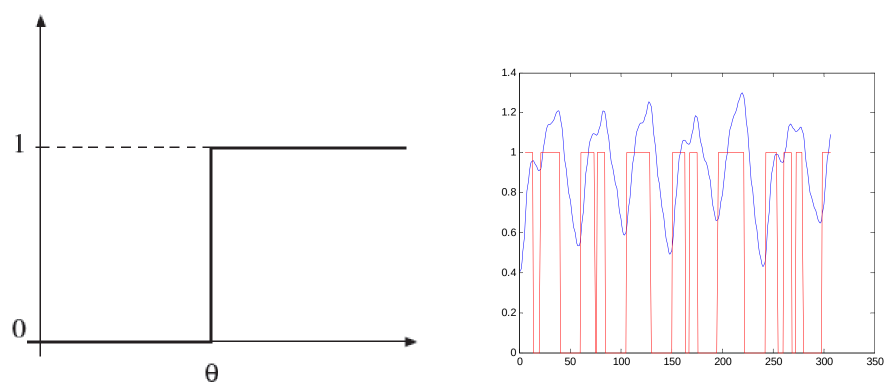
The results of the neural network (in this case the gradient descent backpropagation method) has not been strongly influenced on whether momentum has been applied or not. This is mainly a dependent on the form of the error surface as described above. An example of where momentum would have an effect is shown in the error surface of Fig. 4.2 on page 27.

3.1.5 Learning Rate

In the program a learning rate of $\eta = 0.1$ has been found to be a good measure to start. Depending on the backpropagation method the learning rate performs different functions. The variable learning rate method experiments with a non-constant learning rate.

3.1.6 Threshold function

θ denotes the threshold. In the simulation the optimal threhold is found out through experimentation. When the output neuron passes a threshold, a buying or selling action is performed.



(a) Binary activation function (b) Example of binary activation function

Figure 3.3: Threshold functions

3.2 Handling of time series

In order for the program to be able to handle the time series, a variety of additional enhancements need to be implemented: In the gradient descent method each datasets leads to an adjustment of the weights. The time series then moves one step forward and the input neurons are adjusted accordingly. Input neurons are normally the 5 datapoints preceding the one that is predicted by the output neuron, however the program allows to have the interval between those input neurons to be increased through the interval variable. At the end of the epoque the weights of the neural network need to be saved so that training can continue with another epoque to further adjust the weights. In the program this is achieved as follows: when the variable *skipinit* $= 1$ the weights are no longer reset and randomized at the beginning of the training, but previously calculated weights remain in the memory.

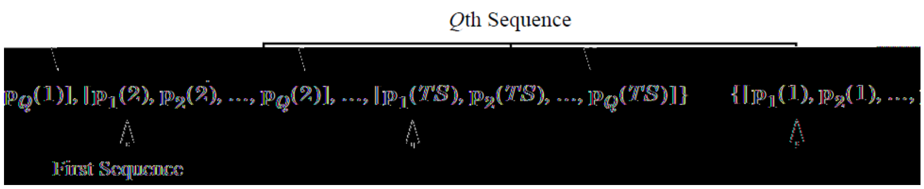


Figure 3.4: Time series handling

3.3 Preprocessing of data

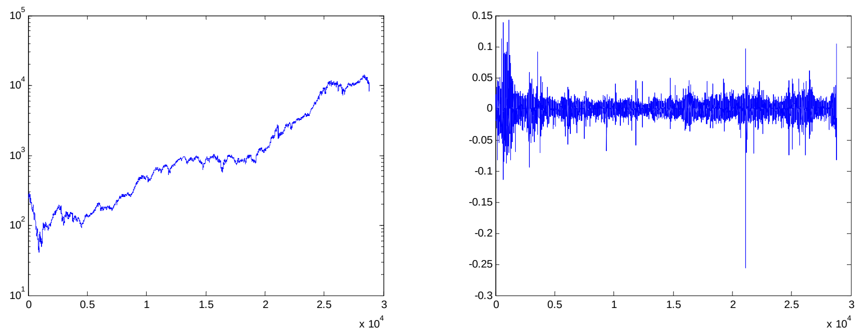
3.3.1 Normalization

Neural networks can be made more efficient when certain preprocessing steps are taken. Normalizing errors when population parameters are known is simply done as follows: $X' = \frac{X-}{}$. When training the network, normalization has shown to be imperative, especially when past daily returns are used. Since they are usually very small numbers, a large number of trainings would need to be used to make sure the neural network output has exactly the same mean as the original time series itself. If this is not the case the bipolar threshold function is no longer symmetrical and buying and selling conditions are messed up.

Without normalization if the input is very large, the weights would have to be very small to prevent the transfer function to be saturated. That's why it is standard practice to normalize the time series before it is applied to the neural network. This needs to be done both to the input vectors and to the target vectors.

3.3.2 Absolute vs. relative changes

While it would theoretically be possible to feed market prices directly into the neural network, it is very likely that better results can be achieved when relative changes to previous periods (i.e. log returns as described in 2.4.2 on page 13) are used, especially as the returns are more likely to approach a normal distribution than the prices themselves.



(a) Dow Jone Indu trial Average 1929-2008: Ab olute value in log cale

(b) Dow Jone Indu trial Average 1929-2008: daily log return

Figure 3.5: Absolute values vs. relative returns

For the purpose of exploiting the strong form of the EMH (for further details see Chapter 2.1 on page 8) it makes sense to use a normalized form of logarithmic returns as we want to extract buying and selling signals. For that we need to know what happens to the returns R in the period $t + x$. An example of the results from a trained network see A.14 on page 52.

3.3.3 Generalization

A well trained network should be able to give good answers even with inputs that have not been seen during the training. The more similar the new input is to the ones that have been used during training, the better are the answers. To improve generalization there are two features that can be used:

1. Early stopping: It is important that the network is not overtrained, as this will lead to much worse results when being applied with untrained input data. In Matlab early stopping can be automatically handled in monitoring the error of the validation set while training on the training set. Training is then stopped when the validation increases over net.trainParam.max_fail iterations (maximum number of validation Increases).

2. Regularization: Regularization can be done by using the Bayesian regularization training function. Any data in the valuation set should be moved to the training set.

3.3.4 Division of data

When training multilayer networks, the general practice is to first divide the data into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set where the error on the validation set is monitored during the training process. The validation error should decrease during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set begins to rise. The test set error (which constitutes the third type of dataset) is not used during training. If the error on the test set reaches a minimum at a significantly different iteration number than the validation set error, this might indicate a poor division of the data set. The division can be done in blocks, randomly, using interleaved sections or by data index. (Mark Hudson Beale [2010])

3.4 Postprocessing and assessment

3.4.1 Performance function - mean square error

A good initial assessment for the neural network is the mean square error (MSE) which gives us indication how the different training epoques improved its performance. It is defined as follows:

$$MSE = \frac{1}{N} \sum_{i=1} (t_i - a_i)^2$$

While it is generally desirable to reduce the MSE as much as possible, overfitting may quickly become a problem when the network is trained too much. An enhancement to the above function gives us the the Bayesian Regularization approach involves modifying the usually used objective function. As shown in Chi Dung Doan [2009] the modification can enhance the model's generalization capability in expanding the term. E_w which is the sum of the square of the network weights.

Matlab offers a variety of tools to further test the validity of the network. The foundations for this are already laid in the steps where the data is preprocessed and the data is split into training, validation and testing set for each epoch for each of the training sets and continues progress of the neural network should be observable. Ideally the validation and test MSE should develop in a similar manner. If the test curve had increased significantly before the validation curve increased, then it would be an indication that overfitting had occurred. An example is shown in Fig. A.5 on page 48 where a repetitive pattern is used as input to BFGS quasi newton method. In Fig. A.14 on page 52 we can see the difference of the results when various training methods are used.

4 Training algorithms

4.1 Overview

4.1.1 Presented methods

While some of the training algorithms were programmed from scratch, I have taken the liberty to use some of the integrated Matlab functions, as they offer much greater efficiency and speed and also flexibility. The following methods have been used for testing:

Function	Algorithm
trainlm	Levenberg-Marquardt
trainbr	Bayesian Regularization
trainbfg	BFGS Quasi-Newton
trainrp	Resilient Backpropagation
trainscg	Scaled Conjugate Gradient
traincgb	Conjugate Gradient with Powell/Beale Restarts
traincgf	Fletcher-Powell Conjugate Gradient
traincgp	Polak-Ribiére Conjugate Gradient
trainoss	One Step Secant
traingdx	Variable Learning Rate Gradient Descent
traingdm	Gradient Descent with Momentum
traingd	Gradient Descent

Table 1: Training algorithms used for testing

Each of the training algorithms have their advantages and disadvantages. While for the overall performance of the trading simulator the actual training algorithm may only have an insignificant effect, as other factors, such as quality of data and correctness of the model specification (i. e. picking the relevant data) is much more important. Nevertheless, for completeness's sake I analyze the performance of the different algorithms and give a short assessment on what their effects are when used with returns on the Dow Jones Industrial Average.

It is complex and and time consuming to compute the Hessian matrix for feedforward neural networks. Algorithms that are based on Newton's method, do not require calculation of second derivatives. These are called quasi-Newton (or secant) methods. They update an approximate Hessian matrix at each iteration of the algorithm. The update is computed as a function of the gradient. The quasi-Newton method that has been most successful in published studies is the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update. This algorithm is implemented in the trainbfg routine.Mark Hudson Beale [2010]

The training algorithms all aim to reduce the error produced by the neural network in finding a way to strive to the ideally global minimum of the n-dimensional error surface.

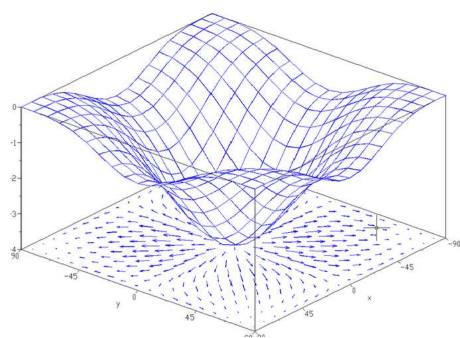


Figure 4.1: Gradients of a vector field

4.1.2 Application in Matlab

At first network and weights need to be initialized and then the network can be trained. The network can be trained for function approximation (nonlinear regression), pattern association, or pattern classification. A set of examples with network input p and target t is required.

For batch gradient descent the *traingd* function is used. The weights will be updated in the direction of the negative gradient.

There are seven training parameters associated with *traingd*:

- epochs: number of epochs

- show: The training status is displayed for every show iterations of the algorithm

- goal: The training stops if the number of iterations exceeds epochs, if the performance function drops below goal
- time: Training stops after a certain amount of time
- min_grad: Training stops if gradient falls below min_grad
- max_fail: If training is lager than maxfail
- lr: The learning rate lr is multiplied times the negative of the gradient to determine the changes to the weights

The network is trained on the training data until its performance begins to decrease on the validation data, which signals that generalization has peaked. The test data provides a completely independent test of network generalization (Mark Hudson Beale [2010]).

4.2 Gradient Descent

Weights and biases are updated in the direction of the negative gradient of the performance function which is the direction where the performance function decreases the most.

Gradient descent iteratively updates w replacing w_t by w_{t+1} using the following update equation where η is a learning rate that typically declines with t. $w_{t+1} = w_t - \eta \nabla \mathcal{L}$. Note that here we are minimizing $\mathcal{L}$ so we want to move in the opposite direction from the gradient.

$$w_{t+1} = w_t - \eta_t w \frac{\partial \mathcal{L}}{\partial w_j}$$

(Mark Hudson Beale [2010] use the notation as follows: $\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g_k}$ where $\mathbf{x}_k$ is a vector of the current weights and $\mathbf{g}_k$ is the current gradient and α_k is the learning rate.)

Through one-dimensional optimization w_j is adjusted while holding the other weights fixed. We do this for each j and because the weights have changed, we may need to repeat the process until all weights are simultaneously at optimal values.

For a normal multilayer perceptron with incremental method the following method has been applied in the computer program:

1. A training sample if presented to the neural network which is normalized. $X_i' = \frac{x_i \quad {}_i}{{}_i}$

2. The z-score normalization uses the mean and the standard deviation for each feature across a set of training data.This produces data where each feature has a zero mean and a unit variance.Priddy [2005]. For our given purposes this appears to be the method of normalization which makes the most sense.

3. The desired output is compared to the actual output after forward propagation.

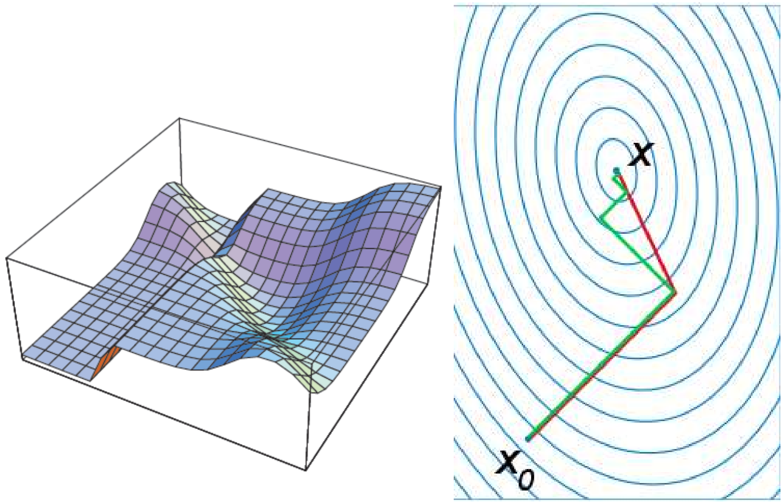
4. β is calculated for each node The beta for the TanH equals $\beta = q \cdot y \cdot (1 - y)$, where q is the weighted sum of the betas of the following layer, where the respective network is connected to. The weights are given by the weights of the connection themselves. y denotes the output signal.

5. The weights of each connection are adjusted with the delta, which is calculated as follows: $\triangle = \beta_i \cdot \eta \cdot y_{xy}$(in the program listing this is neuron(id).beta * learningRate * neuron(neuron(id).connections(n)).outputsignal)

6. The previous steps are repeated until the mean square error is no longer decreasing. This prevents overfitting.

Minimization by gradient descent is based on the linear approximation $E(w + y) \approx E(w) + E'(w)^T y$, which is the main reason why the algorithm is often inferior to other methods. In addition the algorithm uses constant step size, which in many cases are inefficient.

A simple form of gradient descent suffers from serious convergence problems. Ideally we should be able to take steps as large as possible down to the gradient where it is small (small slope) and and take smaller steps when the gradient is large, so that we don't jump of of the minimum. With the update rule of the gradient descent we do right the opposite of that. In addition to that the curvature of the error surface may not be the same in all directions. A long and narrow valley for example the motion could be more along the wall (bouncing from one wall to the other) even though we want to move as fast as possible out of the valley without bouncing the walls. This can be improved with not only using gradient information but also the curvature (i.e. second derivatives). (Ranganathan [2004])



(a) Error function with local minima a decribed in Roja [1996]

(b) Gradient de cent method (green) v . conjugate gradient illu tration (red) a hown in Alexandrov [2010]

Figure 4.2: Moving on the error surface: method comparison

4.3 Gradient Descent with Momentum

In adding momentum to the gradient descent method, the network can also respond to recent trends in the error surface. In that sense momentum can ignore small features on the error surface and act like a lowpass filter. It will help to avoid that the network gets stuck in a shallow local minimum that could be slid through with momentum. (Mark Hudson Beale [2010]).

Including a momentum term in the algorithm is a way to force the algorithm to use second order information from the network. Unfortunately the momentum term is not able to speed up the algorithm considerable, and may cause the algorithm to be even less robust, because of the inclusion of another user dependent parameter, the momentum constant (Møller [1990]). This is in accordance with the observations made in this paper as is visible in Fig. A.12 compared to Fig.A.13.

4.4 Variable Learning Rate Gradient Descent

This method uses a variable learning rate η that is adjusted depending on how the performance of the network evolves during training. η is decreased whenever performance has increased while if performance deteriorates η is increased. As calculation normal gradient descent backpropagation is used as described in Chapter 4.2 is used.

4.5 Gauss Newton method

The Gauss–Newton algorithm is a method used to solve non-linear least squares problems. It can be seen as a modification of Newton's method for finding a minimum of a function. Unlike Newton's method, the Gauss–Newton algorithm can only be used to minimize a sum of squared function values, but it has the advantage that second derivatives, which can be challenging to compute, are not required. The Gauss–Newton algorithm will be derived from Newton's method for function optimization via an approximation. As a consequence, the rate of convergence of the Gauss–Newton algorithm is at most quadratic. The recurrence relation for Newton's method for minimizing a function S of parameters, β is $\beta^{s+1} = \beta^{s} - H^{-1}g$ where g denotes the gradient vector of S and H the hessian matrix of S. Elements of the Hessian are calculated by differentiating the gradient $g_j = 2\sum_{i=1}^{m} r_i \frac{\partial r_i}{\partial \beta_i}$ with respect to B_k $H_{jk} = 2\sum_{i=1}^{m} \frac{\partial r_i}{\partial \beta_j}\frac{\partial r_i}{\partial \beta_k} + r_i \frac{\partial^2 r_i}{\partial \beta_i \partial \beta_k}$.

In order to apply the hessian matrix to the neural network the following methodology has been applied in the computer program

1. Through normal forward propagation the values of each nodes are calculated.

2. β_j is calculated for each node. For the output nodes β_j always takes the value 1. For the inner nodes it will take the value of $\beta^j = q_i \cdot (1-y^2)$ where q_i is again the weighted sum of the following betas (assuming there might be more than one output node) weighted by the connection's weight.

3. Two vectors are calculated for each dataset whereas each value depicts one weight of the network $J_n = \beta_n^j \cdot y_n$ and $G_n \beta_n \cdot y_n$. J in this case is recalculated for each dataset, while G is cumulatively added and averaged at the end of the dataset.

4. The Hessian Matrix is calculated for each dataset so that $H = J' \cdot J$ and is summed up with each dataset (as in Matlab the vector is a column, the multiplication needs to be $J' \cdot J$ and not vice versa).

5. The hessian matrix is summed up and averaged at the end of the batch. To avoid a singular matrix a small value is added to the diagonal of the Hessian after which the deltas can be calculated.

6. At the end of each epoque the delta is calculated in multiplying the G with the inverse hessian. (In Matlab this can be accelerated with $d = g/H$)

4.6 Levenberg-Marquardt

As described in Mark Hudson Beale [2010], similar to the quasi-Newton method, the Levenberg-Marquardt algorithm is designed to approach second-order training speed without having to compute the Hessian matrix. Levenberg-Marquardt can be thought of as a combination of steepest descent and the Gauss-Newton method. When the current solution is far from the correct one, the algorithm behaves like a gradient descent method: slow, but guaranteed to converge. When the current solution is close to the correct solution, it becomes a Gauss-Newton method.

The Hessian matrix is approximated as $\mathbf{H} = \mathbf{J^T J}$ and the gradient $\mathbf{g} = \mathbf{J}^T\mathbf{e}$ where $\mathbf{J}$ is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases and e is a vector of network errors. In vector calculus, the Jacobian marix is the matrix of all first-order partial derivatives of a vector- or scalar-valued function with respect to another vector. In our case it contains the first derivatives of the network errors with respect to the weights and biases.

$$\mathbf{J}(n) = \left.\begin{bmatrix} \frac{\partial e(1)}{\partial w_1} & \frac{\partial e(1)}{\partial w_2} & \frac{\partial e(1)}{\partial w_3} & \cdots & \frac{\partial e(1)}{\partial w_m} \\ \frac{\partial e(2)}{\partial w_1} & \frac{\partial e(2)}{\partial w_2} & \frac{\partial e(2)}{\partial w_3} & \cdots & \frac{\partial e(2)}{\partial w_m} \\ \frac{\partial e(3)}{\partial w_1} & \frac{\partial e(3)}{\partial w_1} & \frac{\partial e(3)}{\partial w_1} & \cdots & \frac{\partial e(3)}{\partial w_m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e(n)}{\partial w_1} & \frac{\partial e(n)}{\partial w_2} & \frac{\partial e(n)}{\partial w_3} & \cdots & \frac{\partial e(n)}{\partial w_m} \end{bmatrix}\right|_{w=w(n)}$$

The importance of the Jacobian lies in the fact that it represents the best linear approximation to a differentiable function near a given point. It is the derivative of a multivariate function. For a function of n variables, n $>$ 1, the derivative of a numerical function must be matrix-valued, or a partial derivative.Haykin [2009]

Finally, the weights are updated as follows:

$$x_{k+1} = x_k - \left(\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}\right)^{-1}\mathbf{J}^T\mathbf{e}$$

.

With a μ is zero the method coincides with Newton's method using the Hessian Matrix as approximation as shown in Chapter 4.5 and implemented manually in Matlab as shown in Chapter D.1.3. With a large μ the method become identical to gradient descent with a very small step size. In different positions on the error surface, one or the other method is at an advantage. Generally it is Newton's method that is faster and more accurate near an error minimum, which means a shifting towards that method is desired as quickly as possible. That's why μ is increased after each successful step.

4.7 Bayesian Regularization

Bayesian regularized neural networks have the advantage that they are difficult to overtrain, as an evidence procedure provides an objective criterion for stopping training. They are difficult are inherently insensitive to the architecture of the network, as long as a minimal architecture has been provided. It has also been shown mathematically that they do not need a test set, as they can produce the best possible model most consistent with the data (Burden and Winkler [2009]).

In our case the network training function updates the weight and bias values according to Levenberg-Marquardt optimization. Backpropagation is used to calculate the Jacobian jX of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to Levenberg-Marquardt,Mark Hudson Beale [2010].

jj = jX * jX

je = jX * E

dX = -(jj+I*mu) \ je

where E is all errors and I is the identity matrix. The adaptive value mu is increased by mu_inc until the change shown above results in a reduced performance value.

In bayesian regularization training not only aims to minimize the sum of squared errors but instead adds an additional term. The objective function becomes

$$F = \beta E_d + \alpha E_w$$

where $E_w$is the sum of squares of the network weights and $E_d$is the sum of the squared errors. $\alpha$and β are coefficients giving different weights to the optimization functions. There is a trade off between reducing the errors more emphasizing network size reduction, which will produce a smoother network response. It minimizes a combination of squared errors and weights, and then determines the correct combination so as to produce a network that generalizes well.

4.8 BFGS Quasi-Newton

As described in Schoenberg [2001], Sir Isaac Newton applied calculus was the optimization of a function. in observing that the derivative of a function being zero gave its extremum. The problem is that finding this extremum for non-quadratic functions is not always easy. Newton proposed an iterative solution in using a Taylor series approximation about some given point of the function's surface. If the function is quadratic, we arrive at the solution in a single step. If the function is not quadratic, we must solve it iteratively. While I don't want to go into further detail how the actual Newton method is working exactly, it is sufficient to understand that the inverse of the Hessian will determine the angle of the direction and the gradient. The issue comes from the fact that a function for computing an analytical Hessian is almost never available and that's why methods have been developed to compute it numerically.

The BFGS Quasi Newton method is a variation of Newton's optimization algorithm, in which an approximation of the Hessian matrix is obtained from gradients computed at each iteration of the algorithm.

As described in Mark Hudson Beale [2010] the BFGS Quasi-Newton method's backpropagation adjusts the weights and biases as $X = X + a \cdot dX$ where dX is the search direction. The parameter a is selected to minimize the performance along the search direction. A line search function is used to locate the minimum point while the first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed as $dX = -H/(gX)^{-1}$ where gX is the gradient and H is a approximate Hessian matrix. The Hessian matrix is defined as:

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

To compute the Hessian matrix is relatively complex and can take a lot of resources. Quasi-Newton methods avoid the calculation in updating an approximate Hessian matrix at each iteration. The update is computed as a function of the gradient. The BFGS is the quasi-Newton method which has been most successful.

The procedure is described in detail in Edwain K.P. Chong [2008].

$$H_k = H_{k-1} + \left(\frac{yy^T}{y^T s}\right) - \frac{H_{k-1} s s^T H_{k-1}}{s^T H_{k-1} s}$$

$$H_k^{-1} = \left(I - \frac{sy^T}{y^T s}\right) H_{k-1}^{-1} \left(I - \frac{ys^T}{y^T s}\right) + \frac{ss^T}{y^T s}$$

where $s = x^{(k)} - x^{(k-1)}$

Like other methods, the BFGS method can also get stuck on a saddle-point. In Newton's method, a saddle-point can be detected during modifications of the (true) Hessian. Therefore, search around the final point when using quasi-Newton methods.

4.9 Resilient Backpropagation (Rprop)

Neural networks usually use sigmoid or TanH transfer functions (activation functions in the hidden layer). As a side effect they sometimes compress a large input range into a limited output range. In both Sigmoid and TanH functions the slopes approach zero when the input gets large. This causes the gradient to have a very small magnitude and therefore weights and biases are changed only slightly from their original value.

The resilient backpropagation tries to mitigate these side effects of the partial derivatives. The principle works as follows: Only the sign of the derivative of the error function can determine the direction of the weight update while the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value which is decreased by a constant factor whenever the derivative with respect to that weight changes sign from the previous iteration. If the derivative is zero, the update value will remain the same. The effect is a decreasing oscillation that updates the weight as whenever an oscillation occurs, the weights change is reduce. If the weight continues to change in the same direction for several iterations, the magnitude of the weight change increases.

Each variable is adjusted according to the following: $dX = deltaX \cdot sign(gX)$ where the elements of $deltaX$ are all initialized to delta0, and gX is the gradient. At each iteration the elements of deltaX are modified.(Mark Hudson Beale [2010])

As described in Riedmiller [1994], for Faster Backpropagation Learning: The RPROP Algorithm, each weight has an individual update value $\triangle_{ij}$ which determines the up.

$$\triangle w_{ij}^{(t)} = \begin{cases} -\triangle_{ij}; \frac{\partial e}{\partial w_{ij}} > 0 \\ +\triangle_{ij}, \frac{\partial e}{\partial w_{ij}} < 0 \end{cases}$$

$$w_{ij}^{(t)} = w_{ij}^{(t)} + \triangle w_{ij}^{(t)}$$

If the partial derivative changes sign, meaning that the previous step was too large and the minimum was missed, the previous weight-update is reverted:

$$\triangle w_{ij}^{(t)} = -\triangle w_{ij}^{(t+1)}, \frac{\partial e}{\partial w_{ij}}^{(t+1)} \cdot \frac{\partial e}{\partial w_{ij}}^{(t)} < 0$$

4.10 One Step Secant

The term secant methods used in is reminiscent of the fact that derivatives are approximated by the secants through two function values although in many dimensions the function values here the function is the gradient do not determine uniquely the secant here the approximated Hessian.

Weights and biases are adjusted as follows in the backpropagation as described in Mark Hudson Beale [2010]:

$X = X + a * dX$ where dX is the search direction. The parameter a is selected to minimize the performance along the search direction. A line search function is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous steps and gradients, according to the following formula:

$dX = -gX + Ac * X_s tep + Bc * dgX$; where gX is the gradient, $X_s tep$ is the change in the weights on the previous iteration, and dgX is the change in the gradient from the last iteration.

4.11 Conjugate Gradient Algorithms

4.11.1 General notes

As described in Mark Hudson Beale [2010]the basic backpropagation algorithm adjusts the weights in the steepest descent direction (negative of the gradient), the direction in which the performance function is decreasing most rapidly. While the function decreases most rapidly along the negative of the gradient, it does not produce the fastest convergence.

In the conjugate gradient algorithms a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions. In most of the algorithms the learning rate η is used to determine the lenght of the weight update. In conjugate gradient algorithms the step size is adjusted at each iteration. A line search determines the step size that minimizes the performance function. Repeated line searches work well when the following condition is satisfied: $\frac{@^2 f}{@w_j @w_k} = 0$. The conjugate gradient method does line searches along the conjugate directions given by the eigenvectors of the Hessian.Haykin [2009]

In conjugate gradient methods and quasi-Newton methods the descent path is not determined by the steepest gradient and a line search routines have to be applied. The search routines have to be applied several times for each individual search, which makes the conjugate gradient methods time-consuming. As a positive factor the methods have a convergence which is usually much quicker as less iterations are needed. They become particularly interesting for large scale networks.

A bisection algorithm is the simplest version of a line search but has usually quite poor performance. The golden section search does not require calculation of the slope either. It just starts by determining an interval, beginning at a given point and extending the interval border in increasing steps until the minimum of the error function is bracketed. The position of the minimum is estimated by narrowing down the current interval. However, a combination of a golden section search and a quadratic interpolation between the intervals is much more effective. It will lead to an asymptotic convergence (Seiffert [2006]).

4.11.2 Scaled Conjugate Gradient

As described in Møller [1990], who developed the algorithm, ,the scaled conjugate gradient method distinguishes itself form the other methods in the fact that it chooses the search direction and step size more carefully by using information from the second order approximation $E(w+y) \approx E(w)+E'(w)^T y+\frac{1}{2}y^T E''(w)y$, where $E(w)$ denotes the error function in a given point $(w+y)$ in $\mathbb{R}^{\mathbb{N}}$.

The scaled conjugate gradient algorithm is based on conjugate directions, but this algorithm does not perform a line search at each iteration which can potentially safe a lot of time. The method may use more iterations than other conjugate gradient methods, but the amount of computations can be significantly reduced as no line search needs to be performed.

As shown in Mark Hudson Beale [2010] most of the optimization methods used to minimize functions are based on the same strategy. The minimization is a local iterative process in which an approximation to the function in a neighborhood of the current point in weight space is minimized. The approximation is often given by a first or second order Taylor expansion of the function. The error function in minimized as follows (Møller [1990]):

1. Choose initial weight vector w_1 and set k = 1
2. Determine a search direction p_k and a step size α_k so that $E(w_k+\alpha_k p_k) < E(w_k)$
3. Update vector: $w_{+1} = w_k + \alpha_k p_k$
4. If $E'(w_k) \neq 0$ then set $k = k+1$ and go to 2 else return $w_k + 1$ as the desired minimum

Through an iterative process the next point is determined. This takes two steps: In the first one the search direction is determined. It is decided in which direction the the search in the weight-space should be performed to find a new point. In a second step it has to be determined how far in the specified direction the search needs to go. If the search direction p_k is set to the negative gradient $-E'(w)$ and the step size α_k to a constant e, then the algorithm becomes the gradient descent algorithm as described in 4.2 on page 25.

4.11.3 Conjugate Gradient with Powell/Beale Restarts

This conjugate gradient method uses a line search to determine the minimum point. The first search direction is the negative of the gradient. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according the the following formula:

$dX = -gX+$ $dX^{old} \cdot Z$ where gX is the gradient. Z can be computed in different ways. The Powell-Beale variation of conjugate gradient is distinguished by two features. First, the algorithm uses a test to determine when to reset the search direction to the negative of the gradient. Second, the search direction is computed from the negative gradient, the previous search direction, and the last search direction before the previous reset. (Mark Hudson Beale [2010])

For all conjugate gradient algorithms, the search direction is periodically reset to the negative of the gradient. In the Powell/Beale restart method the restart occurs if there is very little orthogonality left between the current gradient and the previous gradient.

$$| g_{k-1}^{T} - g_k | \geq \theta \mid g_k |^2$$

Whenever this condition is met, the search condition is reset to the negative of the gradient.

4.11.4 Fletcher-Powell Conjugate Gradient

For the Fletcher-Reeves variation of conjugate gradient it is computed according to Z=normnew_sqr/norm_sqr; where norm_sqr is the norm square of the previous gradient and normnew_sqr is the norm square of the current gradient.

4.11.5 Polak-Ribiére Conjugate Gradient

For the Polak-Ribiére variation of conjugate gradient, it is computed according to Z = ((gX - gX_old)'*gX)/norm_sqr; where norm_sqr is the norm square of the previous gradient, and gX_old is the gradient on the previous iteration.

5 Implemented Matlab Classes

5.1 Neural network

In the simplest case we use a gradient descent method: One way to program the backpropagation algorithm with the flexibility of adjusting the size of the network is in creating a neuron class that handles all the functions itself and where the initial structure is set up through properties of the class itself. Once the Neurons have been instantiated, the addConnection method is called which sets the connections between the neurons and assigns them a certain type. All the different connections have to be defined.

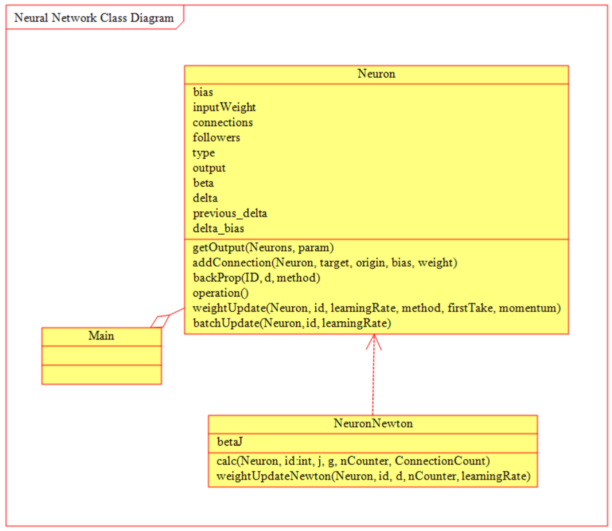


Figure 5.1: UML diagram of Neural Network training

The training can then start in looping through a procedure that first calls the forward propagation. The result is then compared to the desired output and a backpropagation method is called for all the neurons, first for the output layer, then for each of the neurons in the hidden layer. Once all the β and γ are calculated, the weights need to be updated. This is done through the weight updating method.

For the approximate Newton method an additional class (NeutonNeuron) is made available which inherits the Neuron class with all its functions. The full listing is provided in Chapter D.1 on page 70.

5.2 Simulator

The trading simulator is fully object oriented and flexible to assess any trading strategy in changing different parameters and making it easy to assess the outcome through a 3-dimensional graphical depiction. It consists of a variety of classes which an overview is given in Fig. 5.2.

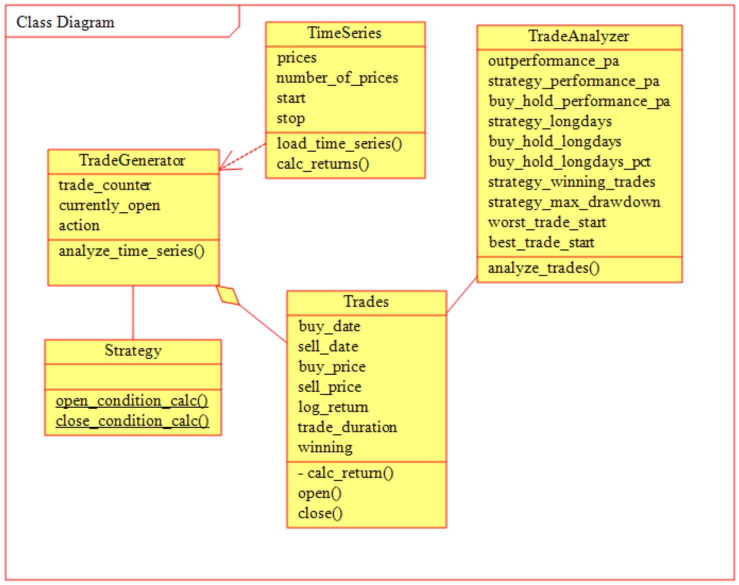


Figure 5.2: UML diagram of trading simulator

The procedure of the simulation is as follows: At first a time series is loaded, this is done in the TimeSeries class as listed in Chapter D.2.2 on page 85. Depending on whether only one series (in the case of an index) or multiple series when various stocks are loaded as in the third example that is presented below. In a next step the trades are generated, based on the strategy that is applied. This is done through the TradeGenerator class and its subclass, which is always a predefined strategy. In our case, the strategy class is generating an output value in the neural network which is then used to make a buying and selling decision. Once all the trades are generated (which each having properties such as buying price, selling price and return, etc), the trades need to be bundled and analyzed. This is the purpose of the trade analyzer class: returns of all the trades are averaged and summed up and a great variety of analysis is done. The different metrics that are calculated are described in more detail in Chapter 2.4.3 on page 14.

Once all the different metrics are stored, they have to be graphically displayed. This is done by the TradeGraphs class (see Chapter D.2.5 on page 96 for a full listing of the class). The difficulty there lies to understandably depict the 3d Arrays that were generated by the TradeAnalyzer class. This is best done with 3d mesh diagrams where all the results are displayed as a function of varying parameters (buying and selling thresholds). In the 3d graphs it is relatively easy to spot the best trading strategy, which can then be analyzed in more detail.

Finally it is always of interest for any strategy to see the actual performance in a simulated scenario, so that the day-to-day behaviour of the hypothetical fund can actually be observed.

6 Empirical results of neural network strategies

6.1 Overview

As outlined earlier it is questionable whether trying to make predictions based on price only are a meaningful method of forecasting prices. But to outline a procedure of how such a testing would have to be done I present some examples that are simply based on past prices. In a first step the neural network is trained according to a certain rule and in a second step the results are simulated and assessed.

In total there are 3 examples: the first two examples are trying to make predictions based on past prices only. The first one focuses on the following 3 days as a whole, assuming that it might be inefficient to try to predict what happens the very next day. The second example focuses on the following day only. Both of them are trading the Dow Jones Industrial index. The third example goes beyond only looking at past prices of an index and incorporates financial factors from different companies to decide, which stock should be bought and sold. The neural network then decides how a given amount of money should be distributed among 100 different stocks, depending on how the financial factors, which are used as input neurons, evolve.

6.2 Expected sum of returns of the next 3 days

6.2.1 Setup

In this setup the input neurons represent the returns of the last 5 days and the output neuron is the sum of the returns of the next 3 days. In a first step we need to train the neural network with all available backpropagation algorithms, so we can decide which was is delivering the best results. Once this is done, the neural network is saved so we can apply it for further testing.

Buying and selling signals are triggered with the binary activation function when the output neuron exceeds the parameter $\theta^b$and $\theta^s$for the buying and selling thresholds. The optimal thresholds are decided through a simulation and by trial and error. The results of the simulation are presented in Fig. A.15 on page 53.

As described in [Rojas, 1996] it is of advantage to use a bipolar network instead of a binary one: “Many models of neural networks use bipolar, not binary, vectors. In a bipolar coding the value 0 is substituted by -1. This change does not affect the essential properties of the perceptrons, but changes the symmetry of the solution regions. It is well known that the algebraic development of some terms useful for the analysis of neural networks becomes simpler when bipolar coding is used.”

6.2.2 Results

Sharp Ratio for Dow Jones of the given period	1.1786
Sharp Ratio for simulated Fund	1.3515
peakToTroughLogFund	-23.7709
peakToTroughLogIndex	-23.7709
Statistical significance of difference	p =0.1759
Outperformance p.a.	0.2062
Buy and hold performance p.a.	14.2208
Strategy performance p.a.	14.4269
Buy and hold longdays	3650
Strategy Longdays	3611

Table 2: Strategy 1 Summary

The results of the trained networks can be seen in Chapter A.14 on page 52 and the actual results of the simulation are printed in Chapter A.15 on page 53. While the above results show that the neural network is not outperforming the market as such, the good news is that the sharp ratio is actually higher. It seems that the neural network is able to avoid certain periods of high volatility.

6.2.3 Statistical significance of out of sample test

Outperformance p.a.	-0.6843
Buy and hold performance p.a.	-1.9609
Strategy performance p.a.	-2.6451
Buy and hold longdays	3225
Strategy Longdays	3208

Table 3: Strategy 1 Summary (out of sample)

The out of sample results are shown in chapter A.16 on page 54.

6.3 10 past days as input predicting next day

6.3.1 Setup

Similar to the above example, we first train the network. The difference to the method user in Chapter 6.2 is that we are using different input and output neurons. As input neurons we go further back: using the returns of the last 10 days as input neurons. As output neuron on the other hand, we only use the return on of the following day.

Once the neural network has been trained with all the different backpropagation methods, we decide which one was most successful. After that we can apply the neural network in a testing environment and try to find out, at which threshold level buying and selling signals should be triggered. They are triggered with the binary activation function when the output neuron exceeds the parameter $\theta^b$and $\theta^s$for the buying and selling thresholds. The results of the simulation are presented in Fig. B.14 on page 62.

6.3.2 Results

Sharp Ratio for Dow Jones of the given period	1.19
Sharp Ratio for simulated Fund	1.22
peakToTroughLogFund	-15.2
peakToTroughLogIndex	-22.7
Statistical significance of difference	p =0.63
Outperformance p.a.	0.71
Buy and hold performance p.a.	14.3
Strategy performance p.a.	15.0
Buy and hold longdays	3650
Strategy Longdays	3388
Number of trades	93

Table 4: Strategy 2 Summary

6.4 High frequency trading example: multiple financial factors as input neurons

6.4.1 Setup

This setup distinguishes itself that it is not simply relying on past prices of the securities but it takes additional company-specific information into account. In this example we no longer examine an index of securities, but we trade the actually securities themselves. The sample consists of 100 securities of the S&P100, for a timeframe of 1980 to 2008, subject to data availability (when one of the factors is not yet available, the period for that stock is ignored). The neural network determines how a given amount of money (starting with e.g. $1000) is distributed among the 100 securities. The neural network is trained in a way that the 3 input neurons will generate one output signal. If the output signal increases relative to the previous day, then this is considered a buying signal. If the output signal decreases to the previous day, it is considered a selling signal.

When the neural network is trained, the output neuron is connected to the next day's return of the respective security. If it is relatively high, it will end up training the network to buy at this point and if it is low, the network will be trained to sell.

The following factors are fed as input neurons into the neural networks:

1. Change in payout ratio over the last day: The payout ratio of a security depicts the portion of net income of the company that is distributed as

a dividend to stockholders. There is no consensus on whether this factor has influence on future stock price development. Nevertheless, traders

7 Conclusion

7.1 Summary of results

Similar to Malikel [1973] view of Technical analysis, it is not possible with a simple neural network to outperform the market when the input neurons are based on price only. In addition to that the simulations have shown that neural networks are unable to outperform the market for an extended period of time when input neurons are connected to p/e, p/b and payout ratios. However in one case an increase in the sharp ratio has been observed.

The different training algorithms have shown relatively large divergence in how they manage to reduce the error. Generally, second order training methods have been most successful while the gradient descent method is clearly not suitable for these kind of large scale problems.

Nevertheless, it is not the backpropagation algorithm which makes a difference whether the market can be outperformed or not, it is much more the specification of the model itself (i.e. the input neurons and the definition of the output neuron) that are important. If the input factors have no significant connection to the output factors, there is no use in training the network.

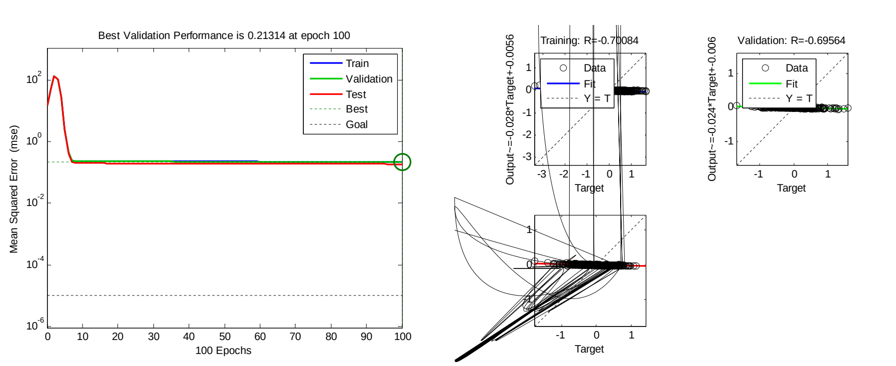
7.2 Suggested future research

In this thesis it was tested whether neural networks could outperform the market over an extended period of time in using past prices and some financial ratios. While there was no restriction on when the trades should be made, the used dataset was using end of day prices. The trades were thus limited on a maximum one one trade per day.

In future research it might make sense to loosen this restriction and allow multiple trades per day. This would require to significantly enhance the dataset: using order book data instead of only actual last prices and use them as input neurons might give a better understanding about the current psychology of the market. Research in this area is relatively limited because datasets are not readily available in information systems such as Bloomberg or Factset. It is likely that when using this additional information and testing the success of a trained neural network based on order book information for a relatively short period of time, success rates could be higher than what is presented in the context of this thesis.

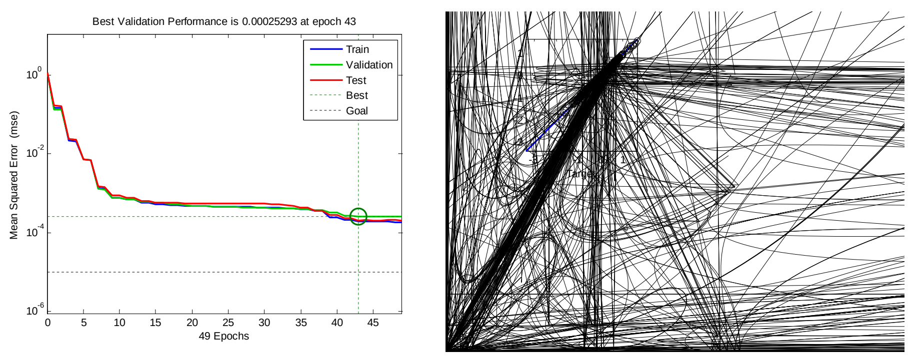
In addition it would be interesting to explore different asset classes. The equity market is generally researched relatively well. Neural networks could find better application when used in connection with derivatives or potentially also forex products, where data is less prone to be contaminated by accounting standards as it is the case in the equity market. In addition it would be interesting to include a quantified form of general newsflow. This would be especially interesting in connection with high frequency trading and detailed information about order books of individual securities. Since this information is not readily available, market inefficiencies are much more likely to surface.

A Figures for strategy 1: 3-day forecast based on past prices



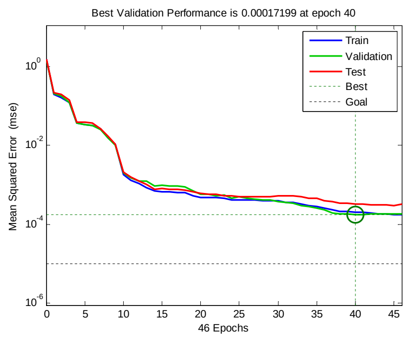
(a) MSE (b) Regre ion

Figure A.1: Gradient Descent (Trained network for DJ Industrial's returns 1990-2000)



(a) MSE (b) Regre ion

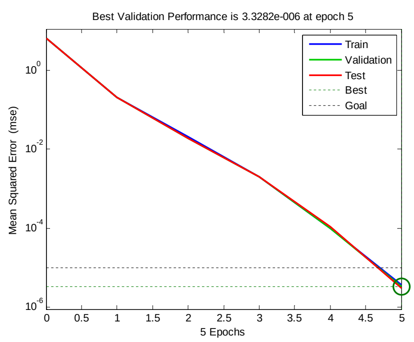
Figure A.2: Scaled conjugate gradient (Trained network for DJ Industrials returns 1990-2000)



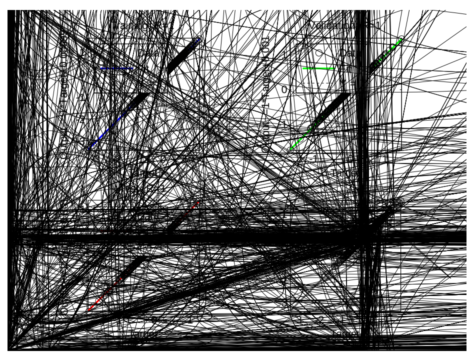
(a) MSE

(b) Regre ion

Figure A.3: BFGS quasi Newton (Trained network for DJ Industrials returns 1990-2000)

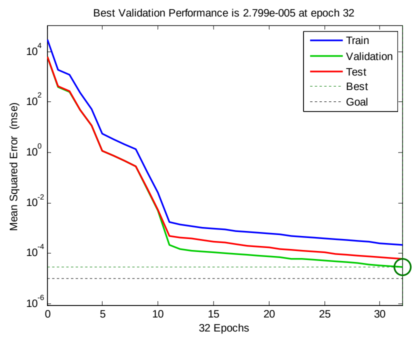


(a) MSE

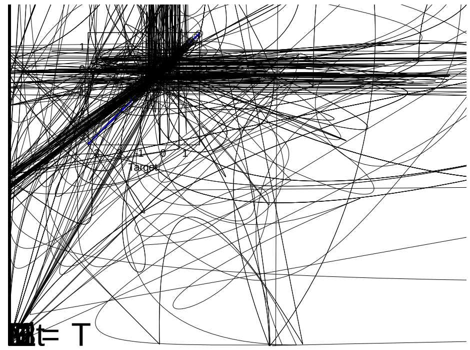


(b) Regre ion

Figure A.4: Levenberg-Marquardt (Trained network for DJ Industrials returns 1990-2000)

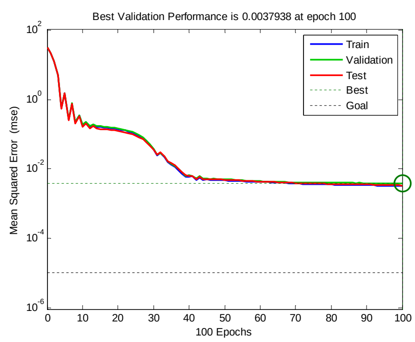


(a) MSE

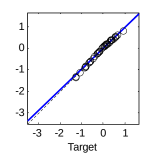


(b) Regre ion

Figure A.5: Bayesian Regularization (Trained network for DJ Industrials returns 1990-2000)

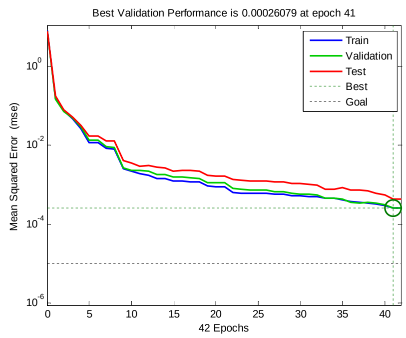


(a) MSE



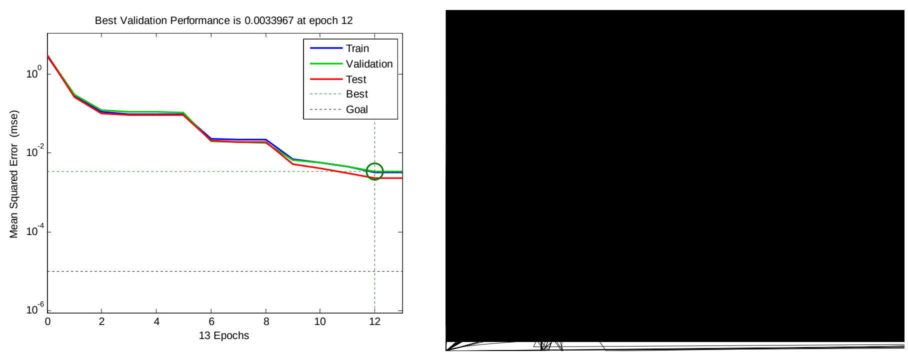
(b) Regre ion

Figure A.6: Resilient Backpropagation (Trained network for DJ Industrials returns 1990-2000)



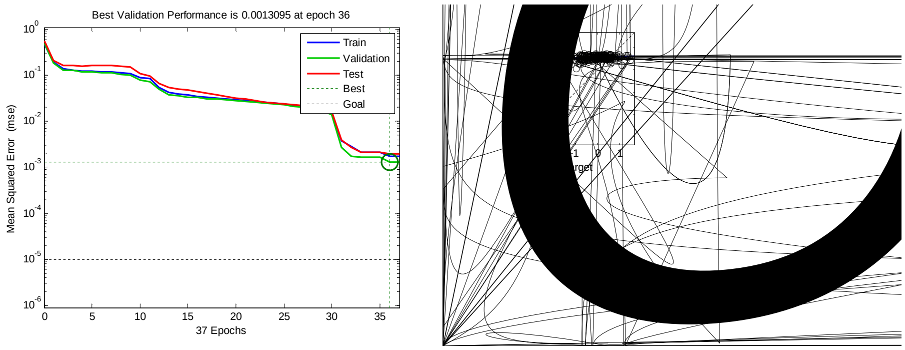
(a) MSE (b) Regre ion

Figure A.7: Conjugate Gradient with Powell/Beale Restarts (Trained network for DJ Industrials returns 1990-2000)



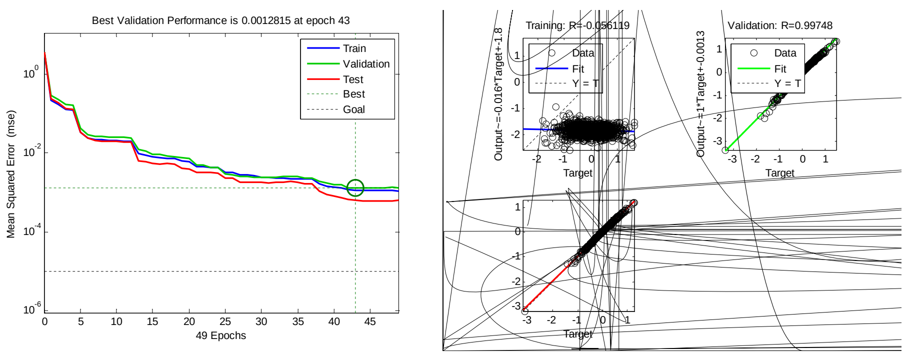
(a) MSE (b) Regre ion

Figure A.8: Fletcher-Powell Conjugate Gradient (Trained network for DJ Industrials returns 1990-2000)



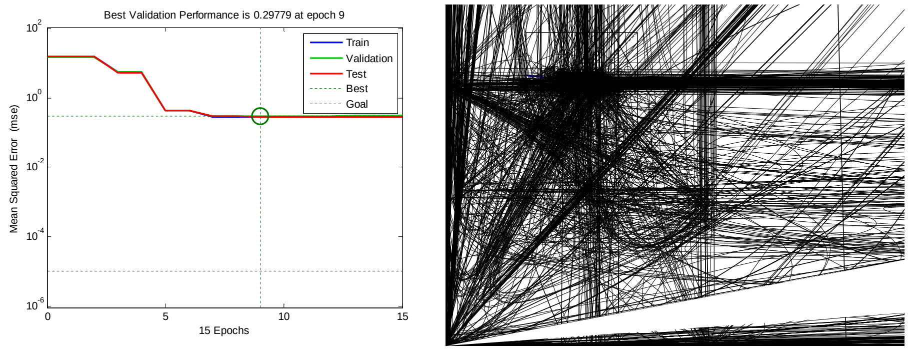
(a) MSE (b) Regre   ion

Figure A.9: Polak-Ribiére Conjugate Gradient (Trained network for DJ Industrials returns 1990-2000)



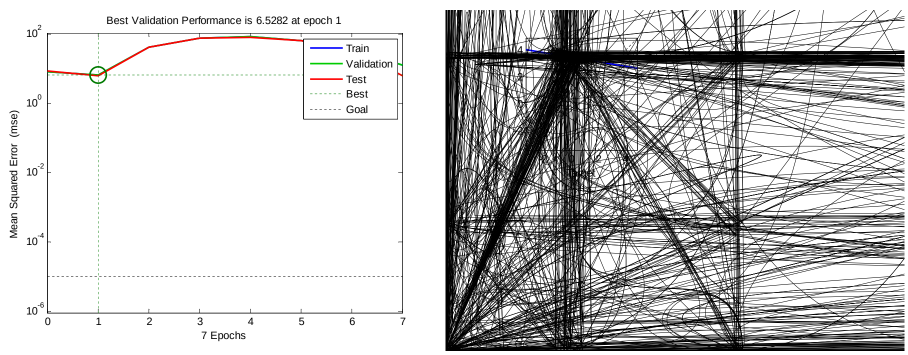
(a) MSE (b) Regre   ion

Figure A.10: One Step Secant (Trained network for DJ Industrials returns 1990-2000)



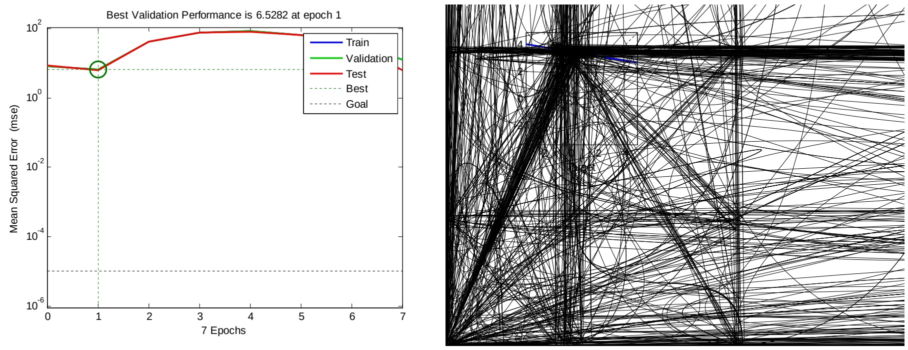
(a) MSE (b) Regre ion

Figure A.11: Variable Learning Rate Gradient Descent (Trained network for DJ Industrials returns 1990-2000)



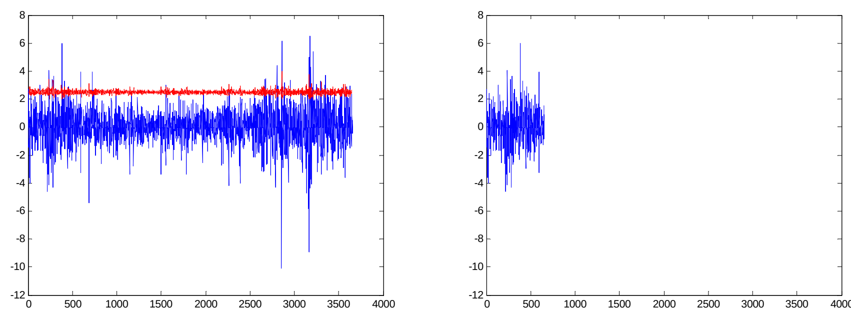
(a) MSE (b) Regre ion

Figure A.12: Gradient Descent with Momentum (Trained network for DJ Industrials returns 1990-2000)



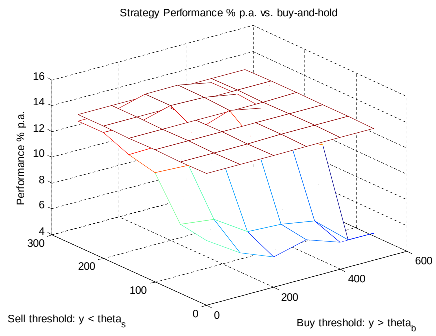
(a) MSE (b) Regre ion

Figure A.13: Gradient Descent (Trained network for DJ Industrials returns 1990-2000)

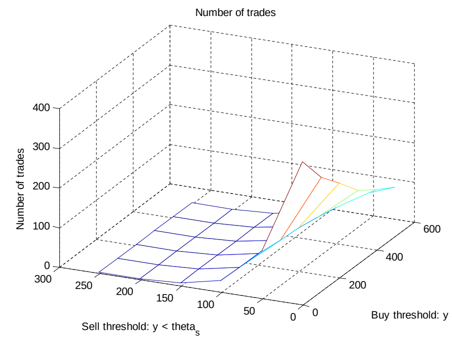


(a) Gradient De cent with Momentum (b) Baye ian Regularization

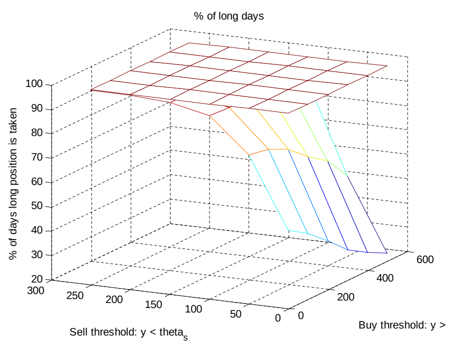
Figure A.14: Trained network for DJ Industrials returns 1990-2000, log returns, comparison between two methods



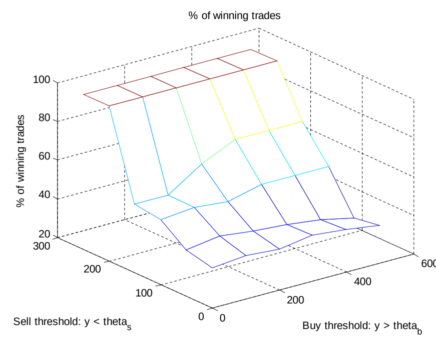
(a) Strategy return p.a.



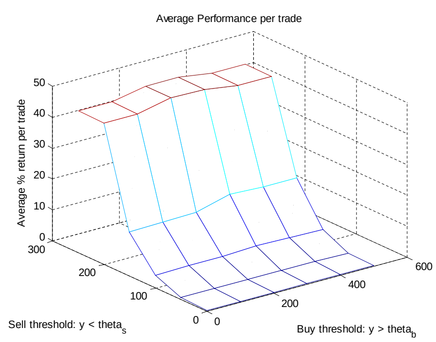
(b) Number of trade with corre ponding trat-egy



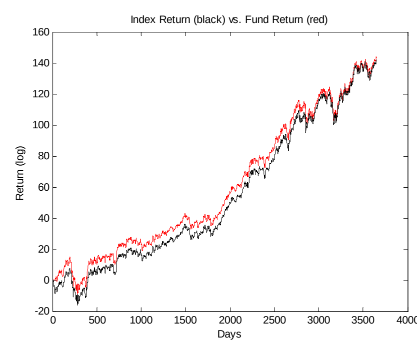
(c) % of day long



(d) % of winning trade

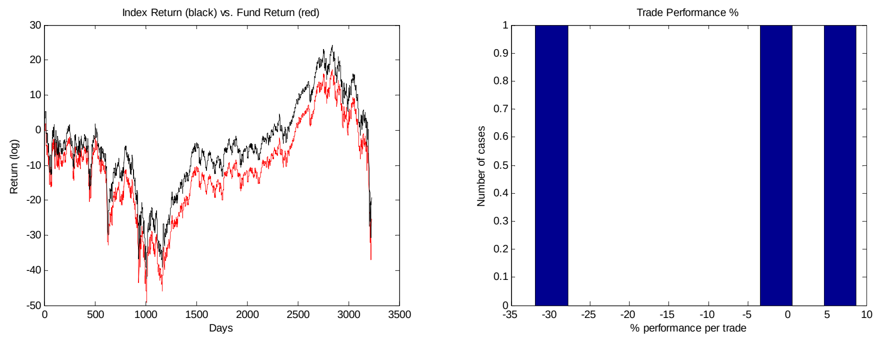


(e) Average performance per trade



(f) Strategy performance a hypothetical fund

Figure A.15: Simulated results of 3-day return forecast with neural network

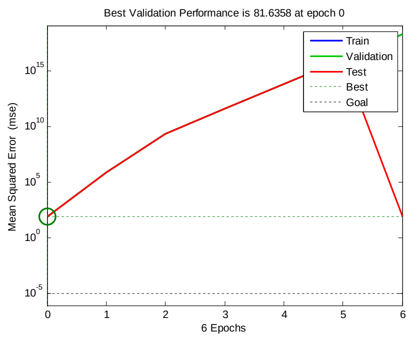


(a) Fund performance

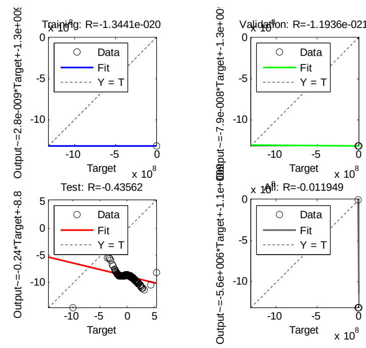
(b) Trade hi togram

Figure A.16: Simulated results of 3-day return forecast with neural network - out of sample application

B Figures for Strategy 2: 1 day price forecast based on past prices

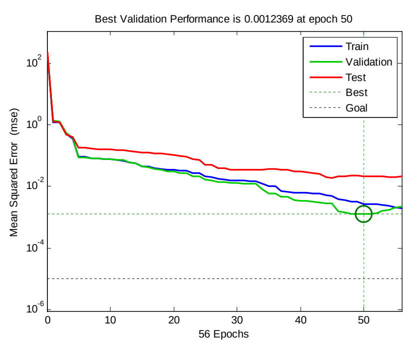


(a) MSE

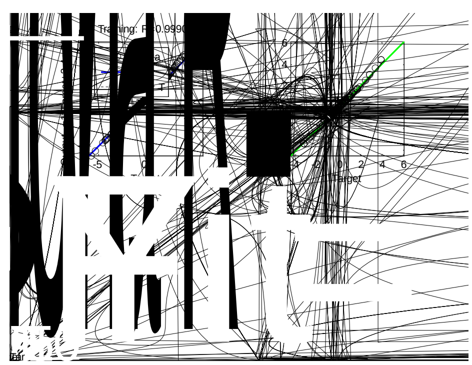


(b) Regre ion

Figure B.1: Gradient Descent (Trained network for DJ Industrials returns 1990-2000)

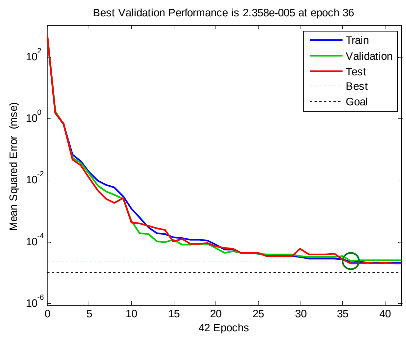


(a) MSE

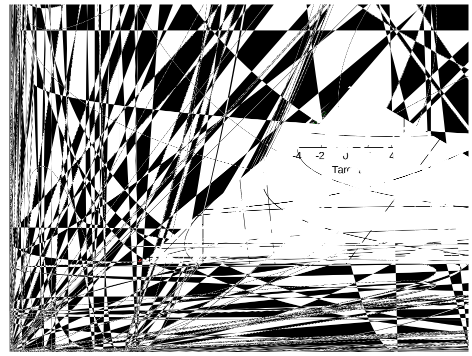


(b) Regre ion

Figure B.2: Scaled conjugate gradient (Trained network for DJ Industrials returns 1990-2000)

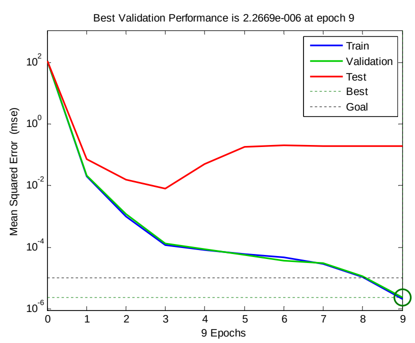


(a) MSE

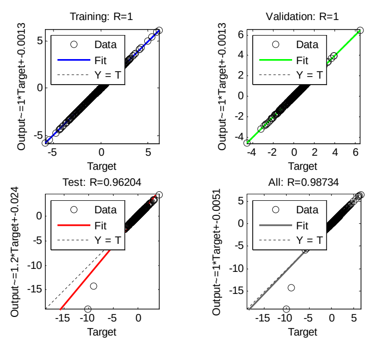


(b) Regre ion

Figure B.3: BFGS quasi Newton (Trained network for DJ Industrials returns 1990-2000)

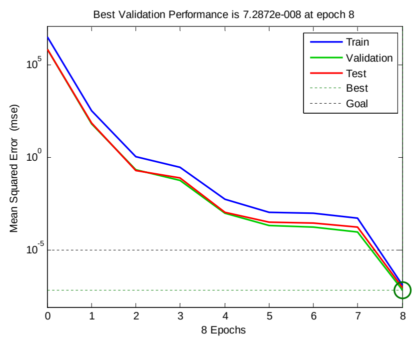


(a) MSE

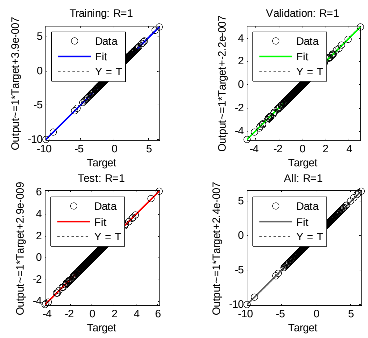


(b) Regre ion

Figure B.4: Levenberg-Marquardt (Trained network for DJ Industrials returns 1990-2000)

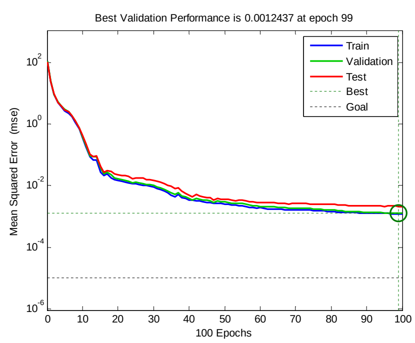


(a) MSE

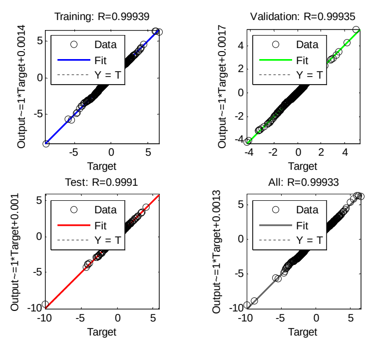


(b) Regre ion

Figure B.5: Bayesian Regularization (Trained network for DJ Industrials returns 1990-2000)

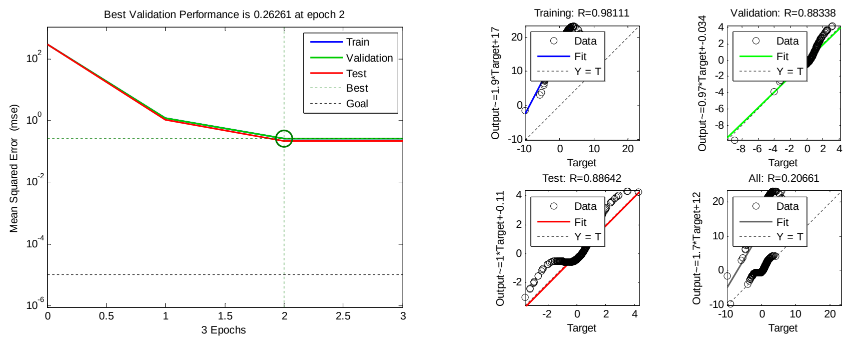


(a) MSE



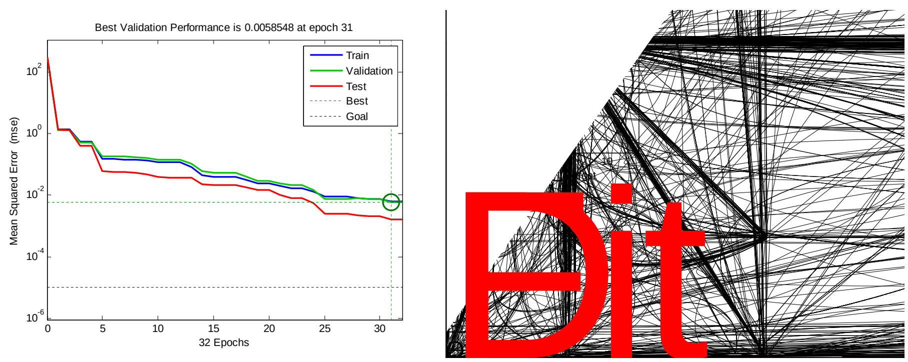
(b) Regre ion

Figure B.6: Resilient Backpropagation (Trained network for DJ Industrials returns 1990-2000)



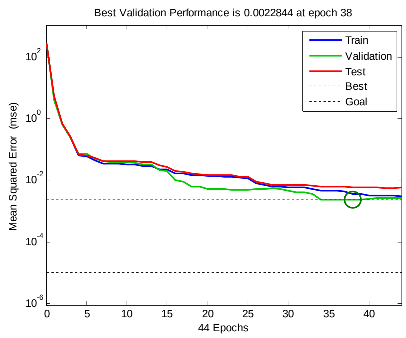
(a) MSE (b) Regre ion

Figure B.7: Conjugate Gradient with Powell/Beale Restarts (Trained network for DJ Industrials returns 1990-2000)

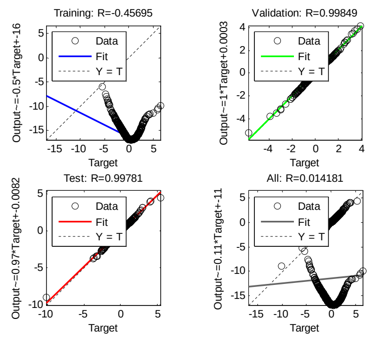


(a) MSE (b) Regre ion

Figure B.8: Fletcher-Powell Conjugate Gradient (Trained network for DJ Industrials returns 1990-2000)

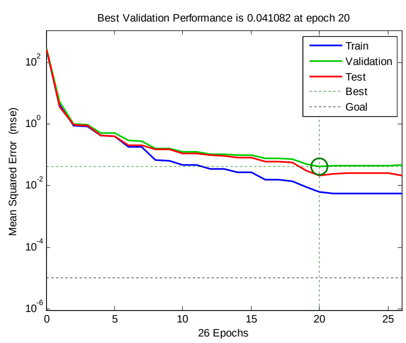


(a) MSE

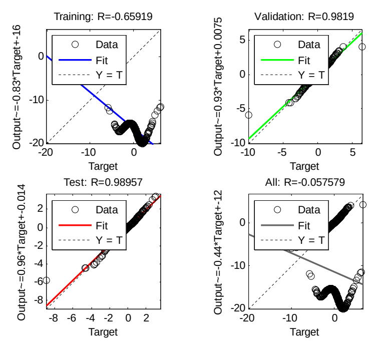


(b) Regre ion

Figure B.9: Polak-Ribiére Conjugate Gradient (Trained network for DJ Industrials returns 1990-2000)

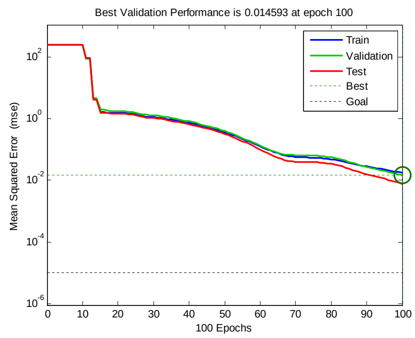


(a) MSE

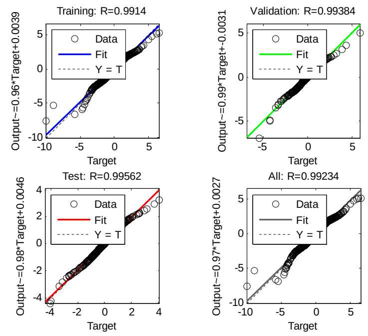


(b) Regre ion

Figure B.10: One Step Secant (Trained network for DJ Industrials returns 1990-2000)

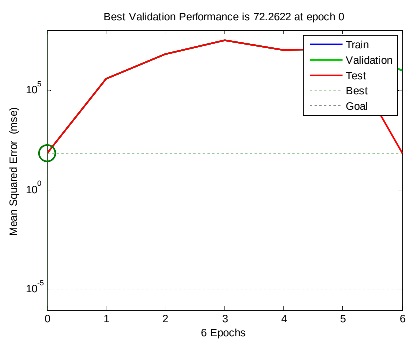


(a) MSE

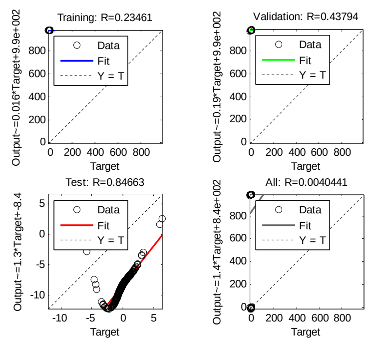


(b) Regre ion

Figure B.11: Variable Learning Rate Gradient Descent (Trained network for DJ Industrials returns 1990-2000)

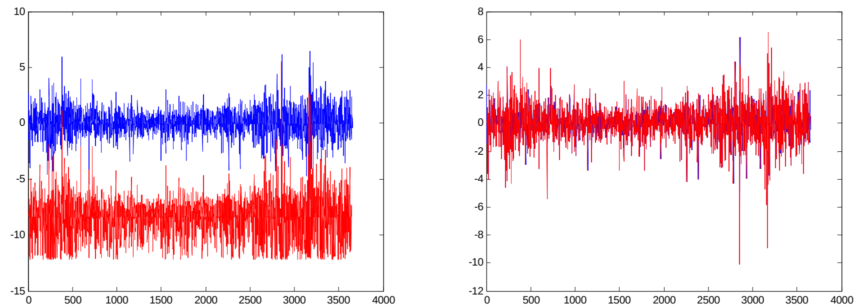


(a) MSE



(b) Regre ion

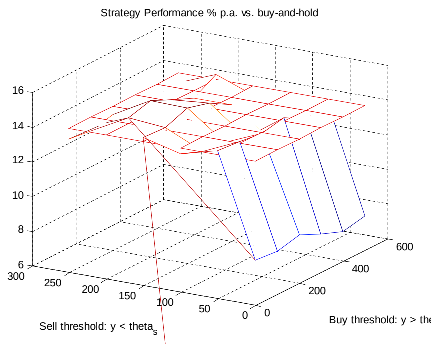
Figure B.12: Gradient Descent with Momentum (Trained network for DJ Industrials returns 1990-2000)



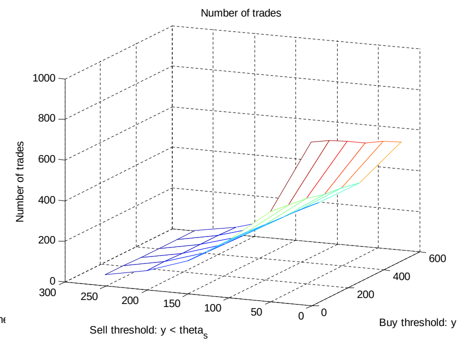
(a) Gradient De cent with Momentum

(b) Baye ian Regularization

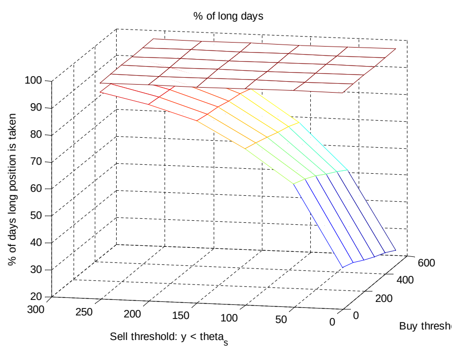
Figure B.13: Trained network for DJ Industrials returns 1990-2000, log returns, comparison between two methods



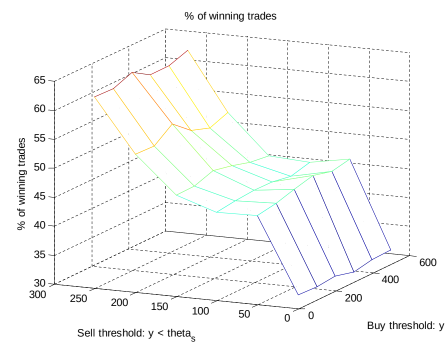
(a) Strategy return p.a.



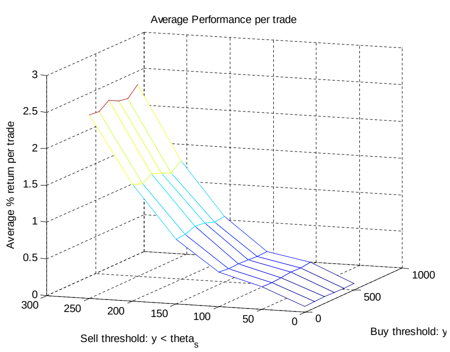
(b) Number of trade with corre ponding trat- egy



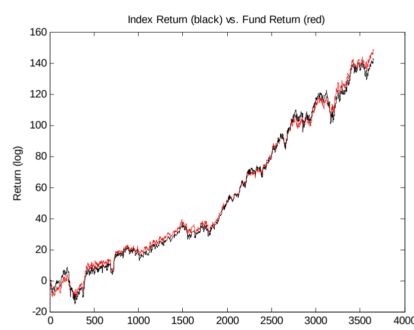
(c) % of day long



(d) % of winning trade



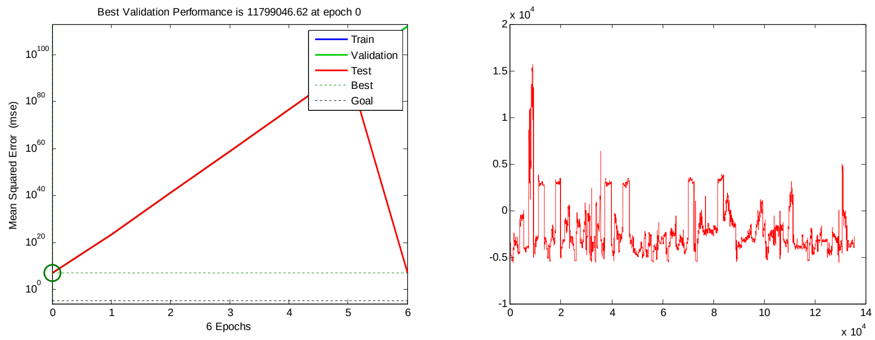
(e) Average performance per trade



(f) Strategy performance a hypothetical fund

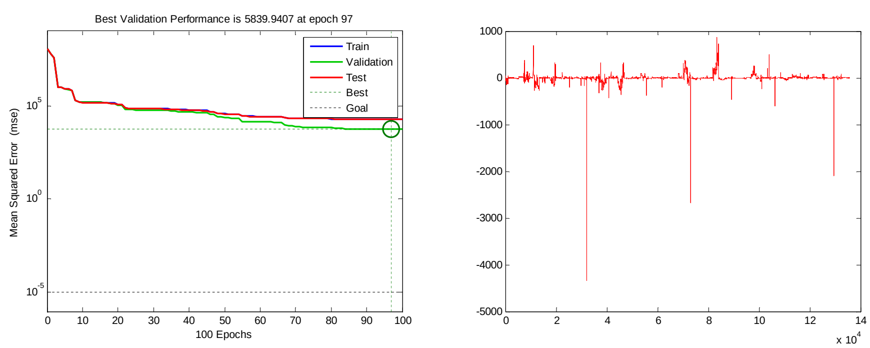
Figure B.14: Simulated results of 1-day return forecast with neural network

C Figures for Strategy 3: forecast based on financial factors



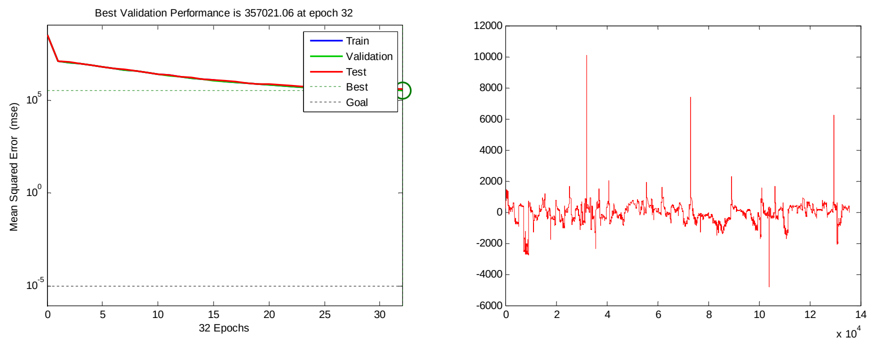
(a) MSE (b) Output ignal

Figure C.1: Gradient Descent (Trained network for DJ Industrials returns 1990-2000)



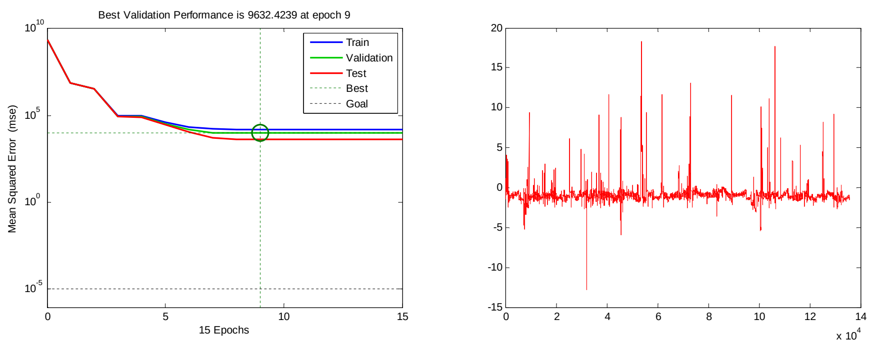
(a) MSE (b) Network output

Figure C.2: Scaled conjugate gradient (Trained network for DJ Industrials returns 1990-2000)



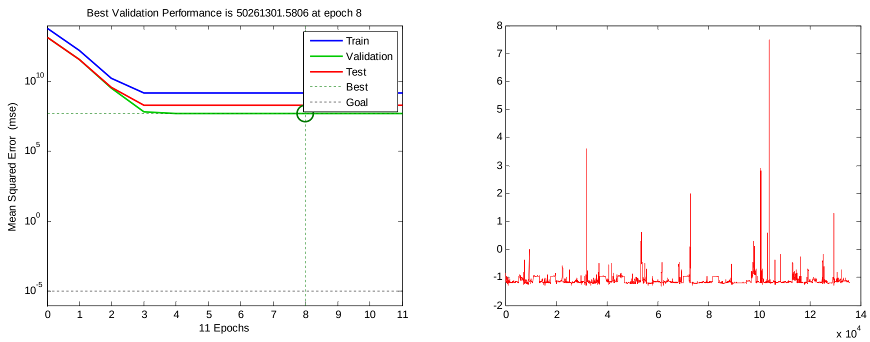
(a) MSE (b) Network output

Figure C.3: BFGS quasi Newton (Trained network for DJ Industrials returns 1990-2000)



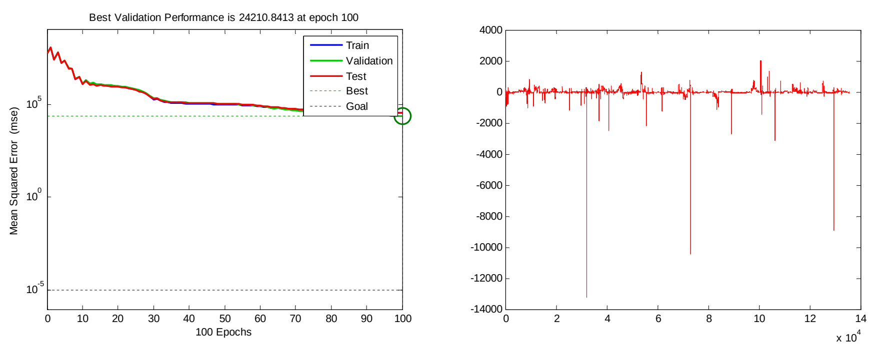
(a) MSE (b) Network output

Figure C.4: Levenberg-Marquardt (Trained network for DJ Industrials returns 1990-2000)



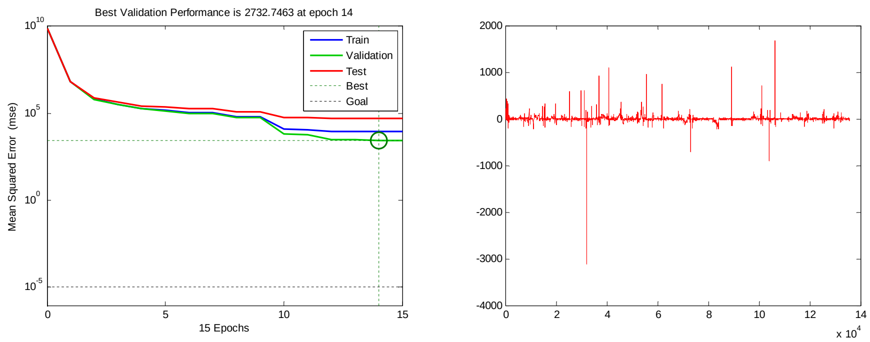
(a) MSE (b) Network output

Figure C.5: Bayesian Regularization (Trained network for DJ Industrials returns 1990-2000)



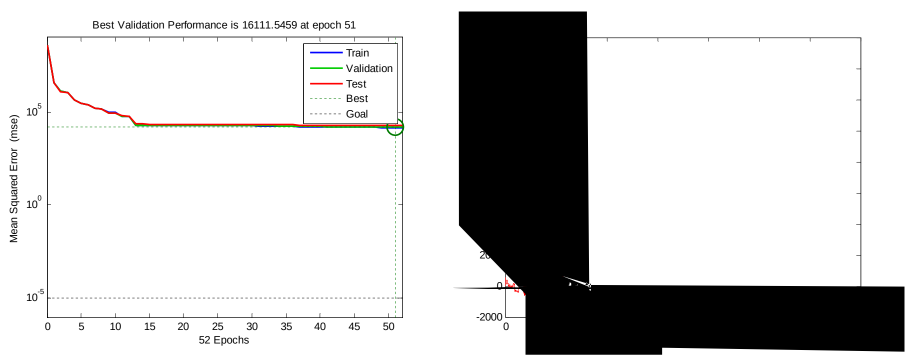
(a) MSE (b) Network output

Figure C.6: Resilient Backpropagation (Trained network for DJ Industrials returns 1990-2000)



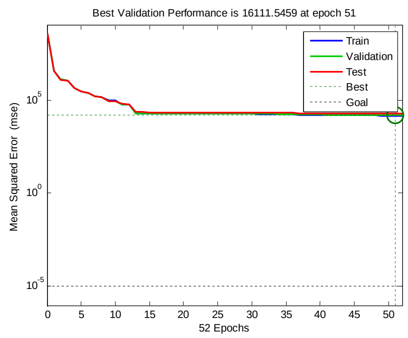
(a) MSE (b) Regre ion

Figure C.7: Conjugate Gradient with Powell/Beale Restarts (Trained network for DJ Industrials returns 1990-2000)

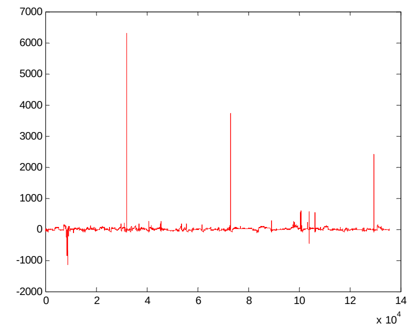


(a) MSE (b) Network output

Figure C.8: Fletcher-Powell Conjugate Gradient (Trained network for DJ Industrials returns 1990-2000)

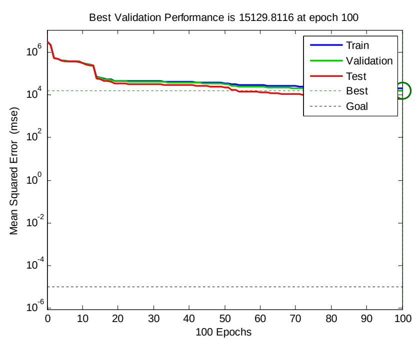


(a) MSE

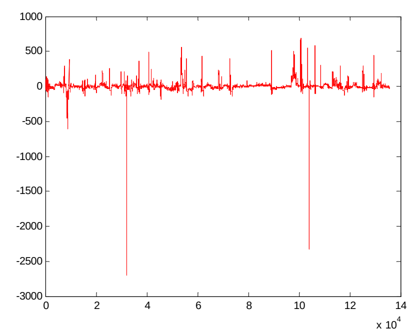


(b) Network output

Figure C.9: Polak-Ribiére Conjugate Gradient (Trained network for DJ Industrials returns 1990-2000)

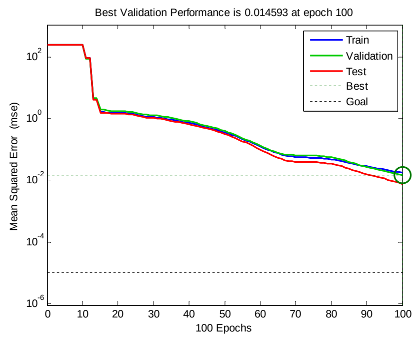


(a) MSE

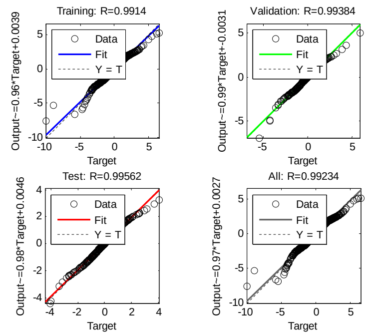


(b) Network output

Figure C.10: One Step Secant (Trained network for DJ Industrials returns 1990-2000)

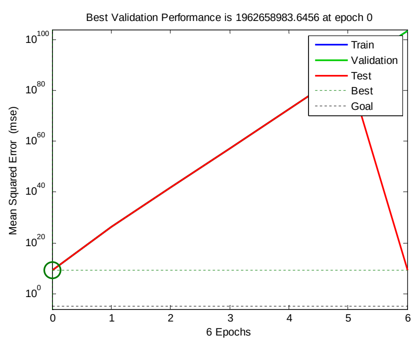


(a) MSE

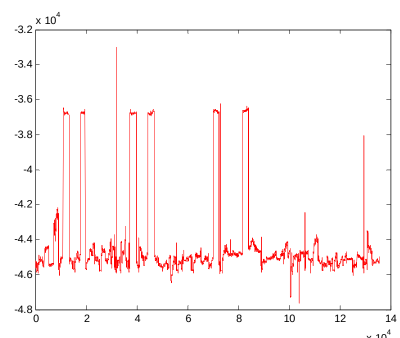


(b) Regre ion

Figure C.11: Variable Learning Rate Gradient Descent (Trained network for DJ Industrials returns 1990-2000)

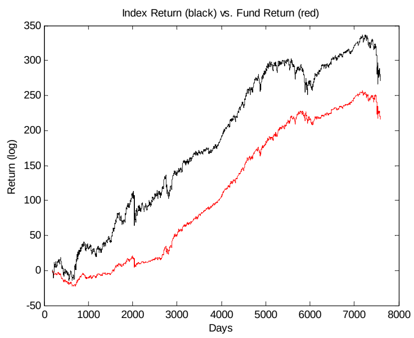


(a) MSE

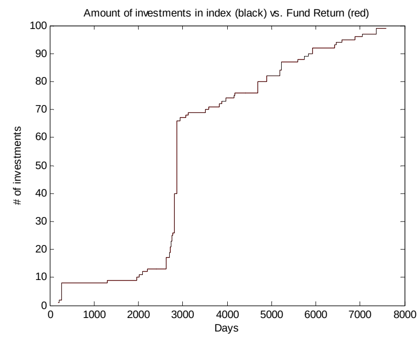


(b) Network output

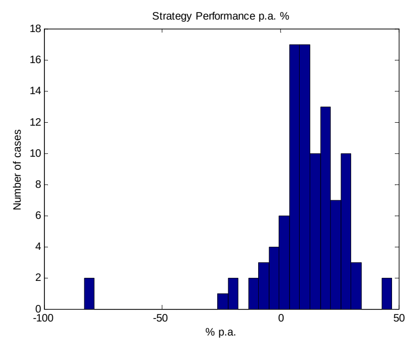
Figure C.12: Gradient Descent with Momentum (Trained network for DJ Industrials returns 1990-2000)



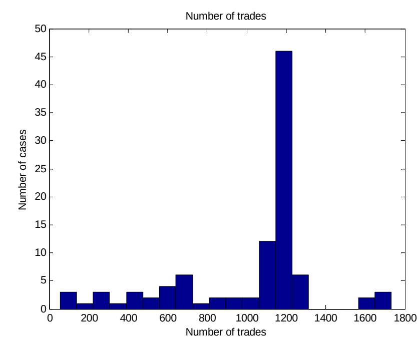
(a) Simulated fund performance (red) v . buy-and-hold performance



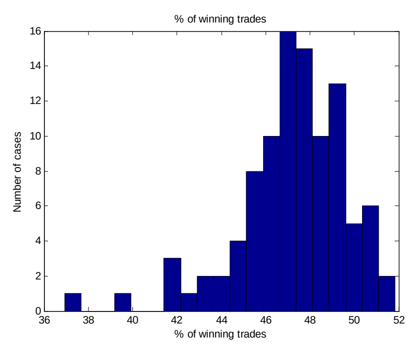
(b) Amount of long po ition in different tock of the S&P100



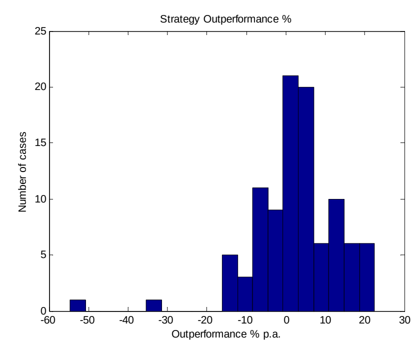
(c) Hi togram with different tock - outperformance of trategy



(d) Hi togram howing number of trade with each tock



(e) Hi togram with % of winning trade



(f) Strategy outperformance

Figure C.13: Strategy results with financial factors as input neurons

D Program Listings

D.1 Neural Network

D.1.1 Neural Network Main program (training) [training.m]

```
clear;
clear n;
method=4; % 0=sigmoid, 1=tanh, 2=linear, 4=Newton method
%output node always uses linear

learningRate=0.15;
normalize=0;
first_input=1;
last_output=8;

for x=1:8
    n(x)=NeuronNewton;
end

% node connections
%n=n.addConnection(3, 1, 1.2, -0.1); % target, origin, weight, bias
%n=n.addConnection(3, 2, 1.25,-0.1);
%n=n.addConnection(4, 1, 1.3, -0.15);
%n=n.addConnection(4, 2, -0.1,-0.15);
%n=n.addConnection(5, 3, -0.1, 0.05);
%n=n.addConnection(5, 4, 0.2,  0.05);

% Nicolas Coursework
%n=n.addConnection(3, 1, 0.23, 0.0); % target, origin, weight, bias
%n=n.addConnection(4, 2, 0.14, 0.0);
%n=n.addConnection(5, 2, 0.3,  0.0);
%n=n.addConnection(5, 1, -0.35,  0.0);
%n=n.addConnection(6, 3, 0.1, 0.0);
%n=n.addConnection(6, 4, -0.2, 0.0);
%n=n.addConnection(6, 5, -0.4,  0.0);
%n=n.addConnection(6, 1, 0.3,  0.0);

% Irati coursework
n=n.addConnection(3, 1, 0.14, 0.0); % target, origin, weight, bias
n=n.addConnection(4, 2, -0.3, 0.0);
n=n.addConnection(5, 2, -0.1,  0.0);
n=n.addConnection(5, 1, 0.5,  0.0);
```

```
n=n.addConnection(6, 3, 0.4, 0.0);
n=n.addConnection(6, 4, -0.4, 0.0);
n=n.addConnection(6, 5, -0.15,  0.0);
n=n.addConnection(6, 1, -0.23,  0.0);


% define structure (type=0: input node, type=1: hidden node, type=2: output
% node
n(1).type=0;
n(2).type=0;
n(3).type=1;
n(4).type=1;
n(5).type=1;
n(6).type=2;




for i=1:1

% input nodes
n(1).output=1;%round(rand(1));  %input values for input nodes
n(2).output=0;%round(rand(1));

endvalue=xor( n(1).output , n(2).output ); %output node correct result


% generate output for forward propagation
n(3)=n(3).getOutput(n,0);
n(4)=n(4).getOutput(n,0);
n(5)=n(5).getOutput(n,0);
n(6)=n(6).getOutput(n,2);


% backward propagation
n=n.backProp(6,endvalue,2);
n=n.backProp(5,endvalue,0);
n=n.backProp(4,endvalue,0);
n=n.backProp(3,endvalue,0);

if method<4 % MPL method
    [n]=n.weightUpdate(6,learningRate);
    [n]=n.weightUpdate(5,learningRate);
    [n]=n.weightUpdate(4,learningRate);
```

```
    [n]=n.weightUpdate(3,learningRate);

elseif method==4 % Newton method
% weight updating
[n,j]=n.weightUpdate(6,j);
[n,j]=n.weightUpdate(5,j);
[n,j]=n.weightUpdate(4,j);
[n,j]=n.weightUpdate(3,j);


    %calculate hessian
    H=j*j';
    h=1/H;
    %update weights
[n]=n.weightUpdateNewton(6,h);
[n]=n.weightUpdateNewton(5,h);
[n]=n.weightUpdateNewton(4,h);
[n]=n.weightUpdateNewton(3,h);

end


%result(i)=round(n(5).output)==endvalue;

end

for nx=3:6
    node=nx %output result
    n(nx).inputweight
end


%figure(1)
%plot((n(6).deltaHistory).^2)
%figure(2)
%plot (result);
```

D.1.2 Neuron Class [Neuron.m]

```
classdef Neuron
    %Neuron for neuronal network
    %

    properties
        bias
```

```
            neuron(origin).followers(numel(neuron(origin).followers)+1)=target;
        end

        function [neuron]=backProp(neuron,id,d,method)
            y=neuron(id).output;
            if neuron(id).type==2                  % end node

                if method==0 neuron(id).beta=(d-y)*y*(1-y); end % d: correct value, y: output value
                if method==1 neuron(id).beta=(d-y)*(1-y*y); end % d: correct value, y: output value
                if method==2 neuron(id).beta=(d-y)*(1); end % d: correct value, y: output value
                if method==3 neuron(id).beta=(d-y)*(1-y*y); end % d: correct value, y: output value
                if method==4 % special case for newton method and output neuron, betaJ needs to be 1
                    neuron(id).betaJ=1; % outbut beta shoud be one
                    neuron(id).beta=(d-y)*1;
                end


            elseif neuron(id).type==1               % inner node
                for f=1:numel(neuron(id).followers)
                    if method==4
                        b (f)=neuron(neuron(id).followers(f)).beta;
                        bj(f)=neuron(neuron(id).followers(f)).betaJ;
                    else
                        b(f)=neuron(neuron(id).followers(f)).beta;
                    end


                    search=numel(neuron(neuron(id).followers(f)).connections);
                    for s=1:search
                        if neuron(neuron(id).followers(f)).connections(s)==id
                            w(f)=neuron(neuron(id).followers(f)).inputweight(s);
                            %break;
```

```
                        %w(f)=interp1 (neuron(neuron(id).followers(f)).connections , neuron(neuron(id).followers(f)).inputweight, id, 'nearest'); % fwd weight lookup
                    end
                end



            end
            q=0;
            qj=0;
            for f=1:numel(neuron(id).followers)
                q= q +w(f)*b (f);
                if method==4
                    qj=qj+w(f); % bj is not multiplied by the beta
                end
            end
            if method==0 neuron(id).beta=(q)*y*(1-y); end
            if method==1 neuron(id).beta=(q)*(1-y*y); end
            if method==2 neuron(id).beta=(q)*(1); end
            if method==3 neuron(id).beta=(q)*(1-y*y); end
            if method==4 neuron(id).betaJ=(qj)*(1-y*y); % bj is not multiplied by the beta
                neuron(id).beta= (q) *(1-y*y);
            end %Newton
        end
        neuron(id).deltaHistory(numel(neuron(id).deltaHistory)+1)=(d-y)*(d-y);%neuron(id).beta;
        %neuron(id).History(numel(neuron(id).biasHistory)+1)=neuron(id).bias;
        neuron(id).firstWeightHistory(numel(neuron(id).firstWeightHistory)+1)=neuron(id).inputweight(1);
    end



    function [neuron]=weightUpdate(neuron,id,learningRate,method,firstTake,momentum)
        neuron(id).delta_bias=neuron(id).beta*learningRate;
        neuron(id).bias    = 0;%neuron(id).bias  + neuron(id).delta_bias;
        for n=1:numel(neuron(id).connections)
            if method==3
```

```
                    if firstTake==true neuron(id).delta(n)=0;
                        neuron(id).delta(n);
                    end
                    neuron(id).delta(n) = neuron(id).delta(n) +
                      neuron(id).beta * learningRate * neuron(
                      neuron(id).connections(n)).output;
                else
                    if firstTake==true neuron(id).previous_delta(
                      n)=0; end
                    neuron(id).delta(n)=neuron(id).beta*
                      learningRate*neuron(neuron(id).connections(
                      n)).output;
                    neuron(id).inputweight(n)   = neuron(id).
                      inputweight(n)  + neuron(id).delta(n)*(1-
                      momentum) + momentum*neuron(id).
                      previous_delta(n);
                end
            end
            neuron(id).previous_delta=neuron(id).delta;
        end


        function [neuron]=batchUpdate(neuron,id,learningRate)
            neuron(id).delta_bias=neuron(id).beta*learningRate;
            neuron(id).bias   = 0;%neuron(id).bias  + neuron(id).
              delta_bias;
            for n=1:numel(neuron(id).connections)
                neuron(id).delta(n);
                neuron(id).inputweight(n)   = neuron(id).
                  inputweight(n)  + neuron(id).delta(n);
                neuron(id).delta(n)=0;
            end
        end
    end
end
```

D.1.3 Neuron Class [NeuronNewton.m]

```
classdef NeuronNewton < Neuron % inherit Neuron class
    %Neuron for neuronal network for quasi-Newton method
    %

    properties
        betaJ
    end
```

```


    methods


        function [neuron,j,g,nCounter,connectionCount]=calc(neuron,id,j,g,nCounter,connectionCount) % override weightUpdate function
            for n=1:numel(neuron(id).connections)
                nCounter=nCounter+1;
                j (nCounter) = neuron(id).betaJ * neuron(neuron(id).connections(n)).output;
                g (nCounter) = neuron(id).beta  * neuron(neuron(id).connections(n)).output + g (nCounter);
            end
        end

        function [neuron,nCounter]=weightUpdateNewton(neuron,id,d,nCounter,learningRate)
            for n=1:numel(neuron(id).connections)
                nCounter=nCounter+1;
                neuron(id).delta(n)=d(nCounter);
                neuron(id).inputweight(n)   = neuron(id).inputweight(n)  + neuron(id).delta(n)*learningRate;
            end
        end
    end
end
```

D.1.4 Forward Propagation Function [NeuronCalc.m]

```
classdef NeuronNewton < Neuron % inherit Neuron class
    %Neuron for neuronal network for quasi-Newton method
    %

    properties
        betaJ
    end


    methods


```

```
        function [neuron,j,g,nCounter,connectionCount]=calc(neuron,id,j,g,nCounter,connectionCount) % override weightUpdate function
            for n=1:numel(neuron(id).connections)
                nCounter=nCounter+1;
                j (nCounter) = neuron(id).betaJ * neuron(neuron(id).connections(n)).output;
                g (nCounter) = neuron(id).beta  * neuron(neuron(id).connections(n)).output + g (nCounter);
            end
        end

        function [neuron,nCounter]=weightUpdateNewton(neuron,id,d,nCounter,learningRate)
            for n=1:numel(neuron(id).connections)
                nCounter=nCounter+1;
                neuron(id).delta(n)=d(nCounter);
                neuron(id).inputweight(n)   = neuron(id).inputweight(n)  + neuron(id).delta(n)*learningRate;
            end
        end
    end
end
```

D.1.5 Sigmoid function [sigmoid.m]

```
function [ y ] = sigmoid( x )
% Outputs the sigmoid function s(x)

y=1./(1+exp(-(x)));
%y=1-2./(exp(2*x)+1);

end
```

D.1.6 Neural Network Training with Toolbox [train_matlab1.m]

```
% Solve an Input-Output Fitting problem with a Neural Network
% Script generated by NFTOOL
%
% This script assumes these variables are defined:
%
% houseInputs - input data.
```

```
% houseTargets - target data.
clear all;

[p,t,series_n]=createInputOutput2; % create continous output

%net = newff(p,t,5,{},'traingd'); % gradient descent
%net = newff(p,t,5,{},'trainscg');% Scaled Conjugate Gradient
%net = newff(p,t,5,{},'trainbfg'); % BFGS Quasi-Newton

%net = newff(p,t,5,{},'trainlm'); % Levenberg-Marquardt
net = newff(p,t,5,{},'trainbr'); % Bayesian Regularization
%net = newff(p,t,5,{},'trainrp'); % Resilient Backpropagation
%net = newff(p,t,5,{},'traincgb');% Conjugate Gradient with Powell/Beale Restarts
%net = newff(p,t,5,{},'traincgf');% Fletcher-Powell Conjugate Gradient
%net = newff(p,t,5,{},'traincgp');% Polak-Ribiére Conjugate Gradient
%net = newff(p,t,5,{},'trainoss');% One Step Secant
%net = newff(p,t,5,{},'traingdx');% Variable Learning Rate Gradient Descent
%net = newff(p,t,5,{},'traingdm');% Gradient Descent with Momentum


net.divideFcn = 'dividerand';

%net.trainParam.show = 50;
net.trainParam.lr = 0.1;
net.trainParam.epochs = 100;
net.trainParam.goal = 1e-5;
net.layers{1}.transferFcn = 'logsig';
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;
%net.trainParam.showWindow=0;


[net,tr]=train(net,p,t);
%plotperform(tr);


figure (1);
plot(series_n(:,1),'b');
hold on;
plot(sim(net,p(:,:)),'r');
```

```
hold off;
```

D.1.7 Neural Network Training with Toolbox [train_matlab3.m]

```
% Solve an Input-Output Fitting problem with a Neural Network
% Script generated by NFTOOL
%
% This script assumes these variables are defined:
%
% houseInputs - input data.
% houseTargets - target data.

%{
clear all;

mode=47;
[x1,xs,x2,y1,ys,y2,z1,zs,z2,n1,ns,n2,c,ms, strategy, filename,
  sheetname, total_calculations] = trading_presets(mode);

TS=TimeSeries;
TS=load_time_series(TS, filename, sheetname, n1,ns,n2,c,ms);   %
  load time serie


%}

% put together training matrixes
a=1;
x=0;
     for n=4:102
         for i=10:TS(n).number_of_returns-31
            % data check
            if ~isnan (sum(TS(n).returns_normalized(i:i+1))) && ~
              isnan (sum(TS(n).second(i-a:i))) && ~isnan (sum(TS(
              n).third(i-a:i))) && ~isnan (sum(TS(n).prices(i-a:i
              )))
             x=x+1;
             % p=inputs
             p(1,x)=TS(n).second(i)/TS(n).second(i-a)-1;
             p(2,x)=TS(n).third(i)/TS(n).second(i-a)-1;
             p(3,x)=TS(n).fourth(i)/TS(n).second(i-a)-1;
             % t=targets or outputs
             t(x)=TS(n).returns_normalized(i+a)/TS(n).
               returns_normalized(i)-1;
               if mod(x,10000)==0
```

```
                        complete_est=x/700000*100
                   end;
               end

            end
        end


%net = newff(p,t,10,{},'traingd'); % gradient descent
%net = newff(p,t,10,{},'trainscg');% Scaled Conjugate Gradient
%net = newff(p,t,10,{},'trainbfg'); % BFGS Quasi-Newton

%net = newff(p,t,10,{},'trainlm'); % Levenberg-Marquardt
%net = newff(p,t,10,{},'trainbr'); % Bayesian Regularization
%net = newff(p,t,10,{},'trainrp'); % Resilient Backpropagation
%net = newff(p,t,10,{},'traincgb');% Conjugate Gradient with Powell/Beale Restarts
%net = newff(p,t,10,{},'traincgf');% Fletcher-Powell Conjugate Gradient
%net = newff(p,t,10,{},'traincgp');% Polak-Ribiére Conjugate Gradient
%net = newff(p,t,10,{},'trainoss');% One Step Secant
%net = newff(p,t,10,{},'traingdx');% Variable Learning Rate Gradient Descent
net = newff(p,t,10,{},'traingdm');% Gradient Descent with Momentum


net.divideFcn = 'dividerand';

%net.trainParam.show = 50;
net.trainParam.lr = 0.1;
net.trainParam.epochs = 100;
net.trainParam.goal = 1e-5;
net.layers{1}.transferFcn = 'logsig';
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;
%net.trainParam.showWindow=0;


[net,tr]=train(net,p,t);
%plotperform(tr);


figure (1);
```

```
%plot(series_n(:,1),'b');
%hold on;
plot(sim(net,p(:,:)),'r');
hold off;
```

D.2 Trading Simulator

D.2.1 Main program [main.m]

```
clear;
clear classes; nn=0;

%% Objects
% TS: time series data
% F: Total Fund returns and Index returns on daily basis
% G: strategy: Subclass of trade_generator which communicates with the strategy
% T: trades
% A: analysis of a specific scenario (i.e. summing up the trades)
% O: optimum and graphs
%% Parameters

%mode=1; % Buying after selloff after several months, then profittaking
%mode=5; % Buying after selloff after several months, x day sell
%mode=2; % Buying after selloff after several days,   then profittaking
%mode=21; % Buying after selloff after several days,   then profittaking one case

%mode=3; % Buying after selloff after several months, buying after days

%mode=4; % Buying after selloff after several days,   then profittaking SMI
%mode=6; % Payout ratio *

%mode=8; % Stop loss and buy after reaching x day low *
%mode=9; % Special Stop loss and buy after reaching x day low
%mode=10; % Analyst recommendations
%mode=11; % Analyst recommendations test
%mode=19; % Special stop strategy: dynamic stop, buy after price> sell price

%mode=29; % Coppock curve
```

```
%mode=30; % Moving average *

%mode=40;  load('trained-5inputNeurons.mat');  % Neural Network 1
%mode=41;  load('trained-5inputNeurons.mat');  % Neural Network 1

%mode=42;  load('trained-10inputNeurons-5d-5t.mat');  % Neural Network 1
%mode=43;  load('trained-10inputNeurons-5d-5t.mat');  % Neural Network 1

%mode=44;  load('net2.mat'); nn=net; % Neural Network 2
%mode=45;  load('net2.mat'); nn=net; % Neural Network 2

%mode=46;  load('net2b.mat'); nn=net; % Neural Network 2

mode=47;  load('net3.mat'); nn=net; % Neural Network 3


%mode=101; % quick: Buying after selloff after several days, then profittaking
%mode=1001; % SP100, one case

%mode=9000; % testsheet

[x1,xs,x2,y1,ys,y2,z1,zs,z2,n1,ns,n2,c,ms, strategy, filename, sheetname, total_calculations] = trading_presets(mode);

%% Start of main program

rf=2; % 2 pct risk free rate

graphmode=1;                                   % 1: more than one case
if x2-x1+y2-y1+z2-z1+n2-n1==0 graphmode=2 ; end      % 2: analyze only one case (2d histogram in analysis)
if n2-n1>0
    graphmode=3;                               % 3: more than one case for various stocks (3d histogram)
    if x2-x1+y2-y1+z2-z1==0 graphmode=4; end  % 4: analyze only one case for various stocks (2d histogram in graph)
end

eta=Eta;
calc=0;
tic;
```

```
TS=TimeSeries;
TS=load_time_series(TS, filename, sheetname, n1,ns,n2,c,ms);   % load time series

if TS(n1).start<z1 && z1==z2
    TS(n1).start=z1;
end


F=FundReturn (min([TS.start]), max([TS.stop]));

for n=n1:ns:n2
  for z=z1:zs:z2
    for y=y1:ys:y2
       for x=x1:xs:x2
            %% progress monitor
            calc=calc+1;                                  % count cases
            eta=calc_ETA(eta,calc,total_calculations)   % calculate and show progress and remaining time


            %% analyze returns
            G=strategy;
            clear T;                                      % clear trades from potential previous runs
            [G,T,F]=analyze_time_series(G, TS(n), F, x,y,z,n,nn,rf); % analyze the time series with the strategy G and create Trades T

            %% analyze trades for a scenatio A
            A(x,y,z,n)=TradeAnalyzer;                     % analyze the generated Trades T and create an object A with the analysis
            A(x,y,z,n)=analyze_trades(A(x,y,z,n), T, G, TS(n), graphmode, rf); % analyze
            A(x,y,z,n)                                    % show the result
       end
    end
  end
end


%% Generate output
```

```matlab

if graphmode==2 || graphmode==4
    lag=round((TS(n).stop-TS(n).start)/10);
    lag=1;
    F=F.plotReturns (lag,1);
end

O=TradeGraphs;                                                % generate the graphs and calculate the optimum in in object O
O=graph (O,A,G,x1,xs,x2,y1,ys,y2,z1,zs,z2,n1,ns,n2,graphmode,G.start);

if graphmode==1
    % calculate again trades for optimum and perform a wilcoxon test on the
    % best annaulized_return strategy
    clear F;
    F=FundReturn (min([TS.start]), max([TS.stop]));
    G=strategy;
    n=1;
    G.start=(O.opt_z-1)+1;                                     % start analysis on day G.start
    [G,T,F]=analyze_time_series(G, TS(n), F, O.opt_x,O.opt_y,O.opt_z,n,nn,rf);
    disp ('Signrank test for avg performance per trade annaulized is different to buy and hold return');
    [p,test]=signrank([T.annualized_return],mean([A.buy_hold_performance_pa]))
    if test==1 && (mean([T.annualized_return]) > mean([A.buy_hold_performance_pa]))
        disp ('!!!!!!!!!!!!!!!!!! POTENTIAL STRATEGY FOUND !!!!!!!!!!!!!!!!!!');
    end
    lag=round((TS(n).stop-TS(n).start)/10);
    lag=1;
    F=F.plotReturns(lag,1);
end
```

D.2.2 TimeSeries Class [Time_series.m]

```matlab
classdef TimeSeries
    %Loads the time series and performs some basic calculations
    %   Detailed explanation goes here

    properties
```

```
        filename
        sheetname
        prices
        second
        third
        fourth
        returns
        returns_normalized
        number_of_prices
        number_of_returns
        column
        start
        stop
        vs
    end

    methods
         function obj=load_time_series(obj,filename,sheetname,n1,ns,n2,c,ms)
            % loads the time series and calls the calc_returns method to
            % calculate the returns

           for n=n1:ns:n2
               disp ('Loading time series...');
               n
               obj(n).vs=0;
               obj(n).filename=filename;
               obj(n).sheetname=sheetname;

               obj(n).column=n;
                if ms==1 % more than one sheet
                    obj(n).sheetname=['sheet' int2str(n)];
                    obj(n).column=1; % first column for prices
                end;

                filedata=xlsread(obj(n).filename,obj(n).sheetname);

                obj(n).prices=(filedata(:,1));                    % row with prices
                obj(n).number_of_prices=size(obj(n).prices,1);    % number of days of which we have returns (number of rows)
                obj(n).number_of_returns=size(obj(n).prices,1)   ;  % number of days of which we have returns (
```

```
number of rows)

            if ms==1
                obj(n).vs=1; % various sheets;
                obj(n).second=(filedata(:,c));    % row with SECOND TIME SERIES

                obj(n).third=(filedata(:,c+1));    % row with THIRD TIME SERIES
                obj(n).fourth=(filedata(:,c+2));    % row with FOURTH TIME SERIES
            end;




            obj(n)=calc_returns(obj(n));
                                  % calculate returns for the time series

        end
    end

    function obj=calc_returns(obj)
       % calculates logarithmic returns for the prices
       obj.returns(1)=0;
       obj.returns_normalized(1)=0;
       obj.start=1;
       obj.stop=obj.number_of_prices;


       for x=2:obj.number_of_prices %nancheck
           if obj.vs==1 % if multiple sheets check for first value in second time series
               if isnan (obj.second(x-1)) && ~isnan (obj.second(x))
                   obj.start=x;
               end
           end

               if ~isnan(obj.prices(x-1)) && isnan(obj.prices(x))    % check for the first nan in prices
                   obj.stop=x;
               end
       end
```

```


            % calculate returns
            disp ('Calculating returns...');
            for x=2:obj.number_of_prices % calculating returns
              for the time series starting from day 2
               obj.returns(x)=log(obj.prices(x)/obj.prices(x-1))
                 *100;
            end

            % calculate normalized returns
            disp ('Calculating normalized returns...');
            m=mean(obj.returns);
            s=sqrt(var(obj.returns));
            for x=2:obj.number_of_prices
               obj.returns_normalized(x)=(obj.returns(x)-m)/s;
            end


            obj.number_of_prices =obj.stop-obj.start;
            obj.number_of_returns=obj.stop-obj.start-1;
         end
    end
end
```

D.2.3 Trade Generator Class [TradeGenerator.m]

```
classdef TradeGenerator % abstract class
    % Trade generator is a super-class which loads the excel file
      ,
    % calcualtes the returns

    properties
        trade_counter=0;
        currently_open
        action
        start
        stop
        number_of_prices
        number_of_returns

        % properties that are used to communicate with the
          subclass of
        % trade_generator and that can be used as conditions to
          generate
```

```
        % trades:
        open_day=0;
        close_day=0;

        open_price=0;
        close_price=0;

        today_price
        today_day

        close_price_override;

        x
        y
        z
        n
    end

    methods
         function [obj,t,F]=analyze_time_series (obj, ts, F, x, y, z, n, nn, rf)
            obj.start=ts.start;
            obj.stop=ts.stop;
            obj.number_of_prices=ts.number_of_prices;
            obj.number_of_returns=ts.number_of_returns;
            obj.currently_open(obj.start)=0;
            obj.x=x;
            obj.y=y;
            obj.z=z;
            obj.n=n;
            obj.open_day=0;
            obj.close_day=0;

            t(1)=Trade;  % generate a trade in case there is none for the entire series

                %% check each day of the returns time series
                for day=obj.start:obj.stop

                        obj.today_price=ts.prices(day);
                        obj.today_day=day;

                        F=addIndexReturn(F,day,ts.returns(day));

                            [open_condition_value, obj]=obj.open_condition_calc (obj,ts,nn);
```

```


            if obj.currently_open(day)==1
                [close_condition_value,obj]=obj.close_condition_calc (obj,ts,nn);
                if obj.close_price_override>0
                    F=addFundReturn(F,day,log(obj.close_price_override/ts.prices(day-1))*100);
                else
                    F=addFundReturn(F,day,ts.returns(day));
                end
            else
                    F=addFundReturn(F,day,rf/365);
            end

        % open position
         if obj.currently_open(day)~=1 && open_condition_value == 1 % open condition
            obj.trade_counter=obj.trade_counter+1;                 % count trades
            obj.currently_open(day+1)=1;                           % flag as currently open trade
            obj.action (day) = +1;
            obj.open_day=day;
            obj.open_price=ts.prices(day);

            % generate trade object
            t(obj.trade_counter) = Trade;                          % generate a new trade object
            t(obj.trade_counter) = open   (t(obj.trade_counter), day, ts.prices(day)); % buy


         % close position
         elseif obj.currently_open(day)==1 && close_condition_value == 1 % close condition

            if obj.close_price_override>0
               price=obj.close_price_override;
               obj.close_price_override=0;
```

```
                        else
                            price=ts.prices(day);
                        end

                        t(obj.trade_counter) = close (t(obj.trade_counter), day, price); % sell
                        obj.currently_open(day+1)=0;                                       % next day the trade will be closed
                        obj.action (day) = -1;                                             % mark in the time series object action
                        obj.close_day=day;
                        obj.close_price=price;

                    else % do nothing
                        obj.currently_open(day+1)=obj.currently_open(day);         % keep current position status
                        obj.action (day) = 0;
                    end
                end
            if obj.currently_open(day+1)==1 t(obj.trade_counter)=close(t(obj.trade_counter),day, ts.prices(day)); obj.action (day) = -1; end % sell
        end
    end
end
```

D.2.4 Trade Analyzer Class [TradeAnalyzer.m]

```
classdef TradeAnalyzer
    % The analyzes the trade objects
    %   Detailed explanation goes here

    properties
        x
        y
        z
        n
        outperformance_pa
        outperformance_pa_leveraged
        strategy_performance_pa
        strategy_performance_pa_leveraged

```

```
        buy_hold_performance_pa
        strategy_performance
        strategy_performance_leveraged

        buy_hold_performance
        strategy_performance_avg

        strategy_number_of_trades
        buy_hold_number_of_trades=1

        strategy_longdays
        buy_hold_longdays
        idle_days
        idle_performance
        strategy_longdays_pct
        buy_hold_longdays_pct=100;
        strategy_avg_trade_duration=0;

        strategy_winning_trades
        strategy_winning_trades_pct
        strategy_max_drawdown
        strategy_max_win

        worst_trade_start
        worst_trade_end
        worst_trade_number
        best_trade_start
        best_trade_end
        best_trade_number
        periods_pa=365;

        strategy_performance_annualized_avg
        strategy_performance_annualized_avg_filtered
        strategy_performance_annualized_avg_filtered_test


    end

    methods
        function obj=analyze_trades (obj, t, g, ts, graphmode, rf
          )
            if g.number_of_prices>0 % only calculate when there
              are more than 1 valid prices in the time series

                rf=rf/obj.periods_pa;

```

```
            obj.idle_days=g.number_of_returns-sum([t.trade_duration]);
            obj.idle_performance=obj.idle_days*rf;

            obj.strategy_performance=sum([t.log_return])+obj.idle_performance;
            obj.strategy_performance_leveraged=sum([t.log_return_leveraged])+obj.idle_performance;

            obj.buy_hold_performance=log(ts.prices(ts.stop)/ts.prices(ts.start))*100;
            obj.buy_hold_longdays=g.number_of_returns;
            obj.strategy_performance_pa         =(obj.strategy_performance + obj.idle_performance ) / obj.buy_hold_longdays * obj.periods_pa;
            obj.strategy_performance_pa_leveraged=(obj.strategy_performance_leveraged + obj.idle_performance )/ obj.buy_hold_longdays * obj.periods_pa;
            obj.buy_hold_performance_pa=obj.buy_hold_performance / obj.buy_hold_longdays * obj.periods_pa;
            obj.outperformance_pa=obj.strategy_performance_pa-obj.buy_hold_performance_pa;
            obj.outperformance_pa_leveraged=obj.strategy_performance_pa_leveraged-obj.buy_hold_performance_pa;
            obj.strategy_longdays=sum([t.trade_duration]);
            obj.strategy_longdays_pct=obj.strategy_longdays/g.number_of_prices*100;
            obj.strategy_number_of_trades=g.trade_counter;
            obj.strategy_winning_trades=sum([t.winning]);
            obj.strategy_winning_trades_pct=obj.strategy_winning_trades/obj.strategy_number_of_trades*100;
            obj.strategy_max_drawdown=min([t.log_return]);
            obj.strategy_max_win=max([t.log_return]);

            obj.strategy_performance_avg =            mean([t.log_return]);
            obj.strategy_performance_annualized_avg = median([t.annualized_return]);

            if obj.strategy_number_of_trades > 1 && sum([t.annualized_return])>0 && sum([obj.buy_hold_performance_pa])>0 % avoid that after
```

```
                removing nans there's nothing left
                obj.
                  strategy_performance_annualized_avg_filtered
                   = median([t.annualized_return]);
                obj.strategy_avg_trade_duration=obj.
                  strategy_longdays/obj.
                  strategy_number_of_trades;
                [p,test]=signrank([t.annualized_return],
                  nanmean([obj.buy_hold_performance_pa]));
            else
                obj.
                  strategy_performance_annualized_avg_filtered
                   = 0;
                test=0; % no test when no trades
            end


            if test == 1
                obj.
                  strategy_performance_annualized_avg_filtered_test
                   = mean([t.annualized_return]);
            else
                obj.
                  strategy_performance_annualized_avg_filtered_test
                   = 0;
            end


            [val,ind]=min([t.log_return]);
           if ind>0 % only do the calculations if there was
             at least one trade
            obj.worst_trade_start=t(ind).buy_date;
            obj.worst_trade_end=t(ind).sell_date;
            obj.worst_trade_number=ind;
                [val,ind]=max([t.log_return]);
            obj.best_trade_start=t(ind).buy_date;
            obj.best_trade_end=t(ind).sell_date;
            obj.best_trade_number=ind;
           end

        end % only calculate when there are more than 1 valid
           prices in the time series

            obj.x=g.x;
            obj.y=g.y;
            obj.z=g.z;
```

```
                obj.n=g.n;


            if graphmode==2

                disp ('++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++');


                figure (1);
                hist([t.log_return]);
                title('Trade Performance %')
                xlabel('% performance per trade')
                ylabel('Number of cases')

                disp ('Signrank test for median different to 0 for trade performance %');
                [p,test]=signrank([t.log_return])


                figure (2);
                hist([t.annualized_return],50);
                title('Annualized Trade Performance %')
                xlabel('% performance p.a. per trade')
                ylabel('Number of cases')

                disp ('Signrank test for median different to 0 for annualized trade performance %');
                [p,test]=signrank([t.annualized_return])



                figure (3);
                hist([t.trade_duration]);
                title('Trade duration')
                xlabel('Trade duration')
                ylabel('Number of cases')

            end

        end
    end
end
```

D.2.5 Trade Graph generator Class [TradeGraphs.m]

```
classdef TradeGraphs
    % Output of graphs for all the various trading parameters
    %   Detailed explanation goes here

    properties
        maxval
        maxval2
        maxval3

        opt_x
        opt_y
        opt_z

        opt2_x
        opt2_y
        opt2_z

        opt3_x
        opt3_y
        opt3_z

    end

    methods (Static)
        function [opt_x,opt_y,opt_z,maxval] = optimize(a, matrix, description, x1,xs,y1,ys,z1,zs) % find out optimum strategy

            [maxval, maxind] = max(matrix(:)); % calculate maximum value and maximum index
            [opt_x, opt_y, opt_z] = ind2sub(size(matrix),maxind); % redistribute maxind on x y and z


            opt_x=(opt_x-1) * xs + x1;
            opt_y=(opt_y-1) * ys + y1;
            opt_z=(opt_z-1) * zs + z1;

            disp(' ');
            disp('Best Strategy: ');
            description
            disp('=============');

            a(opt_x,opt_y,opt_z) % output optimal strategy
```

```
        end
    end

    methods
        function obj = graph (obj,a,g,x1,xs,x2,y1,ys,y2,z1,zs,z2,n1,ns,n2,graphmode,start)

           txtx=g.txtA;
           txty=g.txtB;


          if graphmode==1
            % various cases for an index

%INSERT CODE: ASK AUTHOR FOR MORE INFORMATION
            %matrix2=reshape(cat(2,a(x1:xs:x2,y1:ys:y2,z1:zs:z2).strategy_performance_annualized_avg),(x2-x1)/xs+1,(y2-y1)/ys+1,(z2-z1)/zs+1); % bundle objects
            %[obj.opt2_x,obj.opt2_y,obj.opt2_z,obj.maxval2]  = obj.optimize(a,matrix2,'Annualized per trade',x1,xs,y1,ys,z1,zs); % optimize for strategy_performance_annualized_average

            matrix3=reshape(cat(2,a(x1:xs:x2,y1:ys:y2,z1:zs:z2,1).strategy_performance_annualized_avg_filtered_test),(x2-x1)/xs+1,(y2-y1)/ys+1,(z2-z1)/zs+1); % bundle objects
            [obj.opt3_x,obj.opt3_y,obj.opt3_z,obj.maxval3]  = obj.optimize(a,matrix3,'Annualized per trade filtered test',x1,xs,y1,ys,z1,zs); % optimize for strategy_performance_annualized_average

            [x,y] = meshgrid(y1:ys:y2,x1:xs:x2);

            figure (1);
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z,1).strategy_performance_pa),(x2-x1)/xs+1,(y2-y1)/ys+1));
            xlabel(txtx)
            ylabel(txty)
            zlabel('Performance % p.a.')
            title ('Strategy Performance % p.a. vs. buy-and-hold')
                    hold on;
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z,1).buy_hold_performance_pa),(x2-x1)/xs+1,(y2-y1)/
```

```
      ys+1));
    hold off;

    figure (2);
    mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z
      ,1).strategy_number_of_trades),(x2-x1)/xs+1,(y2-y1)
      /ys+1));
    xlabel(txtx)
    ylabel(txty)
    zlabel('Number of trades')
    title ('Number of trades')

    figure (3);
    mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z
      ,1).strategy_longdays_pct),(x2-x1)/xs+1,(y2-y1)/ys
      +1));
    xlabel(txtx)
    ylabel(txty)
    zlabel('% of days long position is taken')
    title ('% of long days')
           hold on;
    mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z
      ,1).buy_hold_longdays_pct),(x2-x1)/xs+1,(y2-y1)/ys
      +1));
    hold off;

    figure (4);
    mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z
      ,1).strategy_winning_trades_pct),(x2-x1)/xs+1,(y2-
      y1)/ys+1));
    xlabel(txtx)
    ylabel(txty)
    zlabel('% of winning trades')
    title ('% of winning trades')

    figure (5);
    mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z
      ,1).strategy_max_drawdown),(x2-x1)/xs+1,(y2-y1)/ys
      +1));
    xlabel(txtx)
    ylabel(txty)
    zlabel('Worst individual trade % return')
    title( 'Worst individual trade %')

    figure (6);
```

```
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z
              ,1).strategy_max_win),(x2-x1)/xs+1,(y2-y1)/ys+1));
            xlabel(txtx)
            ylabel(txty)
            zlabel('Best individual trade % return')
            title( 'Best individual trade %')

            figure (7);
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z
              ,1).strategy_performance_avg),(x2-x1)/xs+1,(y2-y1)/
              ys+1));
            xlabel(txtx)
            ylabel(txty)
            zlabel('Average % return per trade')
            title ('Average Performance per trade')

            figure (8);
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z
              ,1).strategy_performance_annualized_avg),(x2-x1)/xs
              +1,(y2-y1)/ys+1));
            xlabel(txtx)
            ylabel(txty)
            zlabel('Average % return per trade annualized')
            title ('Average Performance per trade annualized')
             hold on;
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z
              ,1).buy_hold_performance_pa),(x2-x1)/xs+1,(y2-y1)/
              ys+1));
            hold off;

            figure (9);
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z
              ,1).strategy_performance_annualized_avg_filtered),(
              x2-x1)/xs+1,(y2-y1)/ys+1));
            xlabel(txtx)
            ylabel(txty)
            zlabel('Average % return per trade annualized')
            title ('Average % return per trade annualized
              filtered with trade amounts')
             hold on;
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z
              ,1).buy_hold_performance_pa),(x2-x1)/xs+1,(y2-y1)/
              ys+1));
            hold off;

            figure (10);
```

```
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z,1).strategy_performance_annualized_avg_filtered_test),(x2-x1)/xs+1,(y2-y1)/ys+1));
            xlabel(txtx)
            ylabel(txty)
            zlabel('Average % return per trade annualized')
            title ('Average Performance per trade annualized filtered with wilcoxon test')
             hold on;
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z,1).buy_hold_performance_pa),(x2-x1)/xs+1,(y2-y1)/ys+1));
            hold off;

            figure (11);
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z,1).strategy_performance_pa_leveraged),(x2-x1)/xs+1,(y2-y1)/ys+1));
            xlabel(txtx)
            ylabel(txty)
            zlabel('Performance % p.a. leveraged')
            title ('Strategy Performance % p.a. leveraged vs. buy-and-hold')
                    hold on;
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z,1).buy_hold_performance_pa),(x2-x1)/xs+1,(y2-y1)/ys+1));
            hold off;


            figure (12);
            mesh (x,y,reshape(cat(2,a(x1:xs:x2,y1:ys:y2,obj.opt_z,1).strategy_avg_trade_duration),(x2-x1)/xs+1,(y2-y1)/ys+1));
            xlabel(txtx)
            ylabel(txty)
            zlabel('Days')
            title ('Average trade duration')
          end

           % ----------------------------------------
           if graphmode==3
               % 3D histogram

           end
```

```
    % ----------------------------------------

    if graphmode==4
        % special case for various stocks

         figure (1);
         hist([a(x1,y1,z1,n1:ns:n2).strategy_performance_pa],30);
         title('Strategy Performance p.a. %')
         xlabel('% p.a.')
         ylabel('Number of cases')

         figure (2);
         hist([a(x1,y1,z1,n1:ns:n2).strategy_number_of_trades],20);
         title('Number of trades')
         xlabel('Number of trades')
         ylabel('Number of cases')

         figure (3);
         hist([a(x1,y1,z1,n1:ns:n2).strategy_longdays_pct],20);
         title('Longdays %')
         xlabel('Longdays %')
         ylabel('Number of cases')

         figure (4);
         hist([a(x1,y1,z1,n1:ns:n2).strategy_winning_trades_pct],20);
         title ('% of winning trades')
         xlabel('% of winning trades')
         ylabel('Number of cases')

         figure (5);
         hist([a(x1,y1,z1,n1:ns:n2).strategy_max_drawdown],20);
         title ('Worst individual trade % return')
         xlabel('% of winning trades')
         ylabel('Number of cases')

         figure (6);
         hist([a(x1,y1,z1,n1:ns:n2).strategy_max_win],20);
         title ('Best individual trade %')
         ylabel('Number of cases')
         xlabel('Best individual trade % return')

```

```
            figure (7);
            hist([a(x1,y1,z1,n1:ns:n2).strategy_performance_avg],20);
            title ('Average Performance per trade')
            ylabel('Number of cases')
            xlabel('Average % return per trade')

            figure (8);
            hist([a(x1,y1,z1,n1:ns:n2).strategy_performance_annualized_avg],20);
            title ('Average % return per trade annualized')
            ylabel('Number of cases')
            xlabel('Average % return per trade annualized')

            figure (9);
            hist([a(x1,y1,z1,n1:ns:n2).strategy_performance_annualized_avg_filtered],20);
            title ('Average % return per trade annualized filtered with trade amounts')
            ylabel('Number of cases')
            xlabel('Average % return per trade annualized')

            figure (10);
            hist([a(x1,y1,z1,n1:ns:n2).strategy_performance_annualized_avg_filtered_test],20);
            title ('Average Performance per trade annualized filtered with wilcoxon test')
            ylabel('Number of cases')
            xlabel('Average % return per trade annualized')

            figure (11);
            hist([a(x1,y1,z1,n1:ns:n2).outperformance_pa],20);
            title('Strategy Outperformance %')
            xlabel('Outperformance % p.a.')
            ylabel('Number of cases')

            figure (12);
            hist([a(x1,y1,z1,n1:ns:n2).outperformance_pa_leveraged],20);
            title('Strategy Outperformance % with leverage')
            xlabel('Outperformance % p.a. (with leverage)')
            ylabel('Number of cases')

```

```
                figure (13);
                subplot (2,1,1);
                hist([a(x1,y1,z1,n1:ns:n2).buy_hold_performance_pa],30);
                title('Buy-hold-Performance p.a. %')
                xlabel('% p.a.')
                ylabel('Number of cases')

                subplot (2,1,2);
                hist([a(x1,y1,z1,n1:ns:n2).strategy_performance_pa],30);
                title('Strategy Performance p.a. %')
                xlabel('% p.a.')
                ylabel('Number of cases')


                strategy_median=median([a(x1,y1,z1,n1:ns:n2).strategy_performance_pa])
                %strategy_leveraged_mean=mean([a(x1,y1,z1,n1:ns:n2).strategy_performance_pa_leveraged])
                buy_hold_median=median([a(x1,y1,z1,n1:ns:n2).buy_hold_performance_pa])
                outperformance_median=median([a(x1,y1,z1,n1:ns:n2).outperformance_pa])
                %outperformance_leveraged_mean=mean([a(x1,y1,z1,n1:ns:n2).outperformance_pa_leveraged])
                [p,test]=signrank([a(x1,y1,z1,n1:ns:n2).outperformance_pa])

                %[a(x1,y1,z1,n1:ns:n2).strategy_performance_pa]
                %[a(x1,y1,z1,n1:ns:n2).buy_hold_performance_pa]
            end
        end
    end
end
```

D.2.6 Trade Class [Trade.m]

```
classdef Trade
    % Trade objects contain a buying and selling date and price
    %   Each trade object can be bought and sold only once (open and close)

    properties
        buy_date=0;
```

```
        sell_date=0;
        buy_price=0;
        sell_price=0;
        log_return=0;
        log_return_leveraged=0;
        annualized_return=0;
        trade_duration=0;
        winning=0;
        currently_open=0;
        leverage=2;
    end

    methods
        function obj=calc_return(obj)
            obj.log_return=log(obj.sell_price / obj.buy_price) *
              100;
            obj.log_return_leveraged=obj.log_return*obj.leverage;
            obj.annualized_return=obj.log_return / max((obj.
              sell_date-obj.buy_date),1) * 365;
            obj.trade_duration=obj.sell_date - obj.buy_date;
            if obj.buy_price<obj.sell_price obj.winning=1; else
              obj.winning=0; end
        end

        function [obj,ts_obj] = open (obj,date,price, ts_obj) %
          opens a trade
            obj.buy_date=date;
            obj.buy_price=price;
            obj.currently_open=true;
            ts_obj.action (date) = +1;
        end

        function [obj,ts_obj] = close (obj, date, price, ts_obj)
          % closes a trade
            obj.sell_date=date;
            obj.sell_price=price;
            obj=calc_return(obj);
            obj.currently_open=false;
            ts_obj.action (date) = -1;
        end
    end
end
```

D.2.7 Fund Class [FundReturn.m]

```
classdef FundReturn
    %Simulates the returns of the index and the fund with the strategy on a
    %daily basis

    properties
        indexLog
        fundLog
        start
        stop
        indexAmount
        fundAmount
        indexChange
        fundChange
        index
        fund
    end

    methods
        function obj=FundReturn (start,stop)
                obj.indexAmount(start:stop)=0;
                obj.fundAmount (start:stop)=0;
                obj.indexChange (start:stop)=0;
                obj.fundChange (start:stop)=0;
                obj.start=start;
                obj.stop=stop;
        end

        function obj=addFundReturn(obj,day,change)
            if isnan(change)
                change=0;
            end
            obj.fundChange(day)=obj.fundChange(day)+change;
            obj.fundAmount(day)=obj.fundAmount(day)+1;
        end

        function obj=addIndexReturn(obj,day,change)
            if isnan(change)
                change=0;
            end
            obj.indexChange(day)=obj.indexChange(day)+change;
            obj.indexAmount(day)=obj.indexAmount(day)+1;
        end

        function obj=plotReturns(obj,lag,mode)
```

```
            clear obj.indexLog;
            clear obj.fundLog;

            obj.indexLog(obj.start:obj.stop)=0;
            obj.fundLog(obj.start:obj.stop)=0;
            obj.index(obj.start:obj.stop)=0;
            obj.fund(obj.start:obj.stop)=0;

            for x=obj.start+lag : obj.stop
                if obj.indexAmount(x)>0
                 obj.indexChange(x)=obj.indexChange(x)/obj.
                   indexAmount(x);
                 obj.indexLog(x)=obj.indexLog(x-1)+ obj.
                   indexChange(x);
                else
                 obj.indexLog(x)=obj.indexLog(x-1);
                end

                if obj.fundAmount(x)>0
                 obj.fundChange (x)=obj.fundChange (x)/obj.
                   fundAmount (x);
                 obj.fundLog(x) =obj.fundLog (x-1)+ obj.
                   fundChange (x);
                else
                 obj.fundLog(x) =obj.fundLog (x-1);
                end
                %obj.index(x)=(exp(obj.indexLog(x)*0.01)-1)*100;
                %obj.fund(x)=(exp(obj.fundLog(x)*0.01)-1)*100;
            end



            figure (19);
            plot(obj.start:obj.stop-1,obj.fundAmount(obj.start:
              obj.stop-1),'-r');
            hold on
            plot(obj.start:obj.stop-1,obj.indexAmount(obj.start:
              obj.stop-1),'-k');
            hold off
            xlabel('Days')
            ylabel('# of investments')
            title ('Amount of investments in index (black) vs.
              Fund Return (red)')

            figure (20);
```

```
            plot(obj.start+lag:obj.stop-1, obj.fundLog (obj.start+lag:obj.stop-1),'-r');
            hold on
            plot(obj.start+lag:obj.stop-1, obj.indexLog(obj.start+lag:obj.stop-1),'-k');

            if mode==1
             hold off
            end

            xlabel('Days')
            ylabel('Return (log)')
            title ('Index Return (black) vs. Fund Return (red)')

            %figure (21);
            %semilogy(1:obj.stop,obj.fund,'-r');
            %hold on
            %semilogy(1:obj.stop,obj.index,'-k');
            %hold off
            %xlabel('Days')
            %ylabel('Return')
            %title ('Index Return (black) vs. Fund Return (red)')

            sharpe_fund=sharpe(obj.fundLog,0.04)
            sharpe_index=sharpe(obj.indexLog,0.04)

            disp ('Calculating peak to trough...');

            if mode==1
                for x=obj.start+lag:obj.stop
                  minTempFund=min(obj.fundLog(x:obj.stop));
                  ptFund(x)=minTempFund-obj.fundLog(x);
                  minTempIndex=min(obj.indexLog(x:obj.stop));
                  ptIndex(x)=minTempIndex-obj.indexLog(x);
                end
               peakToTroughLogFund=min(ptFund)
               %peakToTroughFund=(exp(peakToTroughLogFund/100)-1)*100

               peakToTroughLogIndex=min(ptIndex)
               %peakToTroughIndex=(exp(peakToTroughLogIndex/100)-1)*100

            disp ('Are time series statistically significantly different?');
```

```
                [p,test]=signrank([obj.indexChange],[obj.
                  fundChange])
             end
         end
     end
end
```

D.2.8 Strategy Class [STRATEGY_NeuralNetwork1.m]

```
classdef STRATEGY_neuralNetwork1 < TradeGenerator
% buy and sell depending on neural network output

    properties
        txtA = ['Buy threshold: y > theta_b'];
        txtB = ['Sell threshold: y < theta_s'];




    end

    methods (Static)
        function [value,g]=open_condition_calc (g,ts,nn)
                    % buying condition
                    %
                      ==============================================

                    % buy if we're above the closing price

                    for d=1:g.z
                        x(d)=ts.returns_normalized(g.today_day-d);
                    end
                    outputNeuron=neuronCalc(6,11,4,x,nn); % tanh
                    %outputNeuron
                    if   (outputNeuron > (g.y/1000))

                    %
                      ==============================================

                    value=true;
                    else value=false;
                    end
        end

```

```
        function [value,g]=close_condition_calc (g,ts,nn)
                  % selling condition
                   % ================================================
                   % sell when x% below high since high since we bought

                   for d=1:g.z
                       x(d)=ts.returns_normalized(g.today_day-d);
                   end
                   outputNeuron=neuronCalc(6,11,4,x,nn); % tanh
                   if   (outputNeuron < (-g.x/1000))
                   % ================================================

                   %if  g.today_price<g.open_price
                   %    g.close_price_override=g.open_price;
                   %else
                   %    g.close_price_override=0;
                   %end

                   value=true;
                   else value=false;
                   end
        end
    end
end
```

D.2.9 Strategy Class [STRATEGY_NeuralNetwork2.m]

```
classdef STRATEGY_neuralNetwork2 < TradeGenerator
% buy and sell depending on neural network output
% Strategy for Neural Network Toolbox

    properties
        txtA = ['Buy threshold: y > theta_b'];
        txtB = ['Sell threshold: y < theta_s'];




    end
```

```
    methods (Static)
        function [value,g]=open_condition_calc (g,ts,nn)
                    % buying condition
                    % ================================================
                    % buy if we're above the closing price

                    for d=1:g.z
                        p(d)=ts.returns_normalized(g.today_day-d+1);
                    end
                    outputNeuron=sim(nn,p');
                    if   (outputNeuron > (g.y/1000))

                    % ================================================

                    value=true;
                    else value=false;
                    end
        end


        function [value,g]=close_condition_calc (g,ts,nn)
                  % selling condition
                    % ================================================
                    % sell when x% below high since high since we bought

                    for d=1:g.z
                        p(d)=ts.returns_normalized(g.today_day-d+1);
                    end
                    outputNeuron=sim(nn,p');
                    if   (outputNeuron < (-g.x/100))
                    % ================================================

                    %if  g.today_price<g.open_price
                    %    g.close_price_override=g.open_price;
```

```
                    %else
                    %     g.close_price_override=0;
                    %end

                    value=true;
                    else value=false;
                    end
            end
        end
end
```

D.2.10 Strategy Class [STRATEGY_NeuralNetwork3.m]

```
classdef STRATEGY_neuralNetwork3 < TradeGenerator
    % Only if the stock market has reached the level of a y day low and
    % and then set a stoploss

    properties
        txtA = ['Buy after increase in Payout ratio'];
        txtB = ['Sell after decrease in Payout ratio'];

        % txtx = ['Buy when y% increase since ',num2str(opt_n),
        % '-day-low'];
    end

    methods (Static)
        function [value,g]=open_condition_calc (g,ts,nn)
                    % buying condition
                    % ===============================================
                    % buy if payoutratio increases
                    outputNeuron(1)=0;
                    outputNeuron(2)=0;
                    for a=1:2
                    i=g.today_day+1-a;
                    if ~isnan (sum(ts.returns_normalized(i:i-1))) && ~isnan (sum(ts.second(i-a:i))) && ~isnan (sum(ts.third(i-a:i))) && ~isnan (sum(ts.prices(i-a:i)))
                    % p=inputs

                      p(1)=ts.second(i)/ts.second(i-a)-1;
                      p(2)=ts.third(i)/ts.second(i-a)-1;
```

```
                p(3)=ts.fourth(i)/ts.second(i-a)-1;
                outputNeuron(a)=sim(nn,p');
            end
            end
            if outputNeuron(1)>0;%outputNeuron(2)
            %
                =============================================

            value=true;
            else value=false;
            end
end


function [value,g]=close_condition_calc (g,ts,nn)
            % selling condition
            %
                =============================================
            % sell if payoutratio decreases
            outputNeuron(1)=0;
            outputNeuron(2)=0;
            for a=1:2
            i=g.today_day+1-a;
            if ~isnan (sum(ts.returns_normalized(i:i-1)))
                && ~isnan (sum(ts.second(i-a:i))) && ~isnan
                (sum(ts.third(i-a:i))) && ~isnan (sum(ts.
                prices(i-a:i)))
            % p=inputs

                p(1)=ts.second(i)/ts.second(i-a)-1;
                p(2)=ts.third(i)/ts.second(i-a)-1;
                p(3)=ts.fourth(i)/ts.second(i-a)-1;
                outputNeuron(a)=sim(nn,p');
            end
            end
            if outputNeuron(1)<-.5;%outputNeuron(2)

            %
                =============================================

            %if  g.today_price>g.open_price
             %   g.close_price_override=exp(g.x*0.01)*g.
                 open_price;
```

```
                        %else
                             g.close_price_override=0;
                        %end

                        value=true;
                        else value=false;
                        end
          end
     end
end
```

E Figures, Tables and Bibliography

List of Figures

List of Tables

References

A direct adaptive method for faster backpropagation learning: The rprop algorithm.

Fathi Abid and Mona Ben Salah. Estimating Term Structure of Interest Rates: Neural Network Vs one Factor Parametric Models. *SSRN eLibrary*, 2002. doi: 10.2139/ssrn.313561.

Oleg Alexandrov. Conjugate gradient. Internet, 2010. URL `http://en.wikipedia.org/wiki/File:Conjugate_gradient_illustration.svg`.

Sanjoy Basu. Investment performance of common stocks in relation to their price-earnings ratios: A test of the efficient markets hypothesis. *Journal of Finance*, pages 663–682, 1977.

Frank Burden and Dave Winkler. Bayesian regularization of neural networks. In David J. Lvingstone, editor, *Artificial Neural Networks*, volume 458 of *Methods in Molecular Biology*, pages 23–42. Humana Press, 2009. ISBN 978-1-60327-101-1.

Andrew P. Carverhill and Terry H. Cheuk. Alternative Neural Network Approach for Option Pricing and Hedging. *SSRN eLibrary*, 2003. doi: 10.2139/ssrn.480562.

Shie-Yui Liong Chi Dung Doan. Generalization for multilayer neural network bayesian regularization or early stopping. *Department of Civil Engineering, National University of Singapore, Singapore*, 2009.

Stanislaw H. Zak Edwain K.P. Chong. *An Introduction to Optimization*. John Wiley & Sons, 2008.

William J. Egan. The Distribution of S&P 500 Index Returns. *SSRN eLibrary*, 2007.

Javier Estrada. Black swans, market timing and the dow. *Applied Economics Letters*, page 1117, 2009.

French K Fama E. The cross-section of expected stock returns. *Journal of Finance*, 47:427–465, 1992.

A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. Martin riedmiller,heinrich braun. *University of Karlsruhe.*

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

Jorge Nocedal Jean Charles Gilberg. Global convergence properties of conjugate gradient methods for optimization. *Rapports de Recherche*, 1268, 1990.

Ojoung Kwon, K.C. Tseng, Jill Bradley, and Luna C. Tjung. Forecasting Financial Stocks Using Data Mining. *SSRN eLibrary*, 2010.

Richard Lowry. The wilcoxon signed-rank test. *Wikipedia*, 2010. URL `http://faculty.vassar.edu/lowry/ch12a.html`.

Burton Malikel. *A Random Walk Down Wall Street.* W. W. Norton & Company, Inc., 1973.

Howard B. Demuth Mark Hudson Beale, Martin T. Hagan. Neural network toolbox 7 - user's guide, 2010.

Womack K. Michaely R, Thaler RH. Price reactions to dividend initiatives and omissions: Overreaction or drift? *Cornell University, Working Paper*, 1993.

Martin F. Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Computer Science Department University of Aarhus Denmark*, 1990.

Dreman David N. and Berry Michael A. Overreaction, underreaction, and the low-p/e effect. *Financial Analysts Journal*, 51 (4):21, 1992.

Francis Nicholson. Price-earnings ratios in relation to investment results. *Financial Analysts Journal*, pages 105–109, Jan/Feb 1968.

Faizul F. Noor and Mohammad F. Hossain. A Quantitative Neural Network Model (QNNM) for Stock Trading Decisions. *Jahangirnagar Review, Part II: Social Science, Vol. XXIX, pp. 177-194, 2005.*

Kevin L. Priddy. Artificial neural networks, an introduction. pages 16–17, 2005.

Ball R. Anomalies in relationships between securities' yields and yield-surrogates. 6:103–126, 1978.

Ananth Ranganathan. *The Levenberg-Marquardt Algorithm.* 2004.

Prantik Ray and Vani Vina. Neural Network Models for Forecasting Mutual Fund Net Asset Value. *SSRN eLibrary*, 2004.

Martin Riedmiller. Rprop - description and implementation details. *University of Karlsruhe*, 1994.

B.d. Riplay. Pattern recognition and neural networks. *Cambridge University Press*, 1996.

Raul Rojas. *Neural Networks, A Systematic Introduction*. Springer, 1996.

Lanstein R Rosenberg B, Reid K. Persuasive evidence of market inefficiency. *Journal of Portfolio Management*, 13:9–17, 1985.

Ronald Schoenberg. Optimization with the quasi-newton method. 2001.

Udo Seiffert. Training of large-scale feed-forward neural networks. *International Joint Conference on Neural Networks*, 2006.

Nassim N. Taleb. *The Black Swan: The Impact of the Highly Improbable*. Random House, April 2007. ISBN 1400063515.

Leonard J. Tashman. Out-of-sample tests of forecasting accuracy: an analysis and review, 2001. URL http://www.forecastingeducation.com/archive/2000/ijfoct-outofsampletests.htm.

Wikipedia, 2010. URL http://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test.

Y. Yuan Y. H. Dai. A three parameter family of nonlinear conjugate gradient methods. *ICM-98-050*, September 1998.