06E042I

IQP MJR-0032 – 48

# Neural Networks for Financial Prediction

An Interdisciplinary Qualifying Project
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

**Submitted By:**
Venera Varbanova

**Submitted To:**
Professor Michael Radzicki (Project Advisor)
Michael J. Radzicki

Date: May 29, 2006
www.wpi.edu/~venera/IQP.pdf
venera@wpi.edu

# Acknowledgements

I would like to express my great appreciation to Professor Michael Radzicki for his patience and support.

# Abstract

The purpose of this project is threefold: firstly, to make an overview of what artificial neural networks are; secondly to demonstrate the process of building and testing a neural network; and thirdly to examine and analyze the use of neural networks as a forecasting tool, specifically a neural network's ability to predict future trends of a Stock Market Index and a publicly traded NASDAQ stock.

# Executive Summary

The technology of artificial neural networks is relatively new. It is still in the phase of constant research and improvement. Along other applications, a very intriguing topic of research nowadays is the application of neural networks for financial markets prediction. In fact, there are already some commercial tools available on the market, which claim to successfully use their neural networks for financial markets prediction.

However, due to the theoretical great return opportunities that may arise from successful neural networks, information on the inputs used, the architecture and logic of the neural networks is not freely and widely published. Thus, this project is aiming to shed some light on what neural networks are, what they can and cannot do, and how they can be utilized for stock market forecasting. More specifically, given the information limitation, the main goal of this project is to design and build a neural network prototype that can be used for stock market prediction.

There are a number of challenges that must be tackled in order to successfully deliver the forecasting neural network prototype, yet training the network, as we shall see later in the paper, is the most crucial part of the entire process. Any neural network could be only as good as the training data it is given. As a result, much attention and time is given to deciding what this simple neural network will be able to handle, and what kind of data and how much it should be given.

Through extensive research and analysis it was decided to build a multi-layer perception neural network and train it on historical S&P 500 closing price data. The Methodology section of this paper explains in detail the motivation behind this choice, and the technical details around the simple prototype neural network. An analogical experiment was performed using Intel Corp historical stock prices. Two applications were produced in the end, a S&P500 predictor, and an INTC predictor.

Testing the prototype network was the most exciting part of the project. Despite the limited time available for training and testing of the network due to the prolonged design and decision period, the test results were promising. The neural network showed that it has a sense of direction of where the market is going, and the numerical predictions it produced were better than expected.

In the process of building the network and doing literature review, many interesting ideas of how to make a neural network work better were conceived. Those ideas are summarized in a chapter on future directions towards the end of the paper, and may serve as a basis for future work on extending this project.

# Table of Contents

# 1 Introduction

Stock trading and investing are becoming ever more popular for the general public; in fact, so popular that many people now act as part-time traders while they are employed at their regular full-time jobs. The primary reason for the expanding popularity of trading is its improved accessibility. Trading over the Internet, or e-trading, has revolutionized the way people buy and sell stocks, and how they make their trading decisions. Computerization in general has had a huge effect on the stock market.

In the beginning, it all happened down on the trading floor, where paper stocks were bought and sold by the yelling and waving of stock brokers. Nowadays, the trend is for nearly everything to be computerized: one of the biggest stock exchanges in the USA, the NASDAQ, is completely computerized and the New York Stock Exchange (NYSE) has recently merged with two fully computerized exchanges, Euronext and Archipelago. For some analysts, the merger spells the end of the famous floor and some specialists speaking privately have said they feared for their jobs under the merger plan (Merkuri, 2006).

Computerized or not, there are two schools of stock analysis that aid analysts and pseudo-analyst to make buy and sell decisions. While fundamental analysis examines the financials of an equity, technical analysis analyses stock's trading patterns through the use of charts, trend lines, and many other mathematical analysis tools. With the advent of technology, technical analysis in particular has been attracting more and more enthusiasts who have began trading and investing. Thus, technical analysis is no longer done with a pen and pencil on the drawing board, but either with the aid of heavy software packages that assist the traders in taking the right decisions, or numerous web sites providing free technical analysis along with technical analysis education.

One of the newest technologies in aiding technical analysis is the neural network technology. This paper will look into it closely, discussing what neural networks are, how they are used by traders and investors, what they are good for, and where they fail. The paper will also examine the effect of neural network over technical analysis and financial markets. Furthermore, the paper will describe the design, development, and testing of a neural network prototype. In the end, ways of improving the neural network will be discussed to provide future directions for development.

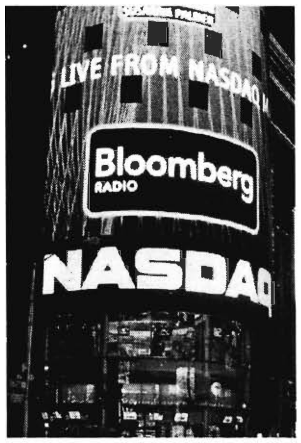
# 2 Background

In this section, we will examine in more detail the two main concepts of this paper: the stock market and neural networks. We are going to make a brief overview of each concept, and then we are going to look into their intersection, that is, where neural networks come into play in the stock trading world.

## *2.1 The Stock Market*

When talking about "the stock market", we are referring to the organized trading of securities through various exchanges and through the over-the-counter market. A "stock exchange" is a specific form of a stock market, a physical location where stocks and bonds are bought and sold, such as the New York Stock Exchange, NASDAQ or American Stock Exchange.

**The New York Stock Exchange**



**NASDAQ has no physical trading floor**

Only publicly held company stock is traded on the stock market, and the associated financial instruments (including stock options, convertibles and stock index futures). Participants in the stock market range from small individual stock investors to large hedge fund traders, who can be based anywhere.

Some exchanges are physical locations where transactions are carried out on a trading floor, by a method known as *open outcry*.[1] Such a stock exchange is The New York Stock Exchange. The other type of exchange is a *virtual* kind, composed of a network of computers where trades are made electronically via traders at computer terminals. Such a stock exchange is the NASDAQ stock exchange.

The stock exchanges are the places where buyers and sellers meet (virtually or physically) and decide on a price. Actual trades are based in the *auction market* fashion, that is, a potential buyer "bids" a specific price for a stock and a potential seller "asks" a specific price for the stock. When the bid and ask prices match, a sale takes place on a first come first serve basis if there are multiple bidders or askers at a given price. The current bid price is the highest amount any buyer is willing to pay and the current ask price is the lowest price at which someone is willing to sell. For a trade to take place, there must be a matching bid and ask price. If there is a spread between the ask and the bid price and no one steps in to buy or sell at the current price, a market maker, or specialist on NYSE, will make the trade. Market makers and specialist are not only required to make the trade when no buyer or seller changes their price, but they actually make their living by trading the spread.

The price of a stock is driven up, when there are more buyers than sellers. Similarly, when there are more sellers than buyers, the stock price goes down. The goal of each trader is to make a profit, and one way to make a profit in the stock market is to buy a stock at a certain price, and sell it later at a higher price. The higher the difference between the prices, the greater the profit is. In order for this to happen, however, the trader should have made a series of right decisions: 1) selecting a stock to trade; 2) selecting the moment to buy the stock; and 3) selecting the moment to sell the stock.

[1] A situation in which traders are wildly throwing their arms up, waving, yelling, and signaling to each other.

Traders use numerous techniques to help them make the right decisions. The techniques vary greatly; they range from rigorous mathematical analysis to following famous traders on TV shows or supposedly helpful websites; from various visual and charting methods to relying on advice from a co-worker; from applying chaos theory, genetic algorithms, evolutionary computing, neural networks, and fuzzy logic to relying on the "gut feeling". Usually it is a combination of all, or at least part, of the above.

## 2.2 Fundamental and Technical Analysis

There are a number of techniques for stock market trading, yet the analysis of financial markets is broadly divided into two main disciplines: fundamental and technical analysis. Fundamental analysis analyses trading or investing in financial markets by attempting to evaluate the intrinsic (true) value of an equity by examining company financials. According to fundamental analysis, if an equity is undervalued then the equity will go up in price and if the equity is overvalued the future price will go in the opposite direction. In order to determine whether an equity is undervalued or overvalued, a firm's financials are also compared by analysts to those of their competitors and to the average values in the S&P 500 (or other appropriate index). Based on industry, company and geopolitical news and financial data analysts forecast expected earnings. Actual earnings relative to expected earnings (or "whisper" earnings) significantly influence a stock's price.

Fundamental analysis assumes that all news and information is absorbed in the equity price. It is generally accepted that fundamental analysis is applicable over the long run. Yet, especially in the short term, the reality is that information is not disseminated perfectly and fundamental analysis may not be reliable.

Technical analysis on the other hand, ignores financial analysis and concentrates only on finding patterns through the use of charts, trend lines, support and resistance levels, and many other mathematical analysis tools, in order to predict future movements in equity's price, and to help identify trading opportunities. The basic foundations of technical analysis are that a stock's current price discounts all information available in the market, that price movements are not random, and that patterns in price movements, in very many cases, tend to repeat themselves or trend in some direction. Technical analysis is the basis of all trading software packages available (About.com. May, 2006).

Technical analysis utilizes a number of technical indicators which have been developed over the years to guide the trading decision-making. Indicators are usually grouped based on their use. Trend, momentum, volume and volatility indicators are the major price/volume indicator categories that are utilized in technical analysis. Trend indicators are used to indicate the direction of a trend. There are many trend indicators, yet some of the most widely used are moving averages, directional movement indicators such as Average Directional Movement Index (*ADX*) which indicates whether the market is trending or ranging, moving average oscillators such as Moving Average Convergence Divergence (MACD) indicating, etc (About.com. May, 2006).

Momentum indicators on the other hand measure the speed at which price is changing. Some of the most used momentum indicators are the Relative Strength Index (RSI) and The Slow Stochastic which are a popular momentum oscillators used to determine overbought/oversold levels. While trending indicators are considered lagging indicators, momentum indicators always precede price movements (Incrediblecharts.com. May, 2006).

Volume indicators are used to confirm the strength of trends. Some of the volume indicators are the Force Index which combines price and volume into one value, attempting to measure the force behind a move in price and the Volume Oscillator (*VO*) which identifies trends in volume. There are a number of other volume indicators that combine one or all of volume, price, or range (Incrediblecharts.com. May, 2006).

Volatility Indicators are used to confirm price behavior. Some of the volatility indicators are the Chaikin indicator and the Volatility Ratio. The Chaikin indicator measures volatility as the trading range between high and low for each period. The Volatility Ratio is designed to highlight breakouts from a trading range (Incrediblecharts.com. May, 2006).

Furthermore, there a hundreds of indicators and new ones are being developed constantly as traders search for a better guidance. Yet, none of the above indicators are used by themselves or are suited to all markets thus technical analysts use a combination of indicators in different market conditions. The craft in technical analysis is therefore exactly in the ability to understand and interpret correctly the many technical indicators. Thus each technical analysis is as good as the interpreter and therefore considerable amount of time must be spent in learning the principles of technical analysis and how it can be used to properly interpret technical indicators.

Because financial markets are extremely complex and human abilities are limited, software models and packages are constantly being developed. Software packages that will better and more accurately find and interpret market conditions and forecast market prices. The neural networks are one of the latest trends in applying new technologies for market prediction in order to rip profits from market inefficiency. The complexity of the neural networks seems to theoretically be very promising for application in the complex financial markets.

## *2.3 Market Predictability*

Before any financial tool for market forecasting is developed one has to answer on the fundamental question whether market prices are predictable at all? Economists are somewhat split over this question. According to the Efficient Market Hypothesis (EMH) developed in 1965 by E. Fama, the financial markets are unpredictable. The EMH, in its weak form, asserts that the price of an asset reflects all of the information that can be obtained from past prices of the asset, i.e. the movement of the price is unpredictable. The best prediction for a price is the current price and the actual prices follow what is called a random walk. The EMH is based on the assumption that all news is promptly incorporated in prices; since news is often unpredictable, prices are unpredictable. Since 1965, the EMH has had broad acceptance in the financial and academic communities.

On the other side, much effort has been expended trying to disprove the EMH. Current prevailing opinion is that the theory has been disproved  S.J. Taylor in his "Modeling Financial Time Series" published in 1994, and L. Ingber in his "Statistical mechanics of nonlinear non-equilibrium financial markets: Applications to optimized trading" (1996) advocate that the capital markets are not efficient. The market inefficiency thus suggests that financial markets are predictable (Taylor, 1994; Ingber, 1996). In this paper we make the assumption that markets are predictable and inefficient in order to develop a neural network for financial prediction.

## *2.4 Neural Networks in Investing and Trading*

Neural networks have undoubtedly already entered the financial world, and are currently used by many big financial firms as an aid to forecasting, classification, decision-making, trading, and many others. New applications of the neural networks technology are constantly being discovered, and the financial field is no exclusion.

Why do some people trust a technology to take the trading decisions for them, and not their minds? One answer is that technology lacks emotion, and emotion often hurts the trading process. For this reason, many traders make an intelligent trading system learn their trading strategies, their decision rules, their rules of thumb, and then let the system make decisions based solely on those strategies and rules. By integrating technology human weaknesses, that is, emotions such as greed and fear that often rule financial markets, are excluded. Indeed, it may be unnatural for a human to let a system decide for him, but in the trading world, this often pays off.

A second answer is the real advantage of neural networks to technical traders, that is, the neural network's ability to recognize patterns - especially patterns that are not obvious to the human eye. Technical trading is all about identifying patterns. Classification of stocks into out-performers and others types is also a unique ability of neural networks, but its full potential power is yet undiscovered.

## *2.5 History of Neural Networks in Trading*

Artificial neural networks have proven to be a useful tool for prediction of non-linear time series, such as financial time series, and for classification of patterns, such as the patterns in the stock market price movements. Thus, from the very beginning of the neural networks research, there were significant endeavors to successfully apply the neural network technology to the financial markets, motivated by great potential profitability.

The earliest work in neural computing goes back to the 1940's when McCulloch and Pitts introduced the first neural network computing model. In the 1950's, Frank Rosenblatt's work resulted in a two-layer network, the perceptron, which was capable of learning certain classifications by adjusting connection weights. Although the perceptron was successful in classifying certain patterns, it had a number of limitations. The perceptron was not able to solve the classic XOR (exclusive or) problem. Such limitations led to the decline of the field of neural networks

In 1969, it became apparent that it was important for neural nets to solve nonlinear classifications (i.e., to adjust the weights and minimize the errors in nonlinear weighting functions in neural nets with more than two layers). In 1974, Paul Werbos solved the nonlinear classification problem by inventing the "backward propagation of errors" technique for his Harvard Ph.D. dissertation. Once the nonlinear classification problem was solved, people began applying neural nets to many problems, including equity trading.

During the 1990s, there was a boom in the application of neural nets to trading, but there weren't too many successes. Lots of institutions spent lots of money, but the results were not very good. One of the main reasons neural nets produced poor results during the 1990s was that the engineers who were writing the code did not adequately understand trading, and the traders using the neural nets did not adequately understand the underlying programming and computer science.

During the bear market of 2000-2003, many traders and institutions lost money and did not have the resources to pour into neural net research, even though neural net technology and computer speed were improving daily. Nowadays (2005-2006) articles in financial journals and papers are expressing their optimism on neural networks, saying: "Now is the time to revisit neural networks." During the years, it became clear to the financial professionals and researchers that the key to developing a good neural network for trading is to first have a trading system that works without the neural network. Once a system is found to work, a neural network can improve it.

## *2.6 Neural Networks Fundamentals*

A first wave of interest in neural networks (also known as “connectionist models” or “parallel distributed processing”) emerged after the introduction of simplified neurons by McCulloch and Pitts in 1943 (McCulloch & Pitts, 1943). These neurons were presented as models of biological neurons and as conceptual components for circuits that could perform computational tasks.

The interest in neural networks re-emerged only after some important theoretical results were attained in the early eighties (most notably the discovery of error back-propagation), and new hardware developments increased the processing capacities. This renewed interest is reflected in the number of scientists, the amounts of funding, the number of large conferences, and the number of journals associated with neural networks. Nowadays most universities have a neural networks group, within their psychology, physics, computer science, or biology departments.

Artificial neural networks can be most adequately characterized as "computational models" with particular properties such as the ability to adapt or learn, to generalize, or to cluster or organize data, and which operation is based on parallel processing. However, many of the above mentioned properties can be attributed to existing (non-neural) models; the intriguing question is to which extent the neural approach proves to be better suited for certain applications than existing models. To date an equivocal answer to this question is not found.

An artificial network consists of a pool of simple processing units which communicate by sending signals to each other over a large number of weighted connections.

A set of major aspects of a parallel distributed model can be distinguished:

- A set of processing units ('neurons,' 'cells');
- A state of activation for every unit, which is equivalent to the output of the unit;
- Connections between the units. Generally each connection is defined by a weight which determines the effect which the signal of the sending unit has on the receiving unit;

- A propagation rule, which determines the effective input of a unit from its external inputs;
- An activation function, which determines the new level of activation based on the effective input and the current activation (i.e., the update);
- An external input (a.k.a. bias, offset) for each unit;
- A method for information gathering (the learning rule);
- An environment within which the system must operate, providing input signals and (if necessary) error signals.

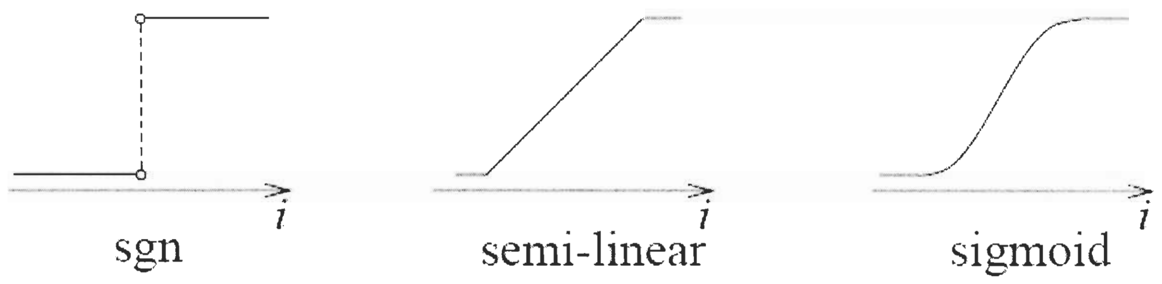
Each processing unit (often called "neuron" after its biological inspiration or "node") performs a relatively simple job: receive input from neighbors or external sources and use this to compute an output signal which is propagated to other units. Apart from this processing, a second task is the adjustment of the weights. The system is inherently parallel in the sense that many units can carry out their computations at the same time.

Within neural systems it is useful to distinguish three types of units: input units which receive data from outside the neural network, output units which send data out of the neural network, and hidden units whose input and output signals remain within the neural network.

In most cases we assume that each unit provides an additive contribution to the input of the unit with which it is connected. The total input to a unit is simply the weighted sum of the separate outputs from each of the connected units plus a bias or offset term. The contribution for positive weight is considered as an excitation and for negative weight as inhibition. In some cases more complex rules for combining inputs are used, in which a distinction is made between excitatory and inhibitory inputs. We call units with a propagation rule sigma units.

We also need a rule which gives the effect of the total input on the activation of the unit. We need a function which takes the total input and the current activation and produces a new value of the activation of the unit. Often, the activation function is a non-decreasing function of the total input of the unit although activation functions are not restricted to non-decreasing functions. Generally, some sort of threshold function is used: a hard limiting threshold function (a sgn function), or a linear or semi-linear function, or a smoothly limiting threshold. For this smoothly limiting function often a sigmoid (S-shaped) function is used (Figure 1).

**FIGURE 1: Activation Functions**



A neural network topology refers to the pattern of connections between the units and the propagation of data. As for this pattern of connections, the main distinction we can make is between:

- Feed-forward networks (our case), where the data flow from input to output units is strictly feed-forward. The data processing can extend over multiple (layers of) units, but no feedback connections are present, that is, connections extending from outputs of units to inputs of units in the same layer or previous layers.
- Recurrent networks that do contain feedback connections. Contrary to feed-forward networks, the dynamical properties of the network are important. In some cases, the activation values of the units undergo a relaxation process such that the network will evolve to a stable state in which these activations do not

change anymore. In other applications, the change of the activation values of the output neurons is significant, such that the dynamical behavior constitutes the output of the network (Pearlmutter, 1990).

A neural network has to be configured such that the application of a set of inputs produces (either "direct" or via a relaxation process) the desired set of outputs. Various methods to set the strengths of the connections exist. One way is to set the weights explicitly, using a priori knowledge. Another way is to 'train' the neural network by feeding it teaching patterns and letting it change its weights according to some learning rule.

We can categorize the learning paradigms in two distinct sorts. These are:

- Supervised learning (our case) or Associative learning in which the network is trained by providing it with input and matching output patterns. These input-output pairs can be provided by an external teacher, or by the system which contains the network (self-supervised).
- Unsupervised learning or Self-organization in which an (output) unit is trained to respond to clusters of pattern within the input. In this paradigm the system is supposed to discover statistically salient features of the input population. Unlike the supervised learning paradigm, there is no a priori set of categories into which the patterns are to be classified; rather the system must develop its own representation of the input stimuli.

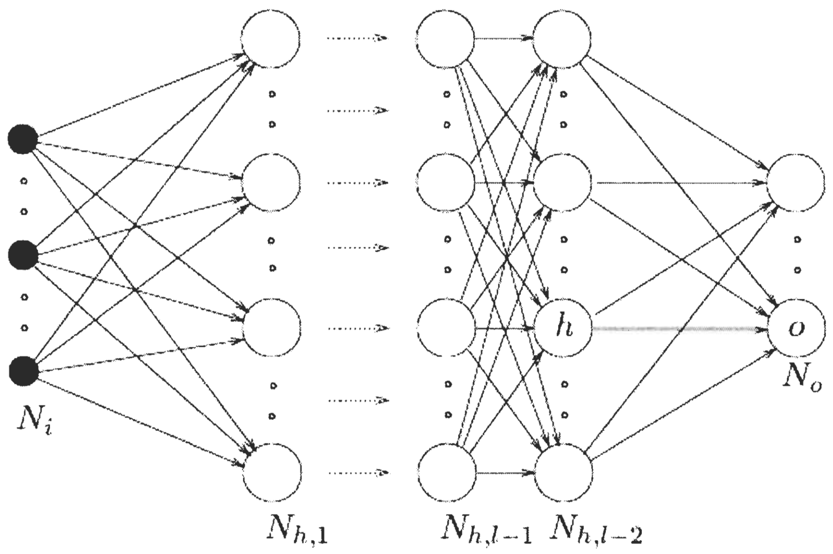
Here is some additional clarification of the most important neural network terminology:

- Output vs. activation of a unit. Since there is no need to do otherwise, we consider the output and the activation value of a unit to be one and the same thing. That is, the output of each neuron equals its activation value.

- Bias, offset, threshold. These terms all refer to a constant (i.e., independent of the network input but adapted by the learning rule) term which is input to a unit. They may be used interchangeably, although the latter two terms are often envisaged as a property of the activation function. Furthermore, this external input is usually implemented (and can be written) as a weight from a unit with activation value 1.
- Number of layers. In a feed-forward network, the inputs perform no computation and their layer is therefore not counted. Thus a network with one input layer, one hidden layer, and one output layer is referred to as a network with two layers. This convention is widely though not yet universally used.
- Representation vs. learning. When using a neural network one has to distinguish two issues which influence the performance of the system. The first one is the representational power of the network; the second one is the learning algorithm.

Below is an introduction of one of the most popular kinds of neural networks, the one that is used for the experiments in this project, namely the multi-layer feed-forward network. As its name suggests, it is a feed-forward network with layers of processing units. The central idea behind the learning of this network is that the errors for the units of the hidden layer are determined by back-propagating the errors of the units of the output layer. For this reason the method is often called the back-propagation learning rule. Back-propagation can also be considered as a generalization of the delta rule for non-linear activation functions and multilayer networks (Figure 2).

**FIGURE 2: Feed-Forward Network**

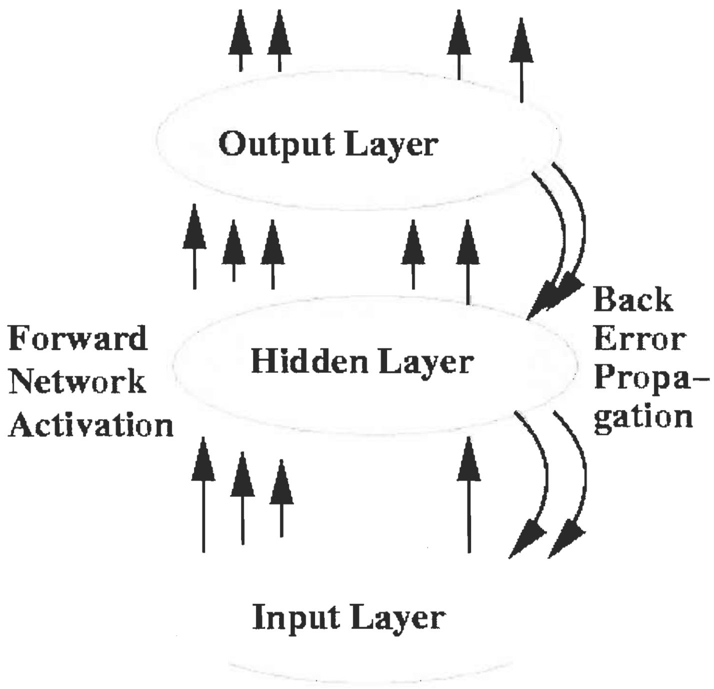


A feed-forward network has a layered structure. Each layer consists of units which receive their input from units from a layer directly below and send their output to units in a layer directly above the unit. There are no connections within a layer. The inputs are fed into the first layer of hidden units. The input units are merely 'fan-out' units; no processing takes place in these units. The activation of a hidden unit is a function of the weighted inputs plus a bias. The output of the hidden units is distributed over the next layer of hidden units, until the last layer of hidden units, of which the outputs are fed into a layer of output units.

Although back-propagation can be applied to networks with any number of layers, just as for networks with binary units it has been shown (Hornik, Stinchcombe, & White, 1989; Funahashi, 1989; Cybenko, 1989; Hartman, Keeler, & Kowalski, 1990) that only one layer of hidden units suffices to approximate any function with finitely many discontinuities to arbitrary precision, provided the activation functions of the hidden units are non-linear (the universal approximation theorem). In most applications a feed-forward network with a single layer of hidden units is used with a sigmoid activation function for the units (Figure 1).

Back-propagation is relatively easy to explain and understand. Here is one non-mathematical explanation of back-propagation. When a learning pattern is clamped, the activation values are propagated to the output units, and then the actual network output is compared with the desired output values. We usually end up with an error in each of the output units. Let's call this error e(o) for a particular output unit o. We have to bring e(o) to zero. The simplest method to do this is the greedy method: we strive to change the connections in the neural network in such a way that, next time around, the error e(o) will be zero for this particular pattern. We know from the delta rule that, in order to reduce an error, we have to adapt its incoming weights.

**FIGURE 3: Back-Error Propagation**



Despite the success of the back-propagation learning algorithm, there are some aspects which make the algorithm not guaranteed to be universally useful. Most troublesome is the long training process. This can be a result of a non-optimum learning rate and momentum. A lot of advanced algorithms based on back-propagation learning have some optimized method to adapt this learning rate, as will be discussed in the next section. Outright training failures generally arise from two sources: network paralysis and local minima, but these are not going to be discussed here.

How good are multi-layer feed-forward networks really? As it will be shown in the experiments later in the paper, the approximation of the network is not perfect. The resulting approximation error is influenced by:

1. The learning algorithm and number of iterations. This determines how good the error on the training set is minimized.
2. The number of learning samples. This determines how good the training samples represent the actual function.
3. The number of hidden units. This determines the 'expressive power' of the network. For “smooth” functions only a few number of hidden units are needed, for wildly fluctuating functions more hidden units will be needed.

Because of time and computing power constraints, for this project the sample size will be relatively small, as well as the number of hidden units. The quality of the produced network is not expected to be very high, but it will be enough to demonstrate the potential of the neural networks technology.

# 3 Methodology

In this section, we will examine in details how to design, build, train, test, and finally use a neural network. As we shall see, there is no single right design and there is not single right testing strategy. In fact, it is not so important what the design is. What is of utmost importance, though, is the training process and the data that is used for it. A neural network should not memorize, it should generalize, and whether it is the former case, or the latter, entirely depends on the training process. Even if the training process is correct and consistent, the neural network is still a useless tool without the right training data.

In order to choose the right training data, we should be very clear on what we expect the network to be able to go. If we give it the right data and we train it properly, the network will give back results in return. But if we incorrectly conclude that there is a certain pattern in a set of data, and there is not one, then we are confusing our network, and the network will not able to generalize anything, or will not be able to extract the pattern that does not even exist. By this point it should be clear that building a working neural network is not trivial. An effective training process may take many days, and even months. Selecting and preprocessing appropriate training data may also take very long. Even when we succeed with training data selection, and with the training process, we cannot say "Here is the complete product" and leave it on the shelf. The neural networks made for dynamic systems like the stock market require daily, if not hourly maintenance, depending on their goal and usage. A smart thing would be to automate these processes, so that the neural network users only have to worry about how to interpret the neural network results.

## *3.1 Identifying Project Goals*

The main goal of this project was to design and build a prototype neural network, and demonstrate the applicability of the neural network model in predicting financial markets. The prototype model was tested on predicting a stock market index, and a stock.

## *3.2 Obtaining and Assessing Available Historical Market Data*

Ideally, any neural network should be trained on all data which is available, such as opening price, closing price, high for the day, low for the day, volume, and all available technical numeric indicators. The more kinds of relevant input data are included in the model, the better the accuracy of the model predictions. However, the trade-off is that the model becomes very complex, and the time to train the model and the computing power requirements sharply rise.

For the prototype neural network build for this project, it was decided to use a reduced amount of data, namely 280 data points, in order to minimize training time and demand for computing power. The S&P 500 stock market index was chosen for the stock market index version of the model, and the Intel Corporation stock, which is traded on NASDAQ as INTC, was chosen for the stock version of the model. For the S&P 500 predictor, the data chosen for training was from January 1975 to February 1976. For the INTC predictor, the data chosen was from April 2005 to May 2006. Each of those sets contained 280 data points, which were selected to be the closing prices for the day.

FIGURE 4: S & P 280 day chart

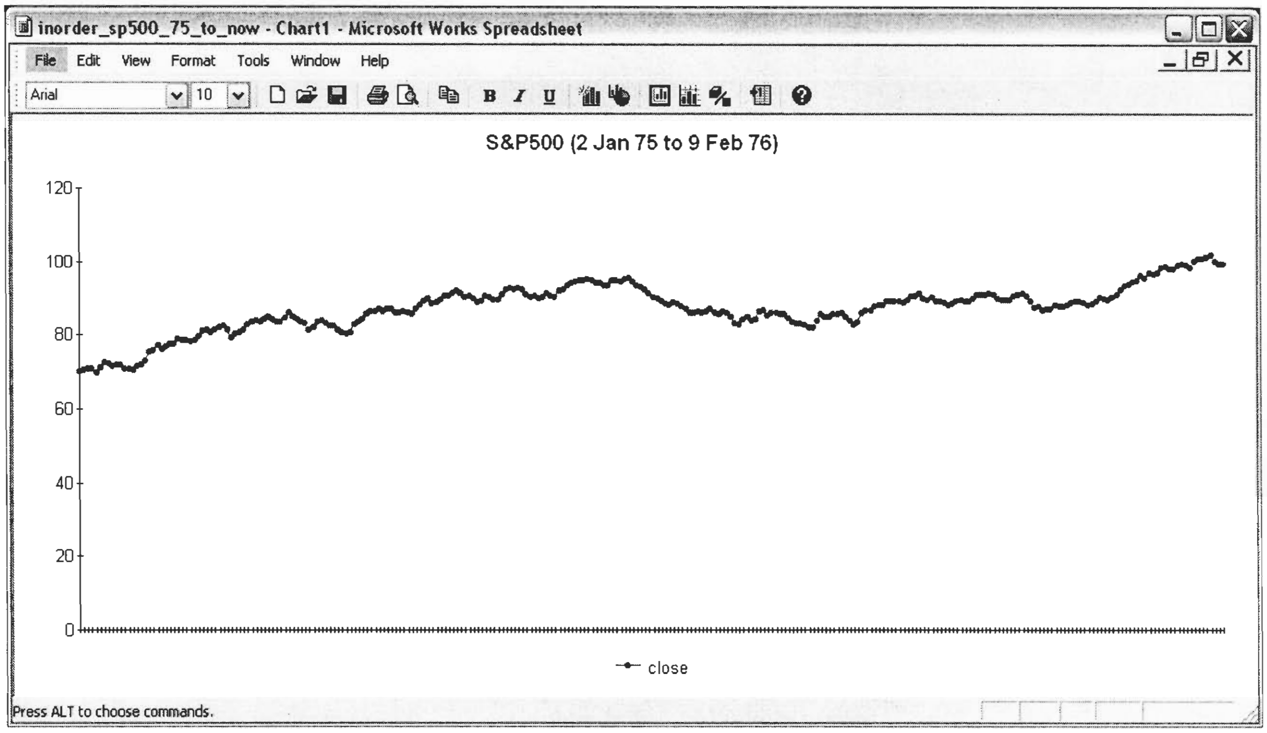
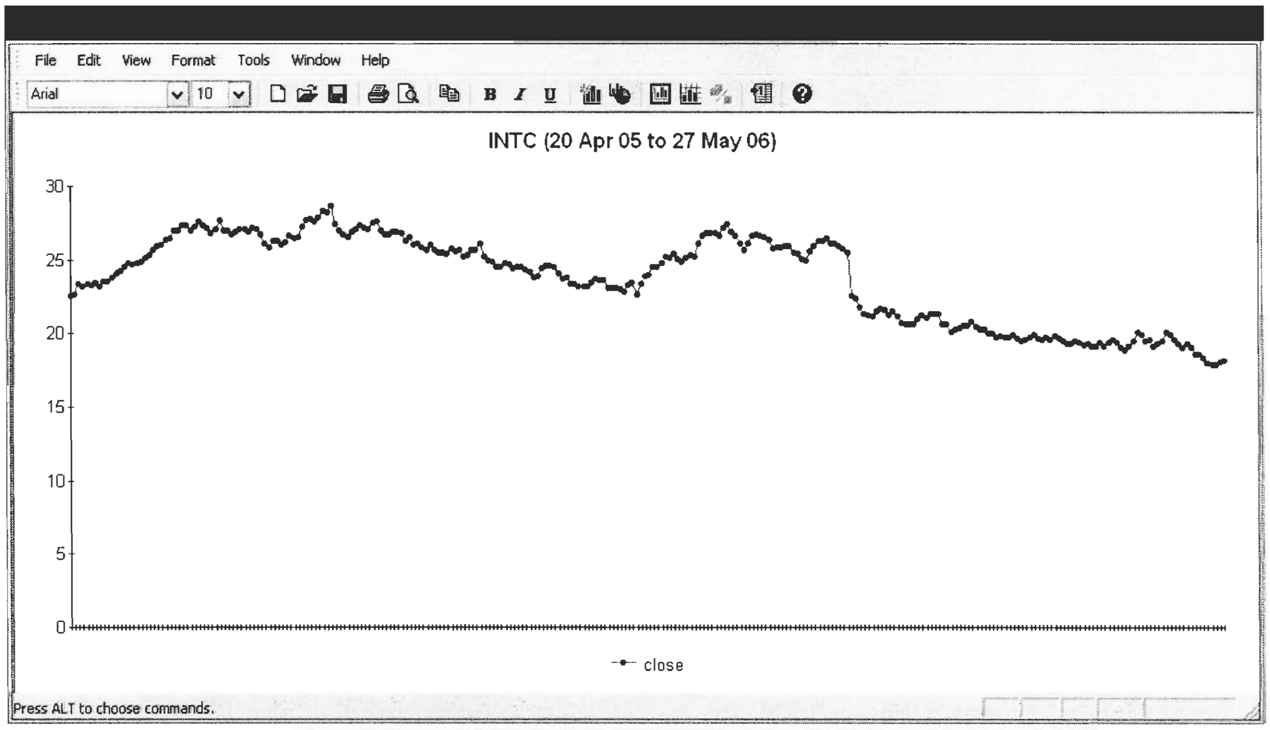


FIGURE 5: INTEL 280 day chart



## *3.3 Designing the System*

The neural network model has 3 layers: 1 input layer, 1 hidden layer, and 1 output layer. The decision to include only one hidden layer was based the literature review. Systems with more than one hidden layer are more prone to overfitting the data than those with 1 hidden layer. Also, the training time increases dramatically when more hidden layers are added, making the experiments too prolonged in time to perform, or requiring unavailable computing resources.

The input layer has 30 input nodes. This means the neural network uses data from 30 days back to make its prediction for the new day. The hidden layer has 10 hidden nodes. The number of nodes in the hidden layer is usually determined by the amount of training data. Since in my case, the training data was minimized, the number of hidden nodes was kept small as well. The output layer has one node, namely the forecasting closing price for the next business day.

The 280 data points available are distributed among the training, evaluation, and test set as follows:

[30-179] TRAINING SET

[180-259] TEST SET

[260-279] EVALUATION SET

The window size, that is, the size of the input vector, was 30 and that is why the training set begins from 30, not from 0.

## *3.4 Implementing the System*

The implementation details, the source code, and the relevant technical details are all described in the Appendix on Programming Details.

## *3.5 Tuning the System*

After the system was ready and able to run, a lot of initial tests were performed. They helped remove some of the source code issues, and determine the needed preprocessing of the data. The data was preprocessed by division by 100. Performing this preprocessing of the data allowed the neural network to finish training faster, as the computed error was minimized faster. The need for saving the results to an output file, in addition to showing them on the screen, was observed, and so code to produce a formatted output file was added. The output file was formatted such that it would be ready to be imported in Excel or some other spreadsheet software system. The file contains 2 columns, Predicted and Actual, and it is very easy to plot a chart and see how successful the neural network was at predicting. Another way of assessing the system would be to see if the direction of change of the predicted values followed the actual direction of change. The difference between the predicted and actual direction change would be used to determine the error rate using any spreadsheet software.

## *3.6 Experiment Setup and Results*

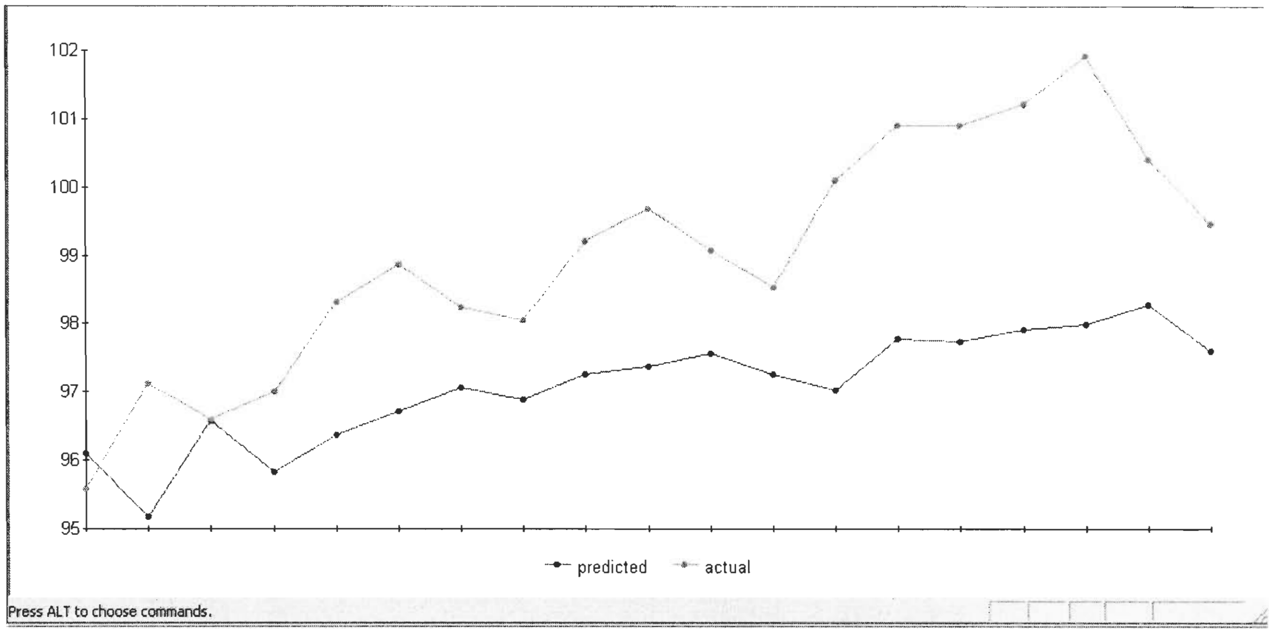
Both experiments were performed using the same source code. The difference between the two experiments is that for the first experiment, the array in backprop.cpp was filled with 280 S&P500 historical values, while for the second experiment, the 280 closing  prices of Intel Corp (symbol: INTC) were used. Before entering the data in backprop.cpp file for each experiment, all closing prices were first divided by 100 to minimize the time to compute the total error.

**Technology used for performing the tests:**

All the tests were performed in exclusion (one by one) on a high processing power laptop with Intel T2400 Core Duo processor, 1GB RAM, abundance of disk space (~80GB), so the times for training shown here may be better than if the system was run on an average personal computer.

## Experiment 1:

**FIGURE 6: S & P 280 predicted vs. actual results**



**Data used**: 280 daily closing prices of S&P500 starting 2 Jan 1975.

**Preprocessing**: Yes, division by 100.

**Time to train**: About 6 hours of full CPU utilization.

**Result**: Overall follows the trend, achieves success rate of 56% (error rate is 44%), numerical price predictions not very accurate.

**Warning**: Is that only possible when the actual direction is up? Will it ever suggest "down" by itself before the real trend goes down?

**Problem**: The numerical predictions are always below in price then the actual data.

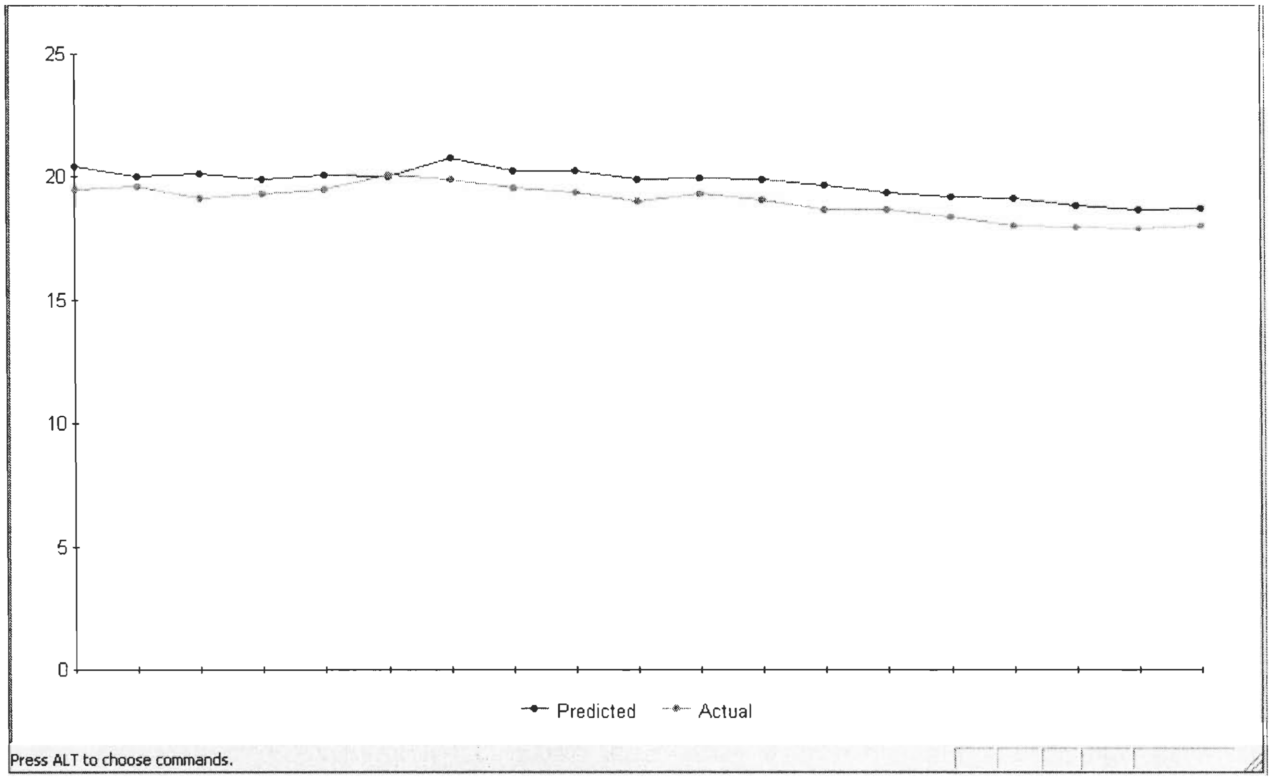
**Possible reason:** Not given enough data so that the network could catch the "speed" of the stock market that is, prices go up faster than the network projects.

**Solution**: More training data - at least several years.

**Error Rate computed:** 44%

## Experiment 2:

**FIGURE 7: INTEL predicted vs. actual results**



**Data used**: 280 daily most recent closing prices of the INTC stock.

**Preprocessing**: Yes, division by 100.

**Time to train:** About 2 hours of full CPU utilization.

**Result**: Follows the trend, relatively low error rate, better numerical price predictions, error rate = 33%, success rate = 67%.

**Warning**: the data the neural network had to predict did not have many ups and downs, the pattern was smoother than in the first experiment, so it is possible this is the reason for the better numeric estimations.

NOTE: For more information on the experiment results, please refer to the attached spreadsheets, which contain plots of the data, error rate computation, and more.

# 4 Future Directions

Here are briefly listed a few suggestions that could be used for further work on this project:

- A full planned statistical experiment with the neural network which would further prove or disprove the effectiveness of the network.
- Comparative experiment between the neural network and a linear forecasting tool which would further prove or disprove the greater effectiveness of the network compared to the effectiveness of available linear forecasting models.
- Comparative experiment between different neural network methods, based on convergence rate, prediction accuracy, training time requirements and stability of results.

Here are some suggested design improvements/experiments.

- Train the neural network on a prolonged period of data (i.e. several years).
- Increase the number of nodes in the hidden layer. Doing so should increase the accuracy of the system.
- Increase the number of hidden layers in the neural network.
- Change the neural network design from a feed-forward one to a recurrent one. Doing so could improve the performance of any time series forecasting network.
- Have the system output its forecast alongside a number indicating its own level of confidence in its prediction. Reject/ignore results whenever the system has low confidence in its prediction.

- Make the neural network read the input data from a CSV file; do not store the training data in an internal array. This would allow for an easy retraining of the neural network with new data.
- Automate the neural network training and retraining process, so that the neural network is periodically retrained with new data from a data feed.

Try hybrid or alternative methods:

- Symbolic processing, which filters noise and could help a recurrent neural network (RNN) model work better, as it is proven RNNs can learn simple grammars.
- Recurrent neural networks, which provide for stronger expression of the temporal relationship of the data points; RNNS could infer rules and construct a deterministic finite state automata.
- Self-organizing modeling.
- Genetic algorithms and neural networks.
- Evolutionary computing, genetic algorithms, and neural networks.
- Neural networks and chaos theory.
- Neural networks and fuzzy logic.
- Neural networks and Bayesian statistical models.
- Have a group of neural network and other models vote independently, and produce a voting result on which the prediction is based.

## 5 Impact on Society

Technology has evolved so quickly in the recent decades that it is now obvious that the trend of applying technology to virtually every aspect of life is going to continue well into the future. The impact of technology on stock trading and investing could be observed in the mergers of international stock exchanges, the increased number of people who are trading stocks and derivatives, the increased number of part-time day traders, the increased number of discount brokers online offering trade execution at lower fees, the new and improved trading systems which incorporate innovative technologies like neural networks and evolutionary computing, and provide for more profitability.

A recent series of mergers on non-electronic stock exchanges with fully electronic ones is very indicative of the direction in which the financial world is going. There are apparent moves by NASDAQ to acquire the London Stock Exchange. At the same time, the NYSE Group (owner of the New York Stock Exchange) offered 8 billion euros ($10.2b) in cash and shares for Euronext on May 22, 2006, outbidding a rival offer for the European Stock exchange operator from Germany's Deutsche Börse, the German stock market. The new entity, to be called NYSE Euronext, intends to form the world's first global stock market, with continuous trading of stocks and derivatives over a 21-hour time span. It is also important to mention Archipelago's acquisition by the NYSE Group. The Archipelago Exchange is a fully electronic stock exchange that agreed to merge with the New York Stock Exchange in April 2005 to form the for-profit NYSE Group. Thus, globalization and computerization are a strong trend in the financial world, and this trend is to a great extend driven by the advancement of technology.

Another clear trend is that more and more people are executing trades online. Low-commission equity trading seems to have been the catalyst for the online brokerage revolution, and is the dominant securities product sold online. Another motivation besides the low commission is the execution speed of the trades.

The impact of electornic trading technology on the financial markets, and indirectly on the society as a whole, could be summarized in four main points. First, electronic trading technology is cost-efficient, both in the sense that it lowers start-up costs for new systems and reduces continuing operating costs substantially. Second, electronic trading technology changes the dynamics of the marketplace by removing physical constraints such as geography and thus increasing number of participants in a market. Third, electronic trading technology has a great potential for disintermediating the markets, allowing buyers and sellers to meet directly. Finally, electronic trading technology has blurred the distinction between broker-dealers and exchanges (Unger, 1999).

Nowadays, thanks again to the advancement of technology and technical analysis, and of their availability and accessibility online, many people whose day job involves work with a computer which is connected to the Internet are acting as part-time traders while at work. This is a controversial practice, since it distracts the workers and reduces their efficiency.

Yet another trend is the increasing usage of trade management platforms, and their integration with innovative technologies like the neural network technology, or evolutionary computing. Most of those systems are still not fully mature, but it will not be a big surprise if several years (or maybe decades) from now a close-to-perfection trading system with a very powerful neural network is available to the general public. It is very probable that such system is very profitable in its early years of adoption, but as its popularity increases and more people start using this same trading system with the same neural net (or another technology) integrated with it, a lot of people will be trying to execute the exact same trades, and the result from that will be greatly reduced profitability. Or maybe another scenario will take place. If the system is greatly customizable, then it is less likely the situation described about would occur. For example, if the trading system is integrated with a neural network whose training and/or architecture are customizable, then it is very unlikely that the systems would produce the same results. Neural networks are very sensitive to the amount and quality and the type of the training data, as well as to their internal architecture, so it will be unlikely that two networks produce the same result.

It will be interesting to observe in the future how, and if, the neural networks technology significantly impacts technical trading and, conversely, how the changing nature of equity, derivative, commodity, bond, currency, etc. markets (i.e., their increasingly interrelated and computerized nature) drives innovations in technical trading such as neural nets. The neural networks technology is expected to improve technical trading, as it would provide for the discovery of time series financial data patterns not readily visible to the human eye. At the same time, the changing nature of the stock exchanges from non-computerized to completely computerized would likely increase the understanding, and consequently, the usage and profitability of technical trading, which is turn will stimulate further research and improvement of technical trading models such as neural net, genetic algorithms, and evolutionary computing models.

In the most extreme case, when all traders use real-time, intelligent trading systems and ideally trained and configured neural networks, all profit opportunities the markets create would immediately be exploited, and the Efficient Market Hypothesis would hold true. In other words, if everybody traded with the perfect trading system using the perfect neural network, absorbing all news in the second it arrives, then nobody will be able to predict the market. Fortunately, this is not likely to become reality.

In conclusion, the financial industry is becoming more and more dependent on advanced computer-technologies in order to maintain competitiveness in a global economy. Neural networks represent an exciting technology with a wide scope for potential applications, ranging from routine credit assessment operations to driving of large scale portfolio management strategies. Some of these applications have already resulted in dramatic increases in productivity, and have thus had a positive effect on society. The hope is that neural networks would prove to be worth the investment in research and development, and would be well-understood and properly used by traders, investors, educators, researchers, and many others.

# 6 Conclusion

Artificial neural networks are universal and highly flexible function approximators first used in the fields of cognitive science and engineering. In recent years, neural network applications in finance for such tasks as pattern recognition, classification, and time series forecasting have dramatically increased. However, the large number of parameters that must be selected to develop a neural network forecasting model have meant that the design process still involves much trial and error. The purpose of this project was to provide an introduction to neural networks, and demonstrate how a prototype forecasting neural network is designed, created, and evaluated. The impact of neural networks and trading technologies on society were examined.

The two instances of the model build had different performance, although the architecture and parameters of both instances were the same, only the training data was different. This clearly demonstrated the importance on the quality of the training data on the performance of the network. The flexibility of the model was also shown, as single model suited two different predictors. Finally, the forecasting potential of the neural network technology was demonstrated, as such a simple system as the one build for this project had success rate of 56% for the first experiment on predicting S&P500, and 67% on predicting INTC. Clearly, the neural networks technology is a powerful one, and its full potential is yet to be uncovered.

## Bibliography:

Andrew W., MacKinlay Craig. *A Non-Random Walk Down Wall Street.* Princeton University Press, 1999.

Back A.D., Tsoi A.C., *FIR and IIR synapses, a new neural network architecture for time series modeling.* Neural Computation, 3(3):375–385, 1991.

Bartlmae K., Gutjahr S., Nakhaeizadeh G. [1997] "Incorporating Prior Knowledge About Financial Markets Through Neural Multitask Learning", University of Karlsruhe, Germany.

Caldwell, R.B. [1994] *Design of Neural Network-based Financial Forecasting Systems: Data Selection and Data Processing.* NEUROVE$T JOURNAL, Vol.2, No.5, pp. 12-20.

Cheng W., Wagner L., and Lin C. [1996] "Forecasting the 30-year U.S. Treasury Bond with a System of Neural Networks, NeuroVe$t Journal, January/February 1996.

Chuah, K. L. [1992] *A Nonlinear Approach to Return Predictability in the Securities Markets Using Feedforward Neural Network*, Ph.D. dissertation, Washington State University.

"Fundamental vs Technical Analysis." About.com. *http://daytrading.about.com/cs/charts/a/anylisis.htm.* accessed May 23, 2005.

Gately, E. [1995] "Neural Networks for Financial Forecasting", John Wiley & Sons, Inc. New York, NY, USA.

Ingber L. *Statistical mechanics of nonlinear nonequilibrium financial markets: Applications to optimized trading.* Mathematical Computer Modeling, 196l.

Ivakhnenko, A.G., Muller, J. A. [] "Recent Developments of Self-Organizing Modeling in Prediction and Analysis of Stock Market", Glushkov Institute of Cybernetics, Ukraine.

Kaastra I., Boyd M. [1995] "Designing a neural network for forecasting financial and economic time series", Canada.

Kamijo, K. and Tanigawa, T. [1990] "Stock Price Pattern Recognition: A Recurrent Neural Network Approach," Proc. of the IJCNN, San Diego, Ca, pp. 215-221.

Kehagias A., Petridis V. *Time series segmentation using predictive modular neural networks.* Neural Computation, 9:1691–1710, 1997.

Kimoto, T., and Asakawa, K. [1990] "Stock Market Prediction System with Modular Neural Networks," Proceedings of the IJCNN, San Diego, Ca, pp. 1-6.

Kutsurelis, J.E. [1998] "Forecasting Financial Markets Using Neural Networks: An Analysis of Methods and Accuracy", NASA, 1998.

Malkiel, B.G. *A Random Walk Down Wall Street. Norton*, New York, NY, 1996.

Merkuri, Ervin. Policy and Management Paper: *Transformation at the New York Stock Exchange: A Constructive Review.* The Heinz School Review. 7 Apr 2006. http://journal.heinz.cmu.edu/Current/NYSE/nyse7.html. accessed May 27, 2006.

Mukherjee S., Osuna E., Girosi F. *Nonlinear prediction of chaotic time series using support vector machines.* In J. Principe, L. Giles, N. Morgan, and E. Wilson, editors, IEEE Workshop on Neural Networks for Signal Processing VII, page 511. IEEE Press, 1997.

NEUROVE$T JOURNAL [1995] *Special issue on Data Selection and Preprocessing*, Vol.3, No.6.

Principe J.C., Kuo J-M. *Dynamic modeling of chaotic time series with neural networks.* In G. Tesauro, D. Touretzky, and T. Leen, editors, Advances in Neural Information Processing Systems, 7, pages 311–318. MIT Press, 1995.

Ruggiero Jr., Murray A. Futures magazine: "Neural Networks: Where They are Now". April 2006.

Russell S., Norvig P. (1995*) Artificial Intelligence: A Modern Approach, Prentice Hall Series in Artificial Intelligence.* Englewood Cliffs, New Jersey.

Saad E.W., Prokhorov Danil V., Wunsch II D. C. *Comparative study of stock trend prediction using time delay, recurrent and probabilistic neural networks.* IEEE Transactions on Neural Networks, 9(6):1456, November 1998.

Taylor S.J., Modeling Financial Time Series. J. Wiley & Sons, Chichester, 1994.

"Technical Indicators." Incrediblecharts.com. *http://www.incrediblecharts.com/technical/indicator_basics.htm.* accessed May 23, 2005.

Tchaban, Griffin, Taylor. (1997) *A Comparison between Single and Combined Back Propagation Neural Networks in the Prediction of Turnover*. Dept. of Comput. Sci., Univ. Coll. Chester (UK).

Unger, Laura. *Impact of Electronic Trading on the Securities Markets.* U.S Securities and Exchange Commission. October 28, 1999. http://www.sec.gov/news/speech/speecharchive/1999/spch313.htm accessed June 1, 2006.

Zemke S. [2003] "Data Mining for Prediction. Financial Series Case", Ph.D. thesis, The Royal Institute of Technology, Sweden.

## Appendix A: Source Code

The source code for this project is based loosely upon code by Karsten Kutza from ww.geocities.com/CapeCanaveral/1624/bpn.html. The code is written in C++ using Microsoft Visual Studio .NET as a development environment.

Net (short for neural network) is a C++ class providing a generic interface for training and executing simple back-propagation neural networks, for performing a variety of tasks.

The source code consists of the following files:

1) Net.h - a header file defining the NeuralNetwork namespace.

2) Net.cpp - defines the neural network interface.

3) NetLayer.cpp - defines a neural network layer interface.

4) backprop.cpp - supports training and running of the neural network.

5) stdafx.cpp - automatically generated by Visual Studio .NET.

6) stdafx.h - automatically generated by Visual Studio .NET.

For each experiment, part of the source code is changed, namely the contents of the historical data array, the name of the binary file containing the neural network, and the name of the output file which contains the results from running the network.

### RUNNING THE NEURAL NETWORKS

Use sp500-predictor.exe to run the s&p500 predictor. It may make up to several hours to train the network, and produce results. The results will be generated both on the black screen which appears when one runs the exe file, and into a file named results-sp500.txt, whose location is the same as the exe file's location.

To better analyze the results, open the exe file with Microsoft Excel or Microsoft Works or other spreadsheet software able to handle data in CSV (comma separated values) format. Best visual effect is achieved by selecting both columns together with the titles "predicted" and "actual", and invoking a charting tool.

Apart from the text file with results, we should not forget there is the black window, in which the program is writing its status. We know that the neural network has finished executing when the message "Press any key to continue..." comes up in the botton of the window. This lets one look at the results on the screen, including the minimized error, and the final error. After pressing a key, the window disappears, that is, the predictor quits.

Running the intc-predictor.exe is completely analogical.

## PROGRAMMING AND DESIGN DETAILS

The neural network class itself is the class NeuralNetwork::Net. It depends on the class NeuralNetwork::NetLayer, both of which are defined in Net.h. The file backprop.cpp is a sample of using Net to train a system to predict stock market prices.

Seed the random number generator (if you wish to randomize weights).

Create an instance of NeuralNetwork::Net, passing in the number of layers, number of perceptrons in each layer, momentum factor, learning rate, and gain for the sigmoid function.

Randomize or clear weights, whichever is desirable, with Net::randomizeWeights() or Net::clearWeights().

Create a class (or two, or more) inheriting from NeuralNetwork::ExampleFactory and implement getExample, which returns via its parameters a new input/output pair each time it's called, and numExamples, which is an estimate of the total number of unique examples. Create an instance of one of these classes for your training set (the set the network attempts to minimize error on) and an instance of one of these classes for your test set (the set used to identify overtraining). These will be passed to autotrain.

Use Net::autotrain() to train the network to its optimum error level on the test set. It will perform a specified number of epochs, then test if error on the test set has exceeded minimum error times the specified cut off error. If so, it will consider training done, and return with the network configured to the weights that gave best results on the test set. Otherwise, it will repeat. If you will not train the network again (without saving and reloading it), you can call Net::doneTraining() to dispose of temporary storage used during training.

Run the neural network on new cases using Net::run(), which simply takes an input and produces an output.

If you wish to save your network for future sessions, use Net::save() with a binary-mode std::ostream. Load it again using the Net constructor taking a binary-mode istream.

Since you might wish to observe the progress of your network's training, you may #define the symbol NEURAL_NET_DEBUG to view the total error each time it is calculated for the test set. When this starts to go up, training is nearing completion.

## FUTURE DIRECTIONS ON SOURCE CODE IMPROVEMENT

In the future, some of these things could be done to improve the neural network.

- Add support for a callback to be called each time the test error is calculated, rather than just a debug trace, so end-users can view progress.
- Add support for delta functions other than the sigmoid function.
- Add support for more direct training, when more control is needed.

- Add better optimization techniques.

## SOURCE CODE LISTING

1) Net.h - a header file defining the NeuralNetwork namespace.

2) Net.cpp - defines the neural network interface.

3) NetLayer.cpp - defines a neural network layer interface.

4) backprop.cpp - supports training and running of the neural network.

5) stdafx.cpp - automatically generated by Visual Studio .NET.

6) stdafx.h - automatically generated by Visual Studio .NET.

### 1) Net.h - a header file defining the NeuralNetwork namespace.

```
/*********************************************************************
Declares everything in the NeuralNetwork namespace: declares the Net
class representing the entire network, the NetLayer class representing
a single layer of the network, and defines several I/O helpers,
accessors, and fixed parameters.
*********************************************************************/

#ifndef _NET_H_
#define _NET_H_

#include <iostream>
#include <string.h>

namespace NeuralNetwork
{

	// The floating-point type used in computations. Using float
	// makes negligible speed difference.
	typedef double real;

	// Useful for reading raw (stored byte-for-byte) objects off an istream
	template <class T>
		inline void readRaw(std::istream& in, T& obj)
	{
		in.read(reinterpret_cast<char*>(&obj), sizeof(T));
	}

	// Useful for reading raw (stored byte-for-byte) arrays off an istream
	template <class T>
		inline void readRawArray(std::istream& in, T* array, int size)
```

```
{
        in.read(reinterpret_cast<char*>(array), sizeof(T)*size);
}

// Useful for storing objects raw (byte-for-byte) on an ostream
template <class T>
        inline void writeRaw(std::ostream& out, const T& obj)
{
        out.write(reinterpret_cast<const char*>(&obj), sizeof(T));
}

// Useful for storing arrays raw (byte-for-byte) on an ostream
template <class T>
        inline void writeRawArray(std::ostream& out, const T* array, int size)
{
        out.write(reinterpret_cast<const char*>(array), sizeof(T)*size);
}

// A layer of a neural network, used by NeuralNetwork::Net
class NetLayer
{
public:
        // Creates a new net layer with the given number of units,
        // and the given immediately-preceding layer (0 for none).
        NetLayer(int initUnits, NetLayer* prevLayer);

        // Loads this layer from a stream where it was previous saved with save()
        NetLayer(std::istream& in, NetLayer* initPrevLayer);

        // Frees buffers used to hold weights, etc.
        ~NetLayer();

        // Gets number of perceptrons in this layer
        int getUnits() const;

        // Gets the output of the outputNum'th perceptron in this layer
        real getOutput(int outputNum) const;

        // Sets the error on the output of the given perceptron
        void setError(int errorNum, real value);

        // Initialises weights of edges going into this layer to random values
        void randomizeWeights();

        // Initialises weights of edges going into this layer to zero
        void clearWeights();

        // Saves weights for later restoring by restoreWeights. Only one
        // set of weights can be saved, usually the best seen so far.
        void saveWeights();

        // Restores the set of weights most recently saved by saveWeights.
        void restoreWeights();

        // Propagates outputs of the previous layer to the outputs of this layer
        void propagate(real gain);
        // Propagates error from this layer back to the previous layer,
        // in preparation for adjustWeights, which uses the error info.
        void backpropagate(real gain);

        // Computes error of the outputs of this layer from a given set of
        // target values, stores these, and returns the mean square error of
        // them all. Usually used on output layer.
        real computeError(real gain, real target[]);

        // Adjusts weights in order to decrease the error as established
        // by previous computeError/backpropagate calls.
        void adjustWeights(real momentum, real learningRate);

        // Gets the values outputted by the perceptrons in this layer and
        // places them in the array outputsHolder
        void getOutputs(real* outputsHolder);

        // Sets the values outputted by the perceptrons in this layer
        void setOutputs(real* newValues);

        // Saves network so it can be later loaded by the (istream&) constructor
        void save(std::ostream& out);

        // Deallocates storage used only during training
        void doneTraining();

private:
        // Gets the weight on the weightnum'th edge coming into unit unitNum
        real& getWeight(int unitNum, int weightNum);
        // Gets the delta-weight on the weightnum'th edge coming into unit unitNum
        real& getDWeight(int unitNum, int weightNum);

private:
        int    units;          // number of units in this layer
        int    weightsPerUnit; // number of conns going into each unit
        real*  output;         // output of ith unit
        real*  error;          // error term of ith unit
        real*  weight;         // connection weights to ith unit
        real*  weightSave;     // saved weights for stopped training
        real*  dWeight;        // last weight deltas for momentum
        NetLayer* prevLayer;   // Pointer to next layer

        // A buffer used and allocated only once for efficiency
        real* weightIntermediate;
};

// Gets number of perceptrons in this layer
inline int NetLayer::getUnits() const
{
        return units;
}

// Gets the output of the outputNum'th perceptron in this layer
inline real NetLayer::getOutput(int outputNum) const
{
        return output[outputNum];
}

// Sets the error on the output of the given perceptron
inline void NetLayer::setError(int nodeNum, real value)
{
        error[nodeNum] = value;
}

// Sets the values outputted by the perceptrons in this layer
inline void NetLayer::setOutputs(real* newValues)
{
        memcpy(output+1, newValues, sizeof(real)*units);
}

// Gets the values outputted by the perceptrons in this layer and
// places them in the array outputsHolder
inline void NetLayer::getOutputs(real* outputsHolder)
{
        memcpy(outputsHolder, output+1, sizeof(real)*units);
}

// Gets the weight on the weightnum'th edge coming into unit unitNum
inline real& NetLayer::getWeight(int unitNum, int weightNum)
{
        return weight[unitNum*weightsPerUnit + weightNum];
}

// Gets the delta-weight on the weightnum'th edge coming into unit unitNum
inline real& NetLayer::getDWeight(int unitNum, int weightNum)
{
        return dWeight[unitNum*weightsPerUnit + weightNum];
}

// Saves weights for later restoring by restoreWeights. Only one
// set of weights can be saved, usually the best seen so far.
inline void NetLayer::saveWeights()
{
        memcpy(weightSave, weight, (units+1)*weightsPerUnit*sizeof(real));
}

// Restores the set of weights most recently saved by saveWeights.
inline void NetLayer::restoreWeights()
{
        memcpy(weight, weightSave, (units+1)*weightsPerUnit*sizeof(real));
}

// Generates training/test examples for the network. Inherit from this
// and pass instances of that subclass into autotrain and test.
class ExampleFactory
{
public:
        // Fills the given arrays with input values and expected output
        // values based on the next training example. Training values
        // should each be enumerated once on average per numOfExamples
        // calls.
        virtual void getExample(int inputSize, real* input,
                int outputSize, real* output) = 0;

        // Returns number of training examples. If randomly generated,
        // pick something large but reasonable.
        virtual int numExamples() = 0;
};

// A complete multilayer feed-forward neural network
class Net
{
public:
        // Creates a new feed-forward neural network with the given number of
        // layers with the specified number of nodes in each, and the learning
        // rate, momentum factor, and gain of the sigmoid function.
        Net(int layers, int layerSizes[],
                real momentumFactor, real learningRate, real gain);

        // Loads a network from a stream where it was previous saved with save()
        Net(std::istream& in);

        // Frees all memory allocated for network
        ~Net();

        // Initializes all weights in network to random values
        void randomizeWeights();

        // Initializes all weights in network to zero
        void clearWeights();

        // Automatically trains the network until its performance on the
        // test set appears to have achieved a maximum. Returns total
        // error on the test set at completion.
        // - epochsBetweenTests establishes how many tests are done between
        //   test set checks for accuracy.
        // - cutOffError establishes how much worse, as a multiple, error has
        //   to be than the minimum error seen before we stop training. Must be >1.
        real autotrain(ExampleFactory &trainingExamples,
                ExampleFactory &testExamples,
                int epochsBetweenTests = 10,
                float cutOffError = 1.1);

        // Runs the network on an input and feeds it forward to produce an
        // output. For usage after training with autotrain.
        void run(real* input, real* output);

        // Tests the network using the given example set, returning total error
        real test(ExampleFactory &testExamples);

        // Deallocates storage used only during training
        void doneTraining();

        // Saves network to a stream, to be read back in with the istream& constructor
        void save(std::ostream& out);

private:
```

```
            // Saves weights for later restoring by restoreWeights. Only one
            // set of weights can be saved, usually the best seen so far.
            void saveWeights();

            // Restores the set of weights most recently saved by saveWeights.
            void restoreWeights();

            // Propagates inputs of the net all the way through to outputs
            void propagate();

            // Backpropagates output errors all the way back through the network
            void backpropagate();

            // Computes and stores error of output layer and each of its components
            void computeOutputError(real* target);

            // Adjusts all weights in network to decrease error recorded by
            // previous calls to backpropagate or computeOutputError.
            void adjustWeights();

            // Sets the values of the inputs to the network
            void setInputs(real* inputs);

            // Gets the outputs and places them in the array outputs
            void getOutputs(real* outputs);

            // Trains a single training example once
            void simpleTrain(real* input, real* expectedOutput);

            // Trains for a given number of epochs on an entire training set
            void train(int epochs, ExampleFactory &trainingExamples);

      private:
            int         numLayers;     // number of layers
            NetLayer**   layer;        // layers of this net
            NetLayer*    inputLayer;    // input layer
            NetLayer*    outputLayer;   // output layer
            real        momentumFactor; // momentum factor
            real        learningRate;  // learning rate
            real        gain;          // gain of sigmoid function
            real        error;         // total net error

            // These are used as temporary buffers in member funcs,
            // allocated once in the constructor for efficiency.
            real*       input;
            real*       expectedOutput;
            real*       actualOutput;
      };

}

#endif /* #ifndef _NET_H_ */
```

**2) Net.cpp - defines the neural network interface.**

```
/*******************************************************************
```

```
Implementation of the class handling the entire network. The
training algorithm is mainly contained here.
*******************************************************************/

#include "Net.h"
#include "stdafx.h"

#ifdef NEURAL_NET_DEBUG
#include <iostream>
#endif

using namespace NeuralNetwork;

// Creates a new feed-forward neural network with the given number of
// layers with the specified number of nodes in each, and the learning
// rate, momentum factor, and gain of the sigmoid function.
Net::Net(int layers, int layerSizes[],
                    real initLearningRate = 0.25,
                    real initMomentumFactor = 0.9,
                    real initGain = 1.0)
                    :momentumFactor(initMomentumFactor),
                    learningRate(initLearningRate),
                    gain(initGain)
{
        // Allocate and initialise layers
        numLayers = layers;
        layer = new NetLayer*[layers];
        layer[0] = new NetLayer(layerSizes[0], 0);
        for (int i=1; i<layers; i++)
        {
                layer[i] = new NetLayer(layerSizes[i], layer[i-1]);
        }

        inputLayer  = layer[0];
        outputLayer = layer[layers-1];

        // Allocate these here to avoid massive allocation slowdowns later
        int inputSize = inputLayer->getUnits();
        input = new real[inputSize];
        int outputSize = outputLayer->getUnits();
        actualOutput   = new real[outputSize];
        expectedOutput = new real[outputSize];

        // Let weights initially be random
        randomizeWeights();
}


// Loads a network from a stream where it was previous saved with save()
Net::Net(std::istream& in)
{
        // Allocate and initialise layers
        readRaw(in, numLayers);
        readRaw(in, momentumFactor);
        readRaw(in, learningRate);
        readRaw(in, gain);
```

```
	layer = new NetLayer*[numLayers];
	layer[0] = new NetLayer(in, 0);
	for (int i=1; i<numLayers; i++)
	{
		layer[i] = new NetLayer(in, layer[i-1]);
	}

	inputLayer  = layer[0];
	outputLayer = layer[numLayers-1];

	// Allocate these here to avoid massive allocation slowdowns later
	int inputSize = inputLayer->getUnits();
	input = new real[inputSize];
	int outputSize = outputLayer->getUnits();
	actualOutput   = new real[outputSize];
	expectedOutput = new real[outputSize];
}

// Saves network to a stream, to be read back in with the istream& constructor
void Net::save(std::ostream& out)
{
	writeRaw(out, numLayers);
	writeRaw(out, momentumFactor);
	writeRaw(out, learningRate);
	writeRaw(out, gain);

	// Save each layer
	for (int i=0; i<numLayers; i++)
	{
		layer[i]->save(out);
	}
}

// Frees all memory allocated for network
Net::~Net()
{
	for (int i=0; i<numLayers; i++)
	{
		delete layer[i];
	}
	delete[] layer;

	delete[] input;
	delete[] actualOutput;
	delete[] expectedOutput;
}

// Initializes all weights in network to random values
void Net::randomizeWeights()
{
	// Simply call randomizeWeights on each layer except the first
	for (int layerNum=1; layerNum < numLayers; layerNum++) {
		layer[layerNum]->randomizeWeights();
	}
}
```

```
// Initializes all weights in network to zero
void Net::clearWeights()
{
        // Simply call randomizeWeights on each layer except the first
        for (int layerNum=1; layerNum < numLayers; layerNum++) {
                layer[layerNum]->clearWeights();
        }
}


// Saves weights for later restoring by restoreWeights. Only one
// set of weights can be saved, usually the best seen so far.
void Net::saveWeights()
{
        // Simply call saveWeights on each layer except the first
        for (int layerNum=1; layerNum < numLayers; layerNum++) {
                layer[layerNum]->saveWeights();
        }
}

// Restores the set of weights most recently saved by saveWeights.
void Net::restoreWeights()
{
        // Simply call restoreWeights on each layer except the first
        for (int layerNum=1; layerNum < numLayers; layerNum++) {
                layer[layerNum]->restoreWeights();
        }
}

// Propagates inputs of the net all the way through to outputs
void Net::propagate()
{
        // Simply call propagate on each layer except the first
        for (int layerNum=1; layerNum < numLayers; layerNum++) {
                layer[layerNum]->propagate(gain);
        }
}

// Backpropagates output errors all the way back through the network
void Net::backpropagate()
{
        // Simply call backpropagate on the layers in reverse order,
        // except the first, thus driving error back from output to input.
        for (int layerNum=numLayers-1; layerNum > 0; layerNum--) {
                layer[layerNum]->backpropagate(gain);
        }
}

// Computes and stores error of output layer and each of its components
void Net::computeOutputError(real* target)
{
        // Just ask outputLayer to compute its error against target
        error = outputLayer->computeError(gain, target);
}
```

```
// Adjusts all weights in network to decrease error recorded by
// previous calls to backpropagate or computeOutputError.
void Net::adjustWeights()
{
        // Simply call adjustWeights on each layer except the first
        for (int layerNum=1; layerNum < numLayers; layerNum++) {
                layer[layerNum]->adjustWeights(momentumFactor, learningRate);
        }
}

// Sets the values of the inputs to the network
void Net::setInputs(real* inputs)
{
        // Sets output values of input-layer perceptrons, which is input
        // of network
        inputLayer->setOutputs(inputs);
}

// Gets the outputs and places them in the array outputs
void Net::getOutputs(real* outputs)
{
        // Gets output of output-layer perceptrons, which is output of
        // network
        outputLayer->getOutputs(outputs);
}

// Trains a single training example once
void Net::simpleTrain(real* input, real* expectedOutput)
{
        // See first what the network produces now for the input
        // and see how far off it is.
        setInputs(input);
        propagate();
        computeOutputError(expectedOutput);

        // Backpropagate that error data and then adjust the weights based
        // on it to reduce total error as quickly as possible.
        backpropagate();
        adjustWeights();
}

// Trains for a given number of epochs on an entire training set
void Net::train(int epochs, ExampleFactory &trainingExamples)
{
        int inputSize = inputLayer->getUnits();
        int outputSize = outputLayer->getUnits();

        // Train on each training example an average of epochs times
        for (int n=0; n < epochs*trainingExamples.numExamples(); n++) {
                trainingExamples.getExample(inputSize, input, outputSize, expectedOutput);
                simpleTrain(input, expectedOutput);
        }
}

// Tests the network using the given examples, returning total error
real Net::test(ExampleFactory &testExamples)
```

```
{
        int inputSize = inputLayer->getUnits();
        int outputSize = outputLayer->getUnits();

        real totalError = 0;

        // Run network once on each example, adding error each time to a
        // running total.
        for (int n=0; n < testExamples.numExamples(); n++) {
                testExamples.getExample(inputSize, input, outputSize, expectedOutput);
                run(input, actualOutput);
                computeOutputError(expectedOutput);
                totalError += error;
        }

        std::cout << "Error: " << totalError << std::endl;
        return totalError;
}

// Automatically trains the network until its performance on
// the test set appears to achieve a maximum. Returns total
// error on the test set after completion.
// - epochsBetweenTests establishes how many tests are done between
//   test set checks for accuracy.
// - cutOffError establishes how much worse, as a multiple, error has
//   to be than the minimum error seen before we stop training. Must be >1.
real Net::autotrain(ExampleFactory &trainingExamples,
                                        ExampleFactory &testExamples,
                                        int epochsBetweenTests,
                                        float cutOffError)
{
        // Get initial error with current weight set
        real minTestError = test(testExamples);
        real testError = minTestError;

        while (testError <= cutOffError*minTestError) {
                // Train for a while on the training examples
                train(epochsBetweenTests, trainingExamples);

                // How good is network now? Save weights if it's the best
                // we've seen so far on the test set.
                testError = test(testExamples);
                if (testError < minTestError) {
                        saveWeights();
                        minTestError = testError;
                }
        }

        // Restore weights so performance on test set is best we ever saw    restoreWeights();

        return minTestError;
}

// Runs the network on an input and feeds it forward to produce an
// output. For usage after training with autotrain.
void Net::run(real* input, real* output)
```

```
{
            setInputs(input);
            propagate();
            getOutputs(output);
}

// Deallocates storage used only during training
void Net::doneTraining()
{
            // Simply call doneTraining() on each layer
            for (int layerNum=0; layerNum < numLayers; layerNum++) {
                        layer[layerNum]->doneTraining();
            }
}
```

### 3) NetLayer.cpp - defines a neural network layer interface.

```
/*********************************************************************
Implements a single layer of the network, handling propagation through
that layer as well as serialization of that layer.
*********************************************************************/

#include "Net.h"
#include "stdafx.h"
#include <stdlib.h>
#include <math.h>
#include <string.h>

using namespace NeuralNetwork;

static const float BIAS = 1.0f;

// Gets a uniform random value in [Low, High]
static real RandomEqualReal(real Low, real High)
{
            return ((real) rand() / RAND_MAX) * (High-Low) + Low;
}

// Creates a new net layer with the given number of units,
// and the given immediately-preceding layer (0 for none).
NetLayer::NetLayer(const int initUnits,
                                                NetLayer* initPrevLayer)
                                                : units(initUnits), prevLayer(initPrevLayer)
{
            output = new real[units+1];
            error  = new real[units+1];

            if (prevLayer)
            {
                        weightsPerUnit = prevLayer->getUnits()+1;

                        int weightArraySize = (units+1)*weightsPerUnit;
                        weight     = new real[weightArraySize];
                        weightSave = new real[weightArraySize];
                        dWeight    = new real[weightArraySize];
            }
            else
```

```
        {
                weightsPerUnit = 0;
                weight = 0;
                weightSave = 0;
                dWeight = 0;
        }

        weightIntermediate = new real[weightsPerUnit+1];

        output[0] = BIAS;
}

// Loads this layer from a stream where it was previous saved with save()
NetLayer::NetLayer(std::istream& in, NetLayer* initPrevLayer)
: prevLayer(initPrevLayer)
{
        readRaw(in, units);

        output = new real[units+1];
        error  = new real[units+1];

        if (prevLayer)
        {
                weightsPerUnit = prevLayer->getUnits()+1;

                int weightArraySize = (units+1)*weightsPerUnit;
                weight     = new real[weightArraySize];
                weightSave = new real[weightArraySize];
                dWeight    = new real[weightArraySize];
                clearWeights();
                readRawArray(in, weight, weightArraySize);
        }
        else
        {
                weightsPerUnit = 0;
                weight = 0;
                weightSave = 0;
                dWeight = 0;
        }

        weightIntermediate = new real[weightsPerUnit+1];

        output[0] = BIAS;
}

// Saves network so it can be later loaded by the (istream&) constructor
void NetLayer::save(std::ostream& out)
{
        writeRaw(out, units);

        if (prevLayer)
        {
                writeRawArray(out, weight, (units+1)*weightsPerUnit);
        }
}
```

```
// Frees buffers used to hold layer info
NetLayer::~NetLayer()
{
	delete[] output;
	delete[] error;

	if (weight)
	{
		delete[] weight;
		delete[] weightSave;
		delete[] dWeight;
	}

	delete [] weightIntermediate;
}

// Initialises weights of edges going into this layer to random values
void NetLayer::randomizeWeights()
{
	for (int i=0; i < (units+1)*weightsPerUnit; i++)
	{
		weight[i]  = RandomEqualReal(-0.5, 0.5);
		dWeight[i] = 0;
	}
}

// Initialises weights of edges going into this layer to random values
void NetLayer::clearWeights()
{
	for (int i=0; i < (units+1)*weightsPerUnit; i++)
	{
		weight[i]  = 0;
		dWeight[i] = 0;
	}
}

// Propagates outputs of the previous layer to the outputs of this layer
void NetLayer::propagate(real gain)
{
	real* currentWeight = &getWeight (1,0);

	for (int unitNum=1; unitNum <= units; unitNum++)
	{
		real sum = 0;
		for (int outputNum=0; outputNum < weightsPerUnit; outputNum++) {
			sum += *currentWeight * prevLayer->getOutput(outputNum);
			currentWeight++;
		}
		output[unitNum] = 1 / (1 + exp(-gain * sum));
	}
}

// Propagates error from this layer back to the previous layer,
// in preparation for adjustWeights, which uses the error info.
void NetLayer::backpropagate(real gain)
{
        int prevUnits = prevLayer->getUnits();

        for (int prevUnitNum=prevUnits; prevUnitNum != 0; prevUnitNum--) {
                real out = prevLayer->getOutput(prevUnitNum);
                real err = 0;
                for (int unitNum=1; unitNum <= units; unitNum++) {
                        err += getWeight(unitNum, prevUnitNum) * error[unitNum];
                }
                prevLayer->setError(prevUnitNum, gain * out * (1-out) * err);
        }
}

// Computes error of the outputs of this layer from a given set of
// target values. stores these, and returns the mean square error of
// them all. Usually used on output layer.
real NetLayer::computeError(real gain, real target[])
{
        real out, err;
        real totalError = 0;

        real* currentTarget = target + 0;
        real* currentOutput = output + 1;
        real* currentError  = error  + 1;

        for(int unitNum=units; unitNum != 0; unitNum--)
        {
                out = *currentOutput;
                err = *currentTarget - out;
                *currentError = gain * out * (1-out) * err;
                totalError += err*err;
                currentOutput++;
                currentTarget++;
                currentError++;
        }

        return 0.5*totalError;
}

// Adjusts weights in order to decrease the error as established
// by previous computeError/backpropagate calls.
void NetLayer::adjustWeights(real momentum, real learningRate)
{
        int localWeightsPerUnit = weightsPerUnit;

        real* currentWeight  = &getWeight (units,prevLayer->getUnits());
        real* currentDWeight = &getDWeight (units,prevLayer->getUnits());

        for (int prevUnitNum=0; prevUnitNum < weightsPerUnit; prevUnitNum++) {
                weightIntermediate[prevUnitNum+1] = learningRate * prevLayer->getOutput(prevUnitNum);
        }

        for (int unitNum=units; unitNum != 0; unitNum--) {
                for (int prevUnitNum=localWeightsPerUnit; prevUnitNum != 0; prevUnitNum--) {
                        real newDeltaWeight = weightIntermediate[prevUnitNum] * error[unitNum];
                        real oldDeltaWeight = *currentDWeight;
                        *currentDWeight  = newDeltaWeight;
```

```
                                        *currentWeight  += newDeltaWeight + momentum*oldDeltaWeight;
                                        currentDWeight--;
                                        currentWeight--;
                        }
            }
}

// Deallocates storage used only during training
void NetLayer::doneTraining()
{
            delete[] error;
            error = 0;
            delete[] dWeight;
            dWeight = 0;
            delete[] weightSave;
            weightSave = 0;
}
```

**4) backprop.cpp - supports training and running of the neural network.**

```
/*********************************************************************
Forecasting Neural Network Prototype. Trained on 280 days worth of closing price data and then predicts over later
days and finds accuracy of those predictions.

This is the file which contains the main function to be executed. The rest of the files just define classes and rules. and
are used by this file.

Development environment used: Microsoft Visual Studio .NET
*********************************************************************/

#include "stdafx.h"
#include "Net.h"
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <fstream>
#include <iostream>

using namespace NeuralNetwork;

// Number of days we have historical closing data for
const int NUM_DAYS = 280;

// Number of previous days given to the network to help predict
const int WINDOW_SIZE = 30;

// Closing price at the end of each workday for each of 280 days in order
// Starting date: January 2, 1975
// Number of weeks we have data on: 56
// Note: The prices have been preprocessed so that they are handled
// more easily by the neural network. The preprocessing action: divide the
// price by 100; reason: make the error smaller, and the training faster
real    Prices [NUM_DAYS] = {

            0.7023,0.7071,0.7107,0.7102,0.7004,
                    0.7117,0.7261,0.7231,0.7168,0.7214,
                    0.7205,0.7096,0.7108,0.707,0.7174,
                    0.7207,0.7298,0.7537,0.7603,0.7726,
                    0.7621,0.7698,0.7782,0.7761,0.7895,
                    0.7856,0.7863,0.7836,0.7858,0.7992,
                    0.8101,0.815, 0.8093,0.8144,0.8221,
                    0.8262,0.8144,0.7953,0.8037,0.8077,
                    0.8159,0.8303,0.8356,0.839,0.8369,
                    0.843,0.8495,0.8436,0.8359,0.8374,
                    0.8476,0.8601,0.8513,0.8434,0.8361,
                    0.8339,0.8142,0.8206,0.8359,0.8385,
                    0.8336,0.8264,0.8243,0.8151 ,0.8088,
                    0.8035,0.8099,0.8284,0.8377,0.8418,
                    0.856,0.863,0.866,0.8725,0.863,
                    0.8723,0.8709,0.8612,0.8604,0.8662,
                    0.8623,0.8564,0.873,0.881,0.8922,
                    0.9008,0.8864,0.8908,0.8956,0.9053,
                    0.9061,0.9158,0.9227,0.9141,0.9043,
                    0.9053,0.9007 ,0.8906,0.8939,0.9058,
                    0.9034,0.8971,0.8968,0.9115,0.9258,
                    0.9289,0.926,0.9269,0.9248,0.9121,
                    0.9044,0.9055,0.9008,0.9052,0.9146,
                    0.9058,0.9039,0.9202,0.9261,0.9362,
                    0.9419,0.9462,0.9481,0.9481,0.9519,
                    0.9485,0.9418,0.9436,0.9354 ,0.9339,
                    0.948,0.9481,0.9466,0.9519,0.9561,
                    0.9461,0.9363,0.932,0.9244,0.9145,
                    0.9018,0.9007,0.8929,0.8869,0.8819,
                    0.8883,0.8875,0.8799,0.8715,0.8623,
                    0.8625,0.863,0.8602,0.8655,0.8712,
                    0.8597,0.856,0.8636,0.862,0.8495,
                    0.8322,0.8307 ,0.8428,0.8506,0.8396,
                    0.8443,0.864,0.8688,0.8548,0.8603,
                    0.862,0.8562,0.8589,0.846,0.8379,
                    0.8345,0.833,0.8288,0.8209,0.8237,
                    0.8406,0.8588,0.8507,0.8494,0.8574,
                    0.8564,0.8619,0.8503,0.8387,0.8293,
                    0.8382,0.8595,0.8688,0.8677, 0.8794,
                    0.8837,0.8821,0.8946,0.8928,0.8923,
                    0.8937,0.8886,0.8982,0.9056,0.9071,
                    0.9124,0.8983,0.8973,0.9051,0.8939,
                    0.8931,0.8904,0.8809,0.8851,0.8915,
                    0.8955,0.8933,0.8934,0.8987,0.9119,
                    0.9104,0.9097,0.9146,0.91,0.8998,
                    0.8964 ,0.8953,0.897,0.9071,0.9094,
                    0.9124,0.9067,0.8933,0.876,0.8784,
                    0.8682,0.8707,0.873,0.8808,0.878,
                    0.8783,0.8809,0.8893,0.8915,0.8943,
                    0.888,0.8814,0.8873,0.8946,0.9025,
                    0.9013,0.8977,0.9019,0.909,0.9258,
                    0.9353,0.9395,0.9458 ,0.9495,0.9633,
                    0.9557,0.9713,0.9661,0.97,0.9832,
                    0.9886,0.9824,0.9804,0.9921,0.9968,
                    0.9907,0.9853,1.0011,1.0086,1.0087,
                    1.0118,1.0191,1.0039,0.9946,0.9962
};
```

```
// Returns a sequence of examples whose desired outputs run from a
// given lower index to a given upper index in the array above.
// Uses the Factory design pattern studied in CS 4233. OBJECT-ORIENTED ANALYSIS AND DESIGN.
class ArrayRangeExampleFactory : public ExampleFactory {
public:
	ArrayRangeExampleFactory(int initLower, int initUpper)
		: currentExample(initLower), lower(initLower), upper(initUpper)
	{ }
	void getExample(int inputSize, real* input, int outputSize, real* output)
	{
		memcpy(input, &Prices[currentExample-WINDOW_SIZE],
			WINDOW_SIZE*sizeof(real));
		output[0] = Prices[currentExample];
		currentExample++;
		if (currentExample > upper)
			currentExample = lower;
	}
	int numExamples() { return upper-lower+1; }

private:
	int currentExample;
	int lower, upper;
};

// Ranges in the Prices array of training set, test set, and evaluation set
// Here:
// [30-179] TRAINING SET
// [180-259] TEST SET
// [260-279] EVAL SET
const int TRAIN_LWB = WINDOW_SIZE;
const int TRAIN_UPB = 179;
const int TEST_LWB  = 180;
const int TEST_UPB  = 259;
const int EVAL_LWB  = 260;
const int EVAL_UPB  = NUM_DAYS - 1;

// Creates a neural network to predict the closing price for a day given the
// previous WINDOW_SIZE days of closing price data, and demonstrates the results
// on the eval set.
int _tmain(int argc, _TCHAR* argv[])
{
	using std::cout;
	using std::endl;

	srand(35233);

	Net *net;

	// Create a new network with 1 hidden layer with 10 nodes, WINDOW_SIZE inputs, 1 output
	// Visually it would look like this:
	// 30 Input Nodes -> 10 Hidden Nodes -> 1 Output Node
	int layerSizes[] = { WINDOW_SIZE, 10, 1 };
	net = new Net(3, layerSizes, 0.05, 0.5, 1.0);
```

```
        // Show the time at which the training was over
        char dateStr [9];
char timeStr [9];
_strdate( dateStr);
_strtime( timeStr );
        cout << "Training starting: " << dateStr<< ", " << timeStr<< endl;



        // Initialize to random weights, then autotrain with training and test sets
        net->randomizeWeights();
        ArrayRangeExampleFactory training(TRAIN_LWB, TRAIN_UPB);
        ArrayRangeExampleFactory test    (TEST_LWB,  TEST_UPB);
        real error = net->autotrain(training, test, 10, 1.05f);
        // Warning: executing the three lines above may take up to a few days

        // Show final test set error, which should be virtually minimized
        cout << "Final test set error: " << error << endl;

        net->doneTraining();

        // Show the time at which the training was over
        char dateStr2 [9];
char timeStr2 [9];
_strdate( dateStr2);
_strtime( timeStr2 );
        cout << "Training started: " << dateStr<< ", " << timeStr<< endl;
        cout << "Training finished: " << dateStr2<< ", " << timeStr2 << endl;

                // Save the results to a file
        FILE*           f;
        f = fopen("results-sp500.txt", "w");
        //fprintf(f, "\n Traing started %s, %s\n Training ended %s %s\n", dateStr, timeStr, dateStr2, timeStr2);
        fprintf(f, "Predicted, Actual \n");

        // Compare results on the evaluation set
        for (int i=EVAL_LWB; i < EVAL_UPB; i++)
        {
                real output[1];
                net->run(&Prices[i-WINDOW_SIZE], output);
                cout.precision(4);
                // Reverse the preprocessing
                cout << "Predicted: " << output[0]*100 << ", Actual: " << Prices[i]*100 << endl;
                //Write the results to a file
                fprintf(f, "%0.3f, %0.3f \n", output[0]*100,Prices[i]*100);
        }

        // Done writing results to the file
        // In order to map the results to a chart, change the extention of the file
        // from txt to csv, and open the file with Excel or Microsoft Works
        // Select both columns, and use the charting tool to view the chart
        fclose(f);

        // Save the neural network to a binary file
        std::ofstream out("sp500.nnw", std::ios::binary);
        net->save(out);
        // Make the system wait for a key to be pressed
        system("PAUSE");
        return 0;
}
```

**5) stdafx.cpp - automatically generated by Visual Studio .NET.**

```
// stdafx.cpp : source file that includes just the standard includes
// backprop.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

// TODO: reference any additional headers you need in STDAFX.H
// and not in this file
```

**6) stdafx.h - automatically generated by Visual Studio .NET.**

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//
#pragma once

#include <iostream>
#include <tchar.h>
#include "Net.h"
```