A Neural Network Based Distributed System for
Forecasting the Stock Market

by

John Edward Riordan

Submitted to the Department of Electrical Engineering and
Computer Science
in partial fulfillment of the requirements for the degree of
Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1991

© John Edward Riordan, MCMXCI.

The author hereby grants to MIT permission to reproduce and
to distribute copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science
May 20, 1991

Certified by

Andrew Lo
Professor
Thesis Supervisor

Accepted by

Leonard A. Gould
Chairman, Departmental Committee on Undergraduate Theses
A Neural Network Based Distributed System for
Forecasting the Stock Market

by

John Edward Riordan

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 1991, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis presents a neural network and genetic algorithm based system for forecasting the stock market. Neural networks and genetic algorithms as they are applied in the model are reviewed. The benefits and drawbacks of combining the two techniques is discussed. The efficient markets hypothesis and how it effects the model is reviewed. The distributed system model and some initial results are presented and discussed.

Thesis Supervisor: Andrew Lo
Title: Professor
Acknowledgments

First and foremost I would like to thank my father and mother for all their support throughout my undergraduate education. Thanks goes to my friend Jason Slibeck for his consistent support and infamous “Slybreaks”. I would like to thank my thesis advisor, Professor Andrew Lo, for his support and faith. Course Six and Athena for providing the computer equipment, Alan Wu and Shawn McKay in particular. Jake Wetzel for helping get data off the Sloan mainframe. Andy, Ivana, and Scott for the early morning trip to the pancake house. And last but not least, my housemate Matt for hanging out watching latenight TV, keeping the apartment livable, and having a sense of humor.
Contents

1 Introduction ................................................. 9
  1.1 The Linear and Nonlinear Sides of the Efficient Markets Hypothesis . 10
  1.2 In Search of Financial Innovation ......................... 11

2 Neural Network Review .................................... 13
  2.1 Networks and Linear Algebra .............................. 13
     2.1.1 The inner product and linear units .................. 15
     2.1.2 Correlation ........................................... 16
     2.1.3 Projection ............................................. 16
  2.2 Networks and Learning .................................... 17
     2.2.1 Hebb’s rule ........................................... 18
     2.2.2 LMS and the Widrow-Hoff rule ....................... 18
     2.2.3 LMS as gradient descent ............................. 19
     2.2.4 Backpropagation ...................................... 20
  2.3 Backpropagation summary ................................ 22
  2.4 Notes .................................................... 23

3 Genetic Algorithms Review ................................ 26
  3.1 An Example .............................................. 27
  3.2 Notes ................................................... 29
  3.3 Genetic Algorithms and Neural Networks .................. 30

4 System Design & Implementation .......................... 31
  4.1 Optimizing Network Architecture ....................... 31
List of Figures

2-1 Two-layer network and three possible activation functions ............ 14
2-2 One node and vector representation ........................................... 15
2-3 Multi-layer network ................................................................. 21

4-1 Forecasting Model ..................................................................... 42

5-1 Returns for Boeing with Neural Network Performance .................. 52
5-2 Returns for Disney with Neural Network Performance .................. 53
5-3 Returns for IBM with Neural Network Performance ...................... 54
5-4 Returns for Kodak with Neural Network Performance ................. 55
5-5 Returns for Phillip Morris with Neural Network Performance ....... 56
Preface

Somewhere in the middle of my third year as an MIT undergrad I was lounging around shooting the stuff with the guys in my place of residence, the back room of second floor (Second Back) in my fraternity, Sigma Alpha Epsilon. At this point in our journey the guys and I were trying to figure out what it was we wanted to do with our lives. This would inevitably lead to a discussion on the meaning of life which inevitably led nowhere. While the question of what to do with my life still goes unanswered, I do remember Tom Fahy '91, a great scholar and noble friend, suggesting that we should all just sit around and do as little as possible for the rest of our lives. This prompted my wise friend Eric Dunne '90 to point out that while Tom’s notion was in indeed grand, adopting such a life philosophy would not permit him to exist in a fashion suitable to his tastes unless he were to become independently wealthy. He followed this point by suggesting that we should create a machine that would make money for us while we hung out on the beach. Initially counterfeiting jumped to my mind, but as a profession it lacks respectability. A machine which would make money off the stock market was more along the lines of what we had in mind, and thus the idea was born. Being a man of computer science, I would be the one who would throw it together, Eric, the budding economist, would fill me in on all I needed to know about the stock market, and Tom, validating our sanity by offering necessary moral support, announced that he would share the boundless profits with us.

It was a great idea. We would have real time data feeds into a computer system which would make all the decisions and manage the money. The computer system would hook directly into the market and execute the trades. We would allow for thirty minutes worth of system up keep every morning and then hit the beach. It was a great idea. However, unable to conceive of a good way to get the computer to make the decisions, we quickly tabled the idea.

A few months after the birth of the idea, I became interested and fairly engrossed in neural networks. About eight months after the birth, while I was applying neural
networks on a handwritten character recognition project during my forth year at MIT, I remembered that day on which the idea was conceived. It was then, in a flash of insight, that I decided to apply neural networks to forecasting the stock market. It was a great idea. My objective was not to do research or discover new techniques, it was to make that autonomous machine which would put us on the beach. Soon there after I wrote a proposal titled “A Proposal to Study the Application of Neural Networks to Forecasting the Stock Market” and began knocking on doors looking for support. No one would touch it.

Thus I began traveling along a path followed by many other souls - making a system to beat the stock market. During the next six months, with objective in hand, I worked over the design, did more research, and took a Grad class at the MIT Sloan School of Management on securities prices. I had decided that since I couldn’t get any support for the great idea I would make it my undergrad thesis. It was January of my fifth year as an undergrad, two years after the birth of the great idea, when with design in hand I began implementation. I had four months to put it together. If all went according to schedule, I would have a working prototype of THE MACHINE by the thesis deadline. When I was a high school student, sixteen weeks was an eternity. Something had happened to time since then.

While I was not able to reach the original objectives by the deadline, I was able to produce a fair amount of “stuff.” A review of this “stuff” follows. For those interested, I have no intentions of tabling THE MACHINE. It will live on. I have been fortunate enough to find a small corporation which will allow its development to continue. As for Eric, he continued whatever it was he was studying in economics and graduated on time. (He always said it was really cool stuff, but was never really able to explain to us exactly what it was he was studying.) Unable to secure a job on his economic background, Eric eventually took up C programming in a small New York based software company. And Tom, riding a roller coaster life, hopes to pick up a computer science degree this spring. Not being particularly fond of computers, Tom has taken an offer at a large investment bank in Manhattan where he will support capitalism. Life goes on.
Chapter 1

Introduction

Many a man has devised a technique, plan, system, or method to forecast the stock market. Wall Street's technical analysts make a living out of using their strategies to attempt to do just that. Popular systems include the "Filter" system, the Dow theory, the relative strength system, price-volume systems, and reading chart patterns. Other popular systems, which are not taken as seriously, include the "bull markets and bare knees" hemline theory, the odd-lot theory, and the sunspot theory. By and large these systems are considered by the academic community to be "usually amusing, often comforting, but of no real value."[15] However, these systems do "sell." For brokerage firms this translates to more commission business.

Finance theory, a product of academia which has produced portfolio theory and asset-pricing theory, has also provided us with the "efficient markets hypothesis" which, in its weak form, asserts that asset prices follow a random walk, and as such are completely unpredictable from publicly available information. This includes information such as the price and volume history for the asset itself or that of any other asset. This theory arises from the notion that given the human and financial resources devoted to the task of detecting and exploiting profit opportunities, these opportunities are exploited as soon as they arise. While there is little evidence against the weak form of the efficient markets hypothesis, there is evidence that stock prices do not follow random walks.[14] Furthermore, the weak form of the efficient markets hypothesis does not suggest that insider (non-public) information cannot be used to
predict price movements.

While efficient markets hypothesis maintains a significant presence, profit opportunities can still be created through financial innovation. Rationality arguments mitigate the efficient markets hypothesis by holding that humans are implicitly limited in their ability to process information and thus efficiency is also limited.[25] Financial and technical innovation can provide an increase in the ability to process information and therefore can increase efficiency. If a financial innovation is not public knowledge, the holder of the financial innovation has in essence created insider information which can be exploited. Of course once this financial innovation becomes public any profitable opportunities will quickly disappear.

1.1 The Linear and Nonlinear Sides of the Efficient Markets Hypothesis

The weak form of the efficient markets hypothesis assert that \( R = E(r_t | I_{t-1}) \) where \( r_t \) is the return of an asset at time \( t \), \( I_{t-1} \) is all the publicly available information at time \( t - 1 \), and \( R \), a constant, is the risk free rate plus a risk premium. Thus the conditional expectation of \( r_t \) given history is a constant, or more plainly, history is of no use in forecasting \( r_t \).

The standard way of testing the weak form of the efficient markets hypothesis is to embed it in a linear autoregressive model for asset returns of the form:

\[
r_t = w_0 + w_1 r_{t-1} + \cdots + w_p r_{t-p} + \epsilon_t
\]

where \( t = 1, 2, \ldots, p \) is a positive integer determining the order of the autoregression, and \( \epsilon \) is a stochastic error assumed such that \( E(\epsilon_t | I_{t-1}) = 0 \). The weak form of the efficient markets hypothesis infers \( \bar{\omega} = (w_0, w_1, \ldots, w_p) \) is a null vector. Any empirical evidence to suggest otherwise, of which there has been little, goes against the hypothesis. However, evidence that \( \bar{\omega} \) is a null vector does not confirm the hypothesis, but merely confirms that there are no linear regularities involved in determining \( r_t \).
Such a confirmation does not rule out the possible existence of nonlinear regularities.

In recent years nonlinear dynamics and its relation to complex behavior displayed by systems in finance and economics has come under study.[3, 22] During the same period research has demonstrated the value of neural networks in nonlinear forecasting tasks.[12] A striking example is given by Lapedes and Farber[13] who have demonstrated that neural networks are capable of decoding deterministic chaos.

1.2 In Search of Financial Innovation

The idea for this project arose in March of 1990 while studying various aspects of neural network design, implementation, and application. Among the various potential applications of neural networks, forecasting holds great promise.[12] Several researchers have reported their experiences with the use neural networks in forecasting, most notably A. Lapedes and R. Farber[13]. Their results are very encouraging, and have prompted more research in recent years in attempts to extract nonlinear regularities from economic time series - especially in the area of asset price movements.[6, 11, 23, 24, 26, 27]. The use of recurrent networks[21] as the computational basis for prediction of time series appears to be most promising. Other techniques which also appear useful include genetic programming,[5, 7] distributed network architectures,[2, 16] and recurrent backpropagation.[1, 19, 21] These techniques are among the most promising advances in parallel computational technology and deserve further investigation.[29]

After considering the forecasting work done to date, that of Rumelhart[20] in particular, a neural network based forecasting system was designed and implemented. The system uses feedforward and recurrent networks with a modified backpropagation learning algorithm. A genetic algorithm is employed to optimize the network architecture and parameter selection. The system automatically generates hundreds to thousands of neural networks, each possessing a unique genome providing the blueprint for its construction, and distributes them over a network of Unix stations where the genetic algorithm dictates their creation and destruction. Generations of
neural networks are bred and trained in this distributed environment where the rules of natural selection favor those networks which possess the best forecasting ability. The environment, and thus the effective computing power, is limited only by the number of available Unix machines. The Center for Research in Security Prices (CRSP)\textsuperscript{1} database at MIT's Sloan School of Management was utilized as the data source.

This thesis presents the neural network and genetic algorithm techniques used in the forecasting system, the design specifics of the forecasting system, a beneficial discovery dubbed "self modifying genomes" which is yielded by combining the genetic algorithm technique with the modified backpropagation learning algorithm, and some initial results.

\textsuperscript{1}The Center for Research in Security Prices is part of the Graduate School of Business at the University of Chicago. CRSP leases MIT the database.
Chapter 2

Neural Network Review

Chosen as the learning rule for the forecasting system design, this brief review covers one of the most well known neural network learning algorithms – backpropagation. Backpropagation is essentially a gradient descent optimization procedure. The review starts with a little background on the mathematical basis of artificial neural network technology followed by a synopsis of the backpropagation rule.

2.1 Networks and Linear Algebra

The state of a neural network ("network" for the rest of this chapter) at any given time can be defined in terms of the activations of the processing units ("nodes") and the weights of the connections between the processing units. Figure 2-1 presents a fully connected two layer network with three input units and three output units. The equation relating the outputs of a layer to the inputs of a layer in this and almost all networks is:

\[ x_j = f\left(\sum_{i=1} w_{ji}x_i\right) \]

where \( x_j \) is a units activation (output) and \( w_{ji} \) is the weight on the connection between \( x_j \) and a node in the previous layer, \( x_i \). The activation function, \( f \), is the processing that a node does on its input. This function can take on different forms. If \( f \) is the identity function (Figure 2-1 top), then the node is a linear unit and the
Figure 2.1: Two-layer network and three possible activation functions
equation reduces to simply \( x_j = \sum_{i=1} w_{ji} x_i \). Other possibilities for \( f \) include threshold actions like \( f(x) = \begin{cases} 1 & \text{if } x > \theta \\ -1 & \text{otherwise} \end{cases} \) which is the function of the Perception unit, and the logistic function \( f(x) = \frac{1}{1+e^{-x}} \) which is the function of backpropagation units. (Figure 2-1 middle and bottom)

### 2.1.1 The inner product and linear units

Figure 2-2(a) shows a single linear unit whose output is, \( x_j = \sum_i w_{ji} x_i \). The relationship between the inputs, weights, and the output of this unit can be thought of in terms of the relationship between vectors. Figure 2-2(b) shows this relationship graphically in weight space.

\[
\vec{x}_i = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \vec{w}_j = \begin{bmatrix} w_{j1} \\ w_{j2} \end{bmatrix}
\]

So that the inner product of \( \vec{x}_i \) and \( \vec{w}_j \) is,

\[
\vec{x} \cdot \vec{w}_j = (x_1 \times w_{j1} + x_2 \times w_{j2}) = x_j
\]
which is \( x_j = \sum_i w_{ji} x_i \).

The linear unit is computing the inner product of the input vector with its weight vector, \( x_j = \vec{x} \cdot \vec{w}_{ji} \).

### 2.1.2 Correlation

From linear algebra, the angle between the two vectors \( \vec{x}_i \) and \( \vec{w}_j \) is defined in terms of the inner product by the following definition:

\[
\cos \theta = \frac{\vec{x}_i \cdot \vec{w}_j}{\| \vec{x}_i \| \| \vec{w}_j \|}
\]

where \( \| \vec{x}_i \| \) is the length of vector \( \vec{x} \).

\[
\| \vec{x}_i \| = \sqrt{\vec{x}_i \cdot \vec{x}_i}
\]

These results could also be found using basic trigonometry, but the inner product method is in general superior. Expanding the above equations with values from the example, it is seen why the inner product is often said to be a measure of similarity. Writing out the equation in terms of our vectors gives

\[
\rho = \cos \theta = \frac{\sum_{i=1}^{2} x_i w_{ji}}{\left(\sum_{i=1}^{2} x_i^2\right)^{1/2} \left(\sum_{i=1}^{2} w_{ji}^2\right)^{1/2}}
\]

This is the formula for the correlation, \( \rho \), between two sets of numbers with zero means.

It is apparent that the node is making a comparison between the input vector and the weight vector.

### 2.1.3 Projection

It is also useful to view the inner product as the projection of the weight vector onto the input vector (Figure 2-2(b)) because the projection is the inner product of the
input vector and the weight vector divided by the length of the input vector.

\[ \text{projection} = \frac{\vec{e}_i \cdot \vec{w}_{ji}}{\| \vec{e}_i \|} \]

This can also be seen from trigonometry where, \( \text{projection} = \| \vec{w}_{ji} \| \cos \theta \), if the equation for \( \cos \theta \) from above is substituted in.

This relationship between the input vector and the weight vector forms the basis for the functioning of all artificial neural networks, and it is this relationship that allows these networks to learn and remember. Note that a single node can calculate the inner product of any n-dimensional input with its weight vector.

### 2.2 Networks and Learning

If the output of a network is a computation based on a relationship between the input vector and the weights the network, it is obvious that if one could find the set weights which gave you the output vectors you desired for a set of input vectors, one would have something which is computationally worthwhile. The excitement around artificial neural networks is mainly focused around a method for doing just that called backpropagation. Backpropagation is an algorithm to change the weights of a network so that it “learns” a set of input/target vectors. Starting with initially random weights, an input is given to the network and an output generated by the network. This output is compared with the desired output, the target output, and the sum-squared error is computed. Backpropagation is an algorithm which minimizes the sum-squared error of the outputs with respect to the target outputs over the entire set of input/target pairs. This algorithm implements gradient descent in sum-squared error. Before an analysis of gradient descent, a review of the early learning rules for networks is helpful.
2.2.1 Hebb’s rule

Hebb is credited with the original insight into the mechanism of learning which is used as the basis of neural network research today. What Hebb essentially said was, “If a neuron is active (firing) and a connection to that neuron is active, then that connection is reinforced.” Upon this the field has been built. Hebb’s rule is:

\[ \Delta \vec{w} = \rho y^* \vec{z} \]

where \( \rho \) is the learning rate (usually \( 0 < \rho \leq 1 \)), \( y^* \) is the target value, and \( \vec{z} \) is the input vector.

This rule is the precursor to the backpropagation algorithm and its similarity will be evident. The procedure of training - giving the network an input, calculating \( \Delta \vec{w} \), and modifying the weights - is called a “trial”. This process changes the weight vector such that \( \vec{z} \cdot \vec{w} \) moves closer to the target, or desired, output. A training period may involve thousands of trials and varies greatly depending on many factors, including net size and complexity, complexity of data, and the learning rate. This rule has two serious problems. First, it is apparent that the number of trails to performed during a training session must be known, otherwise, as the number of trials increases so will the magnitude of the weight vector. Second, the above rule is also limited to one-layer linear networks which does not make it very useful.

2.2.2 LMS and the Widrow-Hoff rule

The Widrow-Hoff learning rule, also referred to as LMS for least-mean-squared, is a modification of the Hebb rule and avoids some of the above problems. It solves the first problem, but it still limits us to linear networks. Backpropagation extends LMS to networks with logistic units, so it is natural to start with an examination of LMS for linear units. The LMS rule is:

\[ \Delta w_{ji} = \rho(y^* - y_j)y_i \]
for an individual weight, or

\[ \Delta \bar{w} = \rho(y^* - y_j)\bar{y}_i \]

for all the weights. Where \( y_j \) is the output of the node \( j \) during the current trial and \( y_i \) is the output of the node in the previous layer (the inputs of a one layer system), and \( w_{ji} \) is the weight on the connection from node \( i \) to node \( j \). From examination of this equation, it is seen that the weight vector changes at a rate proportional to the difference between the target value and the current output value for a given trial.

2.2.3 LMS as gradient descent

The modification of the weights depends on some measure of the error between the target outputs and the actual outputs of the network. One such measure of the error is Square Error:

\[ E = \frac{1}{2} \sum_j (y_j^* - y_j)^2 \]

which is the square of the difference between the target and output, where

\[ y_j = \sum_i w_{ji}y_i \]

is the output of node \( j \) a linear unit, and \( y_i \) is the output a node in the previous layer. Note that Figure 2-1 follows the notation scheme which will be used for now.

The derivative of \( E \) with respect to \( w_{ji} \), which, employing the chain rule, is:

\[ \frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w_{ji}} = -(y^* - y_j) \frac{\partial y_j}{\partial w_{ji}} = -(y^* - y_j)y_i = \delta_j y_i \]

This is derived from the above two equations. Notice the similarity to the LMS rule for weight change, \( \Delta w_{ji} = \rho(y^* - y_j)y_i = -\rho \delta_j y_i \), where \( \rho > 0 \).

\[ \Delta w_{ji} = -\rho \frac{\partial E}{\partial w_{ji}} \]

This corresponds to performing steepest descent on a surface in weight space whose height at any point in weight space is equal to the error measure.
For logistic units the calculations are very similar, although the output of node \( j \) has changed. A logistic unit has output \( y_j = f(\sum_i w_{ji} y_i) \) where \( f(x) = \frac{1}{1+e^{-x}} \). If \( \text{in}_j = \sum_i w_{ji} y_i \) then:

\[
y_j = \frac{1}{1+e^{\text{in}_j}}
\]

and

\[
\Delta w_{ji} \propto -\frac{\partial E}{\partial w_{ji}}
\]

where \( E \) is the sum-squared error function defined from above. Using the chain rule again:

\[
\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial \text{in}_j} \frac{\partial \text{in}_j}{\partial w_{ji}}
\]

The value of \( \frac{\partial \text{in}_j}{\partial w_{ji}} = y_i \) the output of the node in a previous layer. Again applying the chain rule:

\[
\frac{\partial E}{\partial \text{in}_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial \text{in}_j}
\]

The \( \frac{\partial E}{\partial w_{ij}} \) is \( -(y^* - y_j) \) which is again the same computation from above, and the new partial follows:

\[
\frac{\partial y_j}{\partial \text{in}_j} = f'_i(\text{in}_j) = y_j(1 - y_j)
\]

Notice that this is only valid for the output units in the last layer in a network. Bringing all this together:

\[
\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial \text{in}_j} \frac{\partial \text{in}_j}{\partial w_{ji}} = -(y^* - y_j)y_j (1 - y_j)y_i = \delta_j y_i
\]

and therefore \( \Delta w_{ji} = -\rho \delta_j y_i \).

2.2.4 Backpropagation

The backpropagation rule is credited to Werbos, Le Cun, and Rumelhart & McClelland. There are two cases for weight changes. The first occurs when the weights are on the connections to the output units - this is the case that was just calculated above, the second case occurs for the weights associated with all the other layers. A minor shift in notation is now adopted which matches Figure 2-3. Up until this point,
Figure 2-3: Multi-layer network
was used to refer to the output layer. Now, \( j \) will refer to the layer which comes before \( k \) and after \( i \), as in Figure 2-3. Note also that layer \( k \) will refer to the output layer for the following illustration.

The delta rule for the first case was already computed above with \( j \) as the last layer.

Other layers: It is the computation of the \( \delta_j \)'s in the hidden layers which is novel. The interesting result is that there is a simple recursive computation of these \( \delta \)'s which can be implemented by propagating error signals (\( \delta \)'s) backward through the network.

\[
\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial in_j} \frac{\partial in_j}{\partial w_{ji}}
\]

\[
\frac{\partial in_j}{\partial w_{ji}} = y_i
\]

\[
\delta_j = \frac{\partial E}{\partial in_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial in_j}
\]

\[
\frac{\partial y_j}{\partial in_j} = f_j'(in_j)
\]

The novelty:

\[
\sum_k \frac{\partial E}{\partial in_k} \frac{\partial in_k}{\partial y_j} = \sum_k \frac{\partial E}{\partial in_k} w_{kj} = \sum_k \delta_k w_{kj}
\]

So,

\[
\delta_j = f_j'(in_j) \sum_k \delta_k w_{kj}
\]

whenever unit \( j \) is not an output unit. This procedure constitutes the backpropagation learning algorithm for a feedforward network of logistic units, or for that matter for a feedforward network of any semilinear units.

### 2.3 Backpropagation summary

Now the equations above are used to give the procedure of backpropagation in a step by step form. It is assumed that the transfer function is 

\[
f(x) = \frac{1}{1 + e^{-x}}.
\]

Choose a learning rate constant, \( \rho \).
For each sample input:

(a) Compute the δ's for the output layer:

$$\delta_k = -(y_k^* - y_k)f'(in_k)$$

(b) Compute the δ's for the other nodes using:

$$\delta_j = f'_j(in_j) \sum_k \delta_k w_{kj}$$

(c) Compute the weight changes for all the weights using:

$$\Delta w_{ji} = -\rho \delta_j y_i$$

Until satisfactory performance is attained.

Note that since the weight changes are proportional to output errors, the weight changes will become smaller and smaller as the desired outputs are approached. This asymptotic approach to the desired output causes 'learning' to slow as error decreases. For this reason, performance is considered satisfactory when all outputs are within 0.1 percent or so of their desired output. This is one of the reasons that training networks using the backpropagation algorithm can be slow and require many repetitions.

2.4 Notes

The power of neural networks lies in their ability to learn non-linear mappings of vector spaces, to make those mappings using a parallel distributed computational system, to generalize these mappings, and their robustness. The combination of these features makes them candidates to solve "hard" problems such as forecasting nonlinear systems.

The backpropagation algorithm needs a differentiable function for weight change computations, this rules out the threshold function. Not only does the function need
to be differentiable, but for networks comprised of more than one layer the derivative of the function must be nonlinear if backpropagation is to work effectively. If the derivative of the function is linear, there is no way to measure the benefit, δ, of changing a particular weight. This rules out linear activation functions.

It should also be noted that the logistic function was originally chosen because it was a good differentiable approximation of the step function. Although computational complexity and power can be gained using the logistic function, the step function could in general be used to compute the same mappings as the logistic function. However, the backpropagation algorithm can not be used to train a network comprised of nodes using the step function as a activation function. Note also that a network using only linear activation functions can always be reduced to a single layer network (see linear algebra section above) and therefore can only do linear mappings from input to output.

On the ability to generalize, consider the memory of a computer. If you have the correct address of a memory location you will get the correct output, but if your address is one bit off you get complete garbage. In artificial neural network systems if you have the correct address (after learning) you get the correct output, but you also get an output which is correct, or very close to the desired output, if you have an address(input) which is close to the original. While this feature is not of practical use in doing error sensitive mathematical computations, it is very useful in dealing with real world problems. For example, consider the recognition of handwritten digits. When a network is trained to recognize a set of handwritten digits (perhaps using the bit stream of an image scanner as input), it forms representations of each digit internally. These representations are prototypes of the handwritten digits created from examples from the training set. When a handwritten digit which the network has never seen is presented, the network “compares” it to its prototypes and “selects” the closest match.

For the large part, artificial neural networks are currently simulated on serial computers. In doing so, two of the largest benefits of the computational systems are lost; parallelism and robustness. When networks are simulated on serial computers,
the intrinsically parallel computations are done in serial and therefore great speed benefits are lost. Due to their parallel computational structure artificial neural networks are prime candidates to solve continuous time, real world problems in areas from computer vision to robotics. The computational structure also appears to lend itself nicely to any area where the patterns in a particular environment do not present themselves in a straight forward manner, and are therefore very difficult to find using the rule based algorithms of traditional AI systems. Artificial neural networks can be used to “find” these types of patterns even amidst noise, a task serial computers find inherently difficult. An interesting example is the Navy’s use of neural networks interfaced with sonar systems which has greatly increased the functionality of these systems. The Navy has been able to recognize assorted objects in the ocean and on the ocean floor using the sonar and neural networks combination.

Artificial neural networks which are implemented in parallel also have the characteristic of being extremely robust, especially in comparison with traditional computers. If a connection in a network is destroyed (for that matter if many connections are destroyed) the network will most likely function without any apparent, or at least only marginal, perturbation. If a processor is destroyed the same holds true. This of course is far from the case in traditional Von Neuman digital computers.

The explanation of backpropagation in the context above only scratches the surface of the current state of neural network technology. It is also apparent that the technology in is in its infant stage and its potential has just begun to be utilized.
Chapter 3

Genetic Algorithms Review

Genetic algorithms are surprisingly simple search algorithms that transform populations of individual mathematical objects, usually fixed-length binary character strings representing possible problem solutions, into new populations based on the mechanics of natural selection and genetics by using such operations as sexual recombination and fitness proportionate reproduction. Genetic algorithms begin with an initial population of individuals, usually randomly generated, and iteratively (1) evaluates the fitness of each individual in the population and (2) uses operations of natural selection to produce a new population. By using the operations of natural selection the fitness of the members of the population increases thus producing better solutions in the problem environment.

Professor John Holland of the University of Michigan presented the pioneering formulation of genetic algorithms for fixed-length character strings in Adaptation in Natural and Artificial Systems.[9] Holland established, among other things, that genetic algorithms are a mathematically near optimal approach to adaptation when the adaptive process is thought of as a set of simultaneous multi-armed slot machine problems requiring an optimal allocation of future trials given the currently available information. Goldberg[7] provides a survey of recent work in the field.

Given an problem which one wishes to solve by optimization, a simple genetic algorithm approach would proceed as follows:

1. Define an encoding of the parameter set in a character string. In the simple
case, this is a binary string.

2. Define a cost function which evaluates the value of an output and thus the parameter set which produced it. This cost function provides the means to determine the fitness of the individuals in the population.

3. Randomly generate a population of character strings which represent the parameters of the system to be optimized.

4. Then, until a satisfactory fitness level is reached by one or more of the individuals in the population, iteratively

   (a) Determine the fitness of each individual in the population

   (b) Apply the following three genetic operators to produce a new population

      i. Reproduction

      ii. Crossover

      iii. Mutation

3.1 An Example

As an example, consider the following. Given a black box with five binary switches and an output which produces a real number. The task is to determine what switch setting produces the highest output. The parameter set can be encoded in a five bit binary string with each string being represent by one bit. A bit set to one represents the switch being on and a bit set to zero represents the switch being off. The cost function is simply the output of the black box. For the example assume the switch settings which produces the highest output is 01010. For a randomly generated population consider the following four member population:
<table>
<thead>
<tr>
<th></th>
<th>String</th>
<th>Fitness</th>
<th>Switch Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11001</td>
<td>9.1</td>
<td>on, on, off, off, on</td>
</tr>
<tr>
<td>2</td>
<td>10010</td>
<td>11.5</td>
<td>on, off, off, on, off</td>
</tr>
<tr>
<td>3</td>
<td>00111</td>
<td>3.5</td>
<td>off, off, on, on, on</td>
</tr>
<tr>
<td>4</td>
<td>01000</td>
<td>15.1</td>
<td>off, on, off, off, off</td>
</tr>
</tbody>
</table>

The fitness value is the output of the black box for a given switch setting. Application of the genetic operators (except mutation) might start by determining a likelihood of reproduction for each string by determining what percentage of the total fitness each individual has and using that value the probability of reproduction. Thus for the example we might find that strings 2 and 4 reproduce while 1 and 3 do not, and are replaced by the offspring of 2 and 4. Reproduction involves making copies strings 2 and 4 followed by crossover. Crossover splits the copied strings in two at at the same place. For the example, between bits two and three. Swapping of the bits then occurs. In the example, bits one and two of string 2 are swapped with bits one and two of string 4 producing the following new population:

<table>
<thead>
<tr>
<th></th>
<th>String</th>
<th>Fitness</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10010</td>
<td>11.5</td>
<td>parent string 2</td>
</tr>
<tr>
<td>2</td>
<td>01000</td>
<td>15.1</td>
<td>parent string 4</td>
</tr>
<tr>
<td>3</td>
<td>01010</td>
<td>20.0</td>
<td>child string 1</td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
<td>2.1</td>
<td>child string 2</td>
</tr>
</tbody>
</table>

It is seen that child string 1 is the desired result. While it is granted that this was a rigged example, it does convey the intentions of the algorithm. Mutation is used to prevent the algorithm from getting stuck. By randomly switching the value of a bit in a string, mutation creates a new string. On occasion, the genetic algorithm may lock itself out for searching a particular area. Mutation provides a counter to the problem. In practice mutation is used sparingly.
3.2 Notes

Genetic algorithms are a robust adaptive optimization. Robustness is central theme of genetic algorithm research. While traditional optimization methods such as hill climbing and calculus based methods demand a smooth and continuous surface on which to operate, genetic algorithms do not. However, an encoding of the parameter set which is amiable to the genetic algorithm must be possible. This encoding usually takes the form of a finite length binary string. For many problems arriving a good encoding is a very difficult problem (consider an encoding for parameters which are real numbers). The longer the string the larger the population of strings needed and the greater the number of generations which must be traversed to arrive at a solution.

The simplicity which genetic algorithms provide is a great attraction. They are easy to implement and conceptually simple. Most of the analytical literature views genetic algorithms as doing an optimization on the hyperplanes of the parameter space, where the planes are represented by schemata or similarity templates. For example, consider a genome with three bits creating a three dimensional problem where the distribution of possible points lie on a cube. The front face of the cube could be defined as all points that begin with zero. If $*$ is used to represent the "don't care", or wild card, condition, then this plane can also be represented by the schemata $0*$. All bit strings that match a given schemata lie in its hyperplane and every binary string, without wild cards, corresponds to a corner in the hypercube and is a member of $2^n - 1$ hyperplanes where $n$ is the length of the string.

Each schema represents a different genetic fragment. When reproduction occurs the strings exchange hyperplane information and samples a new corner in the hypercube. For obvious reasons, there are techniques to reduce duplicate genomes. As reproduction continues the cost function biases the population toward desirable hyperplanes.
3.3 Genetic Algorithms and Neural Networks

All applications of genetic algorithms to neural networks which were uncovered during research discussed their application to discovering weights or defining the connectivity of a neural network.[8, 18, 28] It not clear what advantage, if any, this approach has over the more mainstream neural network training approaches. In fact, these approaches seem to be at a disadvantage. Weights are normally real values which present an encoding problem for genetic algorithms, the number of connections in a fair sized network is far more than can be reasonably handled by a genetic algorithm, and there is no clear way to relate the operation of the genetic algorithm to the cost function of a traditional neural network learning algorithm. Network connectivity appears to be better determined by learning algorithm related methods – a sampling of which are considered in the next chapter. Genetic algorithms are used by the forecasting system discussed in the next chapter to search the input space of the system which indirectly effects the neural networks by determining the number of nodes in the input layer of the network. They are not used to directly determine network connectivity or weight formation.
Chapter 4

System Design & Implementation

The neural network is the basis of the forecasting system. In choosing the neural network design for the forecasting system, much consideration was given to optimizing network architecture, increasing the rate of convergence, and determining an optimal set of data for input to a given neural network. The first two issues have been studied and ideas from the literature were incorporated into the system design. Rumelhart’s algorithm and Jabob’s heuristics are adopted as part of the neural network system design to deal with these issues. The last issue is particular to problems with very large potential inputs sets which lack easily determined optimal subsets. That is, as in the stock market prediction case, the problem of what input data to choose for a given neural network given a large database of information and no particular idea of what data is significant to the task. Lacking a method to overcome this issue, a unique technique was developed.

4.1 Optimizing Network Architecture

An optimal size neural network is one which handles the task at hand effectively using the smallest number of nodes and connections. From a computational and implementational standpoint it is clear why minimizing the number of nodes and connections is desirable. However, while minimizing the size of a network, attention must be paid to the following relationship between network size and performance:
If a network is too large an overfitting problem may arise and generalization will be poor, if too small the network will not be powerful enough for the task at hand.[4]

As with other methods of function approximation, such as polynomial, too many free parameters will allow the network to fit the training data arbitrarily close, but will not necessarily lead to optimal prediction.[20]

4.1.1 Validation

The gradient descent learning process used by the backpropagation procedure will attempt to fit the major features of the data first. Later those lesser features of the data will begin to be fit. Initially all the hidden units may be viewed as attempting to fit the major features and later differentiating as the lesser features are fit. Still later, as training continues, the network will attempt to fit the noise and features of the data which are not of predictive value. A solution to this problem of overfitting is to halt training just before the network begins to fit the noise and insignificant features in the data.

A neural network is trained to forecast the price of a stock based on historical information. Typically, given historical data for a given period of time, two blocks of data are set aside for training and performance testing. The network in trained on the first set of data – the training set. After training is complete the network performance is tested on the second set of out of sample data set – the prediction set. The two data sets must be exclusive to yield valid performance statistics.

To overcome the overfitting problem the data may be broken into three sets yielding a third set – the validation set. Again all three sets must be exclusive. The validation set can be used to determine when to halt the training of network.[17] This technique entails testing the performance of the network on the validation set periodically during training, but not training the network on the validation set. While the performance on the network on the validation set improves training is continued. Once performance on the validation set begins to diminish training is halted.
While this technique does solve the problem the overfitting problem it does not produce the smallest sized neural network necessary to effectively handle the task at hand. In fact, this technique inherently demands an oversized network.

4.1.2 Optimal Brain Damage

The overfitting problem can also be handled in another manner. Le Cun proposes a method for eliminating unnecessary connections in a network, thus increasing the speed of the network and its ability to generalize. These are both important issues. Another interesting issue arises in network design. He states, “most successful application of neural network learning to real-world problems have been achieved using highly structured networks of rather large size.” He notes that by eliminating particular unnecessary weights that important network design characteristics may become apparent. The technique he introduces is called Optimal Brain Damage (OBD).

He proposes that the parameters (weights) which should be removed are those which have small “saliency”. Here saliency is a measure of the effect which removing such a parameter would have on the error function. Using a Taylor series approximation of the error function, a perturbation, $\delta u$, of the parameter vector will change the objective function by

$$\delta e = \sum_i g_i \delta u_i + \frac{1}{2} \sum_i h_{ii} \delta u_i^2 + \frac{1}{2} \sum_{i \neq j} h_{ij} \delta u_i \delta u_j + o(||\delta u||^3)$$

where the $\delta u_i$'s are the components of $\delta u$, and the $g_i$'s are the components of the gradient $g$ of $e$ with respect to $u$. The $h_{ij}$'s are the elements of the Hessian matrix $h$ of $e$ with respect to $u$, and $g_i = \frac{\partial e}{\partial u_i}$ and $h_{ij} = \frac{\partial^2 e}{\partial u_i \partial u_j}$. Noting that computing this term is impractical due to the size of the Hessian matrix involved, Le Cun makes a few assumptions to depart with all but one of the terms. He assumes the parameter deletion will be done after training when the parameter vector is at a local minimum thus making the first and last terms of the right hand side unnecessary. He also assumes that $h_{ij} = h_{ji}$ in the Hessian matrix $h$. It is not clear that this latter assumption is always valid. It appears that the order in which the partials were taken.
would not be communitie - which Le Cun assumes, but for the sake of making the proposed algorithm tractable it seems like at reasonable (if not necessary) assumption to make. Thus the third term of the right hand side is tossed leaving:

\[ \delta e = \frac{1}{2} \sum_i h_{ij} \delta u_i^2 \]

Le Cun then outlines the calculations of the diagonal second derivatives \( h_{ii} \) and shows that this computation is of the same order of complexity as computing the gradient. The computation of the saliencies for each parameter is then \( s_k = \frac{h_{kk} u_k^2}{2} \). The parameters can then be sorted by saliency and some of the low-saliency parameters can then be fixed at zero, or some variation of this can be done; such as decreasing the magnitude of the parameter, or setting the parameter to zero and then letting it vary again during the next epoch.

Optimizing the size and architecture of a network for a particular data set is the most difficult task in producing functional neural networks, thus solutions to this problem are of paramount importance. Thus Optimal Brain Damage is a step in the right direction, but some of its flaws need to be overcome. The most prominent, which is noted by Le Cun, is that training time increases. Network pruning is done after each training cycle and then the network is trained again. It would be much more efficient to prune during training (of course this is impractical using the presented algorithm). Also, it is not clear how successful this algorithm is on unestablished networks.

### 4.1.3 Rumelhart's Weight-elimination

While Le Cun's algorithm was given consideration, it was decided that Rumelhart's weight elimination procedure would be used. This method assumes that the network which generalizes best is the smallest network still able to fit the training data. The method of weight-elimination involves the extension of the backpropagation gradient method to a more complex cost function. The new cost function has added a term to the standard sum squared error term of the tradition gradient algorithm which acts to
encourage the reduction and eventual elimination of as many weights as possible, given performance on the training set can be obtained with fewer weights, by effectively driving weights to zero. Rumelhart's cost function is:

$$\sum_{k \in s} (target_h - prediction_h)^2 + \lambda \sum_{i,j} \frac{\omega_{ij}^2/\omega_0^2}{1 + \omega_{ij}^2/\omega_0^2}$$

This method suggests starting with a oversized network and allowing the cost function to eliminate connections. As Rumelhart points out, it is useful to begin with \( \lambda \) at zero and to slowly increase the value of \( \lambda \) until performance begins to decline and the thereafter increase or decrease the value of \( \lambda \) so the error on the training set continues to decrease at a steady and slow rate. The scale is given by \( \omega_0 \). For activations between 0 and 1, \( \omega_0 \) is set to 1. If the weight \( \omega_{ij} \) is large compared to \( \omega_0 \), the cost is \( \lambda \). For weights close to zero, the associated cost is also close to zero.

4.2 Increasing Convergence Time

Jacobs[10] outlines four heuristics for increasing the rate of convergence of a neural network relative to the rate of convergence achieved using steepest descent alone, and backs up these heuristics with empirical results. The four heuristics he outlines are:

1. Every parameter of the performance measure to be minimized should have its own individual learning rate – the step size appropriate for any one parameter dimension is not necessarily appropriate for other parameter dimensions.

2. Every learning rate should be allowed to vary over time – it is common for error surfaces to possess different properties along different regions of a single parameter dimension.

3. When the derivative of a parameter possesses the same sign for several consecutive time steps, the learning rate for that parameter should be increased – in this situation it is frequently the case that the slope of the error function will
not change sign (slope has low curvature) for some significant distance in that
direction.

4. When the sign of the derivative of a parameter alternates for several consecutive
time steps, the learning rate for that parameter should be decreased – in this
situation it is frequently the case that the error function has high curvature in
that parameters direction (like a ravine).

Jacobs presents the “Delta-Bar-Delta” rule which consists of a weight update rule
and a learning rate update rule. For the Delta-Bar-Delta rule the weight update
rule is \( w(t+1) = w(t) - \epsilon(t+1) \frac{\delta J(t)}{\delta w(t)} \), where \( w(t) \) is the value of a single weight at
time \( t \) and \( \epsilon(t) \) is the learning rate value corresponding to \( w(t) \) at time \( t \). This is
backpropagation with local learning rates. The learning rate update rule proposed is:

\[
\Delta \epsilon(t) = \begin{cases} 
  k & \text{if } \delta(t-1)\delta(t) > 0 \\
  -\phi \epsilon(t) & \text{if } \delta(t-1)\delta(t) < 0 \\
  0 & \text{otherwise}
\end{cases}
\]

where \( \delta(t) = \frac{\delta J(t)}{\delta w(t)} \) and \( \delta(t) = (1-\theta)\delta(t)+\theta\delta(t-1) \) with \( \delta(t) \) is the partial derivative of
the error with respect to \( w \) at time \( t \) and \( \delta(t) \) is an exponential average of the current
and past derivatives with \( \theta \) as the base and time as the exponent, and \( k \) is a constant.
This rule increments the learning rates linearly and decrements them exponentially,
which hopefully prevents the learning rate from becoming too large, and allows for a
rapid decrease of the learning rates while preventing them from going negative.

Jabobs empirical performance evaluations of the Delta-Bar-Delta(DBD) rule com-
pared to gradient descent alone, gradient descent with momentum (traditional mo-
mementum), gradient descent with DBD, and a hybrid; with both momentum and DBD
indicate that in all cases the methods using the heuristics converge faster, by a large
factor, than gradient descent alone. The DBD and the hybrid methods performed
best. This improvement would be expected.

The benefits of this rule and the adopted heuristics seems clear, and there doesn’t
appear to be a downside – the added computation is minimal. The idea of localized
learning rates is appealing to the biologically inclined, while the computation of the learning rate change is only as satisfying as backpropagation.

4.3 Determining Optimal Input Set

There are three ingredients required to specify a model for time series prediction:

1. Choose an embedding for the time series \( \{x_t\} \). The structure can be captured by expressing the current value of \( x_t \) as a function of a set of parameters which can be determined at sometime prior to \( t \). Thus, \( x_t = \mathcal{F}(\text{past}) = \mathcal{F}(\tilde{n}_{t-1}, \tilde{n}_{t-2}, \tilde{n}_{t-3}, ..., \tilde{n}_{t-d}) \). The matrix \((\tilde{n}_{t-1}, \tilde{n}_{t-2}, \tilde{n}_{t-3}, ..., \tilde{n}_{t-d})\) lies in the \( d \)-dimensional time delay space or lag space with the total dimensional equal to the sum of all the parameters in each vector in the matrix.

2. Assume approximating a continuous surface to the points \( \{x_t, (\tilde{n}_{t-1}, \tilde{n}_{t-2}, \tilde{n}_{t-3}, ..., \tilde{n}_{t-d})\} \) will produce a good model of the time series. Approaches in time series prediction differ mainly in the choice of the primitives (splines, radial basis functions, polynomials, sigmoids, ...) and in the choice between a global fit in lag space versus many local fits.

3. Choose a cost function which provides a measure of how well the approximated surface fits the points. Assuming Gaussian errors the cost function is the sum of squared differences.

A major problem in attempting to use neural networks to forecast the price of a given stock at time \( t \), \( x_t \), is to determine what to give the neural network as input. That is, determining what \( \tilde{n}_{t-1}, \tilde{n}_{t-2}, \tilde{n}_{t-3}, ..., \tilde{n}_{t-d} \) will produce good \( x_t \)'s. One could attempt to forecast the price based on past prices of a given stock alone, but that does not seem like it would be the most fruitful approach given the large amount of data available. Indeed, many would jump to argue that such an approach would be fruitless if the random walk hypothesis for stock market returns is taken into consideration. Thus, given an embedding, a method for approximation, and a cost
function the problem becomes how to choose those data which are most significant in forecasting the future price of a given stock within some period of its history. In addition, the smaller the size of the significant set the better, for a set too large will be intractable by a neural network. As with all forecasting methods, it is assumed that once this significant set of forecasting data is determined for a period it will remain significant during a period of time following the period used to determine the significant set. The validity of this assumption is course debatable.

4.3.1 Optimal input set by natural selection

To solve this problem, it was decided to go with a brute force natural selection process. The idea is to generate hundreds to thousands of neural networks each attempting to forecast the same stock. Each network has a unique input set which is defined in a genome associated with that network. Those networks which fail to perform well are "killed" and replaced by children created from the genomes of those networks which are performing well. In addition, the smaller the size of the input set of a network as defined by its genome the healthier it is considered to be. Through a process of natural selection, small networks with high predictive ability are created. While this approach will produce good results in theory, there are the practical limitations of computer power and time - real time.

As with all genetic algorithm approaches, the necessity for a large population becomes particularly acute in this case. The time needed for the genetic algorithm to find a good solution is proportional to population size and inversely proportional to computational power. The amount of processing power needed to train and maintain such a population in a reasonable fashion is far more than current PCs or workstations have at their disposal. However, there is no reason why the population needs to be maintained at a central location.

Taking advantage of this fact, it was decided that the population of neural networks would be distributed over a network of machines with each machine in the network trains and maintains a group of networks. The processing power in this case is limited only by the number of machines connected to the network and is thus virtu-
ally limitless. Using this approach, the virtual power of a supercomputer is achieved with only a handful of UNIX workstations leading to a very small cost vs. power ratio.

In this fashion, the problem of determining what \( \tilde{r}_{t-1}, \tilde{r}_{t-2}, \tilde{r}_{t-3}, \ldots, \tilde{r}_{t-d} \) will produce good \( z_t \)'s is handled.

### 4.3.2 Genomes and weight elimination

Using Rumelhart's weight elimination scheme enhances the natural selection environment described above by accelerating the search done by the genetic algorithm. As Rumelhart's weight elimination reduces the number of connections a network has by driving weights to zero, nodes in the network are eliminated when all the weights on the connections leaving a node are zeroed. When an input node is eliminated, the bit of the genome associated with that node is set to zero; eliminating that hyperplane from the search. I call this effect "self modifying genomes."

It is assumed that this characteristic is positive, for it accelerates the optimization entailed by the genetic algorithm by reducing both the search space and the size of the related networks. Furthermore, the genome size is reduced. The genetic algorithm implemented for this system uses variable sized genomes which include information on only those datum of the input set which are being used by the associated neural network.

Each datum in the database under consideration forms the input set. Each datum is assigned an integer triple, \( (i, j, k) \). The first integer, \( i \), identifies the lag of the variable relative to the current time, \( t \). While the neural network is training, it is moving through history and as such has a current time frame; a time frame which is variable. A network may wish to have the return of some stock at some point in history relative to its current time; for example, the return of stock X at lag three, or time \( t - 3 \), is the return of stock X three time units before current time.

The second integer, \( j \), of the pair uniquely identifies the stock and the third, \( k \), identifies the parameter type. Parameters are items such as return or volume. For example, the return on stock A with no lag might be assigned \( (0, 12, 2) \) while the
return on stock B with lag five might be assigned (5, 123, 2). A subset of integer triples is used as the “bits” of given genome. When genomes are recombined the integers are sorted and duplicates are eliminated. While the input set, possibly the entire database, may be very large, the genomes are only as large as the size of the subset being used by their associated neural networks.

Following the example in Chapter 3, all the integer triples can be considered “ones,” and all those integer triples in the input set which are not included in the a genomes subset can be considered “zeros.” Since the integer triples essentially encode the position of the “ones” in the genome, the “zeros” can be eliminated.

A mapping from the integer triples to their respective real data is maintained by the database manager process which is described below. This mapping is transparent to the rest of the system, and the input set can be increased without effecting the system.

The implemented genetic algorithm starts with “large” genomes which naturally reduce via the genetic algorithm’s cost function, which favors smaller genomes, and as the weight elimination scheme operates to reduce network size. For obvious reasons, a manual limit is set on the magnitude of the lag index of the genomes. The genomes, which define the neural networks the training units create and maintain, also define the inputs which the neural network will use, the initial architecture - number of nodes in each layer, and the type of network - feedforward or recurrent. Associating genomes with the input parameters of dynamic systems, such as the neural networks used in this forecasting system, appears to be useful and has biological implications.

4.4 Design Specifics

4.4.1 Equipment

The forecasting system was developed as a distributed application written in the C programming language in a UNIX environment with socket based interprocess communication. All the software comprising the system, approximately 8000 lines of
C code, is original to this project. MIT's Athena network was used as the hardware platform for development and testing. The Undergraduate Electrical Engineering and Computer Science Department supplied a partition on a file server for data storage. The CRSP database of securities prices which is licensed to MIT's Sloan School of Management for academic research was utilized as the data source.

4.4.2 Implementation

The implementation was broken up into two phases; the communication framework, and the system code. The socket based communication framework was implemented and tested separately from the rest of the system. Aside from allowing for easier testing, this division creates a clean separation between the communication framework and the system code and prevents the system code from becoming dependent on the underlying interprocess communication; an important feature for portability. The communication framework consisted of writing to communication framework for each of the five main processes of the system plus a remote slave process. The five main processes are diagrammed in Figure 4-1.

Aside from the communication framework and the remote slave process, there are five main parts to the design; the forecaster process, the resource manager process, the stock expert process, the training unit process, and the database manager process.

Remote Slave

One remote slave server process resides on each machine which is part of the system. It is a server to the forecaster, resource manager, and stock expert processes. The remote slave process provides a socket though which its local machine can communicate with other machines. It allows the creation and destruction of stock expect and training unit processes on its machine as well as providing load information to the resource manager.
Figure 4.1: Forecasting Model

Generic Algorithms & Neural Networks
Forecaster

The forecaster provides user interface to the rest of the system. The forecaster is a client to users and is at the top of the client/server hierarchy. The forecaster is a client to the resource manager, the stock experts which it creates, and to remote slave process which facilitate the creation of stock experts. The forecaster is started by the user via a C shell call. At start time the forecaster figures out what the local machine name is and creates a resource manager on the same machine. After doing a communications handshake with the resource manager, the forecaster starts a menu driven user interface. The following functions are provided to the user via this menu:

1. Add machine — Add machine adds a new machine to the system. The machine must be accessible on the network and have a remote slave server process running. The address of the remote slave (an address of a process is comprised of the machine name on which the process is running and the port number though which that process may be accessed) is needed for successful addition of a machine.

2. Remove machine — Remove machine removes a machine from the system. As with add machine the address of the remotes slave server process of the machine to be removed needs to be known.

3. Create stock expert — Create stock expert creates a stock expert process on the lowest load machine currently being used by the system. The ticker name of the desired stock is needed to create a stock expert for that stock.

4. Destroy stock expert — Destroy stock expert destroys a stock expert process. As with create stock expert, the ticker name of the desired stock must be known.

5. Get forecast from stock expert — Get forecast returns a forecast on the future value of the desired stock if possible. The ticker name of the desired stock must be known and a stock expert for that stock must exist. The date of the day for which the forecast is desired must also be given.
6. Set memory allocation for stock expert — This is part of the memory allocation scheme. Currently the larger the amount of space allocated to a stock expert the more training units it will be able to maintain and thus the faster it will be able to generate a forecasts.

7. Set database — Set database adds a new database manager interface to the system. The address of the database manager must be known to set a database. More than one database manager may be used at a time which can eliminate a bottle neck which may occur if there are a large number of training units requesting data from a single database manager.

8. Shutdown system — Self explanatory.

Resource Manager

The resource manager process resides on the same machine which the forecaster resides. After completing a communications handshake with the forecaster which started it, the resource manager exists as server to the forecaster and to the stock experts which the forecaster creates. The resource manager is a client of remote slave processes which provides it with load information. The resource manager has two functions:

1. To keep track of all the machines on the network which are currently being used by the system. The resource manger maintains a list of the addresses. The addresses are comprised of the machine name on which a stock expert resides and the socket number though which the remote slave process can be accessed.

2. As a service to the forecaster process, which may wish to start a new stock expert process, and to the stock expert processes, which will wish to start new training unit processes, the resource manager locates and returns the remote slave address of the lowest load machine currently active in the system.
Stock Expert

The stock expert process can reside on any machine with a remote slave process. There is one stock expert per stock for which forecasts are being attempted. The stock expert exists as a server to the forecaster for which it provides forecasts for its unique stock. The stock expert is a client to the resource manager process, the remote slave processes, and the training unit processes which it creates. The stock expert has two functions:

1. To provide, upon request, forecasts for the stock which the stock expert is an "expert" on to the forecaster. Once the training units produced by the stock expert begin to produce valid forecasts, the stock expert creates a "customer service" which is comprised of those neural networks from the set of all training units which are producing valid forecasts. The customer service is comprised of a subset of all the training units which the stock expert maintains. This subset is updated periodically so that the top ranking neural networks fall within the subset. A forecast is produced by a democratic voting procedure. The tabulated output is reported as the forecast.

2. To maintain a population of training units, each containing a unique neural network defined by a genome, using a genetic algorithm to produce and an environment where natural selection will create valid forecasting neural networks. After a network matures (when training has plateaued), the genetic algorithm ranks the network against its peers based on the networks performance. Those networks which fail to make the ranks are "killed" and replaced by the offspring of networks which are in the ranks. If a newly ranked network makes the ranks, it will displace a network from the ranks if the ranks are full. The displace network is "killed" and replaced in the same fashion as previously mentioned.

When a stock expert is created, it creates as many training units as possible given the memory allocated. The initial training units are created with pseudo-random genomes. Many of the genomes are biased initially based on fundamental and practical knowledge from standard finance theory.
Training Unit

The training unit process can reside on any machine with a remote slave process. There are zero or more training units per stock expert with an upper bound on the number of training units dictated by the amount of memory allocated to the associated stock expert. The training unit process exists as a server to a stock expert process to which it is accountable. The training unit process is a client to the database manager process which provides it with the data necessary to train its neural network. The data needed is defined by the training units genome which is provided by the associated stock expert. The training unit has two functions:

1. Primarily to train the neural network which is defined by the genome which it is handed upon its creation. The data necessary to train the network is provided by the database manager process.

2. To provide status and performance information to its parent stock expert upon request. Status information provides the information necessary to determine the maturity of the training unit. That is, information necessary to determine if the neural network of the training unit is ready to be ranked. Performance information provides the information necessary to determine the rank of the training units neural network.

The training unit process is divided into four modules; the controller, the neural network, the performance evaluator, and the data driver. The data driver is further divided into two parts; the main driver and the subdriver. The subdriver is the back end to the data driver which provides a communication scheme dependent interface to the database manager and thus can be considered part of the communications framework, but, due to its close (but independent) relationship to the training unit, the data driver it is lumped in with the training unit. The function of the four modules are:

1. Controller — The controller directs the actions of the training unit. The controller operates in essentially two states. The first is the training state in which
the training unit continuously trains its neural network. The second is server mode in which the controller directs the training unit in a fashion which produces information requested by the stock expert - specifically status and performance information.

2. Neural network — The neural network module holds the neural network and provides functions to build, manipulate, and train the neural network. The actions taken on the neural network are controlled by the controller.

3. Performance evaluator — The performance evaluator stores and updates the performance and status statistics of the neural network. The performance evaluator is controlled by the controller.

4. Data driver — The data driver provide a transparent interface for the training unit to the database manager. The data driver handles all the data manipulation so that the controller can update the input and target data of the neural network with a single call - get_next_data(). All the details of data organization and processing are handled transparently to the rest of the training unit. The data necessary is defined by the training unit’s genome. The sub driver does the actual data transfer transparent to the main driver. Thus this double transparency facilitates easy moving to new database sources and communication frameworks.

The training unit process can create many different neural network types and architectures (ie. backpropagation, recurrent, four layer, three layer, ...). The types and architectures are defined by the genome passed to the training unit and are easily expandable to create almost any conceivable configuration. The training unit process initializes a connection with the database manager when it is started up. This initialization is handled by the subdriver of the training unit. The subdriver notifies the database of the data set which will be required and there after only requests a new block of data of a given size. The data can be passed in sequential or random order over a time period defined at initialization time.
Database Manager

The database manager process resides on a user defined machine. There are one or more database managers per forecasting system. The database manager exists as a server process to the training unit processes. The database manager acts as an interface to the database which is of course database dependent. The database manager passes blocks of data to training units on request and maintains information regarding the data which each training unit is working on and passes blocks of data of a specified size to the training unit on request. The database manager is split into two parts; a front end and a back end. The front end provides an interface for the training unit processes. The back end is database dependent. If a new database is to be utilized, only the back end need be modified to suit the needs of the new database.
Chapter 5

Results

The initial results reported in this chapter were produced using a portion of the system described in chapter four. This portion lacks the genetic algorithm control structure and a full database. The remaining system generates traditional neural networks like those outlined in previous work on applying neural networks to time series data such as that by Rumelhart[20] and White[27]. The results suggest that while the neural networks showed promising results on in sample data, the neural networks failed to produce significant predictions on out of sample data.

Five stocks, Boeing Co., Disney Walt Co., IBM, Eastman Kodak, and Phillip Morris Inc., were chosen from the Dow Jones 30 Industrials as targets for prediction of return. The database consisted of returns for Dow Jones 30 Industrials from October 1, 1985 to December 29, 1989. The data was broken into three subsets, a training set (October 1, 1986 to September 31, 1987), a presample prediction set (October 1, 1985 to September 31, 1986), and a post sample prediction set (October 1, 1987 to December 29, 1989). Theses subsets were the same for each of the five stocks. The Dow Jones 30 Industrials included the following stocks:
<table>
<thead>
<tr>
<th>Alcoa</th>
<th>Allied</th>
<th>American Express</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT&amp;T</td>
<td>Bethlehem Stl.</td>
<td>Boeing</td>
</tr>
<tr>
<td>Caterpillar</td>
<td>Chevron</td>
<td>Coca Cola</td>
</tr>
<tr>
<td>Disney</td>
<td>Dupont</td>
<td>Exxon</td>
</tr>
<tr>
<td>General Electric</td>
<td>General Motors</td>
<td>Goodyear</td>
</tr>
<tr>
<td>IBM</td>
<td>International Paper</td>
<td>J. P. Morgan</td>
</tr>
<tr>
<td>Kodak</td>
<td>McDonalds</td>
<td>Merck</td>
</tr>
<tr>
<td>Minnesota Mng &amp; Mfg</td>
<td>Phillip Morris</td>
<td>Procter &amp; Gamble</td>
</tr>
<tr>
<td>Sears</td>
<td>Texaco</td>
<td>Union Carbide</td>
</tr>
<tr>
<td>United Technologies</td>
<td>Westinghouse</td>
<td>Woolworth</td>
</tr>
</tbody>
</table>

Thirty neural networks were randomly generated for each stock with the following restrictions:

1. **Input data was limited to returns of the Dow Jones 30 Industrials.**

2. **Data lag was limited to twenty, that is, if a network is attempting prediction of return at time \( t + 1 \), the network could have access only to data from time \( t - 20 \) to time \( t \).**

3. **The number of input units for a network fell in the range of 20 to 100, and the number of hidden units was restricted to the range \( \frac{1}{2} \text{number - of - input - units} \) to \( \frac{1}{4} \text{number - of - input - units} \).**

4. **The networks were non-recurrent.**

One neural network with inputs receiving only lag data, through \( t - 30 \), on the stock it was assigned to predict was also created for each stock. This manufactured network otherwise met the restrictions above. The genomes for both the randomly generated and manufactured networks were used by the forecasting system to launch neural networks. No genetic algorithms were used produce new generations of genomes. The networks were trained until maturity as dictated by the training unit performance.
evaluator. The 155 networks were distributed over twenty Vaxstation 3100 workstations. All 155 networks trained to maturity in less than thirty hours. Performance data on in sample and out of sample data sets was recorded for all 155 networks.

Figures 5-1 through 5-5 display the actual return with the performance data for a chosen network. The y-axis marks return value (the neural networks were trained and tested on return data which was multiplied by one hundred) and the x-axis indexes the date starting with 10/1/85 and running to 12/29/89. The training set data runs from approximately index 250 to index 750. The out of sample data lies along indexes 0 through 250 and 750 through 1100 which comprise the presample prediction and postsample prediction sets respectively.

The network outputs displayed are those of a network which performed well on the training set. On out of sample data, all the networks failed to suggest any predictive ability. While in depth analysis was not entailed, the manufactured network performed as well or better on the in sample data as the randomly generated networks, and most networks tended to recognize the large magnitude returns within the training set, especially around the October 1987 period.
Figure 5-1: Returns for Boeing with Neural Network Performance
Figure 5-2: Returns for Disney with Neural Network Performance
Figure 5-3: Returns for IBM with Neural Network Performance
Figure 5.4: Returns for Kodak with Neural Network Performance
Figure 5-5: Returns for Phillip Morris with Neural Network Performance
Chapter 6

Conclusion

At this point the system cannot be declared a money machine. While the initial results suggest that the neural network system does not have the ability to forecast out of sample data, this by no means proves such systems are incapable profitable behavior. While the neural network system as outlined in the thesis has not been fully tested, what has been tested supports the results of the research on which much of the system has been designed. The neural networks clearly learn the data on which they are trained, but at this point, fail to show an ability generalize on stock market returns. Whether or not this is work of the efficient markets hypothesis will be determined at some future point.

While the value of applying genetic algorithms and recurrent networks to this task is yet to be determined, the distributed design allowed for efficient construction and training of many neural networks concurrently - a technique that lends itself easily to other neural network projects.

All in all, it has been a lot of fun so far.
Bibliography


