A COMPARISON OF STOCHASTIC GLOBAL OPTIMIZATION METHODS: ESTIMATING NEURAL NETWORK WEIGHTS

By

LONNIE KENT HAMM

Bachelor of Science
Oklahoma State University
Stillwater, Oklahoma
1991

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1995

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
August, 2003

A COMPARISON OF STOCHASTIC GLOBAL OPTIMIZATION METHODS: ESTIMATING NEURAL NETWORK WEIGHTS

Thesis Approved:

Wade Brorsen

Thesis Adviser

[illegible]

[illegible]

[illegible]

[illegible]

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my major advisor, Dr. B. Wade Brorsen for his constructive guidance in conducting this research. This research would have been considerably more difficult without his encouragement, patience, and friendship. I would especially like to thank Dr. Brorsen for the financial resources he provided to pay my assistantship salary and purchase computers and software. In addition, I would like to thank the Department of Agricultural Economics for their financial support during my graduate studies. I would also like to thank Dr. Martin Hagan for his helpful comments and his willingness to share his extensive knowledge of neural networks with me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

SUMMARY

1.1 Introduction

A neural network is an analytical tool which models the relationship between a given set of independent and dependent variables. Since their introduction in the mid-80's by Rumelhart, Hinton, and Williams, neural networks have received considerable attention. At times, this attention may have been nothing more than marketing hype. However, over time, neural networks have become accepted as a mainstream analytical tool. Neural networks can be found in statistical packages such as SPSS, SAS (Enterprise Miner), Statistica, and RATS. Companies such as MasterCard, American Express, Wal-Mart, and KayBee toys are using neural networks and applications range from fraud detection, product marketing, and financial prediction, to medical diagnosis (White, 2002).

The most widely used type of neural network, and the object of this study, is the feedforward type of multilayer perceptron (MLP). For much of the remaining discussion in this and following chapters, the MLP type of neural network will be referred to generically as a "neural network". Much of the excitement surrounding the MLP, and other types of neural networks, is due to their ability to model highly nonlinear relationships. Neural networks have been shown to be universal approximators capable of approximating almost any function (Cybenko; Hornik, Stinchcombe, and White, 1989; Funabashi). The neural network approximates the relationship between the dependent and independent or explanatory variables as well as the interaction between the explanatory variables. The universal approximation capability of a neural network means the functional form of the model does not have to be explicitly specified.

A more detailed discussion of neural networks and the universal approximation property is deferred until chapter 2. At this point it suffices for the reader to understand that a neural network can be viewed as estimating a map $f:\Re^n \to \Re^m$ where n is the number of explanatory variables and m is the number of dependent variables. The relationship between the space of explanatory variables $\mathbf{X}$ and the space of dependent variables $\mathbf{Y}$ is embodied by

(1.1) $$f:\mathbf{X}\to\mathbf{Y}.$$

The relationship in (1.1) is estimated empirically from data that is composed of T repeated measurements of $\mathbf{X}$ and $\mathbf{Y}$, namely $n\times 1$ and $m\times 1$ vectors $\mathbf{x}_t$ and $\mathbf{y}_t$, $t=1,\ldots,T$.

The dependent variable(s) $\mathbf{Y}$ are rarely completely determined by the explanatory variables $\mathbf{X}$, therefore, our model with the as of yet unknown function f can be written as:

(1.2) $$\mathbf{Y}=f(\mathbf{X})+\varepsilon,$$

where ϵ is a random error term. We would like to estimate the function f in such a way that minimizes the error ε.

The universal approximation property means that a neural network can estimate the function $f(\mathbf{X})$ in (1.2) arbitrarily well. The neural network can be written as

$f(\mathbf{X},\theta,\alpha)$, where θ is a vector of model parameters that must be estimated, and α is a vector of parameters that controls the mathematical form of the neural network, i.e. the number of hidden layers and hidden neurons. Setting the variable α to an appropriate value and estimating θ controls the flexible functional form of a neural network. Therefore, given a fixed α, the ability of the neural network to estimate the mapping rests upon our ability to estimate the parameter values θ. The selection of the values for α is also central to the networks ability to estimate the mapping, however, for purposes of this research these parameters are considered fixed. More will be said on this subject in chapter 2. The process of estimating the weights or parameters of a neural network is commonly referred to in the neural network literature as "training", "supervised learning", or simply "learning". The next section briefly discusses learning in neural networks in the context of global optimization.

1.2 Neural Network Learning

The goal of neural network learning is to minimize some error function. One of the most common is the least squares error function given by:

$$Q_\alpha(\theta) = n^{-1}\sum_{t=1}^{T}[y_t - f(\mathbf{x}_t,\theta,\alpha)]^2 \tag{1.3}$$

which minimizes the mean squared error. The function Q is also called the objective or cost function. The least squares estimator $\hat{\theta}$ solves

$$\min_{\theta\in S} Q_\alpha(\theta), \tag{1.4}$$

where S is the set of all feasible model parameters. Given a sufficient number of observations or training examples, standard estimation techniques can be used to achieve optimal approximation (Kuan and White). Since the cost function for this problem is nonconvex, nonlinear optimization algorithms are required to estimate θ.

In the early years of neural network research, a gradient descent algorithm was commonly used to estimate the parameters of neural networks. Application of the gradient descent method to neural networks was known as backpropagation (BP)[1]. Gradient descent has long been known to be an inefficient estimation method. See Hagan, Demuth, and Beale and Sima for a discussion of some of the drawbacks of backpropagation for training neural networks. The estimation of neural networks can be viewed as being equivalent to estimating a nonlinear regression. Neural network researchers have drawn from the large body of knowledge that exists for estimation of nonlinear regressions and applied efficient estimation techniques such as various conjugate gradient methods (Møller; Smagt; Kinsella; Johansson, Dowla, and Goodman), Broyden-Fletcher-Golfarb-Shanno (BFGS) (McLoone and Irwin), and Levenberg-Marquardt (Hagan and Menhaj). A survey of various training methods is given by Battiti, and Sjöberg et al.

In spite of the increased efficiency of the aforementioned estimation techniques, they are still classified as local search techniques. In other words, they suffer from stopping or converging at a local minimum. A local minima is defined as follows. Let S denote the set of all feasible model parameters θ and $\theta^* \in S$ denote the location of a local minimum for $Q(\theta^*)$ defined in (1.3). Define a $\delta-$neighborhood, $\mathrm{N}(\theta^*,\delta)$,

[1] We use the term backpropagation to encompass a strict gradient descent method as well as other heuristic modifications of gradient descent such as addition of a momentum term or a variable learning rate.

around θ^* as the set of feasible points $\theta \in S$ such that $0 < \| \theta - \theta^* \| < \delta$. Then a local minimum is defined as[2]

(1.5) $$Q(\theta^*) < Q(\theta) \text{ for all } \theta \in \mathrm{N}(\theta^*, \delta) \in S.$$

In other words, the objective function value at the local minimum is less than any other objective function value in a neighborhood around the local mimimum. However, if we go far enough away from the local minima, i.e. somewhere outside the δ – neighborhood, we may find a function value that is less than the value at the local minimum. The objective function for training a neural network is multi-modal and thus any local search algorithm will suffer from local minima. A local search technique will generally proceed downhill in the search space from its starting point towards the nearest minimum. What we want to find is the global minimum. A global minimum would be defined as[3]

(1.6) $$Q(\theta^*) < Q(\theta) \text{ for all } \theta \in S.$$

In other words, no matter where we go in the feasible search space, we will not find an objective function value that is less than the value at the global minimum.

Many tricks have been invented to address the local minima issue when using a local search algorithm to train neural networks. For example, training with noisy exemplars, and perturbing the parameters after convergence to try to escape from what may be a local minima. One of the most common methods is to restart a local optimization routine with a new random set of starting values for the weights. This

[2] Technically, (1.5) is defined as a strong local minimum. A weak local minimum is defined as in (1.5) with the $<$ relation replaced by $\leq$.
[3] The relation $<$ in (1.6) could be replaced by $\leq$ for some problems. For example, in the type of neural network of concern in this study, symmetries in the mathematical form of the network causes certain permutations of the elements of θ to produce identical outputs from the neural network.

restarting method is sometimes called a multi-start algorithm. The quality of the final solution from a multi-start algorithm will depend upon the number of restarts.

While the methods above may lead to improved solutions, there is no guarantee that such minima will not also be only locally optimal. Global search techniques are an alternative estimation technique. Global optimization algorithms are a class of algorithms that seek to avoid getting trapped in local minimums. Two classes of global optimization methods exist: stochastic or deterministic approach. It should be noted that although methods such as multi-start local optimization algorithms are not generally considered a global optimization algorithm, they could be considered a quasi-global algorithm. This is because as a practical matter many global algorithms, especially the stochastic kind, only offer asymptotic guarantees of a global minimum. Therefore, global algorithms may increase the probability of obtaining a good solution but do not offer any real guarantees of reaching a global minimum. Therefore, for some problems, a multi-start algorithm could be considered competitive with a more traditional global algorithm, especially a stochastic algorithm.

Some deterministic global methods are branch and bound, Lipschitz programming, outer approximation, and concavity cuts. By taking advantage of the mathematical structure of the problem, these methods can guarantee, within a specified level of accuracy, convergence to a global minimum in a finite amount of time (Horst, Pardalos, and Thoai; Ryoo and Sahinidis). For a review of these deterministic approaches see Horst and Tuy.

There have only been a few uses of deterministic optimization techniques for learning in neural networks. Tang and Koehler used a Lipschitz optimization approach.

Their procedure required solving a number of nonlinear nonconvex maximization problems in a recursive manner to find local approximations of the Lipschitz constant. Therefore, the algorithm was very numerically demanding. Shang and Wah applied a deterministic algorithm they called NOVEL to neural network learning. The NOVEL method, introduced by Shang and Wah, is a hybrid global/local minimization method. Starting points for a gradient based local search technique are selected by solving an ordinary differential equation specifying a trajectory through the search space. Shang and Wah indicated that NOVEL performed better than gradient descent, and evolutionary algorithms on some benchmark problems. However, the algorithm is computationally demanding because it requires the evaluation of differential equations in each global search phase.

Because of their computational requirements for problems with more than a few parameters, deterministic algorithms will not be explored in this study. However, it is worth noting that the field of deterministic global optimization is fairly new and active and new methods may some day be developed which could be appropriately applied to learning in neural networks.

Stochastic global optimization methods have been widely applied to optimization of neural networks. Stochastic methods employ random elements in their search procedure. None of these stochastic methods guarantee a global solution but if used they should increase the probability of obtaining a good solution. Most of these algorithms can be shown to converge to the global optimum with a probability approaching one as the number of iterations of the algorithm approaches infinity. Simulated annealing (SA) and evolutionary algorithms are two of the most commonly used global optimization algorithms and are the two global methods used in this research. The term evolutionary algorithm (EA) is used here as an umbrella term encompassing genetic algorithm (GA), evolutionary strategies (ES), and evolutionary programming (EP) methods. SA and EA methods have been used by many researchers to estimate the parameters of neural networks.

1.3 Literature

Evolutionary algorithms have been used to estimate neural networks by a great many researchers (e.g. Chen and O'Connell; Frenzel; Neruda (1997, 2000); and Yan and Zhu). Yao provides a good literature review of combining EA's with neural networks. An indexed bibliography of EA's and neural networks is given by Alander. In this study, we are only concerned with estimating the weights of neural networks with a fixed architecture. That is, a neural network with a fixed number of hidden layers and hidden neurons. However, EA's have also been used to evolve the architecture of neural networks as well as the weights (e.g. Maniezzo; Harp, Samad, and Guha; Miller, Todd, and Hegde; Angeline, Saunders, and Pollack; Pujol and Riccardo). In addition, EA's have also been used to select training data and interpret the outputs of neural networks (Guo and Uhrig; Chang and Lippmann; Brill, Brown and Martin). A review of different types of applications of EA's to neural networks is given by Yao; Whitley; Schaffer, Whitley, and Eshelman.

Genetic algorithms are the most widely used type of EA for estimating neural networks. Sexton, Dorsey, and Johnson (1998) compared a GA based training algorithm with BP for 7 test problems. Overall, the solutions obtained by the GA dominated the solutions obtained by BP. In addition, the solutions obtained by the GA had significantly less variation in the in-sample root-mean-squared error across different runs. Bartlett and Downs found that a GA was an order of magnitude faster than BP on the 7-bit parity problem. However, on the smaller XOR problem data set, Bartlett and Downs found that a GA training method was slower than backpropagation.

In spite of the encouraging, although mixed at times, results of many researchers, there are theoretical reasons why a genetic algorithm may not perform satisfactorily for training neural networks. Because of symmetries in the mathematical functional form of a neural network, there can be numerous equivalent solutions to the optimization problem. This is called the competing conventions problem (Whitley) or permutations problem (Radcliffe). The permutation problem reduces the performance of a genetic algorithm because of its negative effect on one of the basic operational components of a genetic algorithm, the crossover or recombination operator. The permutation problem interferes with the crossover operators ability to combine solutions from previous iterations or generations into new superior solutions. Hancock, however, concluded that the permutation problem is not as bad as has been suggested for genetic algorithms. Nonetheless, several researchers have proposed solutions to this problem for genetic algorithms. Montana and Davis attempted to incorporate knowledge of the functional aspects of hidden neurons into a crossover operator for their genetic algorithm. They compared their GA to BP for optimizing a neural network for the classification of sonar data. It was found that the GA outperformed BP by a large margin. Rooij, Jain and Johnson proposed a crossover operator similar in concept to that of Montana and Davis. When compared against BP on 5 classification problems, Rooij, Jain, and Johnson found that their genetic algorithm was less susceptible to becoming stuck in local minimums, however, overall the GA was only marginally better than BP and BP was significantly faster on some problems.

Evolutionary programming and the closely related evolutionary strategies are two evolutionary algorithms that typically do not employ crossover operators but instead rely mainly on mutation operators to modify the chromosomes. Since they do not use crossover, they may theoretically perform better than GA's for training neural networks. Porto, Fogel, and Fogel compared an evolutionary programming method with backpropagation for a sonar classification problem. They found that evolutionary programming performed better than backpropagation. Backpropagation repeatedly stalled at suboptimal weight sets that did not yield satisfactory results. The drawback of the study was they only used one data set and compared evolutionary programming to inefficient BP.

In addition to EA's, SA algorithms have been applied to a wide extent in training neural networks. In the study by Porto, Fogel, and Fogel mentioned above, an SA algorithm was also included in the comparison. The SA algorithm performed similarly to the EP algorithm. Sexton, Dorsey, and Johnson (1999a) compared the performance of a simulated annealing algorithm from Goffe, Ferrier, and Rogers against BP on the same 7 test problems from Sexton, Dorsey, and Johnson (1998). The simulated annealing algorithm exhibited superior performance with respect to both in-sample root-mean-square error as well as out-of-sample interpolation and extrapolation. Cohen, Saad, and Marom used the adaptive simulated annealing algorithm of Ingber for training of a time delay recurrent neural network (TDRNN). The training of TDRNN is known to be a very difficult problem. Day and Camporese used a SA algorithm to train a network with non-differentabilites in the objective function with success. However, they also reported that a stochastic tunneling algorithm required less time than the SA algorithm. Others successfully using SA or a modified SA algorithm are Fang and Li; and Barnes, O'Neill and Wood.

Evolutionary algorithms and simulated annealing are not efficient at fine tuning a local search but are designed to be adept at exploring the search space and finding regions that may contain a good solution. Therefore, an alternative use of these global algorithms is for finding a good set of initial weights for a local optimization routine. This type of hybrid method would be expected to outperform either a local gradient method or a global algorithm used individually. Many researchers have used this type of hybrid training (e.g. Chen and O'Connell; Lee;Omatu and Deris; Erkmen and Ozdogan; Omatu and Yoshioka; and Xinxing and Licheng). Belew, McInerney, and Schraudolph used a hybrid approach with a genetic algorithm to generate starting values for a conjugate gradient or backpropagation local optimization algorithm. They found that training times could be reduced by as much as two orders of magnitude. However, their research was limited to a single data set. Skinner and Broughton reported that for a small network (18 weights), the local conjugate-gradient algorithm outperformed a GA in addition to a SA and a swarm search algorithm. However, for a larger more complex parameter space, (98 weights), a hybrid scheme with simulated annealing or genetic algorithms in combination with conjugate-gradient local search technique showed a dramatic improvement in convergence. They also reported that they have successfully used their hybrid approach to train networks with as many as 600 weights. Heistermann reported that a hybrid EA and gradient algorithm outperformed the gradient algorithm alone for large problems. Although he also reported that for small to medium size problems the gradient algorithm was more efficient than the hybrid method. Other studies have also found superior performance for hybrid techniques (Likartsis, Vlachavas, and Tsoukalas; Yan, Zhu, and Hu ; and Knowles, Corne, and Bishop). However, Kitano reported contradictory results. Kitano found that a hybrid GA-BP method was at best equally efficient to faster variants of BP in neural networks of small size and were far less efficient in large networks.

Besides simulated annealing and evolutionary algorithms, other types of stochastic optimization algorithms have been proposed. Baba et al. presented a hybrid algorithm that used a combination of the Solis and Wets random optimization method and conjugate gradient training. They compared the hybrid algorithm to local optimization routines on 3 different problems and found that their hybrid method was very efficient at finding low error values as compared to conjugate gradient and backpropagation training. Brunelli introduced a new stochastic algorithm called iterated adaptive memory stochastic search (IAMSS) and found that it performed better than backpropagation on two test problems. Barnard also proposed a new stochastic training technique that performed well against various local optimization techniques.

1.4 Research Objectives

Overall, with some exceptions, the literature shows that global optimization algorithms such as evolutionary algorithms or simulated annealing, used alone or in combination with a local search algorithm, offer some advantages. From an experimental evaluation standpoint, many of the studies reviewed in the previous section are lacking in quality. Many of the studies looked at the performance of the algorithms on only few data sets, and in some cases only a single data set. In addition, the majority of the data sets were classification problems. Little attention has been paid to function approximation problems. The majority of the studies above compared global algorithms to BP. It is well known that there are much more efficient alternatives than BP local optimization routines. Yao's review concludes that contradictory results are partially due to the fact that in some studies, the EA's were compared with the relatively slow BP algorithm. Also, few of the studies have compared results across different global optimization routines. This study attempts to provide a more rigorous comparison of several global algorithms against efficient local optimization routines across a wide variety of data sets, both real-world and simulated data, in a function approximation context. The objectives of this study are as follows.

General Objective:

> Determine the relative speed and accuracy of alternative global optimization methods in estimating the weights of neural networks.

Specific objective:

> Determine the relative speed and accuracy of 10 alternative global optimization algorithms for estimating the weights of neural networks by performing multiple estimations from random starting values on 6 function approximation problems and analyzing the running time and distribution of the final objective function values over the multiple estimations.

1.5 Procedure

The objectives given above are addressed by estimating the parameters of neural networks trained on 6 different function approximation problems in a Monte-Carlo setting. This is done by repeating the estimations on each of the data sets numerous times from different starting values. The objective function values after convergence from each of these restarts are saved for further analysis.

This study is limited to estimation of the parameters of a particular type of feedforward type of neural network, the multilayer perceptron (MLP). The number of hidden layers and hidden neurons is chosen based on previous usage of the particular data set of interest or on an ad-hoc basis. Depending upon the size of the estimation problem, either a quasi-Newton or conjugate gradient algorithm is used for the local optimization routine. The global optimization algorithms investigated are two simulated annealing algorithms, one simple random stochastic algorithm, one genetic algorithm and five evolutionary strategy algorithms. All of the global optimization algorithms are a hybrid between the aforementioned global algorithms and one of the local search techniques. The weights after convergence of the global algorithm are used as starting values for the local optimization algorithm.

The results of the Monte-Carlo estimations will be displayed both numerically and graphically. The results will be displayed graphically using histograms and boxplots of the final converged objective function values. In addition, basic statistics such as mean, median, standard deviation and maximum and minimum values will be computed for the objective function values for each data set and each algorithm. In addition to the basic statistics, analysis will be performed which takes into account the computing time involved with a particular algorithm. An algorithm that converges to a minimum quicker than another algorithm could be restarted from different starting points more times in a given amount of computing time than a slower algorithm. Therefore, even though a slower algorithm, e.g. a genetic algorithm, may be more successful at finding lower minimums than say a relatively faster local search algorithm, e.g. a conjugate gradient algorithm, the local search algorithm could be rerun more times in a given time frame. Therefore, the local search technique may be competitive with the global algorithm.

1.6 Organization

Chapter 2 discusses the theory of neural networks. The history and development of neural networks is briefly discussed and some applications of neural networks are presented. The theory of the multiplayer perceptron type of feedforward neural network will be presented in detail in addition to a brief discussion of a few other types of neural networks.

Chapter 3 discusses some global optimization algorithms including evolutionary algorithms, simulated annealing, and a simple stochastic optimization algorithm introduced by Solis and Wets. The first section introduces two evolutionary algorithms, the genetic algorithm and several evolutionary strategy algorithms. It also discusses some issues related to implementing an evolutionary algorithm for the training of a neural network. The next section discusses two different simulated annealing routines and finally the last section discusses the stochastic optimization routine proposed by Solis and Wets.

Chapter 4 presents the details of the methods used to accomplish the research objectives. The first section presents some of the details and the relevant parameter setting of the optimization algorithms used in this research. Next is a description of the data sets. The last section describes how the results of the simulations will be presented and contains a description of some of the statistics used to summarize the results.

Chapter 5 presents and discusses the results of the study. The results of the estimation of the parameters of the neural network models on the data sets across the various training algorithms are presented. Mean, median standard deviation, and maximum and minimum values obtained across restarts are presented for each of the data sets and training methods. Graphical displays of the distribution of objective function values after convergence from the various training algorithms is displayed in histograms and box plots. The results are discussed in the context of the research objectives.

The last chapter summarizes the study's results and conclusions. General conclusions on the applicability of the various optimization algorithms used in this study for estimation of multiplayer perceptrons are presented. Some directions for future research are also discussed.

CHAPTER 2

NEURAL NETWORKS

2.1 Introduction

The human brain is a marvel of nature, for many tasks it is superior to the most complex supercomputer. The human brain is especially adept at processing visual information; recognizing objects, faces, and so on. A brain can adjust to a new environment by "learning" and it can deal with information that is fuzzy, noisy, or otherwise inconsistent. Because of these factors, researchers have sought to use the biological concepts of the brain and its neurons to develop new computing and pattern recognition paradigms. These efforts led to the development of various biologically inspired input-output models in the 1950's and 60's. Development in this area virtually ceased when Minsky and Papert showed in 1960 that a particular type of these models, perceptrons, could not solve some very simple problems.

Research in biologically-inspired models began anew in the early 1980's and culminated in the work of Rumelhart and the PDP Group. The work of Rumelhart and the PDP Group is credited for much of the revitalized research in biologically inspired input-output models, hereafter, generically referred to as neural networks. Rumelhart, Hinton and Williams developed what has become known as the backpropagation neural network. The backpropagation neural network is referred to as a feedforward neural network in this research. The backpropagation neural network overcame many of the shortcomings of the perceptron which was criticized by Minsky and Papert. It should be noted that Werbos in 1974 developed the mathematical framework for the backpropagation neural network, however, his work went unnoticed at the time.

The next section of this chapter briefly discusses some applications of neural networks and alternative neural network paradigms. Section 2.3 presents the feedforward type of neural network in detail. Section 2.4 discusses feedforward neural network's flexible functional form and their abilities as universal approximators. The last section discusses and presents some methods for estimating the parameters of feedforward neural network.

2.2 Applications and Types of Neural Networks

Neural networks are flexible and have been used to solve many different problems. Some of the applications have been to perform coordination tasks (Hougen, Fischer, and Johnam), decode deterministic chaos (Lapedes and Farber; Gallant and White), and recognize hand-printed characters (Fukushima and Miyake). Trippi has assembled various papers which use neural networks in financial market forecasting, macro economic prediction, credit risk classification, exchange rate prediction and other applications related to finance and economics.

The most common uses of neural networks can be classified into the following categories: classification, associative memory, and autoassociative memory. An example of classification would be to classify sonar signals as those reflecting from a submarine or from a naturally occurring underwater object. An example of an associative memory application would be any time series model or a price prediction model. An autoassociative network is one in which some pattern that has been corrupted by noise is presented to the network and the network reproduces the original uncorrupted pattern. In general a neural network can be viewed as estimating a map $f : X \rightarrow Y$ where *X* is the space of inputs or independent variable and *Y* is the space of outputs or dependent variables. In the case of classification *Y* is a $n \ x \ 1$ vector of variables, each of which indicate inclusion or exclusion in one of *n* different categories. In an associative memory application *Y* is a vector containing that which is to be predicted. In an auto associative application *Y=X,* where *X* is the uncorrupted version of the input pattern.

The term "neural network" can mean different things to different people. The term neural network defined in its most general sense is an architecture in which its operations are distributed among many relatively simple processors (Masters, 1993). This definition suggests a great deal of flexibility in what computing paradigms can be called neural networks. Indeed, a great deal of research has been devoted to developing different types of neural networks. The literature is extensive and developing rapidly and therefore a complete review of the subject is beyond the scope of this research. However, for the interest of those readers seeking to do research in this area, several different types of neural networks are briefly discussed below.

Some models that are decades old have received renewed interest because they are easily recast as a neural network. For example Donald Specht's probabilistic neural network which is used for classification is identical to kernel discriminant analysis (Sarle 1994b). Another example would be the functional link network developed by Yoh-Han Pao. The functional link network is simply a multiple regression with a nonlinear front-end, and a nonlinear transformation applied to the output (Masters, 1993). These two types of modeling techniques suddenly attracted attention when they were presented in the context of a neural network.

Other types of neural networks such as feedforward neural networks and radial basis function (RBF) networks are more unique. However, there are some similarities between these types neural networks and existing modeling techniques. It will be shown later that the standard feedforward type network could be thought of as a form of nonlinear regression. Xu, Krzyzak, and Yuille have established some useful connections between kernel regression estimators and RBF networks. Feedforward neural networks are the focus of this research and are discussed in detail in the next section.

2.3 Feedforward Neural Networks

In light of the considerable hype which sometimes surrounds neural networks, it would be useful to discuss what a feedforward neural network is not before discussing what a feedforward neural network is. Neural networks were originally inspired by the way in which a group of biological neurons process information. Therefore, the development of neural networks has its roots in neuroscience. There are obvious analogies that can be drawn between the functioning of artificial neural networks and their biological counterparts. However, an artificial neural network is a much simplified model of the way a collection of brain cells operate. In fact, beyond simple analogies, the neurons in an artificial neural network share little in common with their biological counterparts.

The word neural probably leads people to sometimes write that a neural network simulates the behavior of the human brain. The human brain contains about 1.5×10^{10}

neurons of various types and each neuron receives signals from 10 to 10^4 other neurons (Ripley). Therefore, an artificial neural network is a much simplified mathematical representation of the way a relatively small collection of biological neurons operate. The process by which biological neurons process information is complex. The communication between neurons is both electrical and chemical and each of these communication process is complex. As will become clear in the next section, the neurons or processing elements in an artificial neural network are simple nonlinear functions and the "communication" between the neurons is linear. However, even though a neural network shares little in common with the workings of biological neurons, they are powerful enough to possess the ability to "learn" from experience, develop rules, and recognize patterns in data.

If an artificial neural network is not a model of the brain, the question is what is a neural network? Before proceeding with the answer to this question, it would be useful to associate some of the terminology used in the neural network literature to the corresponding terminology used in statistics or econometrics. The neural network literature refers to (Sarle 1994b):

- independent variables as inputs
- dependent variables as targets
- predicted values as outputs
- individual variables as a feature
- estimation as training, learning, adaptation, or self-organization.
- observations as training patterns
- parameter estimates as synaptic weights or connection strengths.

The following discussion describes how a feed forward type network with one hidden layer produces its output given some input. The notation in the following sections borrows heavily from that used in Kuan and White and Frances and Dijk. Figure 2.1 provides a reference for the discussion. The figure is a graphical representation of a feedforward neural network with 3 inputs or independent variables, one hidden layer with 2 hidden neurons, and 1 output or dependent variable. The neurons in a neural network are arranged in layers. The input layer contains the inputs or independent variables at time *t* and the output layer contains the output(s) or dependent variable(s) at time *t*. Note that similar to a vector autoregression model, there could be more than one dependent variable.

Assume we are given a set of *T* observations or data pairs $\{(\mathbf{x}'_t, \mathbf{y}'_t)\}_{t=1}^{T}$ where $\mathbf{x}_t$ is a $k \times 1$ vector of explanatory or independent variables and $\mathbf{y}_t$ is a $d \times 1$ vector of dependent variables. Then for each observation *t*, the *k* input neurons send the signals $x_{i,t}$, $i = 1, \ldots, k$ to the *h* neurons in the hidden layer via connection weights or model parameters γ_{ij}, $j = 1, \ldots, h$. Note that in figure 2.1 there is an input $x_{0,t}$. This input or neuron is sometimes called a bias neuron and its value is defined to be always 1. Therefore, we have inputs to the network defined by $\widetilde{\mathbf{x}}_t = (x_{0,t} = 1, \mathbf{x}'_t)$. Each hidden unit j takes a linear combination of the inputs by summing the product of the weights connecting the inputs to itself times those inputs, or in other words taking the dot product $\widetilde{\mathbf{x}}'_t \boldsymbol{\gamma}_j$ where $\mathbf{y}_j = (\gamma_{0,j}, \gamma_{1,j}, \ldots, \gamma_{k,j})$. By setting the bias neuron $x_{0,t} = 1$, the weight or parameter $\gamma_{0,j}$ for each hidden neuron *j* is somewhat analogous to the intercept in a linear regression.

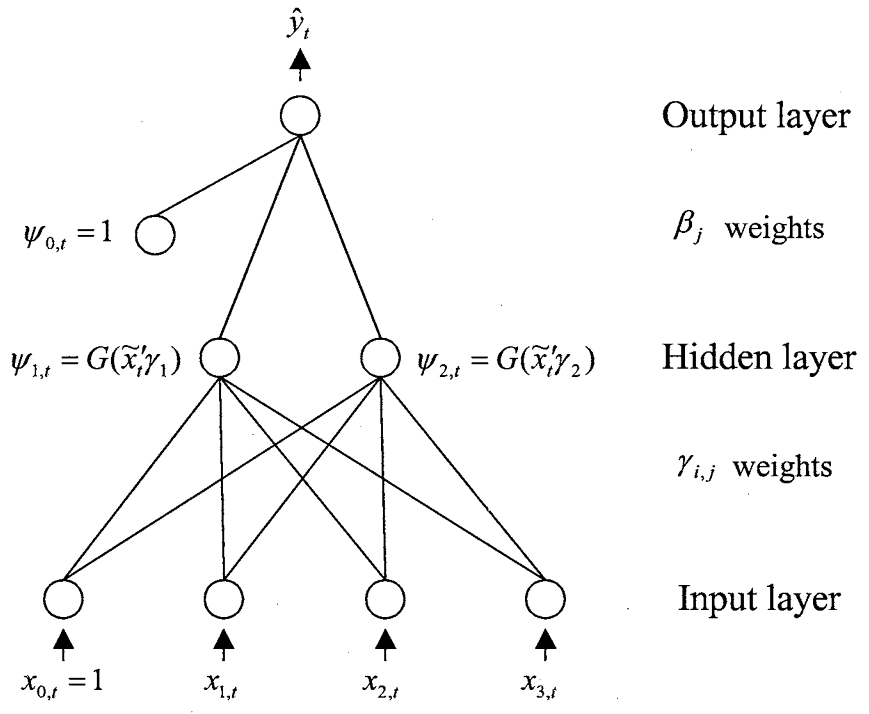

Figure 2.1 Feedforward Neural Network With One Hidden Layer

The linear combination of the network inputs to each hidden neuron is processed by a nonlinear 'activation function' $G:\Re\rightarrow\Re$. The output or activation of hidden neuron *j* is $G(\tilde{\mathbf{x}}_t'\boldsymbol{\gamma}_j)$ or alternatively

$$G(\gamma_{0,j}+\sum_{i=1}^{k}(x_{i,t}\cdot\gamma_{i,j})). \tag{2.1}$$

In other words, each hidden neuron in figure 2.1 is a nonlinear single (scalar) valued function whose input is a linear combination of the networks inputs or independent variables. The form of the activation function $G()$ can be chosen quit freely, however the function is generally monotonically increasing. In addition, the activation function in the hidden layer must be nonlinear. A nonlinear activation function is responsible for the nonlinear approximation capabilities of the feedforward type neural network. The nonlinear approximation capabilities of neural networks will be discussed in more detail in a later section. The two most commonly used activation functions, and the ones used in this research, are the logistic and hyberbolic tangent functions. The logistic function is defined by

$$G(z) = 1/[1 + \exp(-z)], \tag{2.2}$$

and the hyperbolic tangent by

$$G(z) = \tanh(z) = (e^z - 1)/(e^z + 1). \tag{2.3}$$

The activations or outputs from the hidden neurons are passed to the output neuron(s) in an analogous manner as from the input layer to the hidden layer. Let the output from each hidden neuron *j* be represented by $\psi_j = G(\tilde{\mathbf{x}}_t'\boldsymbol{\gamma}_j)$. The hidden layer sends the signal $\tilde{\Psi} = (\psi_0 = 1, \psi_1, \ldots, \psi_h)'$ to each of the q neurons in the output layer via weights or parameters $\beta_{i,j}$, $i = 0, \ldots, h,\ j = 1, \ldots, q$. The term $\psi_{0,t} = 1$ serves the same purpose as $x_{0,t}$ does in the input layer. The output neuron(s) process the signals from the hidden layer in the same way that the hidden neurons process the signals from the input layer. That is by taking a linear combination of the outputs of the previous hidden layer and passing it through an activation function. Assuming an output activation function *F*, the output from neuron *i*, which is the neural networks estimated value of $y_{i,t}$ would be $\hat{y}_{i,t} = F(\tilde{\Psi}_t'\boldsymbol{\beta}_i)$ where $\boldsymbol{\beta}_i = (\beta_{0,i}, \beta_{1,i}, \ldots, \beta_{q,i})'$ or alternatively

$$\hat{y}_{i,t} = F(\beta_{0,i} + \sum_{j=1}^{h}(\psi_{j,t} \cdot \beta_{i,j})), \tag{2.4}$$

where h is the number of hidden neruons. Note that in figure 2.1, there is only one output neuron. The activation function in the output layer is analogous to the activation function in the hidden layer. The discussion above generalizes to the case of a neural network with more then one hidden layer. In that case, the outputs from the neurons in the previous hidden layer become inputs to the neurons in the next hidden layer, and so on, until the output layer.

Using the notation and discussion from above, the function relationship in a feedforward neural network with one hidden layer between the independent variables and the estimated value for a particular dependent variable i and observation t is:

$$\hat{y}_{i,t} = f(\mathbf{x}_t, \boldsymbol{\theta}_i) = F(\beta_{0,i} + \sum_{j=1}^{h} \beta_{j,i} G(\tilde{\mathbf{x}}_t' \gamma_j)) \tag{2.5}$$

where $\boldsymbol{\theta}_i = (\beta_{0,i}, \beta_{1,i}, \ldots, \beta_{h,i}, \gamma_1', \gamma_2', \ldots, \gamma_h')$ is the vector of model parameters or weights and h is the number of hidden neurons in the single hidden layer. It is not necessary to have an activation function in the output layer for a feedforward neural network to be a universal approximator. Therefore for function approximation types of problems, the activation function in the output layer is often dropped. If we assume that the activation function F is the identity function *$F(a)$* $= a$ and for simplicity there is only one output or dependent variable, equation (2.5) reduces to

$$f(\mathbf{x}_t, \boldsymbol{\theta}) = \beta_0 + \sum_{j=1}^{h} \beta_j G(\tilde{\mathbf{x}}_t' \gamma_j) \tag{2.6}$$

where $\theta = (\beta_0, \beta_1, \ldots, \beta_h, \gamma_1, \gamma_2, \ldots, \gamma_h)'$ is the $n \times 1$ vector of parameters or weights that must be estimated.

The input variables can be included as linear regressors by using direct connections between the inputs and outputs. Modifying (2.6) we have:

$$f(\mathbf{x}_t, \boldsymbol{\theta}) = \mathbf{x}_t'\boldsymbol{\phi} + \beta_0 + \sum_{j=1}^{h} \beta_j G(\tilde{\mathbf{x}}_t'\boldsymbol{\gamma}_j) \tag{2.7}$$

where $\phi = (\phi_1, \phi_2, \ldots, \phi_k)$ and $\theta = (\phi_1, \phi_2 \ldots, \phi_k, \beta_0, \beta_1, \ldots, \beta_h, \gamma_1', \gamma_2', \ldots, \gamma_h')$. If the output activation level is the identity function, as it is in (2.7) above, we have a standard linear regression model augmented by nonlinear terms. The hidden layer neurons in (2.7) can be viewed as latent variables that enrich the linear model (Kuan and White).

From the preceding discussion, it is clear that the neurons in a neural network need not be thought of as mysterious. All neurons in a neural network are mearly "processing elements". Neurons in the input layer serve as "input terminals" to the network for the independent variables. The neurons in the hidden layer(s) are processing elements that take a linear combination of the outputs from the neurons in the previous layer and passes this value through nonlinear activation function. The neurons in the output layer are also processing elements that perform a linear combination of the outputs from the neurons in the last hidden layer. The neuron(s) in the output layer may perform no further processing, as in (2.6), or may apply an activation function such as in (2.5). It can be seen from equations (2.5)-(2.7) above that a feedforward neural network can be considered a nonlinear regression. Standard iterative estimation techniques familiar to econometricians for estimation of nonlinear models can be used to estimate the *n* model parameters in $\boldsymbol{\theta}$. However, as opposed to most nonlinear regression model, neural networks, because of the nonlinear activation functions in the hidden layer, are flexible function forms capable of approximating almost any function. Neural networks are thus said to be universal approximators.

2.4 Neural Networks as Universal Approximators

It has been shown that single hidden layer feedforward neural networks of the type discussed in the previous section and depicted in figure 2.1 are "universal approximators". In other words, given sufficiently many hidden units and properly adjusted model parameters $\boldsymbol{\theta}$, a neural network can approximate an arbitrary mapping arbitrarily well for a large class of functions. The theoretical function approximation capabilities of feedforward neural networks have been explored by Hornik, Stinchcombe, and White (1989), Cybenko, and Carroll and Dickinson. Barron showed that the approximation capabilities of feedforward neural networks require the number of parameters grow linearly. Other function approximation methods, e.g. polynomial, spline, and trigonometric expansions, require that the number of parameters grow exponentially for comparable approximation. The universal approximation properties of neural networks are the key to the demonstrated usefulness of neural networks in many applications as well the potential usefulness of neural networks in economics. With a neural network there is no need to explicitly identify the functional form. Only the variables relevant to the particular problem need be identified.

To be more precise, the universal approximation property means that for any continuous function $g(\mathbf{x},\boldsymbol{\xi})$, every compact subset K of $\Re^k$, and every $\delta > 0$, there exists a neural network $f(\mathbf{x},\boldsymbol{\theta})$, such that

$$\sup_{\mathbf{x}\in K}\left|f(\mathbf{x},\boldsymbol{\theta}) - g(\mathbf{x},\boldsymbol{\xi})\right| < \delta . \tag{2.8}$$

In (2.8), $g(\mathbf{x},\boldsymbol{\xi})$ represents the true (unknown) model that we are trying to approximate. In reality, unless we are modeling a deterministic process with known inputs, what we generally have is $f(\mathbf{x}',\boldsymbol{\theta})$ or $f(\mathbf{x}',\mathbf{z},\boldsymbol{\theta})$ where $\mathbf{x}'\subseteq\mathbf{x}$ and $\mathbf{z}$ is some other vector of inputs that are superfluous to the process we are modeling.

Given a sufficient number of hidden neurons, the approximation capability of a neural network is dependent upon our ability to set the values of the parameters in $\boldsymbol{\theta}$ appropriately. The next section discusses the procedure to set the values for $\boldsymbol{\theta}$. This procedure is referred to as estimation in statistics or economics and learning in the neural network literature.

2.5 Learning (Estimation) in Neural Networks

The objective of training a neural network is to find an optimal set of weights $\boldsymbol{\theta}$ such that some objective or cost function is minimized. The most common objective or cost function is the least squares function. Suppose we are given a set of training data composed of T observations or data pairs $\{(\mathbf{x}'_t,\mathbf{y}'_t)\}_{t=1}^{T}$ where $\mathbf{x}_t$ is a $k\times 1$ vector of explanatory or independent variables and $\mathbf{y}_t$ is a $q\times 1$ vector of dependent variables. Then the training of a neural network $f(\mathbf{x}_t,\boldsymbol{\theta})$ by the least squares objective function is defined by:

$$\min_{\theta\in\Theta} Q(\theta)=\sum_{t=1}^{T}[y_t-f(\mathbf{x}_t,\theta)]^2 \tag{2.9}$$

where y_t is the dependent variable, and Θ is the space of feasible weights or model parameters and $f(\mathbf{x}_t,\boldsymbol{\theta})$ is from say (2.7).

A term that penalizes large weights is sometimes added to the objective function. Addition of this penalty term is referred to as weight decay. Various penalty terms may be used but the most common is the sum of squared weights times a decay constant. This form of weight decay in a linear model is equivalent to ridge regression. See Bishop for other forms of weight decay. Augmenting the cost function in (2.9) with terms that penalize the squared value of large weight values yields:

$$\text{(2.10)} \qquad \min_{\theta\in\Theta} Q(\theta) = \sum_{t=1}^{T}[y_t - f(\mathbf{x}_t,\theta)]^2 + r_\phi \sum_{i=0}^{k}\phi_i^2 + r_\beta \sum_{j=0}^{h}\beta_j^2 + r_\gamma \sum_{j=1}^{h}\sum_{i=0}^{k}\gamma_{ij}^2 ,$$

where r_ϕ, r_β, and r_γ are weight decay constants. The weight decay constants may be set to for example $r_\phi = .01$, and $r_B = r_\gamma = .0001$, as they are in Franses and Dijk. The weight decay penalty term in (2.10) will cause the weights to converge to smaller values then they would under the objective function in (2.9).

Large weights can hurt the generalization performance of a neural network. Excessively large weights leading to hidden neurons can cause "saturation" of those neurons. A saturated hidden neuron will produce outputs at the extremes of its activation function's range for all or most of the observations of training data. For example, if the activation function is the sigmoid function given in (2.2), the neurons output will be near 0 or 1. This causes the outputs from the neurons to be too "rough". Excessively large weights leading to output units can cause outputs beyond the range of the data. In other words, large weights leading to the hidden layers and/or output layers can cause excessive variance of the outputs (Geman, Bienenstock, and Doursat). Statistical theory tells us that a neural network with a large number of weights relative to the number of observations in the training data may have poor generalization performance. Bartlett claims that the size of the weights is more important then the number of weights.

Many algorithms from the field of nonlinear optimization have been applied to minimize (2.9) or (2.10), including gradient or steepest descent, conjugate gradient, Newton, and Quasi-Newton methods (Smagt). The aforementioned optimization algorithms are iterative procedures. Given a function to be optimized, an initial weight vector $\boldsymbol{\theta}^{(0)}$ is chosen. In practice the initial weight vector $\boldsymbol{\theta}^{(0)}$ is usually chosen randomly. For each iteration *i*, a direction $\mathbf{u}^{(i)}$ and a stepsize $\alpha^{(i)}$ are calculated by the optimization algorithm and the weight vector is updated as

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} + \alpha^{(i)}\mathbf{u}^{(i)}. \tag{2.11}$$

Assuming we are minimizing the objective function, the goal of (2.11) is to decrease the value of the objective function with each iteration. Thus the problem of minimization of a function by iterative methods is finding the values for $\mathbf{u}^{(i)}$ and $\alpha^{(i)}$ to accomplish this. This task is made difficult due to the fact that we only have information about the objective function in a small neighborhood around $\boldsymbol{\theta}^{(i)}$.

Derivatives of the objective function with respect to $\boldsymbol{\theta}^{(i)}$ provide information about the behavior of the objective function in a small neighborhood around $\boldsymbol{\theta}^{(i)}$. Rumelhart et al. derived analytical derivatives for neural networks and proposed an algorithm that become known as backpropagation. It is a well known fact from elementary calculus that the value of any function $f(\boldsymbol{\theta})$ decreases quickest in the direction $-\nabla f(\boldsymbol{\theta})$, or in other words, in the negative direction of the gradient. Thus it would seem reasonable to let $\mathbf{u}^{(i)}$ in $(2.11) = -\nabla f(\boldsymbol{\theta})$. Backpropagation utilizes this principle. Backpropagation is similar, and in some cases, equivalent to the traditional gradient descent algorithm. Backpropagation, as it was originally implemented, departs from a true gradient descent by adjusting the weights after the presentation of each observation as opposed to adjusting the weights after presentation of all the observations in a training set.

During the early development of neural networks, and sometimes still, backpropagation was the most commonly used training algorithm and was some times viewed with Mystique. As Kuan and White write

> For a period, artificial neural network models coupled with the method of backpropagation came to be viewed as magic, with considerable accompanying hype and extravagant claims.

Those familiar with nonlinear optimization theory know that gradient descent is inferior to many other algorithms which are available. Gradient Descent is very slow to converge and in addition, if the error surface has "valleys", it can suffer from a condition known as hemstitching (Avriel). Hemstitching is a condition in which the weight changes "bounce" from wall to wall, making little progress down the valley. Hagan, Demuth, and Beale in addition to Sima also discuss of some of the drawbacks of backpropagation for training neural networks. The preferable optimization routines would be Levenberg-Marquardt (Hagan and Menhaj), various Newton and Quasi-Newton methods (Smagt; Battiti), and conjugate gradient methods (Kinsella; Barnard). Despite the efficiency of the aformentioned algorithms in training neural networks, the estimation procedure is still problematic. The optimization algorithm mentioned above are local search algorithms. They proceed downhill to the nearest minimum. The objective function to be minimized when training neural networks contains numerous local minima. Therefore, some users have used so called global search algorithms as an alternative to the more traditional local search algorithms.

CHAPTER 3

GLOBAL OPTIMIZATION

3.1 Introduction

Many optimization solutions to real world modeling problems in areas such as finance, statistics, and engineering design are characterized by multimodal, nonconvex, objective functions. Standard optimization methods may fail to find adequate solutions to these problems since they may only find a local minimum. Because of this, interest and application of global optimization methods has been increasing. Global optimization is concerned with finding the true global minimum of nonlinear functions. The increasing interest in global optimization methods is partly fueled by the rapid increase in computing power available. Researchers have been emboldened to tackle increasingly difficult optimization problems that would have been computationally impractical a few years ago.

Global optimization algorithms can be divided into two broad catagories, stochastic and deterministic. Deterministic algorithms will not be discussed here because of their significant computational requirements for larger optimization problems such as neural network estimation. Instead, stochastic global algorithms will be presented in this chapter. Stochastic global algorithms have been applied to a very wide range of problems, including large-scale optimization problems. Some examples of stochastic search algorithms are simulated annealing and various evolutionary algorithms such as genetic algorithms, evolutionary strategies, and evolutionary programming.

The aforementioned stochastic global optimization algorithms have similarities and differences among themselves with respect to their main operating mechanisms. However, a concept at the core of all stochastic global optimization algorithms is the generation of a stochastic move from the current point. For an iteration i of the algorithm, this move from a point $\boldsymbol{\theta}^{(i)} \in \Re^n$ can be characterized as follows:

$$\widetilde{\boldsymbol{\theta}}^{(i)} = \boldsymbol{\theta}^{(i)} + \mathbf{r}, \tag{3.1}$$

where $\mathbf{r}$ is a random vector drawn from some probability density function $p(\boldsymbol{\theta}^{(i)}, \varphi)$ where φ represents the parameter(s) of the p.d.f., for example the standard deviation. The stochastic or random move is referred to as a mutation in the evolutionary algorithm literature.

The point $\widetilde{\boldsymbol{\theta}}^{(i)}$ generated by a stochastic move such as that in (3.1) is sometimes called a trial point. The point $\boldsymbol{\theta}^{(i)}$ will be called the predecessor point in this discussion. In some algorithms, such as some evolutionary strategies, if $Q(\widetilde{\boldsymbol{\theta}}^{(i)}) < Q(\boldsymbol{\theta}^{(i)})$, then we automatically set $\boldsymbol{\theta}^{(i+1)} = \widetilde{\boldsymbol{\theta}}^{(i)}$ and the algorithm proceeds to the next iteration. In other algorithms, the trial point is accepted, i.e. replaces its predecessor, based on some probability. This latter approach is taken in simulated annealing algorithms, genetic algorithms, and some evolutionary strategy algorithms. Some stochastic search algorithm, such as genetic algorithms, and some evolutionary strategies, generate multiple trial points in parallel. The generation of trial points may look like:

(3.2)
$$\begin{aligned}\tilde{\boldsymbol{\theta}}_1^{(i)} &= \boldsymbol{\theta}_1^{(i)} + \mathbf{r}_1 \\ &\vdots \\ \tilde{\boldsymbol{\theta}}_p^{(i)} &= \boldsymbol{\theta}_p^{(i)} + \mathbf{r}_p,\end{aligned}$$

Where $\mathbf{r}$ is similarly defined as in (3.1) and p is the number of trial points. In the case of (3.2), the acceptance of a trial point for the next iteration of the algorithm may depend upon its performance in relation to its predecessor point as well as all other trial points and/or their predecessor points.

Some algorithms, such as genetic algorithms and evolutionary strategies, generate trial points by combining the "information" in two randomly chosen trial points in addition to mutations such as those in (3.2). This may be performed by randomly picking two integers $i, j \in [1, p]$ where $i \neq j$ and performing the following operation:

(3.3)
$$\begin{aligned}\tilde{\boldsymbol{\theta}}_i^{(k)} &= \boldsymbol{\alpha}_i \boldsymbol{\theta}_i^{(k)} + \boldsymbol{\alpha}_j \boldsymbol{\theta}_j^{(k)} \\ \tilde{\boldsymbol{\theta}}_j^{(k)} &= \boldsymbol{\alpha}_j \boldsymbol{\theta}_i^{(k)} + \boldsymbol{\alpha}_i \boldsymbol{\theta}_j^{(k)},\end{aligned}$$

where $\boldsymbol{\alpha}_i$ and $\boldsymbol{\alpha}_j$ are appropriately chosen $n \times 1$ vectors. In other words, generating two new trial points by taking a linear combination of both predecessor points. Generating trial points in this way is often referred to as crossover or recombinaton. In a genetic algorithm and evolutionary strategy, the crossover operation is normally performed before mutation. Genetic algorithms emphasize crossover operations like that performed in (3.3) and evolutionary strategies emphasize random mutation like that performed in (3.2).

A stochastic mutation like that in (3.1) or (3.2) would move to a new point in the neighborhood of the old point. The magnitude, or probability radius, of such a move would be dependent upon the degree of perturbation, i.e. standard deviation. This perturbation, and for some algorithms, the probability of accepting an uphill move, enables stochastic global algorithms to escape a local minimum. An operation such as that pefomed in (3.3) could move the search to a new region of the solution space. These characteristics, along with the parallel search characteristics of some of the algorithms, allows a wide search range of the solution space. The goal being to find a global minimim point, $\boldsymbol{\theta}_{\min}$, such that

$$\forall_{\boldsymbol{\theta}\in S} : Q(\boldsymbol{\theta}_{\min}) \leq Q(\boldsymbol{\theta}), \tag{3.4}$$

where S is the solution space and $Q()$ is the function being minimized or the objective function. Stochastic global optimization methods in general do not guarantee to find the global minimum given in (3.4). Statistically we can prove that stochastic methods coverge to a global minimum with a probability approaching one as their running time goes to infinity. This is not necessarily an impressive feat since we are interested in algorithms that can be run in a reasonable amount of time. However, in comparison to traditional local optimization methods, they theoretically should increase the likelihood of finding a "good" solution.

References for further aspects of global optimization not addressed in this text are Floudas; Pinter; Horst and Pardalos; Gray et. al., and Neumaier. The following sections review the two most common evolutionary algorithms, namely genetic algorithms and evolutionary strategies, in addition to simulated annealing and a simple stochastic algorithm attributed to Solis and Wets. Section 3.2 covers genetic algorithms. Section 3.3 discusses the other major type of evolutionary algorithm, the evolutionary strategy. Finally simulated annealing and the simple stochastic algorithm of Solis and Wets are covered in sections 3.4 and 3.5 respectively.

3.2 Genetic Algorithms

3.2.1 Introduction

Genetic algorithms were introduced and investigated by John Holland, along with colleagues and students, at the University of Michigan. A book by Holland as well as research by one of his students, De Jong, describe the theory and implementation of their proposed genetic algorithm (GA). The genetic algorithm model introduced by Holland still applies to much of the current theory. In addition, the Simple Genetic Algorithm (SGA) of Holland, still serves as a template for all genetic algorithms. The intent of Holland was not to develop an algorithm for the solution of optimization problems. Holland's goal was to study the process of adaptation in nature and to develop computer models of this natural adaptation. Nonetheless, since their introduction, genetic algorithms have been developed as algorithms to solve optimization problems, and have been applied to a wide variety of problems. Since a biological motivation underlies the original development of genetic algorithms, terms from biology are used to describe their algorithmic operations and mechanisms.

It should be noted that the terminology in the genetic algorithm literature is not always consistent. For example, A real-valued vector, or a vector of bit strings as the case may be, representing a potential solution to the optimization problem, may be referred to as a chromosome, gene, or individual. In addition, the terminology is sometimes logically inconsistent across the literature and even compared to the meaning of the equivalent terms in biology. For example, the term chromosome is sometimes used to refer to a specific term or parameter in a vector representing a solution to a problem, instead of to the whole vector. This is also inconsistent because in biological terms, the genetic content of an individual is distributed over more than one chromosome. This text will endeavor to be more consistent.

A genetic algorithm works from a set of potential solutions to the optimization at hand. In the case of estimating the weights of a neural network, each potential solution represents a set of neural network weights $\boldsymbol{\theta}$. The set of potential solutions is referred to as a population. In the literature, each potential solution is often referred to as a chromosome or individual and a collection of individuals is referred to collectively as a population. The terms chromosome and individual will at times be used interchangeably in this text. Each iteration of a genetic algorithm is referred to as a generation. The individuals in a population are each assigned a value called the fitness value, which measures its goodness with respect to solving the optimization problem. The fitness values are assigned by a fitness function that is in turn a function of the objective function value associated with that individual. Operators are sets of functions or procedures which operate on the population to form the next generation from the current generation with the goal of on average finding a better solution to the problem in each generation. The following summarizes some of the genetic algorithm jargon:

Chromosome or individual – A potential solution to the optimization problem at hand. In the case of this research, each potential solution represents a set of neural network weights $\boldsymbol{\theta}$.

Population – A set of chromosomes (individuals) referred to collectively as the population.

Fitness – Each individual in a population is assigned a value called the fitness, which measures its goodness with respect to solving the optimization problem.

Fitness function – The fitness values are assigned by a fitness function that is in turn a function of the objective function value associated with that individual.

Generation – Each iteration of a GA is referred to as a generation.

Operators: selection, mutation, crossover – Functions or procedures that operate on the current generation of individuals to form the next generation of individuals. The primary operators of genetic algorithms are selection, mutation and crossover operators.

With this terminology in hand, we can sketch out the basic outline of a genetic algorithm:

Step 0: Randomly initialize a population of individuals. Using the fitness function, evaluate the fitness of each individual in the population.

Step 1: Test for termination criterion, e.g. elapsed time or generations, best fitness value, etc.

Step 2: Apply the selection operator to the current population to form an intermediate population of parents for offspring production.

Step 3: Apply the following reproduction operators in turn to the intermediate population:
crossover operator – recombine the "genes" of selected parents,
mutation operator – randomly perturb individuals.

Step 4: Evaluate the fitness of each individual in the intermediate population. Based on the fitness values, select the survivors (offspring) from the intermediate population.

Step 5: Form the population for the next generation by replacement of individuals in the original population with offpsring. Return to Step 1.

It can be seen that the Darwinian process of natural selection or survival of the fittest drives the genetic algorithm. In step 4, the individuals in the intermediate population with higher fitness values have a higher probability of surviving and making it to the next generation.

The operators of genetic algorithms, namely mutation, crossover, and selection, will be explored in more detail in sections to come. First, however, two main defining characteristics of genetic algorithms are covered in the following two sections. Section 3.2.2 examines the choice between a binary or floating-point representation for the individuals in the population and section 3.2.3 examines the choice between a steady-state and generational genetic algorithm.

3.2.2 Binary Versus Floating-Point Representation

A basic design issue when implementing a genetic algorithm is the choice of a binary or floating-point encoding mechanism for chromosomal representation of the parameters of the optimization problem at hand. In a floating-point or real-valued encoding, each chromosome or individual is simply the vector of floating point numbers representing the optimization problem's parameters. For a binary encoding of a real valued optimization problem, a suitable encoding of real-valued vectors $\boldsymbol{\theta} \in \Re^n$ as binary strings $\boldsymbol{v} \in \{0,1\}^l$ is required. Some optimization problems, such as combinatorical optimization, are naturally represented by a binary encoding. Genetic algorithms have been successfully applied to combinatorial problems such as knapsacks (Khuri, Bäck, and Heitkötter 1994a), scheduling (Khuri, Bäck, and Heitkötter 1994b), and graph problems (Khuri and Bäck 94; Bäck and Khuri 94). Genetic algorithms that work on binary strings are sometimes referred to as canonical genetic algorithms.

The various methods for encoding real numbers as binary or bit strings will not be discussed in detail here. However, a simple method for translating between binary and real-valued numbers would be the following. Suppose a continuous variable is defined in a range from –.75 to .75. This variable could be encoded to a given precision or number of decimals places by multiplying the real value by an appropriate integer, say 100, and dropping the decimal portion of the product. Hence, the real numbers would be mapped to integers in the range $[-750,750]$ and the corresponding binary code for each integer can be easily computed. The binary codes of all the variables are then concatenated to obtain a binary string.

It may seem unusual to encode the parameters of a real-valued optimization problem as binary strings. However, as a method to study the process of adaptation in nature, the binary representation of Holland's genetic algorithm was most natural. In addition, fundamental genetic algorithm theory such as the Schema Theorem and the Building Block Hypothesis rely on a binary or bit string implementation (Holland; Goldberg 1989b). Therefore, binary representation of chromosomes has been used historically because of its presumed superiority. The Schema Theorem and Building Block Hypothesis will not be expounded upon in this text. Interested readers are referred to Fogel (1994), Whitely (1994), Michalewicz, or van Rooj, Jain, and Johnson. In spite of the apparent theoretical foundations of binary string representation, researchers have debated their necessity (Vignaux and Michalewicz; Antonisse; Michalewicz). The schema theorem, the theoretical underpinning of genetic algorithms, has been criticized by several researchers (Wright; Whitley). Empirical findings have shown that real-valued encoding has worked well (Syswerda; Wright; Janikow and Michalewicz). Michalewicz reported superior results for a variety of problems using a floating-point representation as compared to a binary representation.

The evidence seems to indicate that as a general rule, real-valued representation should be used when the underlying optimization problem is real-valued. There are many potential reasons for the demonstrated success of real-valued representations. In a binary representation, the reproduction operators operate at the bit level. Large changes in the chromosome can result by changing a single bit in the chromosome. This has the result of reducing the correlation between parents and offspring with respect to their fitness values. A procedure known as gray coding reduces but does not eliminate this problem (Eshelman and Schaffer). Related to the correlation between parents and offspring is the fact that a genetic algorithm with a floating point representation is closer to the problem space. Two potential solutions that are close to each other in the representation space are also close in the problem space. In a binary representation, the distance would be defined by the number of different bit positions. Researchers report other negative consequences, such as hamming cliffs, from bit mutation (Janikow and Michalewicz; Rooij, Jain, and Johnson).

There is invariably a loss of precision when converting from a floating-point to binary representation. If too few bits are used to encode the weights, some combinations of real-valued parameters may be impossible or difficult to achieve (Goldberg 1991). On the other hand, an increase in precision increases the size of the individuals or chromosomes. Michalewicz claims that genetic algorithms can be inefficient at manipulating bitstrings with thousands of bits. Others claim that binary encoding does not scale up well (Whitley, Starkweather, and Bogart). Increasing the size of the chromosomes also decreases the computational efficiency of the algorithm. Each application of the fitness function to a chromosome requires an evaluation of the objective function being optimized. This in turn necessitates converting the bit string to floating-point numbers.

3.2.3 General Types of Genetic Algorithms

Genetic algorithms can be categorized by the survival policy used in the procedure for replacing individuals in each generation. Recall from the outline of the genetic algorithm presented in section 3.2.1 that in step 5, the population of the next generation is formed by replacing individuals in the current population with survivors or offspring. The two most common approaches to replacement in a population are generational and steady-state. The traditional approach is the generational scheme but the steady-state approach is increasingly popular.

In a generational genetic algorithm, the new population (offspring) entirely replaces the original population in each generation. The steady-state type of genetic algorithm replaces only a portion of the population, permitting offspring to compete directly with parents in the next generation, or in some approaches in the current generation. Generally, a steady-state genetic algorithm creates only a small number of offspring in each generation to replace the worst performing individuals in the population. In some implementations, there can be a competition for survival between the offspring and current population. One disadvantage of the steady-state scheme is that because of the extreme selection pressure, they can quickly lose the diversity in their population of individuals. In other words, each potential solution is very similar to most others, thereby negating one of genetic algorithms inherent benefits, namely parallel search. This causes premature convergence of the algorithm, which begins performing a simple stochastic hill-climbing search in which new potential solutions are similar to the old solutions.

3.2.4 Fitness Evaluation

In each generation or iteration of a genetic algorithm, the fitness of each member of the population evaluated. Fitness evaluation measures the relative performance of the chromosomes in the population. This is done by use of a fitness function that yields a single real-valued parameter that reflects the individuals success at solving the problem at hand. The fitness function could be the same as the objective function, i.e. the function we are trying to optimize, however, in most cases, the fitness function is a function of the objective function. For example, in this research, we are trying to minimize the sum-of-squared error (SSE), our objective function. The objective function values cannot be used directly because a lower (higher) SSE indicates a better (worse) solution, and hence a higher (lower) fitness value. Therefore, the fitness function has to perform some sort of inversion operation. Also, it is sometimes convenient to normalize the fitness values to a range of 0 to 1. Some authors emphasize the potential disconnect between these two functions and use the term fitness and evaluation separately (Whitley).

3.2.5 Selection Operators

Individuals are selected from the current generation by selection operators which form an intermediate population on which breeding and mutation are to take place. Based upon natures "survival of the fittest" mechanism, individuals with a higher fitness value are more likely to be selected. Those individuals that are selected are said to be parents of the next generation. There are two important factors to consider when applying a particular selection operator, namely, population diversity and selective pressure (Michalewicz). Selection pressure refers to the ability of the operator to select those individuals with higher fitness values. Population diversity is the degree to which the individuals in the population differ from each other.

An effective selection operator should exert sufficient selective pressure so as to avoid stagnation in the evolutionary process (Goldberg, 1989a). This can be seen more readily in the later stages of a genetic algorithms search where the diversity of the chromosomes is low. Low selective pressure in this situation could easily lead to stagnation because of the low variance in the fitness values. On the other hand, strong selective pressure can lead to premature convergence of the genetic algorithm search. Strong selective pressure could lead the search to focus on a few good individuals in the population early in the genetic algorithms search. Therefore, genetic or population diversity would be lost preventing an adequate exploration of the search space. It is important for the selection operator to balance the two opposing factors.

Many selection strategies are available. Some of the commonly used are tournament, roulette wheel, and proportionate selection. Some examples of selection strategies are discussed in more detail in the following sections.

Roulette wheel selection

In roulette wheel selection, individuals are selected with a probability proportional to their relative fitness values. The probability that a particular chromosome $\mathbf{x}$ is chosen is given by

$$p_{\text{select}}(\mathbf{x}) = \frac{f(\mathbf{x})}{\sum f} \tag{3.5}$$

where $f()$ is the fitness function. The roulette wheel selection method is a proportionate selection method. This method can be described by visualizing a roulette wheel where each chromosome or individual occupies an area that is relative to its fitness value. A fixed marker selects a particular chromosome when the wheel stops. By repeatedly spinning this roulette wheel the intermediate population is formed. Obviously, a chromosome with a higher fitness value will occupy a larger proportion of the roulette wheel and hence have a higher probability of being chosen.

Integral selection

Integral selection is a modification of roulette wheel selection. In roulette wheel selection the expected number of times that a chromosome $\mathbf{x}$ would be selected is given by

$$E_{\text{select}}(\mathbf{x}) = n \cdot p_{\text{select}}(\mathbf{x}), \tag{3.6}$$

where n is the population size. The number of offspring allocated to a chromosome may differ significantly from the expected number. Integral selection seeks to reduce the role of chance by guaranteeing that each chromosome is selected as many times as its corresponding expected value in (3.6). Since this method will most of the time lead to the allocation of fractional numbers of a individual chromosomes, the actual number selected must be rounded up or down. The rounding method includes some randomization to avoid biases toward a particular chromosome.

Rank selection

Rank selection is a modification of roulette wheel selection where the fitness values are used to rank the chromosomes. The probability of selection is proportional to the rank rather than the raw fitness values. One possible ranking scheme is linear ranking. The individual with the lowest fitness value are assigned a rank of 0, the next worse a rank of 1, and so on. The individuals are then selected based upon some linear function of its sorted rank. The linear function ensures that there is always a fixed ratio between the best and worst chromosomes in a population. The other individuals will be linearly spaced between the two. We can assign to the individual at rank i a probability of selection given by

$$p_i = \frac{1}{n}(2 - c + (2c-2)\frac{i-1}{n-1}), \tag{3.7}$$

where n is the size of the population and $1 \leq c \leq 2$ is the selection bias. The higher the value of c the higher the selection pressure. That is, the more the algorithm will favour the better chromosomes. As a genetic algorithm progresses there is smaller and smaller variance in the fitness values across the population. The rank selection method ensures that even after the performance of the individual chromosomes in the population converge, the best chromosome will be favoured over the worst to the same extent they were in the beginning. This method helps to avoid premature convergence and stagnation. One computational drawback of this method is that it requires sorting of the entire population at each generation. Tournament selection can be used to avoid this problem.

Tournament selection

In tournament selection, a typically small number, m, of chromosomes is randomly chosen from the population. The selection is independent and with replacement so an individual could be chosen more than once. The best or fittest chromosome is chosen from this pool of individuals to be passed on to the intermediate population. The size of the pool, i.e. m, is called the tournament size. The higher the value of m, the more selection pressure the operator will exert. Conversely, if $m = 1$, then the operator picks randomly. In the genetic algorithm literature, a value of $m = 2$ is not very selective and $m = 7$ is considered relatively highly selective.

3.2.6 Crossover (recombination)

The crossover operator is the distinguishing operator of genetic algorithms (Davis). Crossover has traditionally been viewed as the main search operator with mutation being only a background operator (Holland). Crossover is the process by which genetic material from different individuals is combined to create offspring. This is the so-called mating or breeding portion of the genetic algorithm. Pairs of strings or chromosomes are picked at random from the population to serve as parents. These parents are subjected to crossover to form offspring. The theory underlying crossovers predominant role in the success of a GA is the building-block hypothesis first introduced by Goldberg (1989b). The building-block hypothesis sais that the "building blocks" are subparts of individuals that are considered good. As evolution progresses, through crossover, these building blocks can be transferred from individual to individual spreading throughout the population.

As an example of how crossover operators work, some of the more common are illustrated and/or discussed below. Assume we start with the following two chromosomes:

$$s = (s_1, \ldots, s_n)$$
$$v = (v_1, \ldots, v_n)$$

The following crossover operators will operate on the above chromosomes in the following manner where $k, l \in (1, \ldots, n)$ are random numbers:

a) One-point crossover

$$s' = (s_1, \ldots, s_{k-1}, s_k, v_{k+1}, \ldots, v_n)$$
$$v' = (v_1, \ldots, v_{k-1}, v_k, s_{k+1}, \ldots, s_n)$$

b) Two-point crossover

$$s' = (s_1, \ldots, s_{k-1}, s_k, v_{k+1}, \ldots, v_{l-1}, v_l, s_{l+1}, \ldots, s_n)$$
$$v' = (v_1, \ldots, v_{k-1,}, v_k, s_{k+1}, \ldots, s_{l-1}, s_l, v_{l+1}, \ldots, v_n)$$

c) Linear interpolation one-point crossover

$$s' = \left(\frac{2s_1 + v_1}{3}, \ldots, \frac{2s_{k-1} + v_{k-1}}{3}, \frac{2s_k + v_k}{3}, \frac{s_{k+1} + 2v_{k+1}}{3}, \ldots, \frac{s_n + 2v_n}{3} \right)$$
$$v' = \left(\frac{s_1 + 2v_1}{3}, \ldots, \frac{s_{k-1} + 2v_{k-1}}{3}, \frac{s_k + 2v_k}{3}, \frac{2s_{k+1} + v_{k+1}}{3}, \ldots, \frac{2s_n + v_n}{3} \right).$$

The simplest operator and the one employed in Holland's SGA is the one-point crossover illustrated in a) above. Assuming the string or chromosome is of length *n*, a crossover point is randomly chosen in the range 1 to *n*-1. The portions of the strings that lie beyond the crossover point are exchanged between the two strings. Similar in concept is the two-point crossover in which there are two potential crossover points. This operator is illustrated in b) above. In general, an *m*-point crossover scheme can be used where $m<n$. In uniform crossover, each point between genes is a potential crossover point. Each of these potential points has a probability of .5 that it will be a crossover point. Note that as discussed in previous sections, in some traditional implementations of a genetic algorithm, the individual elements in the chromosomes s and v would be binary numbers. For our purposes in this research, we are only concerned with real-valued chromosomes. Real-valued strings offer the potential for many other crossover operators. For example, as illustrated in c) above, the crossover operator could employ a linear combination of the genes. Regardless of the crossover operator used, genetic algorithms typically employ a crossover rate p_c. The crossover operator is employed only if $p_c > r$ for a random number $r \in [0,1]$. The crossover rate, p_c, is typically set close to 1. If the crossover operator is not employed then the strings remain unaltered.

A disadvantage of one-point crossover is that given two individuals, some combinations of their building blocks can not be achieved in the offspring. Two-point crossover is a more flexible operator because it has two segments, one in each parent, that

can be swapped. An extreme case is uniform crossover in which each point in both parents is subject to crossover with a probability of .5. Uniform crossover is very flexible, and any combination of individuals in the chromosomes can be achieved. On the other hand, it is the most disruptive to the building blocks.
Many other crossover operators have been investigated, including linear and non-linear representations. See Booker, Fogel, Whitley, and Angeline and Michalewicz for further discussion of the subject.

3.2.7 Mutation

After crossover, each string is subject to mutation. Mutation is useful for introducing new genetic material and keeping the genetic diversity in the population (Bäck, Fogel, Whitley, and Angeline). For binary strings, mutation operates independently on each bit of the string. Mutation is simply a matter of flipping a bit, for example from 0 to 1. For real-valued genes, mutation is usually accomplished with the addition of a normally distributed variable, i.e. Gaussian noise, to the values of the parameters in the chromosome. For real-valued chromosomes, other mutation options are available. For example, inversion of the genes or distributions other than normal could be used for additions to the values in the chromosome.

In most genetic algorithm implementations, each of the individuals in the population is subject to mutation with a probability p_m. This value is normally close to zero, however, some have argued that for real-valued chromosomes p_m can be quite high, for example .5 (mutate 50% of the chromosomes)(Rooij, Jain, Johnson). Such a high rate of mutation for a binary representation would be very disruptive. When a bit is mutated, it is switched to its opposite state. This could have a large effect on the chromosome as a whole. However, for a floating-point representation, assuming the variance of the random number addition is not too large, the mutation is much less disruptive. Therefore, a higher rate of mutation can be justified.

3.3 Evolutionary Strategies

3.3.1 Introduction

Evolutionary strategies were born out of an attempt to solve an engineering optimization problem, namely, the optimal shapes of bodies in a flow. To solve this problem, Schwefel and Rechenberg collaborated in the 1960's to develop the evolutionary strategy (ES) approach to function optimization. Evolutionary strategies are designed to optimize functions of continuous variables (Michalewicz). However, Evolutionary strategies have also been extended to discrete problems (Bäck, Hoffmeister, and Schwefel; Herdy). Similar to genetic algorithms, modern evolutionary strategies operate on a population of potential solutions. However, in contrast to genetic algorithms, mutation is the primary operator and crossover is only a background operator. Much of the terminology for genetic algorithms introduced in section 3.2 is also used to described evolutionary strategies. For example, a potential solution to the optimization problem at hand is referred to as an individual. A set of individuals is referred to collectively as a population. In addition, many of the same genetic operators such crossover, mutation, and selection are used in evolutionary strategies. However, their fundamental methods of operation are different then the corresponding operators for genetic algorithms.

The following notation is commonly used to describe particular forms of evolutionary strategy algorithms:

1. $(1+1)-ES$,
2. $(\mu+1)-ES$,
3. $(\mu+\lambda)-ES$,
4. $(\mu,\lambda)-ES$.

The enumiration above also represents the historical development of evolutonary strategy algorithms with the $(1+1)-ES$ being the earliest and simplest ES and the $(\mu+\lambda)-ES$ and $(\mu,\lambda)-ES$ representing the latest and most sophisticated implementations. The symbol μ denotes the number of parents or individuals in the population and the symbol λ denotes the number of offspring created by the parents within a generation. The notation representing a particular strategy also serves to characterize the selection operator that is used to select individuals from the population of potential solutions. For example, in the $(\mu+\lambda)-ES$, the best μ individuals out of the union of parents and offspring survive. In $(\mu,\lambda)-ES$ the next generation is formed by selecting the best μ individuals from a population of potential solutions of size λ ($\lambda > \mu$ is necessary). Each of the four ESs are described in detail in the following sections.

3.3.2 (1+1) and (μ+1) – Evolutionary Strategies

The earliest developed evolutionary strategy was the $(1+1)-ES$. The $(1+1)-ES$ strategy is based on a population of only one individual and mutation is the only genetic operator. The $n\times 1$ vector of optimization variables $\mathbf{x}$ are mutated according to

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} + N(0,\sigma^2) \tag{3.8}$$

where t is the generation, $N(0,\boldsymbol{\sigma}^2)$ is a normally distributed random vector of size $n \times 1$ with mean zero and standard deviation given by the $n \times 1$ vector $\boldsymbol{\sigma}$. In nature, small mutations occur more often than larger ones, therefore, the choice of perturbations from a normal random variable in (3.8) is somewhat intuitively appealing. Assuming we are minimizing a function, an offsping $\mathbf{x}^{(t+1)}$ replaces its parent $\mathbf{x}^{(t)}$ iff $f(\mathbf{x}^{(t+1)}) < f(\mathbf{x}^{(t)})$, otherwise $\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)}$ and the algorithm proceeds with another Gaussian mutation of $\mathbf{x}$.

The $(1+1)-ES$ is called a "two-membered evolution strategy" because the offspring competes with its parent to make it to the next generation and at least temporarily, there are two individuals in the population. The algorithm above is mearly a random search algorithm recast with some evolutionary terminology. Nonetheless, it was a start toward the more sophisticated evolutionary strategies. In spite of the simplicity of the $(1+1)-ES$, this type of algorithm has been shown to converge to the global optimum with probability one for sufficiently long search time (Michalewicz). However, convergence with probability one sais nothing about a particular algorithms potential use as a practical optimization algorithm, especially with respect to convergence rate. To improve the convergence rate of the $(1+1)-ES$, Rechenberg proposed a "1/5 success rule". That is, the ratio φ of successful mutations to all mutations should be 1/5. If φ is greater than 1/5 the variance is increased, otherwise it is decreased. Rechenberg derived this somewhat ad-hoc rule on the basis of optimizing the convergence rates on two particular functions. The rule is intuitive in the sense that if there is a large percentage of successes, then larger steps should be taken to explore a wider region of the search space. Alternatively if there is a small percentage of successes, the search should be focused on a smaller region.

The search described above could lead to premature convergence on some types of functions (Michalewicz). An increased population size was proposed to address this problem. Rechenberg proposed a multimembered evolutionary strategy with the $(\mu+1)-ES$ algorithm where μ=population size. A multimembered evolutionary strategy uses a crossover or recombination operator to combine $\mu>1$ individuals to form one offspring. This is unlike a typical genetic algorithm which generally has a fixed population size from generation to generation or at the very least produces more than one offsping in each generation. The $(\mu+1)-ES$ strategy was never widely used, however, it provided a transition to the $(\mu+\lambda)-ES$ and $(\mu,\lambda)-ES$ introduced by Schwefel (1977, 1981). These two strategies, and especially the latter one, are more state-of-the-art than their predicessors. The next section discusses these particular evolutionary strategeis.

3.3.3 (μ+λ) and (μ,λ) – Evolutionary Strategies

Like the $(\mu+1)-ES$, $(\mu+\lambda)-ES$ and $(\mu,\lambda)-ES$ introduce the possibility of a crossover or recombination operator for a multimembered population. The $(\mu+\lambda)-ES$ and $(\mu,\lambda)-ES$ algorithms are distinguished from each other by their selection mechanism. In the former, μ parents create $\lambda \geq 1$ offspring by recombination and mutation. The selection mechanism picks the μ best individuals out of the union of the parents and offspring to form the next generation. The later algorithm creates λ offspring, where $\lambda > \mu$, by recombination and mutaton. The best μ individuals out of these λ offspring are selected for the next generation. Both of these algorithms use self-adaption of the mutation variances. The self-adaption is an improvement over an ad-hoc criterion, such as the 1/5 success rule. Each individual in the population is composed of the $n \times 1$ parameter vector $\mathbf{x}$ plus up to $n(n+1)/2$ variances and covariances for an n-dimensional normal distribution used to generate perturbations to the $\mathbf{x}$ vector. In the evolutionary strategy literature, the variances and covariances are called strategy variables and the individual components of the parameter vector $\mathbf{x}$ are called object variables.

The general functioning of the $(\mu+\lambda)-ES$ and $(\mu,\lambda)-ES$ algorithms can be sketched out as follows:

Step 0: Randomly initialize a population of size μ. Evaluate the objective function value for each of the individuals in the population.

Step 1: Test for termination criterion, e.g. elapsed time or generations, best objective function value, etc.

Step 2: Apply the following reproduction operators in turn to the population to produce λ offspring:
crossover operator – recombine the "genes" of selected parents,
mutation operator – randomly perturb individuals.

Step 3: Evaluate the objective function value for each of the individuals in the population.

Step 5: Form a population for the next generation by selecting the best μ individuals from:
for $(\mu+\lambda)-ES$: the parents μ plus the offspring λ,
for $(\mu,\lambda)-ES$: the offspring λ where $\lambda > \mu$.

Step 5: Return to Step 1.

The specifics of each of the steps above, i.e. initialization, crossover, mutation, and selection, are presented in the paragraphs below. The discussion and notation in the following sections borrows heavily from Bäck and Schwefel (1993), Bäck and Schwefel (1996), and Bäck, Rudolph, and Schwefel (1993). The notation commonly used in the evolutionary strategy literature will be introduced in the section on mutation.

Initialization

Initialization of the object variables can be handled in the same manner as you might for a regular nonlinear optimization algorithm. For the standard deviations, Schwefel (1981) recommended $\sigma_i(0) \approx \Delta x_i / \sqrt{n}$ where Δx_i is the estimated distance between the starting point and optimum. As a practical matter, we might not have an idea what Δx_i should be. For the neural network problem, care should be taken not to make the initial values of the standard deviation too large, otherwise, this will tend to saturate the values of the hidden neurons. If we have previously run a gradient based algorithm, we could use the difference between the mean of these weight values and the expected mean of the random initial weights as an estimate of Δx_i should be. The self-adaption mechanism can scale the standard deviations into a more appropriate range if the initial values are not too large. It should be noted that constraint handling can be included in the algorithm but is beyond the scope of this discussion.

Mutation Operator

The mutation is guided by an n-dimensional normal distribution having a probability density function

(3.9) $$p(\mathbf{x}) = \frac{\exp(-\frac{1}{2}\mathbf{x}'\mathbf{C}^{-1}\mathbf{x})}{\sqrt{(2\pi)^n \det \mathbf{C}}}$$

where $\mathbf{x}' = (x_1, \ldots, x_n)$ is a vector of the choice variables, $\mathbf{C}$ is the covariance matrix for $\mathbf{x}$, and $\det \mathbf{C}$ represents the determinant of the covariance matrix. The choice variables $\mathbf{x}$ are called object variables in the evolutionary strategy literature. The variances and covariances also evolve in modern ESs and are thus subject to mutation in addition to recombination. The variances and covariances are known as strategy variables in the ES literature. The strategy variables are composed of up to n different variances, $\boldsymbol{\sigma} = (c_{ii} = \sigma_i^2,\ \ i \in \{1,...,n\})$, and $n(n-1)/2$ covariances, $\boldsymbol{\alpha} = (c_{ij},\ \ i \in \{1,...,n-1\}, j \in \{i+1,...,n\})$. Thus, we have $w = n(n+1)/2$ strategy variables representing the variances and covariances (1/2 the off-diagonal terms from $\mathbf{C}$) that can be varied during the operation of the algorithm. An individual $\mathbf{a}$ in the population therefore consists of up to 3 components, $\mathbf{a} = (\mathbf{x}, \boldsymbol{\sigma}, \boldsymbol{\alpha})$. To ensure positive-definiteness of the covariance matrix, or equivalently to ensure that the coordinate system remains orthogonal, rotation angles $\alpha_j, 0 \le \alpha_j \le 2\pi$ are used in place of the covariance coefficients c_{ij}. The rotation angles are related to the covariances by the following

$$\tan(2\alpha_{ij}) = \frac{2c_{ij}}{\sigma_i^2 - \sigma_j^2} \tag{3.10}$$

For more detail on algorithmic implementation of the above and further discussion of the reason for using rotation angles see Bäck (1996).

With the above information we can proceed with mutation of the strategy and object variables. The following are the main possibilities for mutation:

1. $n_\sigma = 1, n_\alpha = 0$: All object variables have identical standard deviation σ, and the covariances are zero.

$$\sigma' = \sigma \exp(\tau_0 N(0,1))$$
$$x_i' = x_i + \sigma' N(0,1)$$

2. $n_\sigma = n, n_\alpha = 0$: All object variables have their own standard deviation σ_i, with covariances of zero.

$$\sigma_i' = \sigma_i \exp(\tau' N(0,1) + \tau N_i(0,1))$$
$$x_i' = x_i + \sigma_i' N(0,1)$$

3. $n_\sigma = n, n_\alpha = n*(n-1)/2$: All object variables have their own standard deviation σ_i, with non-zero covariances.

$$\begin{aligned}
\sigma_i' &= \sigma_i \exp(\tau' N(0,1) + \tau N_i(0,1)) \\
\alpha_j' &= \alpha_j + \beta N_j(0,1) \\
\mathbf{x}' &= \mathbf{x} + N(0,\mathbf{C})
\end{aligned}$$

It is suggested that the constants τ, τ', and β be set according to (see Bäck, 1996, pg. 72):

$$\tau_0 \propto (2\sqrt{n})^{-1/2}$$

$$\tau' \propto (2n)^{-1/2}$$

$$\beta \approx 0.0873\,(5^\circ)\,.$$

It is suggested that the algorithm is robust with respect to the values of these parameters (Bäck, Rudolph, and Schwefel), however, the specific optimal values undoubtedly depend upon the particular topological characteristics of the objective function (Bäck and Schwefel, 1993). The factors τ and τ' are sometimes referred to as "learning rates", similar in concept to the learning rate for backpropagation in neural networks or the step size in many nonlinear optimization algorithms.

Crossover or Recombination

In $(\mu+\lambda)-ES$ and $(\mu,\lambda)-ES$, the object variables $\mathbf{x}$ as well as the strategy variables σ and α are subject to recombination. In addition, the recombination operator may be different for the object variables, standard deviations, and rotation angles. Various recombination operators are used in evolution strategies. A single offspring may be produced by using information from two parents chosen from the population or the creation of the individual may involve up to all parent individuals, depending upon the operator used.

Traditionally, two different general forms of recombination operator have been used for evolution strategies: discrete recombination and intermediate recombination. Three different versions of these operators exist giving 6 different possibilities for operators. Notationally, for a specific element x_i from the object variable vector $\mathbf{x}$ we have the following possibilities:

$$x_i' = \begin{cases} x_{s,i} \text{ or } x_{t,i} & \text{discrete} & r_d' \\ x_{s,i} \text{ or } x_{t_j,i} & \text{panmictic discrete} & r_D' \\ x_{s,i} + (x_{t,i} - x_{s,i})/2 & \text{intermediate} & r_i' \\ x_{s,i} + (x_{t_j,i} - x_{s,i})/2 & \text{panmictic intermediate} & r_I' \\ x_{s,i} + \chi * (x_{t,i} - x_{s,i})/2 & \text{generalized intermediate} & r_g' \\ x_{s,i} + \chi * (x_{t_j,i} - x_{s,i})/2 & \text{panmictic generalized intermediate} & r_G' \end{cases}$$

where $i = 1,...,n;\ j,s,t \in \{1,...,\mu\}$ and $\chi \in [0,1]$ is a uniform random variable. In the panmictic generalized intermediate form, χ_i denotes that the random variable is resampled for each possible value of *i*, or in other words for each component of the new individual $\mathbf{x}'$. The indices *s* and *t* denote two separate parents selected at random from the population P and the index *j* in t_j indicates *t* to be sampled anew for each value of *i*.

Selection Operator

The selection mechanism is what distinguishes the $(\mu + \lambda) - ES$ from $(\mu, \lambda) - ES$. The notation in fact characterizes the type of selection operator used for the respective strategies. To be more precise, if *s* is the selction operator, the respective operators perform the following operations or mappings:

$$\begin{array}{ll} (\mu + \lambda) - \text{selection} & s_{(\mu+\lambda)} : I^{\mu+\lambda} \rightarrow I^{\mu} \\ (\mu, \lambda) - \text{selection} & s_{(\mu,\lambda)} : I^{\lambda} \rightarrow I^{\mu} \end{array}$$

The latter selects the μ best individuals out of the offspring only while the former selects the μ best individuals out of the parents plus the offspring. It should be obvious that for the $(\mu,\lambda)-ES$, $\lambda > \mu$, otherwise, no selection takes place. The $s_{(\mu+\lambda)}$ selection operator is an elitist scheme where the best individuals are guaranteed to survive. The $s_{(\mu,\lambda)}$ operator on the other hand, restricts each individual to a single generation. This would seem to be a disadvantage, however, the ability to loose good solutions actually allows the algorithms to escape local minima. This operator, however, also facilitates the extinction of bad individuals. Bäck (1996) recommends intermediate recombination for the strategy parameters. The ratio μ/λ drives the character of the evolutionary search. If we decrease μ, the algorithm will be more path-oriented and converge quicker while increasing μ leads to a wider or more global search of the parameter space. A suggested value for these parameters is $\mu = 15$ and $\mu/\lambda \approx 1/7$ (Bäck, 1996).

3.4 Simulated Annealing

3.4.1 Introduction

Simulated annealing as an algorithm originated from an analogy between the process of slowly cooling a solid to reach a low energy ground state or thermal equilibrium and minimizing the cost function of a combinatorial (discrete) optimization problem (Kirkpatrick, Gelatt, and Vecchi). The physical annealing process contains the following two steps (Aarts and Korst)

- Increase the temperature of the heat bath to a maximum value at which the solid melts.
- Decrease carefully the temperature of the heat bath until the particles arrange themselves in the ground state of the solid.

In the liquid phase, the particles move about freely and are arranged randomly. As the liquid cools this mobility is lost. If it is cooled slowly enough, the particles will align themselves to each other to form an ordered crystalline structure that is the minimum energy state for the system. On the other hand, if the liquid is cooled to fast, it will end up in a polycrystalline state having a higher energy than the minimum energy state.

The physical annealing process can be modeled with computer simulations (Binder). Metropolis, et al. first proposed a simple algorithm to simulate the annealing of a solid to thermal equilibrium. Their algorithm, known as the Metropolis algorithm, generates a sequence of energy states for the system. Let E_i be the current energy state for the system. A subsequent energy state E_j can be generated by applying a small perturbation to the system, such as a random displacement of an atom. If the energy has decrease, that is $E_j - E_i \leq 0$, then the state *j* is accepted and the algorithm proceeds to the next iteration with a perturbation to the system in state *j*. If $E_j - E_i > 0$ then state *j* is accepted with a probability given by

$$\exp\left(\frac{E_i - E_j}{k_B T}\right) \tag{3.11}$$

where k_B is a physical constant known as the Boltzmann constant and *T* is the temperature of the heat bath. The acceptance rule in (3.11) is known as the Metropolis criterion. If state *j* is not accepted then the algorithm starts over beginning with a new perturbation to the system in state *i*. By repeating the previous steps many times, the metropolis algorithm simulates the thermal motion of atoms in thermal contact with a heat bath at temperature *T*. If the temperature is lowered sufficiently slow, the solid can reach thermal equilibrium at each temperature. Thermal equilibrium in a system at a temperature T is given by the Boltzmann probability distribution

(3.12) $$\text{Prob}(E) \sim \exp(-E / kT).$$

Kirkpatrick, Gelatt, and Vecchi recognized the analogies between the evolution of a solid at a given temperature and the solution of a combinatorial optimization problem.[1] They applied the metropolis algorithm to function optimization by substituting the unknown parameters for the particles of the solid and the associated cost or objective function value for the energy of the system. In this "simulated" annealing algorithm, the parameters of the function to be optimized are randomly perturbed to create a new set of parameters. These parameters are accepted if they result in a lower value for the cost function (assuming minimization), otherwise, if the result is an increase in the cost function, the new parameters are accepted or rejected based upon a probabilistic acceptance criterion such as the metropolis criterion in (3.11).

Temperature plays the role of a control parameter in simulated annealing. The temperature often controls the amount of perturbation to the parameters being optimized. In addition, the acceptance criterion is a function of the temperature. At lower temperature, there is a decreasing probability that a set of parameters which results in an increased cost function, will be accepted. The temperature, which is a control parameter, is initially set to a high level. The high temperature, which results in large perturbations, allows the algorithm to perform a wide search of the parameter space. The high temperature also leads to an acceptance of a higher percentage of steps or perturbations

[1] Kirkpatrick, Gelatt and Vecchi are generally credited with the development of a optimization algorithm from statistical mechanics concepts. However, Laarhoven and Aarts report that Cerny, along with the earlier works of Pincus as well as Khachaturyan, Semenovskaya, and Vainshtein, also recognized the analogies between statistical mechanics concepts and optimization.

that result in an increase in the cost function. This also allows for a wider search of the parameter space. The acceptance of parameters with a higher cost also allows the algorithm to escape local minima.

The occasional acceptance of an uphill step by the acceptance criterion, as well as a sufficiently high beginning temperature level and its subsequent lowering, are the keys to an affective "global" simulated annealing algorithm. The converse of the process of annealing is quenching in which the temperature is rapidly lowered. In the physical annealing process, quenching is very likely to result in a freezing of the particles of the solid into suboptimal structure. Similar to quenching, local optimization routines move rapidly downhill toward the nearest minimum. A local optimization routine is greedy in the sense that it always seeks to take a step downhill. By executing the simulated annealing algorithm at a sequence of slowly decreasing temperature values, and allowing uphill moves with a non-zero but gradually decreasing probability, simulated annealing is allowed to explore the parameter space but eventually settle into what is hopefully the global minimum.

Examples of early applications of SA were to designing integrated circuts (Vecchi and Kirkpatrick), pollution control (Derwent), and the famous traveling salesman problem (Aarts and Korst). It was reported to perform well in the presence of a high number of variables (Kirkpatrick, Gelatt, and Vecchi; White, 1984). Vanderbilt and Louie described the first application of a simulated annealing algorithm to optimization of a function with continuous parameters (Boender and Romeign). Corona, Marchesi, Martini, and Ridella also presented a simulated annealing algorithm for functions of continuous variables. Corona et al. as well as Goffe and Ferrier compared this algorithm to conventional local optimization algorithms for minimizing several test functions and found the algorithm to be very reliable at finding the global minimum.

There are many different simulated annealing algorithms, however, the following three functions characterize all simulated annealing algorithms (Ingber, 1989). Let $\mathbf{x} \in \Re^n$ be a vector of continuous variables:

1) $p(\mathbf{x})$: The probability density function (p.d.f.) of the distribution to perturb or generate the parameters, e.g. the gaussian or uniform distributions. The degree of dispersion is usually controlled by the temperature T, in which case we can write $p(\mathbf{x},T)$. For example, assuming a normal distribution, the standard deviation could be a function of T.

2) $g(\Delta C,T)$: The p.d.f. for accepting a new set of parameters after perturbation, e.g. the Metropolis criterion as given in (3.11). The temperature is given by T and ΔC is the change in the value of the cost function from the perturbaton is a decreasing function of the temperature.

3) $h(k)$: The function which controls the cooling schedule for the temperature T. The function slowly decreases the temperature T as k, the number of iterations of the algorithm, increases. A simple example would be $T_k = T_{k-1} \times p$ where $p \in [0,1]$.

Using the three functions above, the following is an outline of a simulated annealing algorithm for minimization of a cost or objective function $C(\mathbf{x})$:

step 1: Pick an initial temperature $T_0 > 0$ and point $\mathbf{x}_0$ in the parameter space and calculate the corresponding function value $C(\mathbf{x}_0)$.

step 2: Randomly pick a new point $\mathbf{x}'_k = \mathbf{x}_k + \Delta\mathbf{x}$ in the parameter space using the p.d.f. $p(\Delta\mathbf{x}_k)$ and calculate the corresponding function value $C(\mathbf{x}'_k)$.

step 3: If $C(\mathbf{x}'_k) < C(\mathbf{x}_k)$ then set $\mathbf{x}_{k+1} = \mathbf{x}'_k$, otherwise generate a uniform random number $r \in [0,1]$ and decide to accept the inferior $\mathbf{x}'$ according to the probabilistic criterion $r \le g(\Delta C, T_k)$, where $g() \in [0,1]$ and $\Delta C = C(\mathbf{x}') - C(\mathbf{x})$.

step 4: After M points or perturbations have been considered, i.e. repeating steps 2-3 M times, reduce the temperature by $T_{k+1} = h(k) \quad T_{new} = T_{old} \times p$, where $p \in [0,1]$. Repeat steps 2-4 until the stopping criterion has been reached.

Note that this is only a rough outline of an annealing algorithm. There could be many suttle variations. The following two sections discusses the two main variations of simulated annealing, namely, Boltzmann annealing and Fast Simulated Annealing.

3.4.2 Boltzmann Annealing

Boltzmann annealing (BA), sometimes referred to as classic simulated annealing (CSA) (Szu and Hartley), is based upon the metropolis (monte carlo) algorithm presented in the previous section. Referring back to the basic structure of a SA algorithm which was presented above, boltzmann annealing chooses the normal distribution for $p()$. The p.d.f. of a n-dimensional multivariate normally distributed variable $\mathbf{x}$ is given by:

$$p(\mathbf{x}) = (2\pi)^{-n/2} |\mathbf{R}|^{-1/2} \exp\{-\tfrac{1}{2}(\mathbf{x}-\boldsymbol{\mu})'\mathbf{R}^{-1}(\mathbf{x}-\boldsymbol{\mu})\}, \tag{3.13}$$

where $\boldsymbol{\mu}$ is a $n \times 1$ vector of means, $\mathbf{R}$ is a $n \times n$ covariance matrix, and $|\mathbf{R}|$ is the determinant of the covariance matrix. If assume $\boldsymbol{\mu} = 0$ and the covariances are zero with a standard deviation σ for each component of $\mathbf{x}$, the normalized multivariate normal p.d.f. is easily derived from (3.13) above:

$$p_{\mathbf{x}}(\mathbf{x}) = \sigma^{-d} (2\pi)^{-d/2} \exp\left\{\frac{-\mathbf{x}'\mathbf{x}}{2\sigma^2}\right\}. \tag{3.14}$$

Using (3.14) above, the p.d.f. $g(\Delta\mathbf{x})$ for step 2 of BA is then given by:

$$p(\Delta\mathbf{x}) = r^{-d} (2\pi)^{-d/2} \exp\left\{\frac{-\Delta\mathbf{x}'\Delta\mathbf{x}}{2r^2}\right\}, \tag{3.15}$$

where $r = \sigma T_k$ and $\Delta \mathbf{x} = \mathbf{x}_{k+1} - \mathbf{x}_i$. Since the standard deviation is a function of the temperature T, it is easy to see how the dispersion of perturbations will get smaller as the temperature T is slowly decreased[2].

The acceptance probability from step 3 is given by

$$g(\Delta C, T_k) = \min\{1, \exp(\Delta C / cT_k)\} \tag{3.16}$$

where $\Delta C = C(\mathbf{x}') - C(\mathbf{x})$ is the change in the cost function being minimized and c is a constant scaling factor. If $\Delta C \leq 0$ the step is automatically accepted, otherwise, as detailed in step 3, it is accepted according to the probability given by $\exp(\Delta C / cT_k)$ [3].

Geman and Geman proved that given $g()$ in (3.16), and a sufficiently high temperature T_0, asymptotically the algorithm will find the global minimum provided the reduction in T is not faster than:

$$T_k = h(k) = \frac{T_0}{\ln(k)}. \tag{3.17}$$

The proof is also sketched in Ingber (1989) and Szu and Hartley. As a practical matter, the cooling schedule in (3.17) is very slow. Many researchers use faster cooling schedules. Global convergence is no longer guaranteed, however this does not preclude the algorithm from still being useful for solving optimization problems. Since as a practical matter, the computing power is often not available to "ensure" global convergence for many larger problems, it suffices to obtain reasonably good answers to

[2] The standard deviation can be a function of T or alternatively we can set the standard deviation equal to T. The algorithm is invariant to which choice we make. We need only consider any necessary scaling with respect to our choice for T, c in (3.16) and if appropriate, the standard deviation σ.

[3] Some authors, such as Ingber and Szu and Hartley, give a description of Boltzmann annealing with the alternative acceptance criterion given in (3.21) later in this text. However, it is this authors opinion that most "classic" implementations of Boltzmann annealing used the acceptance criterion given in (3.16) above. See section 3.4.3 and figure 3.2 for a comparison of the two.

the problem. Ingber (1993) refers to the use of faster cooling schedules as simulated quenching rather than simulated annealing.

3.4.3 Fast Simulated Annealing

The choice of $p()$ and $g()$ in (3.15) and (3.16) above for Boltzmann Annealing comes from physical principles underlying concepts from statistical mechanics. However, there is no reason why these choices should be the optimal for function optimization. Researchers have used other functions leading to algorithms that are theoretically more efficient. Szu and Hartley introduced several modification which theoretically make the simulated annealing algorithm much more efficient. They introduced what they called fast simulated annealing (FSA) by substituting the cauchy distribution for $p()$ and introducing a different acceptance criterion $g()$. The cauchy distribution with a median of θ and scale parameter λ is given by (Johnson and Kotz, 1970)[4]:

(3.18) $$(\pi\lambda)^{-1}[1+\{(x-\theta)/\lambda\}^2]^{-1}.$$

The upper and lower quartiles are given by $\theta \pm \lambda$. A standard cauchy distribution with $\theta = 0$ and $\lambda = 1$ is a central t distribution with one degree of freedom. To generate a step for $\mathbf{x}$ in $\Re^n$, one cannot simply sample the univariate cauchy distribution for each component of $\mathbf{x}$. Instead, the step must be generated from a multivariate cauchy distribution. An n-dimensional multivariate cauchy distribution is given by (Johnson and Kotz, 1972; Styblinski and Tang):

(3.19) $$p(\Delta\mathbf{x}) = \left(\Gamma/\pi\right)^{(n+1)/2} \frac{\lambda}{(\lambda^2 + \Delta\mathbf{x}'\Delta\mathbf{x})^{(n+1)/2}}.$$

The cauchy distribution does not possess finite values of mean and standard deviation (Johnson and Kotz, 1970). The cauchy distributions infinite variance gives the distribution fatter tails as compared to the normal distribution. Therefore, the cauchy distribution provides for more occasional long jumps while retaining local sampling as compared to the gaussian. Figure 3.1 shows a comparison between cauchy and normal distributions. This trade-off between local and global search allows for a much faster cooling schedule given by

$$T_k = \frac{T_0}{k}. \tag{3.20}$$

The cooling schedule in (3.19) is exponentially faster than that given for BA while still maintaining the property of global convergence (Szu and Hartley). Szu and Hartley also introduced an acceptence criterion which is different than that for BA:

$$g(\Delta C, T_k) = \frac{1}{1+\exp(\Delta C / cT_k)}. \tag{3.21}$$

Figure 3.2 shows a comparison between the Szu acceptance criterion (3.20) and the metropolis acceptance criterion.

[4] A mathematically equivalent form which is often given in the simulated annealing literature is $\lambda / \{\pi[\lambda^2 + (x-\theta)^2]\}$.

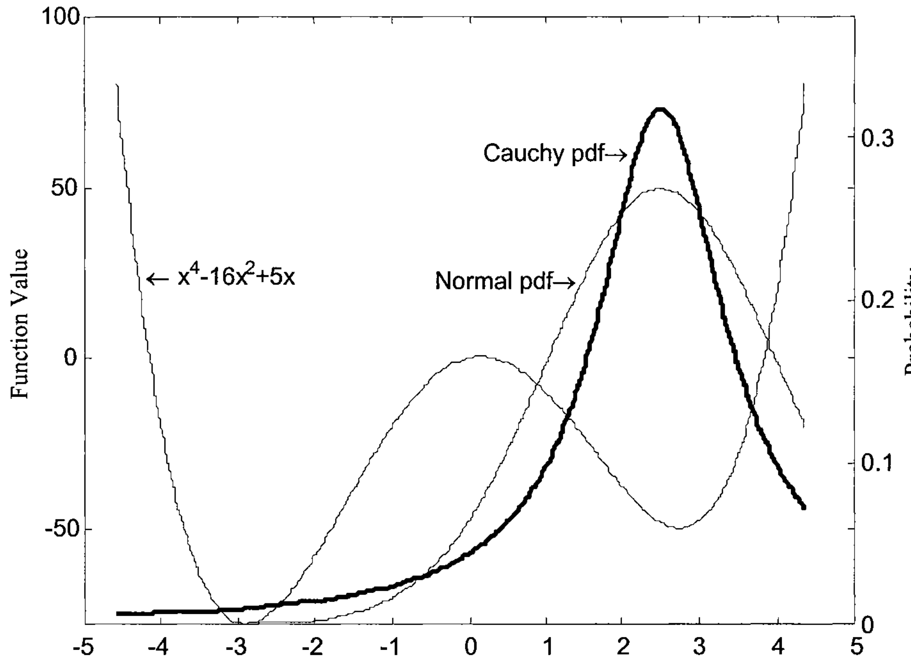

Figure 3.1. Comparison of the Normal and Cauchy Probability Density Functions

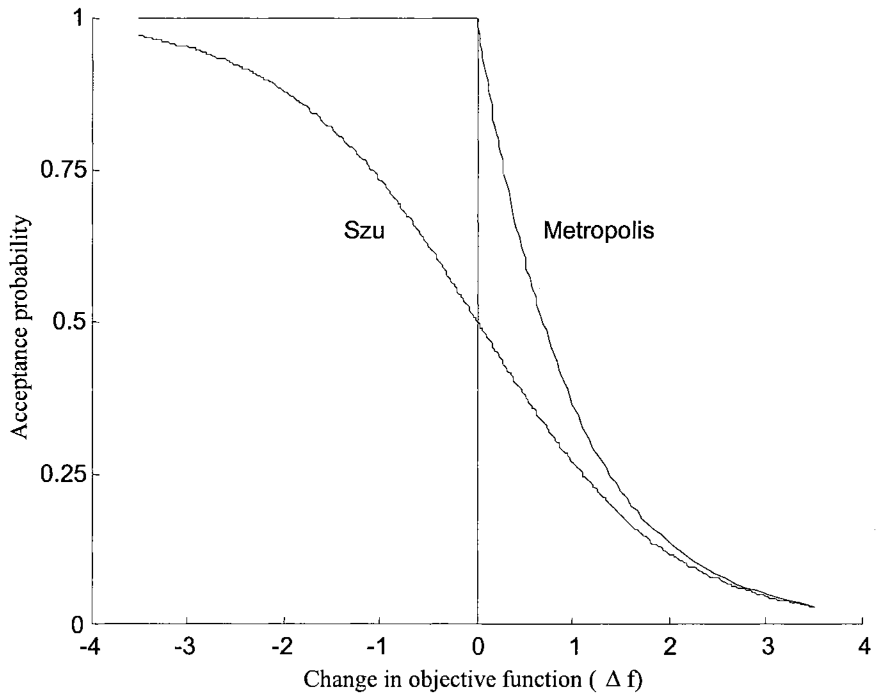

Figure 3.2 Comparison of Probabilistic Selection Criterion

CHAPTER 4

DATA AND PROCEDURES

4.1 Introduction

This chapter explains the data and methods used to accomplish the research objectives given in chapter 1. Section 4.2 enumerates the neural network architectures used with various training data sets. In addition, several cost functions used to train the neural networks are presented. Section 4.3 describes the neural network training data sets as well as the configurations of the neural networks associated with each of those data sets. Section 4.4 describes the local and global optimization algorithms used to train the neural networks. Finally, Section 4.5 describes the procedures used to determine the relative speed and accuracy of the alternative global optimization methods used in this research to train neural networks.

4.2 Neural Network Architectures and Cost Functions

This section presents the specific neural network forms used in this research. For a more general discussion of different neural networks, the reader is referred to chapter 2. This study is restricted to training of a feedforward multilayer perceptron (MLP). The output from output neuron *d* of a feedforward MLP with one hidden layer is:

$$f_d(\mathbf{x}_t, \theta_d) = \beta_{0d} + \sum_{j=1}^{h} \beta_{jd} G(\tilde{\mathbf{x}}_t' \gamma_j) \tag{4.1}$$

where h is the number of hidden neurons in the single hidden layer, $\mathbf{x}_t$ is a $k \times 1$ vector of inputs or explanatory variables for observation t, $\tilde{\mathbf{x}}_t = (1, \mathbf{x}_t')$, $\gamma_j = (\gamma_{0j}, \ldots, \gamma_{kj})'$ is a vector of weights connecting the inputs to hidden neuron j, $\beta_d = (\beta_{0d}, \ldots, \beta_{hd})'$ is a vector of weights connecting the hidden neurons to output neuron d, $\theta_d = (\beta_d', \gamma_1', \ldots, \gamma_h')$ is the vector of model parameters or weights, and $G(\bullet)$ is a hidden layer activation function. The two most commonly used activation functions, and the ones used in this research, are the logistic and hyberbolic tangent functions. The logistic function is defined by:

(4.2) $$G(z) = 1/[1 + \exp(-z)],$$

and the hyperbolic tangent by:

(4.3) $$G(z) = \tanh(z) = (e^z - 1)/(e^z + 1).$$

A feedforward MLP with one hidden layer and only one output neuron is easily derived from (4.1):

(4.4) $$f(\mathbf{x}_t, \theta) = \beta_0 + \sum_{j=1}^{h} \beta_j G(\tilde{\mathbf{x}}_t' \gamma_j),$$

where $\theta = (\beta_0, \ldots, \beta_h, \gamma_1', \ldots, \gamma_h')$ is the $n \times 1$ vector of parameters that must be estimated. In (4.4), the number of model parameters $n = 1 + h(k+1)$. The input variables can be included as linear regressors by using direct connections between the inputs and outputs. Modifying (4.4) we have:

(4.5) $$f(\mathbf{x}_t, \theta) = \tilde{\mathbf{x}}_t' \phi + \sum_{j=1}^{h} \beta_j G(\tilde{\mathbf{x}}_t' \gamma_j),$$

where $\phi = (\phi_0, \ldots, \phi_k)$ and $\theta = (\phi_0, \ldots, \phi_k, \beta_1, \ldots, \beta_h, \gamma'_1, \ldots, \gamma'_h)$ is the $n \times 1$ vector of parameters that must be estimated. In (4.5), the number of model parameters is $n = 1 + k + h(k+2)$.

Several different cost or objective functions are used in this research to train the neural networks. Given a set of training data with T observations and assuming the neural network form given in (4.4), the least squares cost function is defined by:

$$\min_{\theta \in \Theta} Q(\theta) = \sum_{t=1}^{T} [y_t - f(\mathbf{x}_t, \theta)]^2 \tag{4.6}$$

where y_t is the dependent variable, and Θ is the space of feasible weights or model parameters. Augmenting the cost function in (4.6) with terms that penalize large weight values yields:

$$\min_{\theta \in \Theta} Q(\theta) = \sum_{t=1}^{T} [y_t - f(\mathbf{x}_t, \theta)]^2 + r_\phi \sum_{i=0}^{k} \phi_i^2 + r_\beta \sum_{j=0}^{h} \beta_j^2 + r_\gamma \sum_{j=1}^{h} \sum_{i=0}^{k} \gamma_{ij}^2, \tag{4.7}$$

where r_ϕ, r_β, and r_γ are weight decay constants. Following Franses and van Dijk, for all problems on which weight decay is used, the weight decay parameters are set equal to $r_\phi = .01$, and $r_B = r_\gamma = .0001$.

4.3 Training Data Sets

A variety of training data sets are used in this research to evaluate the training algorithms. The data sets include financial, scientific, and synthetically generated data. The size of the data sets and associated neural network models varies from 250 observations and 2 input variables for a neural network with 15 weights to 533 observations and 22 input variables for a neural network with 211 weights. Table 4.1

Table 4.1 Summary of Training Data Sets and Neural Network Models.

Data Set	Obs[a]	Neural Network Architecture[b]	NNW[c]	NNF[d]	NNOF[e]
Bilinear	250	2-3-1, dc; logistic-identity	15	(4.5)	(4.7)
DAX	360	4-4-1, dc; logistic-identity	29	(4.5)	(4.7)
JYUS	364	2-3-1, dc; logistic-identity	15	(4.5)	(4.7)
JYUSTTR	326	2-3-1, dc; logistic-identity	15	(4.5)	(4.7)
Flare	533	22-8-3; logistic-identity	211	(4.1)	(4.6)
Mackey-Glass	500	5-6-1; logistic-identity	43	(4.4)	(4.7)

[a]The number of observations in the data set.
[b]The number of neurons in consecutive layers are enumerated as input-hidden-output, with a dc following indicating a direct connection between the input and output neurons. Likewise, following the layout of the neurons, the activation functions are enumerated beginning with the first hidden layer.
[c]The number of neural network weights or parameters that must be estimated.
[d]Equation number of the neural network functional form used for that data set.
[e]Equation number of the objective function form used to train the neural network for that data set.

summarizes the characteristics of the data sets. Section 4.3.1 presents the synthetically generated training data and section 4.3.2 the real-life training data. The abbreviation for a particular data set, if any, are in parentheses after the section heading bearing the data sets name.

4.3.1 Synthetic data

Mackey-Glass

The Mackey-Glass time series has appeared numerous times in the neural network literature as a benchmark time-series for prediction and estimation, for example, Chow and Leung; Ergezinger and Thomsen; Goffe, Ferrier, and Rogers; Sexton, Dorsey, and Jonson (1999b). Mackey and Glass were the first to investigate the series. The Mackey-Glass time series $x(t)$ is produced by the numerical solution of the Mackey-Glass differential-delay equation:

$$\frac{dx}{dt} = -bx(t) + \frac{ax(t-\tau)}{1+x^c(t-\tau)}, \tag{4.8}$$

where a, b, c, and τ are parameter constants and $x(t)$ is the value of x at time t. The constants a, b, and c are typically set to 0.2, .1, and 10 respectively. The constant τ is called the delay parameter and it determines the chaotic behavior displayed by the series. Farmer has studied the behavior of (4.8). For values of $\tau > 16.8$, the series exhibits chaotic behavior. This study uses a discrete version of the Mackey-Glass equation as used in Gallant and White:

$$g^{*}(x_{t-5}, x_{t-1}) = x_{t-1} + 10.5\left[\frac{0.2x_{t-5}}{1+x_{t-5}^{10}} - 0.1x_{t-1}\right]. \tag{4.9}$$

The Mackey-Glass series is said to be qualitatively like financial market data (Gallant, Hsieh, and Tauchen). The series can exhibit long stretches of volatile data of apparently random duration.

The Mackey-Glass data for this research were generated from (4.9) with starting values of $x_0 = 1.6$ and $x_i = 0$ for $i = -4, \ldots, -1$. One thousand observations are generated with the first 500 discarded leaving 500 observations for training data. Figure 4.1 is a graph of the first 100 observations of the Mackey-Glass series as used in this research. The neural network model has five inputs consisting of five lags of the Mackey-Glass series. As can be seen from (4.9), only lags t-1 and t-5 are necessary to approximate this series. However, in most actual applications of neural networks, the true dimension of the problems is unknown. Therefore, superfluous inputs are commonly part of neural network modeling. The neural network model has one hidden layer with 6 neurons, logistic activation functions for the hidden layer neurons and an identity transfer function for the output neuron.

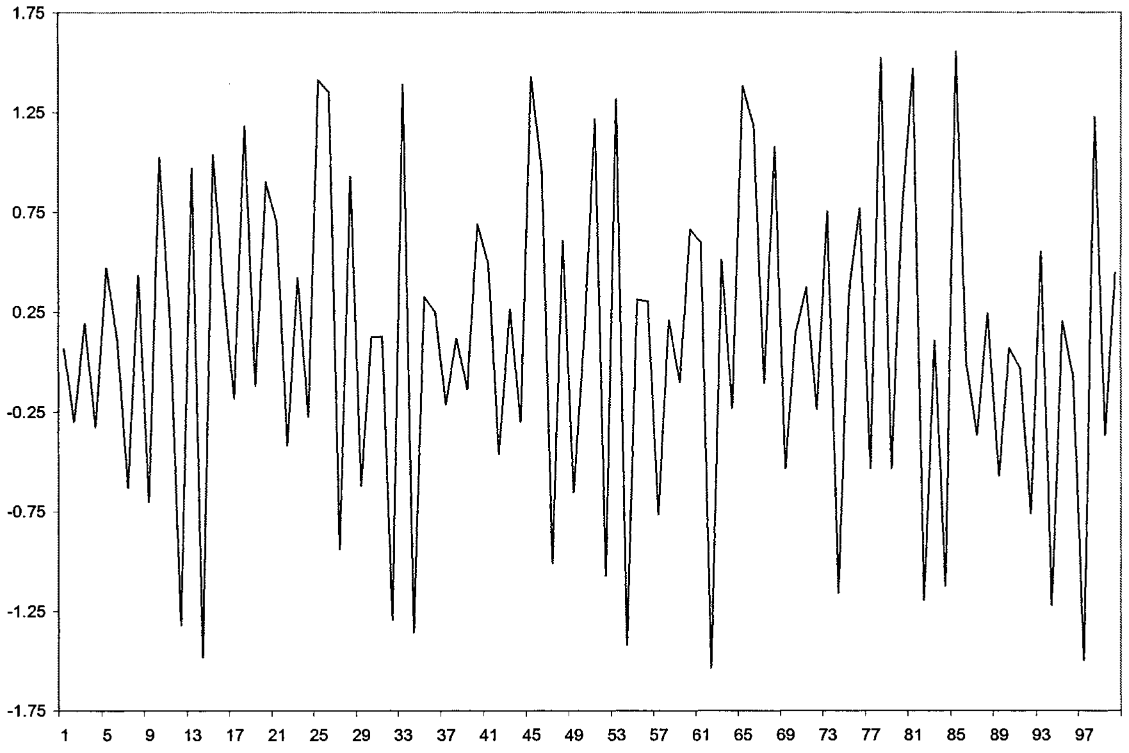

Figure 4.1. Mackey-Glass Time Series

Bilinear Model

The time series from the bilinear model is generated by:

$$y_t = \beta y_{t-2}\varepsilon_{t-1} + \varepsilon_t, \tag{4.10}$$

with $\beta = 0.6$. This series displays occasional sharp spikes. Its characteristics make it of interest in econometrics and control theory (Mohler). Granger and Anderson showed this model has zero autocorrelations at all lags. Therefore, linear models will not be successful in modeling this series. Franses and Dijk modeled the series in (4.10) and reported that a neural network with at least two lags as inputs showed considerable improvement over a linear model.

The bilinear series in this study is generated by setting $y_0 = y_{-1} = 0$ and drawing ε_t from a Normal(0,1) distribution. A total of 350 observations are generated with the first 100 discarded leaving 250 observations for the neural network training set. A graph of this series is given in figure 4.2. Based upon results in Franses and Dijk, the neural network for this series has 2 lags of (4.10) as inputs, 3 hidden neurons in a single hidden layer with logistic activation functions, and an identity activation function for the output layer.

4.3.2 Real-Life Data

Japanese Yen-US Dollar Exchange Rate (JYUS)

The Japanese Yen-US Dollar Exchange Rate (JYUS) data are weekly returns on the Japanese yen-US dollar exchange rate from Franses and Dijk. The weekly returns are given by:

$$r_t = \ln P_t - \ln P_{t-1}, \tag{4.11}$$

where r_t is the return for week t and P_t is the level of the Japanese yen-US dollar exchange rate for week t. Franses and Dijk demonstrated that the relationship between the JYUS return series and its lags is nonlinear.

Following Franses and Dijk, the training data consists of 364 observations from January 1986 through December 1992. The neural network model uses two lags of (4.11) as inputs. We use a feedforward network with logistic activation functions for 3 hidden neurons in a single hidden layer and an identity activation function for the output neuron.

Japanese Yen-US Dollar Exchange Rate with Technical Trading Rules (JYUSTTR)

This second model to predict the Japanese yen-US dollar exchange rate is constructed by using technical trading rules as inputs to a neural network prediction

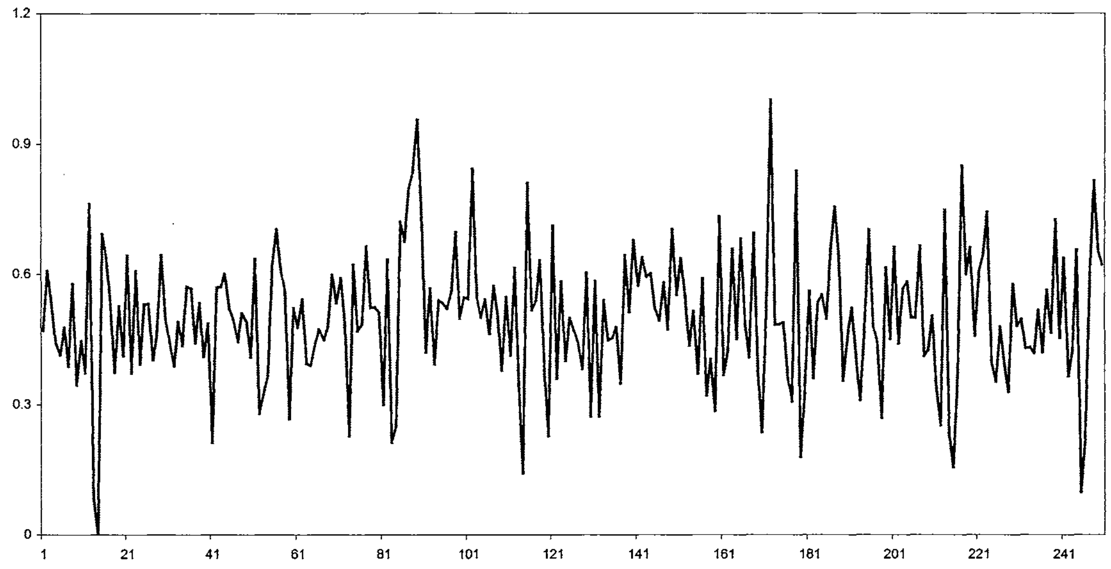

Figure 4.2 Bilinear Time Series

model. Specifically, moving average trading signals are used. Following the notation of Franses and Dijk, we define a moving average of length τ for period *t* as:

$$m_t(\tau) = \frac{1}{\tau}\sum_{i=0}^{\tau-1} p_{t-i} . \tag{4.12}$$

A moving average technical trading rule can be constructed from (4.12) as follows:

$$s_t(\tau_1, \tau_2) = m_t(\tau_1) - m_t(\tau_2), \tag{4.13}$$

where $\tau_1 < \tau_2$. Equation (4.13) defines what is commonly called a dual moving average crossover. Following Franses and Dijk, τ_1 and τ_2 are set to 1 and 40. The time periods for this data are the same as that for the JYUS data set. Based on results presented in Franses and Dijk, three lags of (4.13) are used as inputs and the neural network is chosen to have 3 hidden neurons in a single hidden layer with logistic activation functions, and the identity function for the activation function in the output neuron. This data set will be referred to in this research as the JYUSTTR data set.

DAX

The DAX data are weekly absolute returns on the DAX stock index of the Frankfurt stock exchange as used in Franses and Dijk . The weekly absolute returns are as given in equation (4.11) except that the absolute value is taken. This task is given to be harder than predicting just return levels. Franses and Dijk show evidence of nonlinearity between this series and its lags. The time periods for the data are the same as that for the JYUS and JYUSTTR data sets. Based on results presented in Franses and Dijk, four lags of the absolute returns on the DAX stock index are used as inputs. The neural network is chosen to have a single hidden layer with 4 hidden neurons and logistic activation functions. The output neuron has the identity function for its activation function.

Solar-Flare (Flare)

The solar-flare data were obtained from the Proben1[1] benchmark data set (Prechelt). The objective is to predict the number of small, medium, and large size flares that will happen during the next 24-hour period in a fixed active region of the sun's surface. There are 3 dependent variables in the data set, one each to predict the number of small, medium, and large solar flares. There are 22 inputs describing the type and history of the active region and the previous flare activity.[2]

Following the Proben1 guidelines, the first 533 observations from the data file flare1.dt are used for training. Based upon the training and prediction results on this data set from Prechelt, a network with 8 neurons in a single hidden layer with logistic transfer

[1] The Proben1 benchmark data set is accessible via anonymous FTP on ftp.ira.uka.de as /pub/neuron/proben1.tar.gz.

[2] Prechelt reports 24 inputs, however, processing of this data by Prechelt leaves 2 inputs with constant values of zero in the file flare1.dt. Therefore, these 2 inputs are dropped for this research.

functions is chosen and the identity function for the activation functions in the output layer. The scaling of the data is left as it is in the original file flare1.dt. The input variables are scaled from 0 to 1. The three output variables have minimum values of 0 and maximum values of .75, .375, and 1.00 respectively.

4.4 Optimization Algorithms

The purpose of this section is to present the specific optimization algorithms used in this research. For a more general and theoretical discussion of these algorithms, the reader is referred to chapter 3. It is assumed in this section that the reader is familiar with the general concepts and terminology of the algorithms presented in this chapter. Therefore, only a very brief introduction is given to each of the algorithms presented. Some of the terminology in this section for the evolutionary algorithms may differ from that used in some of the literature. In particular, various presentations of evolutionary algorithms exist in the literature. Also, the conventional terminology associated with each of the algorithms will be used in this research to refer to the parameter array being optimized. For example, the model parameters θ may be referred to as trial or candidate solutions in the simulated annealing literature, an individual in the evolutionary strategy literature, and a chromosome in the genetic algorithm literature.

The stochastic global optimization algorithms used in this research are hybrid algorithms combining a local optimization with the stochastic global algorithm by using the parameters obtained from the global algorithm as starting values for the local routine. Stochastic global optimization algorithms are theoretically good at widely exploring the potential solution space. However, they are poor at honing in on a particular solution once a promising area of the solution space is found. On the other hand, a local routine will quickly converge to a local minimum. Combining the two types of algorithms exploits the advantages of both local and global types of optimization algorithms. This hybrid approach has been used for training neural networks (Yan, Zhu, and Hu; Skinner and Broughton; Knowles, Corne, and Bishop; Heistermann). For a particular data set and neural network model, the local optimization routine used in the hybrid global algorithm depends upon the size of the neural network model. For the largest neural network models, a conjugate gradient routine is used and a quasi-Newton algorithm for the smaller neural network models.

Since global algorithms are not very good at fine tuning a local minimum, convergence criterions that are used for local routines, such as the magnitude of the gradient, are not appropriate. Therefore, for simplicity, all of the stochastic global routines are run for 100,000 function evaluations. The local routine then takes over and is run to convergence. Unless otherwise stated, for the local algorithm or any of the global algorithms, the starting values of the neural network weight vectors θ, as given in equation (4.4), are randomly initialized uniformly between $\pm .3$. The details of the local and global optimization algorithms are presented in the following sections.

4.4.1 Local Optimization

Several local optimization routines, namely a quasi-Newton and conjugate-gradient algorithm, are used in this study. The quasi-Newton algorithm uses information from the first derivatives as well as a BFGS approximation of the hessian. The algorithm is very efficient at converging to a local-minimum in a minimum number of iterations. However, the quasi-Newton algorithm can be computationally demanding for larger problems because it requires calculating and storing the approximation to the hessian.

The conjugate-gradient algorithm only requires calculating and storing the first derivatives and therefore is commonly used for problems with a large number of variables. In this study, the quasi-Newton algorithm is used for the smaller network models and the conjugate-gradient method for the largest network models. The quasi-Newton algorithm used is the DUMING subroutine from the IMSL subroutine libraries (Visual Numerics) and the conjugate-gradient algorithm is the DUMCGG routine, also from the IMSL libraries. The DUMCGG routine is based on the conjugate gradient method in Powell. For the DUMING and DUMCGG routines, all user definable parameters, other than those associated with the stopping criterions, are set to their default. The starting values of the weight vectors θ, as given in equation (4.4), are randomly initialized uniformly between $\pm .3$. The stopping criterions and associated user parameters are discussed below.

The local algorithms are run until a convergence test is met. For both the conjugate-gradient and quasi-Newton algorithms, the convergence test is based on two termination criteria. The algorithm continues until one of the termination criteria is met. One of the termination criteria is based on the computational effort expended. The quasi-Newton algorithm has 3 computational criteria on which to stop the algorithm: the maximum number of iterations, function evaluations, or gradient evaluations. The computational stopping criterion for the conjugate gradient algorithm is based on the maximum number of function evaluations. For the quasi-Newton algorithm, the maximum number of iterations is set to 20,000 and the maximum number of function and gradient calculations is set to 30,000. For the conjugate-gradient routine, the maximum number of function evaluations is set to 60,000.

The other stopping criterion for both algorithms is based on the magnitude of the gradient. A small magnitude of the gradient would indicate that the algorithm has converged very near the minimum. Therefore, the goal is to obtain convergence based upon the criterion of a small gradient. The criterion based on computational effort is only a fall back if the algorithm is stuck. For the conjugate gradient routine DUMCGG, the algorithm will stop when the square of the Euclidian norm or two-norm of the gradient is less than a given gradient tolerance. The two-norm is defined as the square root of the sum of squares of the components of the gradient. A more appropriate norm for large-scale problems would be the infinity norm which is defined as the maximum of the absolute values of all elements in the vector. The large number of terms contributing to the calculation of a two-norm of the gradient will make conventionally accepted values for the gradient tolerance too stringent (Gill and Murray, pg. 307). Therefore, for a given optimization problem in this research, some amount of experimentation may be needed to discover an appropriate value for the gradient tolerance.

The stopping criterion for the quasi-Newton algorithm criterion occurs when the infinity norm of the scaled gradient is less than a gradient tolerance. The *i*-th component of the scaled gradient at the point $\mathbf{x}$ is given as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}, \tag{4.14}$$

where g is the gradient, s is a scaling matrix for the variables, and f_s is a scaling factor for the function being optimized. In this research f_s and the elements of s are set to 1. The gradient tolerance is set to $\sqrt[3]{\varepsilon}$ (approximately 6.055E-6), where ε is the double precision machine tolerance. The quasi-Newton algorithm also has a second stopping criterion not based on computation effort. This stopping criterion occurs when the scaled distance between the last two steps is less than a step tolerance. The i-th component of the scaled step tolerance between the last two consecutive steps x^j and x^{j-1} is given by

$$\frac{\left|x_i^j - x_i^{j-1}\right|}{\max(\left|x_i^j\right|, 1/s_i)}. \tag{4.15}$$

The elements of s are set to 1 in this study. The step tolerance is set to $\varepsilon^{2/3}$ (approximately 3.666E-11), where ε is the double precision machine tolerance.

4.4.2 Genetic Algorithm

The genetic algorithm used in this study uses a real-valued representation of the model parameters, as opposed to a binary representation. See section 3.2 for a discussion of real versus binary valued chromosomes. The genetic algorithm in this research is patterned after the genetic algorithm used in Rooij, Jain, and Johnson to train neural networks. The defining feature of this genetic algorithm is the use of a neural network specific crossover operator. The neural network specific operator addresses what Rooij, Jain, and Johnson refer to as neuron disruption. This neuron disruption is caused by the functional characteristics of neural networks and interferes with a normal crossover operators ability to form superior solutions in neural network training. The genetic algorithm used in this research will be referred to as the neural network genetic algorithm (NNGA). In addition to a neural network specific crossover operator, the NNGA uses a uniform crossover scheme. A uniform crossover operator further minimizes potential disruptive behavior caused by phenomena that Rooij, Jain, and Johnson refer to as representational bias of disruption and (reverse) hitch-hiking. See section 3.2 for an illustration of a uniform crossover scheme with the neural network specific crossover operator.

Assuming the notation for the neural network form given in (4.4) and the cost function in (4.6), the NNGA algorithm proceeds as follows:

Step 0: Set the generation counter $r = 0$, pick the values for various parameters of the algorithm and generate a random population P of chromosomes $\theta_i^{(0)} \in \Re^n$, $i = 1,\ldots,p$.

Step 1: Calculate the fitness g_i of each chromosome in P using the fitness function $G: -Q(\theta_i^{(r)}) \Rightarrow [1,b]$, $i = 1,\ldots,p$ where b is the bias. Save the fittest chromosome for possible insertion back into the next generation.

Step 2: Evaluate the stopping criterion. If $r \geq R$,where R is the maximum number of generations allowed, halt the algorithm and return the fittest chromosome found across all generations, else continue to step 3.

Step 3: (a) Using the fitness values, calculate the selection probability sp_i of each chromosome according to:

$$sp_i = g_i \Big/ \sum_{j=1}^{n} g_j \,,\ i = 1,\ldots,n \,.$$

(b) Form an intermediate population P' of chromosomes from the current population using a roulette wheel selection scheme by randomly selecting, based on the selection probabilities, chromosomes from the current population with replacement.

Step 4: Randomly select, without replacement, two chromosomes from P' and apply the following operators to the pair of chromosomes:

(a) Generate a random number $i \in [0,1]$. If $c_p > i$, where c_p is the crossover probability, apply the neural network specific uniform crossover operator to the pair of chromosomes.

(b) For each gene in each chromosome, if $m_p > i$, where m_p is the mutation probability and $i \in [0,1]$ is a random number, apply the mutation operator by adding a random number from a $\mathrm{N}(0,s)$ to the gene.

Step 5: (a) If the fittest chromosome of the population, as calculated in step 1, has not survived without being altered by the genetic operators, re-insert the chromosome into the population replacing a randomly selected chromosome.

(b) Set $P = P'$ and return to step 1.

In Step 0, the chromosomes or weight vectors are randomly initialized uniformly between $\pm .3$. In Step 1, according to an elitism scheme, the fittest chromosome in each generation is saved for possible insertion back into the population. If the fittest chromosome does not survive selection in Step 3, or is altered in Step 4 by the crossover or mutation operators, it is reinserted back into the population for the new generation in Step 5. If reinserted, the chromosome replaces a randomly chosen chromosome. The fitness function G used in Step 1 transforms the cost function values to produce a high fitness value from a low cost function value. The fitness function also normalizes the transformed cost function values over the range $[0,b]$ so that the ratio of the best to worst fitness values is fixed. As the algorithm progresses and the ratio of the worst to best cost function values decreases, the fixed ratio of fitness values improves convergence. In Step 4, the crossover operator is applied to the pair of randomly selected chromosomes, however the mutation operator is applied to each gene of each chromosome (each weight in each weight array) individually. The size of the population p, the bias b, the standard deviation of mutation s, and r_c and r_m, the probability of crossover and mutation respectively are the user definable algorithm parameters that must be set. The setting of these parameters will be discussed in section 4.5.1.

4.4.3 Evolutionary Strategies

Five different Evolutionary Strategies taken from Schwefel (1995) are used in this research. One of the algorithms is the two-membered evolutionary strategy Schwefel (1995) refers to as EVOL. The other four algorithms are variations of a multi-membered evolutionary strategy that Schwefel (1995) calls the KORR algorithm. The source code included with the book from Schwefel (1995) is utilized to implement the evolutionary strategy. Some modifications are made to the code as explained below in the sections detailing the specific implementations of the algorithms.

The EVOL algorithm is a very simple evolutionary strategy algorithm referred to as a $(1+1)-\mathrm{ES}$ algorithm in the literature. This notational representation of the algorithm characterizes the operation of the algorithm. In each iteration or generation of the algorithm, a single individual produces one offspring by mutation of itself. The selection mechanism then picks the superior of the parent or offspring to survive to the next generation. In spite of its simplicity, Schwefel (1995, pg 151) claims the $(1+1)-\mathrm{ES}$ type of evolutionary strategy has been more widely used than any other evolutionary strategy algorithm in practice.

Assuming the notation for the neural network form given in (4.4) and the cost function in (4.6), the EVOL algorithm proceeds as follows:

Step 0: Set the generation counter $r=0$ and auxiliary counter $r'=0$. Pick the values for various parameters of the algorithm and randomly initialize the parent $\theta^{(0)} \in \Re^n$.

Step 1: (a) Mutate the parent to form an offspring according to $\widetilde{\theta}^{(r)} = \theta^{(r)} + \mathbf{v}$ where $\mathbf{v}$ is a random vector drawn from a Gaussian distribution with mean 0 and standard deviation $s^{(r)}$,
(b) If $Q(\widetilde{\theta}^{(r)}) < Q(\theta^{(r)})$ let $\theta^{(r+1)} = \widetilde{\theta}^{(r)}$, else $\theta^{(r+1)} = \theta^{(r)}$.

Step 2: (a) Set $r = r+1$ and $r' = r'+1$,
(b) If $r' = 10 \cdot n$ then:
(i) adjust the standard deviation of mutation according to:

$$s^{(r+1)} = \begin{cases} s^{(r)} \cdot c_\sigma, & \text{if } \varphi(p) < 1/5, \\ s^{(r)} \cdot 1/c_\sigma, & \text{if } \varphi(p) > 1/5, \\ s^{(r)}, & \text{if } \varphi(p) = 1/5, \end{cases}$$

where c_σ is a predefined constant and $\varphi(p)$ is the success ratio of the mutation operator during the last p generations,

(ii) set $r' = 0$.

Step 3: If $r = R$, where R is the maximum number of generations allowed, halt the algorithm, otherwise return to step 1.

In Step 0, the parent or neural network weight vector is randomly initialized uniformly between $\pm .3$. The setting of the values for c_σ and s are discussed in section 4.5.1.

The KORR algorithm is a more modern and sophisticated algorithm as compared to the very simple EVOL evolutionary strategy above. The KORR algorithm has a much more sophisticated mechanism for adjusting the mutation variances than does the EVOL algorithm. One of the defining characteristics of a modern evolutionary strategies algorithm is its ability to evolve or self-adapt the variances and sometimes covariances of the Gaussian mutations. Each model parameter has a standard deviation of mutation associated with it that evolves or varies through out the operation of the algorithm. In addition, as opposed to a single parent producing one offspring in every generation, the KORR algorithm is multi-membered. Similar to a genetic algorithm, the KORR algorithm works from a population of individuals or candidate solutions in parallel. Another similarity to genetic algorithms is that modern evolutionary strategies introduce recombination as an operator. However, as opposed to a genetic algorithm, mutation remains the primary operator with recombination a background operator. The KORR algorithm includes many options for various types of recombination. The four variations of The KORR algorithms used in this research differ in the type of recombination that is used.

The KORR algorithms used in this research are referred to in the literature as $(\mu,\lambda)-\text{ES}$ types of evolutionary strategies. The $(\mu,\lambda)-\text{ES}$ algorithm works from a population with μ individuals. Each individual is composed of the model parameters plus the standard deviations of the mutations associated with that individual. The population *P* for iteration or generation *r* can be written as $P^{(r)} = (a_1^{(r)},\ldots,a_\mu^{(r)})$ with the individuals in *P* given by $a_j^{(r)} = (\theta_{1j}^{(r)},\ldots\theta_{nj}^{(r)}, s_{1j}^{(r)},\ldots,s_{nj}^{(r)}),\ j=1,\ldots,\mu$, where θ_{ij} are the model parameters and s_{ij} are the standard deviations of mutation associated with θ_{ij}. The u individuals in the population are referred to as parents. In each generation, the parents produce λ offspring, where $\lambda > \mu > 1$, using mutation and possibly recombination. A selection operator then selects the best μ individuals from the offspring to form a population for the generation. Assuming the notation for the neural network form given in (4.4) and the cost function in (4.6), the four variations of the KORR algorithm, KORR1, KORR2, KORR3, and KORR4 proceed as follows:

Step 0: Set the generation counter $r = 0$, pick the values for various parameters of the algorithm, and initialize the population *P* by setting the starting values for the model parameters $\theta_{ij}^{(0)} \in \Re^n,\ i=1,\ldots,n,\ j=1,\ldots,\mu$, and the standard deviation of the mutations $s_{ij}^{(0)},\ i=1,\ldots,n,\ j=1,\ldots,\mu$.

Step 1: Create a population of offspring $\widetilde{P}$, of size $\lambda > \mu$, by repeatedly selecting two parents at random from P and applying the following operators to the parents:

(a) Recombination operator:
  - (i) for KORR1: no recombination is used
  - (ii) for KORR2: intermediary recombination of pairs of parents for the model parameters,
  - (iii) for KORR3: intermediary recombination of pairs of parents for the standard deviation of the gaussian mutations,
  - (iv) for KORR4: intermediary recombination of pairs of parents for both the model parameters and standard deviation of the gaussian mutations.

(b) Apply the mutation operator in turn to the standard deviation of mutations and the model parameters :
  - (i) $\widetilde{s}_{ij}^{(r)} = s_{ij}^{(r)} \cdot \exp(\tau' \cdot \mathrm{N}(0,1) + \tau \cdot \mathrm{N}_{ij}(0,1)),\ i \in \{1,\ldots,\lambda\},\ j \in \{1,\ldots,n\}$,
  - (ii) $\widetilde{\theta}_{ij}^{(r)} = \theta_{ij}^{(r)} + \widetilde{s}_{ij}^{(r)} \cdot \mathrm{N}(0,1),\ i=1,\ldots,\lambda,\ j=1,\ldots,n$.

Step 2: Evaluate the cost functions $Q(\theta_i^{(r)})$, $i = 1, \ldots, \lambda$, where $\theta_i^{(r)} \in \widetilde{P}$.

Step 3: Apply the selection operator which selects the best μ individuals from the population of offspring $\widetilde{P}$ to form the next generation P.

Step 4: Set $r = r + 1$. If $r = R$, where R is the maximum number of generations allowed, exit the algorithm and return the best solution found, otherwise, return to Step 1.

Note that as opposed to the NNGA presented in the previous section, the KORR algorithm is not an elitist strategy. In each generation, the offspring replace all the parents. In Step 0, the parents or neural network weight vectors are randomly initialized uniformly between $\pm .3$. In Step 1 part (b), the subscripts on $\mathrm{N}_{ij}(0,1)$ indicate that a new random number is drawn for the mutation of each individual standard deviation of mutation. The setting of the values for the number of parents μ, number of offspring λ, the adjustment factors for the standard deviation of mutation τ and τ', and the beginning standard deviations of mutation $s_{ij}^{(0)}$ are discussed in section 4.5.1.

4.4.4 Simulated Annealing

Two simulated annealing algorithms are used in this research, a classic simulated annealing (CSA) routine that uses Gaussian mutations and a fast simulated annealing (FSA) routine that uses Cauchy mutations and a faster cooling scheme than CSA. In this research, the CSA routine is referred to as SA1 and the FSA routine SA2. Assuming the notation for the neural network form given in (4.4) and the cost function in (4.6), the algorithms proceed as follows:

Step 0: Pick the maximum number of iterations R, the number of iterations per temperature reduction R', initial acceptance criterion temperature $T_a^{(0)} > 0$, initial parameter temperatures $T_p^{(0)} > 0$, final parameter temperature $T_p^{(R)} > 0$, the final temperature ratio parameter

$\alpha = T_a^{(R)} / T_p^{(R)}$, estimate an appropriate scale factor c, initialize the starting point $\theta^{(0)} \in \Re^n$, calculate the corresponding cost function value $Q(\theta^{(0)})$, set $Q_{\min} = Q(\theta^{(0)})$, and set the iteration counter $r = 0$ and auxiliary counter $r' = 0$.

Step 1: Randomly pick a trial point $\tilde{\theta}^{(r)} = \theta^{(r)} + \Delta\theta$ in the parameter space where the step $\Delta\theta$ is drawn from the following distributions:

(a) for SA1 (CSA): a normalized multivariate normal p.d.f., with the parameter temperature $T_p^{(r)}$ playing the part of the standard deviation:

$$p(\Delta\theta) = \frac{1}{(2\pi)^{n/2}(T_p^{(r)})^n} \exp\left[\frac{-(\Delta\theta)'(\Delta\theta)}{2(T_p^{(r)})^2}\right],$$

(b) for SA2 (FSA): a normalized (median of zero) multivariate cauchy distribution, with scale parameter given by the parameter temperature $T_p^{(r)}$, given by (Johnson and Kotz, 1972; Styblinski and Tang):

$$p(\Delta\theta) = (\Gamma/\pi)^{(n+1)/2} \frac{T_p^{(r)}}{((T_p^{(r)})^2 + (\Delta\theta)'(\Delta\theta)^{(n+1)/2}}.$$

Step 2: Calculate the cost function value $Q(\tilde{\theta}^{(r)})$, $\Delta Q = Q(\tilde{\theta}^{(r)}) - Q(\theta^{(r)})$, and set $Q_{\min} = Q(\tilde{\theta}^{(r)})$ if $Q(\tilde{\theta}^{(r)}) < Q_{\min}$. Accept the trial point according to:

(a) For SA1 (CSA): if $Q(\tilde{\theta}^{(r)}) < Q(\theta^{(r)})$, set $\theta^{(r+1)} = \tilde{\theta}^{(r)}$, otherwise calculate an acceptance probability p_a according to

$$p_a = \mathrm{MIN}(1, \exp(-\Delta Q / cT^{(r)})).$$

If $b < p_a$, where $b \in [0,1]$ is a random number, then accept the inferior point $\tilde{\theta}^{(r)}$, otherwise let $\theta^{(r+1)} = \theta^{(r)}$.

(b) For SA2 (FSA): calculate an acceptance probability according to

$$p_a = \frac{1}{1 + \exp\left(\frac{\Delta Q}{cT^{(r)}}\right)}.$$

If $b < p_a$, where $b \in [0,1]$ is a random number, then accept the point $\tilde{\theta}^{(r)}$, otherwise let $\theta^{(r+1)} = \theta^{(r)}$.

Step 3: Set $r = r + 1$ and $r' = r' + 1$.

(a) If $r = R$, exit the algorithm and return $Q_{\min}$.

(b) Else if $r' = R'$, reduce the acceptance criterion temperature T_a according to:

$$T_a^{(r+1)} = T_p^{(r+1)} \cdot \exp(\log(\alpha) / R')^{(r+1)},$$

where $T_p^{(r+1)}$ is calculated according to:

(i) for SA1 (CSA): $T_p^{(r+1)} = T_p^{(r)} \cdot h$ where $h = (T_p^{(R')} / T_p^{(0)})^{1/(R'-1)}$,

(ii) for SA2 (FSA): $T_p^{(r+1)} = \dfrac{T_p^{(0)}}{1 + h \cdot (k+1)}$ where $h = \dfrac{T_p^{(0)} - T_p^{(R')}}{T_p^{(R')} \cdot (R'-1)}$.

Step 4: Return to step 1.

In Step 0, the network weight vector is randomly initialized uniformly between $\pm .3$. The two separate temperatures T_p and T_a allow for more control of the algorithm. Setting the parameter α controls the ratio of the two temperatures at the end of R iterations. Another parameter that has control over the probability of accepting an inferior trial point is the scale factor c. The scale factor c is a critical parameter and its appropriate magnitude depends upon the characteristics of the particular function being optimized. Because of the factor ΔQ, the acceptance criterions in step 2 above are sensitive to the amount of variation in the cost function. As was done in Masters (1995), an appropriate scale factor c is estimated in step 0 by sampling the parameter space numerous times and calculating the standard deviation of the associated cost function values. The scale factor is then calculated as $c = \sigma_c / T_\text{c}^{(0)}$ where σ_c is the standard deviation of the cost functions. The multivariate normal distribution in step 1 assumes zero covariances. The Cauchy distribution is calculated as in Styblinski and Tang. The setting of the values for α along with $T_p^{(0)}$ and $T_a^{(0)}$ are discussed in section 4.4.1.

4.4.5 Solis and Wets

The random optimization method of Solis and Wets is a simple algorithm that was used by Baba et. al. and Baba to train a neural network. Assuming the notation for the neural network form given in (4.4) and the cost function in (4.6), the algorithm proceeds as follows:

Step 0: Select a starting point $\theta^{(0)} \in \Re^n$, a standard deviation s, the maximum number of iterations R, and set r=0 and $\mathbf{b} = 0$.

Step 1: Generate a trial point $\tilde{\theta}^{(r)} = \theta^{(r)} + \mathbf{v}$ where $\mathbf{v}$ is drawn from a Guassian distribution with a mean of $\mathbf{b}$ and a standard deviation of s.

Step 2: (a) If $Q(\tilde{\theta}^{(r)}) < Q(\theta^{(r)})$, let $\theta^{(r+1)} = \tilde{\theta}^{(r)}$ and $\mathbf{b}^{(r+1)} = 0.4\mathbf{v}^{(r)} + 0.2\mathbf{b}^{(r)}$,
(b) else if $Q(\tilde{\theta}^{(r)}) \geq Q(\theta^{(r)})$, take a step in the opposite direction from the original point: $\hat{\theta}^{(r)} = \theta^{(r)} - \mathbf{v}$. If $Q(\hat{\theta}^{(r)}) < Q(\theta^{(r)})$ let $\theta^{(r+1)} = \hat{\theta}^{(r)}$ and $b^{(r+1)} = b^{(r)} - 0.4\mathbf{v}^{(r)}$, otherwise let $\theta^{(r+1)} = \theta^{(r)}$ and $\mathbf{b}^{(r+1)} = 0.5\mathbf{b}^{(r)}$.

Step 3: If r=R, exit the algorithm, otherwise set $r = r + 1$ and go to step 1.

In Step 0, the network weight vector is randomly initialized uniformly between $\pm.3$. Note that the mean for the Gaussian perturbations, represented by **b**, varies for each element in the weight vector θ for each iteration of the algorithm. However, the standard deviation of the Gaussian perturbations, represented by s, is the same for all elements. Given a starting vector $\theta^{(0)}$, there are two user definable parameters that must be set for this algorithm, namely the standard deviation deviation s and the maximum number of iterations R. The setting of these parameters will be discussed in section 4.5.1.

4.5 Simulation Details

4.5.1 Picking the Global Optimization Parameters

There are many parameters for the stochastic global optimization algorithms that must be chosen wisely for these algorithms to perform well. For example, the standard deviation of mutation or the temperature in the simulated annealing algorithm. Often, these parameters are chosen on an ad hoc basis. For some of the parameters, guidance exists as to reasonable values. In those cases, the recommended values will be used in this research. For other parameters, a more systematic methodology is employed to determine the appropriate values. Obtaining the results to be presented in chapter 5 for the stochastic global algorithms can be viewed as a two-stage procedure. In the first stage, a small number of preliminary restarts is run on each data set with each of the global algorithms to determine an appropriate set of algorithm parameters for that particular data set and algorithm. In the second stage, the set of algorithm parameters chosen in stage 1 is used to run a large number of restarts and the results from these simulations are presented in chapter 5.

In Stage 1, the procedure for choosing the parameters of the stochastic global algorithms for a particular data set is based on investigating the performance of a variety of combinations of algorithm parameters on that data set with a small number of restarts. Each specific combination of algorithm parameter values is referred to as a specific configuration of the algorithm. The performance of each of these configurations is based on the mean of the final cost function values calculated across the limited number of preliminary restarts. The number of preliminary restarts is 5 for the larger, and hence more computationally demanding, problems Flare and Mackey-Glass, and 10 for the smaller problems Bilinear, DAX, JYUS, and JYUSTTR. The specific configuration with the aforementioned lowest mean cost function value is the set of parameters on which either 250 or 500 restarts, depending upon the size of the problem, are run. The number of restarts in stage two is 250 for the larger problems Flare and Mackey-Glass and 500 for the remaining smaller problems. The results from these restarts are those that are reported in Chapter 5.

Tables 4.2 and 4.3 present some of the details of the above-described procedure for each of the algorithms and training data sets. Table 4.2 lists the range of values

Table 4.2 Range of Values Investigated for Parameters of the Global Algorithms.

Algorithm	Parameter	Range of Values
NNGA	b (bias)	2, 5, 10, 20
	r_c (probability of crossover)	0.80, 0.20
	r_m (probability of mutation)	0.20, 0.60, 0.80
	s (standard deviation of mutation)	0.03, 0.06, 0.12, 0.25, 0.50, 1.00
EVOL	s (standard deviation of mutation)	0.03, 0.06, 0.12, 0.25, 0.50, 1.00
	a_s (adjustment factor for s)	0.85, 0.99
KORR1-4	τ' (adjustment factor for s)	$1/\sqrt{2n}$, $1/(2\sqrt{2n})$, $1/(4\sqrt{2n})$
	τ (adjustment factor for s)	$1/\sqrt{2\sqrt{n}}$, $1/(2\sqrt{2\sqrt{n}})$, $1/(4\sqrt{2\sqrt{n}})$
	s (standard deviation of mutation)	0.03, 0.06, 0.12, 0.25, 0.50, 1.00
SW	s (standard deviation of mutation)	0.03, 0.06, 0.12, 0.25, 0.50, 1.00

Note: For a detailed explanation of the algorithms and their parameters, see section 4.4.2 for the NNGA algorithm, section 4.4.3 for the EVOL and KORR algorithms, section 4.4.5 for the SW algorithm.

investigated for certain parameters of the various global algorithms. The standard deviation of mutation s, is a parameter common to all the algorithms in table 4.2. The value for s is an important parameter. A value too large can result in saturation of the hidden neurons. Saturation occurs when large weight values cause most or all of the hidden neurons to attain values at or near their threshold values, for example, 0 or 1 for the sigmoid activation function given in (4.2). It can be hard for neurons to come off their saturated levels because a large change in relevant weights may be necessary. A small change in a weight or weights may not be enough to bring the activation levels down enough to come off saturation. On the other hand, a standard deviation of mutation value that is too small may result in a longer time than necessary for the ES to obtain satisfactory results. More importantly, the algorithm may never obtain satisfactory results because it fails to explore a wide enough area of the model parameter space.

It should be noted that a key feature of the evolutionary strategies type of algorithm is the ability to adjust the standard deviation of mutation as the algorithm progresses. Nonetheless, different values for the beginning standard deviation of mutation for the evolutionary strategies EVOL and KORR1 through KORR4 are investigated in this research. The reasons given above for trying various values for the beginning standard deviation of mutation may still apply, albeit to a lesser extent than for the other algorithms that do not adapt the standard deviation as the algorithm progresses.

Table 4.3 shows the number of preliminary configurations and restarts as well as the final number of restarts for each of the algorithms and data sets. For example, for the NNGA algorithm, 72 configurations representing various combinations of algorithm parameters are investigated for each of the data sets. Depending upon the size of the neural network model, either 5 or 10 random restarts or runs are estimated for each of these configurations. Based upon the mean cost function values computed across the preliminary restarts, the top performing preliminary configuration of algorithm parameters is chosen to run a full scale number of restarts, either 250 or 500, depending upon the size of the neural network models. As can be seen in table 4.3, there are no preliminary configurations for the local optimization routine, LO, and therefore, depending upon the size of the problem, only 250 or 500 final restarts are run for a single configuration. The procedure for choosing the parameters of the two simulated annealing algorithms, SA1 and SA2, does not follow exactly with that of the other global algorithms. The procedure for picking these values is discussed in the appropriately labeled sections to follow.

It should be noted that the sort of procedure described above for choosing the parameters of the algorithms gives an unfair advantage to the global algorithms. It could be argued that in practice, this sort of computationally demanding procedure is infeasible. However, in this research, if the local optimization routine outperforms the global

Table 4.3 Number of Preliminary Configurations and Restarts and Final Restarts for Each of the Algorithms and Data Sets.

Algorithm	Preliminary Configurations	Data Set	Number of Preliminary Restarts	Number of Final Restarts
LO	-	Bilinear	-	500
		Dax	-	500
		JYUS	-	500
		JYUSTTR	-	500
		M-G	-	250
		Flare	-	250
NNGA	72	Bilinear	10	500
		Dax	10	500
		JYUS	10	500
		JYUSTTR	10	500
		M-G	5	250
		Flare	5	250
EVOL	14	Bilinear	10	500
		Dax	10	500
		JYUS	10	500
		JYUSTTR	10	500
		M-G	5	250
		Flare	5	250
KORR1	18	Bilinear	10	500
		Dax	10	500
		JYUS	10	500
		JYUSTTR	10	500
		M-G	5	250
		Flare	5	250
KORR2	18	Bilinear	10	500
		Dax	10	500
		JYUS	10	500
		JYUSTTR	10	500
		M-G	5	250
		Flare	5	250
KORR3	18	Bilinear	10	500
		Dax	10	500
		JYUS	10	500
		JYUSTTR	10	500
		M-G	5	250
		Flare	5	250

(continued)

Table 4.3 (continued) **Number of Preliminary Configurations and Restarts and Final Restarts for Each of the Algorithms and Data Sets.**

Algorithm	Preliminary Configurations	Data Set	Number of Preliminary Restarts	Number of Final Restarts
KORR4	18	Bilinear	10	500
		Dax	10	500
		JYUS	10	500
		JYUSTTR	10	500
		M-G	5	250
		Flare	5	250
SA1	-	Bilinear	5	500
		Dax	5	500
		JYUS	5	500
		JYUSTTR	5	500
		M-G	5	250
		Flare	5	250
SA2	-	Bilinear	5	500
		Dax	5	500
		JYUS	5	500
		JYUSTTR	5	500
		M-G	5	250
		Flare	5	250
SW	6	Bilinear	10	500
		Dax	10	500
		JYUS	10	500
		JYUSTTR	10	500
		M-G	5	250
		Flare	5	250

routines, the difficulty of choosing the global optimization algorithm parameters would add weight to the favorable results obtained by the local algorithm. The following sections outline, for each of the algorithms, either the values for specific algorithm parameters, or the details of the procedures to obtain them.

NNGA

The parameters for the neural network genetic algorithm (NNGA) that remain to be determined are the values for p, the size of the population, b, the bias, s, the standard deviation of mutation, and r_c and r_m, the probability of crossover and mutation. The size of the population, p, is fixed at 50 as it was in the study by Rooij, Jain and Johnson. This population size is a trade-off between a smaller population size which would have a faster convergence and a larger population which would provide a higher probability of obtaining a good solution at the expense of higher computational costs. A range of values for the bias parameter b are investigated. Rooij, Jain, and Johnson used a value of 10 in their study. This value is investigated along with values of 2, 5, and 20.

Rooij, Jain, and Johnson reported good results with mutation from .4 to .8 and settled on a rate of .6. As discussed in chapter 3, these rates of mutation are high in comparison to the values normally used for binary genetic algorithms. Rooij, Jain, and Johnson indicate that settings of this magnitude produced the best results in their simulations. Mutation on binary chromosomes produces a high degree of disruption since the gene in question is switched to its opposite state. Mutation on real-valued chromosomes through the addition of a probabilistic value is less disruptive and therefore a higher mutation rate is feasible. Rooij, Jain, and Johnson reported that there was little difference in performance between values of .6, .8, and 1.0 for the probability of crossover for the neural network specific uniform crossover operator. Three values are investigated in this research .8, .6, and a relatively low value of .2. A low value for the probability of crossover of .2, in conjunction with a high probability of mutation, causes the NNGA to approach the operation of a multi-membered evolutionary strategy. Table 4.2 lists the range of values investigated for the parameters of the NNGA algorithm

EVOL

Table 4.2 lists the range of values investigated for two parameters of the EVOL algorithm, s, the standard deviation of mutation, and c_σ, an adjustment factor for the standard deviation of mutation. Two values for c_σ are considered, .85 and .99. Schwefel (1995, pg 368) recommends a value of .85 for c_σ and Keane states that for highly multi-modal functions, a value of .99 or even higher, improves performance of the EVOL algorithm. Section 4.3.3 gives a detailed explanation of the EVOL algorithm. Two fixed values for the EVOL algorithm are not detailed in section 4.3.3. These are listed as arguments EA and EB to the EVOL subroutine in Schwefel (1995, pg. 368). These values are both set to $\sqrt{\varepsilon}$, where ε is the double precision machine epsilon (approximately 2.22E-16). The values for the parameters LS, TM, EC, and ED for the EVOL subroutine in Schwefel are not relevant because the code was modified to suppress the intrinsic convergence tests.

KORR1, KORR2, KORR3, and KORR4

The distinguishing feature between the four KORR algorithms is the differing application of recombination. All other parameters for the algorithms are identical. The parameters τ and τ' listed in table 4.2 are used to determine the degree of adjustment of the standard deviations of mutation in each generation. See section 4.3.3 for more details of these parameters. Schwefel (1995) recommends values of $\tau' = c/\sqrt{2 \cdot n}$ and $\tau = c/\sqrt{2\sqrt{n}}$ with a value of $c = 1$ likely to work well for the KORR implementations used in this study. For each of the parameters τ and τ', two other values which are ½ and ¼ of the recommended values are also investigated. Six different values for the beginning standard deviation of mutation s are listed in table 4.2.

The fixed parameters for the KORR algorithms are the number of parents in the population, μ, the number of offspring produced, λ as used in the specification of the algorithms in section 4.3.3. Typical values for μ and λ would be 10 and 100 respectively (Schoenauer and Michalewicz) representing a $(10,100)-\mathrm{ES}$ scheme. However, since some of the neural network models in this research contain a large number of parameters and the calculation of the cost functions are expensive, the number of offspring λ will be limited in this study. Schwefel (1995, pg. 145) recommends that the ratio λ/μ should not be less than 5 or 6. Values of $\mu = 10$ and $\lambda = 60$ are used for all KORR algorithms. Therefore, all KORR algorithms in this research will implement a $(10,60)-\mathrm{ES}$ scheme.

SA1 and SA2

To establish appropriate values for the various parameters of the simulated annealing routines SA1 and SA2, a procedure using guidelines given in Masters (1995, pg. 89) is followed. These guidelines, quoting Masters (1995), are as follows:

- The acceptance rate should be high at first. Many experts recommend about 80 percent of trial points be accepted in the early stages. Choose the user scale accordingly.
- After annealing has progressed for a while, the acceptance rate should have dropped to a fairly low value. Failure to do so often indicates a problem. The user scale may be too high, the temperature may be dropping too quickly, or the perturbations may be inappropriately scaled.
- When annealing has progressed to the point of diminishing returns, the acceptance rate will usually stabilize around some moderate asymptote. If it is still dropping, progress is being made.

The user scale c, as given in the explanation of the SA1 and SA2 algorithms in section 4.3.4, is automatically estimated. Therefore, the relevant parameters for the simulated annealing that can be set to affect the above mentioned factors are the beginning and final temperatures $T_a^{(0)}$, $T_p^{(0)}$, and $T_p^{(M)}$, and the ratio of the ending acceptance criterion and parameters temperatures given by $\alpha = T_a^{(M)} / T_p^{(M)}$. These parameters are varied to get the beginning acceptance criterion between 70 and 80 percent and the ending acceptance criterion below 20 percent.

Solis and Wets

The SW algorithm only has one adjustable algorithm parameter, s, the standard deviation of mutation. Table 4.2 lists the range of values investigated for these parameters. See section 4.3.5 for a detailed explanation of the SW algorithm.

4.5.2 Algorithm Evaluation and Comparison

The purpose of the comparison of various optimization algorithms in this research is to determine the relative speed and accuracy of alternative global optimization methods in estimating the weights of neural networks. To accomplish this objective, the performance of each of the algorithms is evaluated through Monte-Carlo simulations. Each of the algorithms for each data set are retrained numerous times from different random starting points. The final cost function values for each of these runs are saved and various statistics are then computed from these values. The mean, median, and standard deviation across the runs will be presented in chapter 5. In addition to the statistics concerning the different runs, the results of the runs are also presented graphically. The distribution of the cost function values after convergence over the different runs will be displayed in histogram format. In addition to the histogram displays, box plots will be displayed to better compare the distributions from different algorithms.

4.5.3 Neural Network Software and Computing Environment

The neural network software was programmed in Fortran 90 using Compaq's Visual Fortran, version 6.6A compiler for Windows. Excluding the evolutionary strategy algorithms, as explained in section 4.3.3, all code was written by the author. The simulations are performed on two computers, each using a single 1 Ghz Intel PIII processors with 256 Meg of memory and running the Windows 2000 operating system.

All reasonable efforts were made to optimize the code for speed, both with compiler optimization switches as well as efficient coding. Where applicable, use was made of the optimized Math Kernel Libraries (MKL) from Intel. The MKL contains vector math functions that are highly optimized for Intel processors. The MKL includes optimized functions from the BLAS libraries, which perform vector and matrix multiplies, and vectorized transcendental functions, which were used for the activation functions of the neural networks.

CHAPTER 5

RESULTS AND DISCUSSION

5.1 Introduction

This chapter presents the results of the simulations carried out to accomplish the research objectives given in section 1.4. Section 5.2 briefly discusses the selection of the user-definable parameters of the stochastic global optimization algorithms. These parameters were obtained by the procedure given in section 4.4.1. The specific values for the parameters are presented and discussed in more detail in appendix A. Section 5.3 presents the results of the comparisons of the global optimization algorithms against the local optimization algorithm. Finally, Section 5.4 summarizes the results and presents the conclusions in the context of the research objectives.

5.2 Stochastic Global Optimization Algorithm Parameters

The performance of the stochastic global optimization algorithms may depend upon wisely choosing the values for various user-definable algorithm parameters. Section 4.4.1 presented the details of the procedure for picking a good combination of algorithm parameters from a set of many combinations of algorithm parameters. This section will discuss the algorithm parameters only briefly. The specific values of the parameters and a more detailed discussion of them are given in Appendix A.For most of the algorithm parameters, it is difficult to see any discernable pattern in the values chosen. However, a few general observations can be made about some of the parameters. With some exceptions, it appeared that a smaller value for the standard deviation of mutation was most appropriate. Values as low as .03 were optimal for some of the problems. The adjustment factor for the standard deviation of mutation in the EVOL algorithm, given by a_s, was with one exception chosen to be .999. The rather small adjustment of the standard deviation of mutation in each iteration is consistent with the results of Keane who found that smaller adjustments were required in highly multi-modal problems. For the NNGA algorithm, a rather high rate of probability of mutation was chosen for the majority of the problems.

A set of parameters that works well for one neural network model is unlikely to work for another. The inconsistency of performance for a given set of algorithm parameters across different optimization problems is a drawback of the stochastic global algorithms. The reader is also reminded that a computationally expensive procedure, such as the one used in this study to determine reasonable value for the global optimization algorithm parameters, would be impractical to use in most modeling situations. This extra computational effort is essentially ignored in the remainder of the results presented in this chapter. Therefore, the stochastic global algorithms are theoretically given an unfair advantage over restarts of a local optimization routine. However, if a local optimization routine outperforms the global algorithms, it only adds weight to the results.

5.3 Simulation Results

The simulation results presented in this section are from algorithm runs in which the local or hybrid global/local algorithm converged. Convergence was reached either through a test such as the magnitude of the gradient, or by reaching the maximum number of iterations as given in section 4.4.1 of this study. No algorithm run "bombed" or produced floating-point exceptions that would have caused the algorithm to cease operation.

Figures 5.1-5.6 show for each of the 6 training data sets the cost function values in histogram form for each of the 10 optimization routines. For the Bilinear, DAX, JYUS, and JYUSTTR data sets, the histograms contain 500 objective function values after convergence. For the Flare and Mackey-Glass data sets, the histograms contain 250 objective function values. To make it easier to compare the various optimization algorithms, the x-axis is scaled identically for each of the 10 histograms displayed in each figure. The labels for each histogram correspond with the particular optimization algorithm used. The abbreviation LO represents the particular local optimization routine employed in this training data set: for the Flare and Mackey-Glass data sets a conjugate gradient algorithm and for the rest of the data sets a quasi-Newton algorithm. The remaining algorithms are global optimization algorithms with the following meaning: NNGA – a neural network specific genetic algorithm, SW – the Solis-Wets algorithm, EVOL – an evolutionary strategy (ES) from Schwefel (1995), KORR1 through KORR4 – 4 different variations of ES algorithms from Schwefel (1995), and SA1 and SA2 are two variations of simulated annealing. See section 4.4 for details of the algorithms. The numbers above the left two bins are the percentage of values contained in each of the two

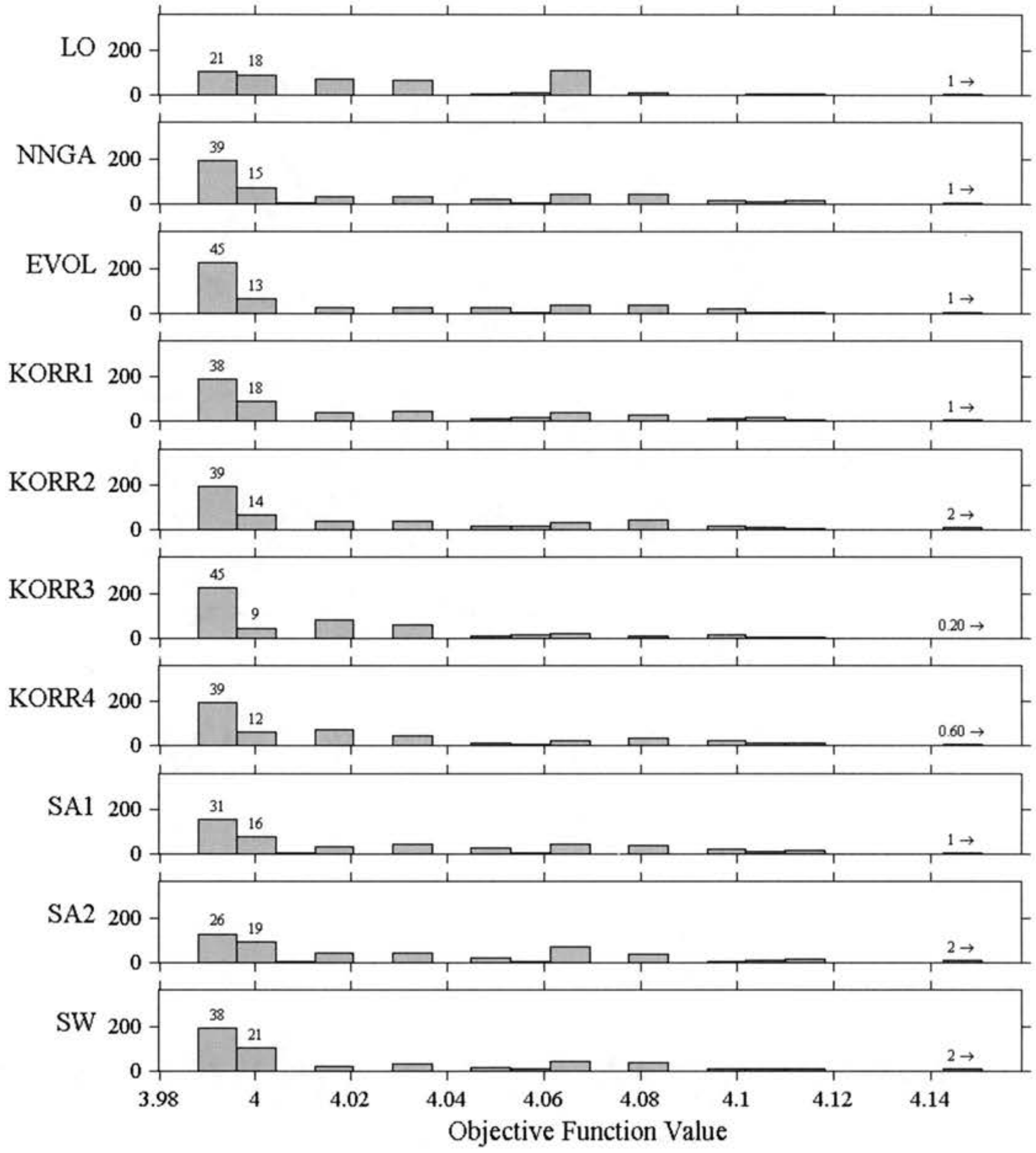

Figure 5.1 Histograms of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the Bilinear Training Data. Each of the nine histograms above contain 500 objective function values after convergence from 500 different random starting values. See table 1 for an explanation of the labels in each histogram representing the various optimization algorithms. The numbers above the left two bins are the percentage of values contained in each of the two bins. If a number with an arrow appears over the far right bin, it indicates that the histogram has been truncated to better display the results. In that case, the number is the percentage of values contained in and to the right of the bin.

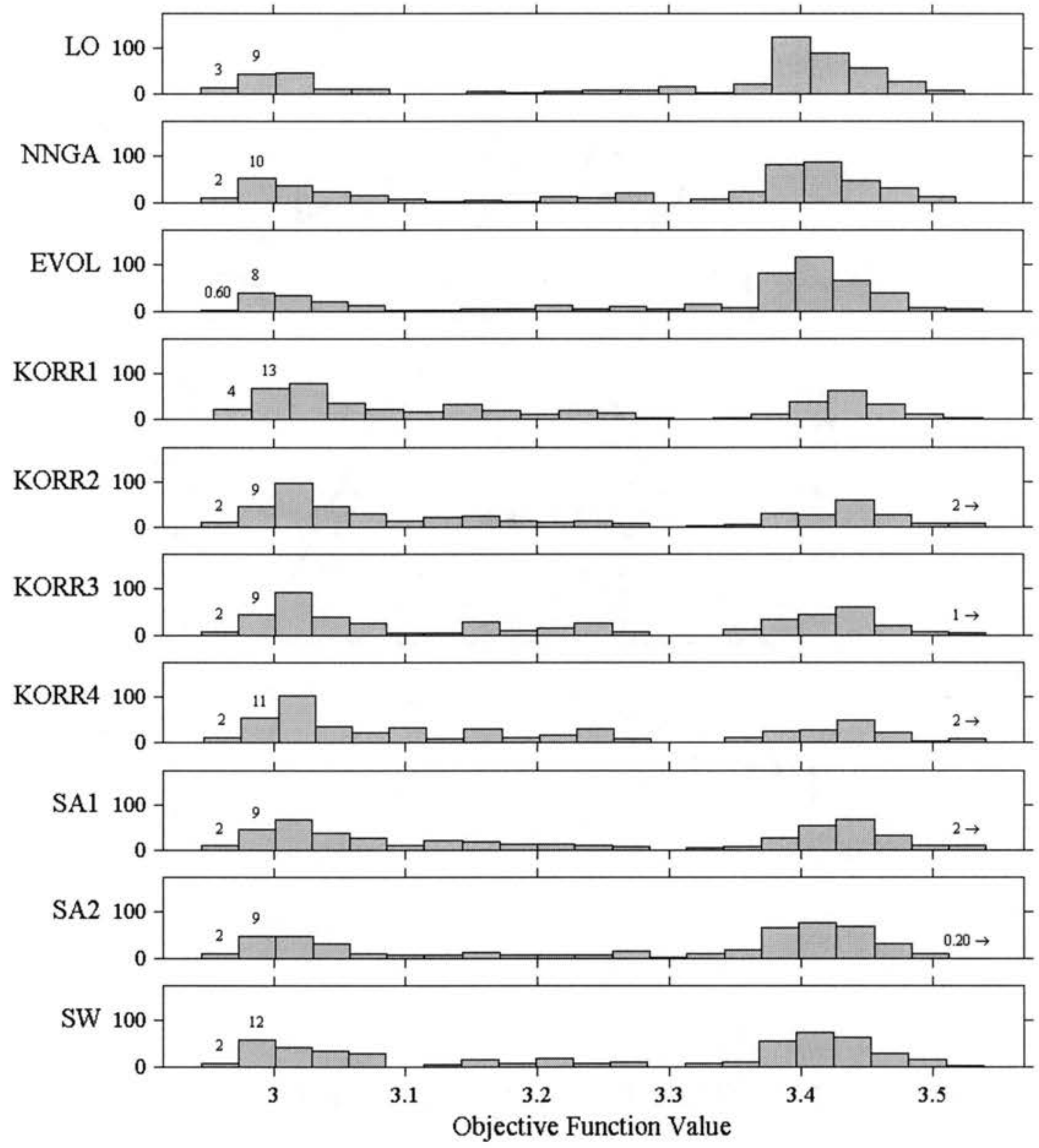

Figure 5.2 Histograms of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the Dax Training Data. See figure 5.1 for a more detailed explanation of the information in this figure.

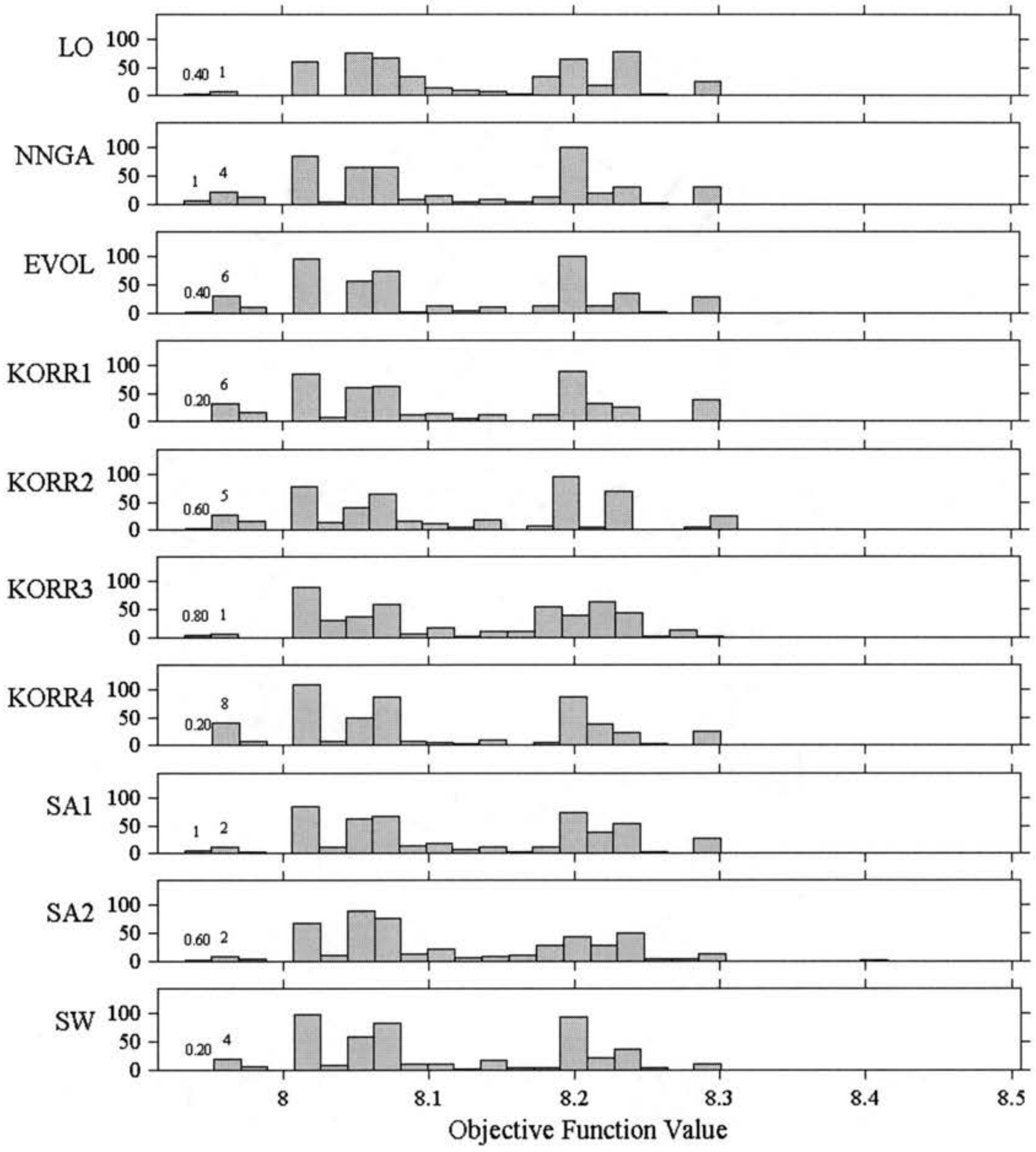

Figure 5.3 Histograms of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the JYUS Training Data. See figure 5.1 for a more detailed explanation of the information in this figure.

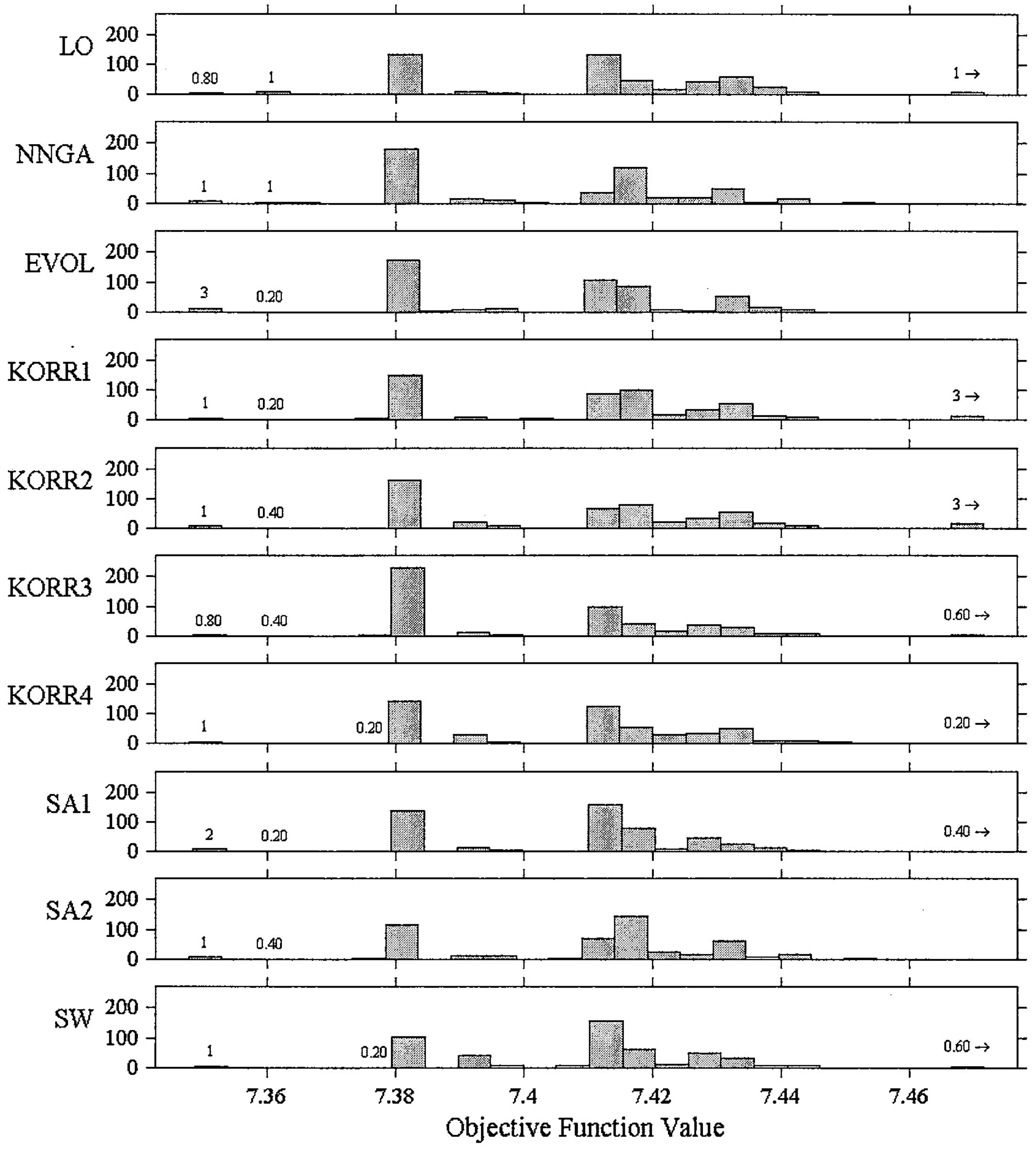

Figure 5.4 Histograms of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the JYUSTTR Training Data. See figure 5.1 for a more detailed explanation of the information in this figure.

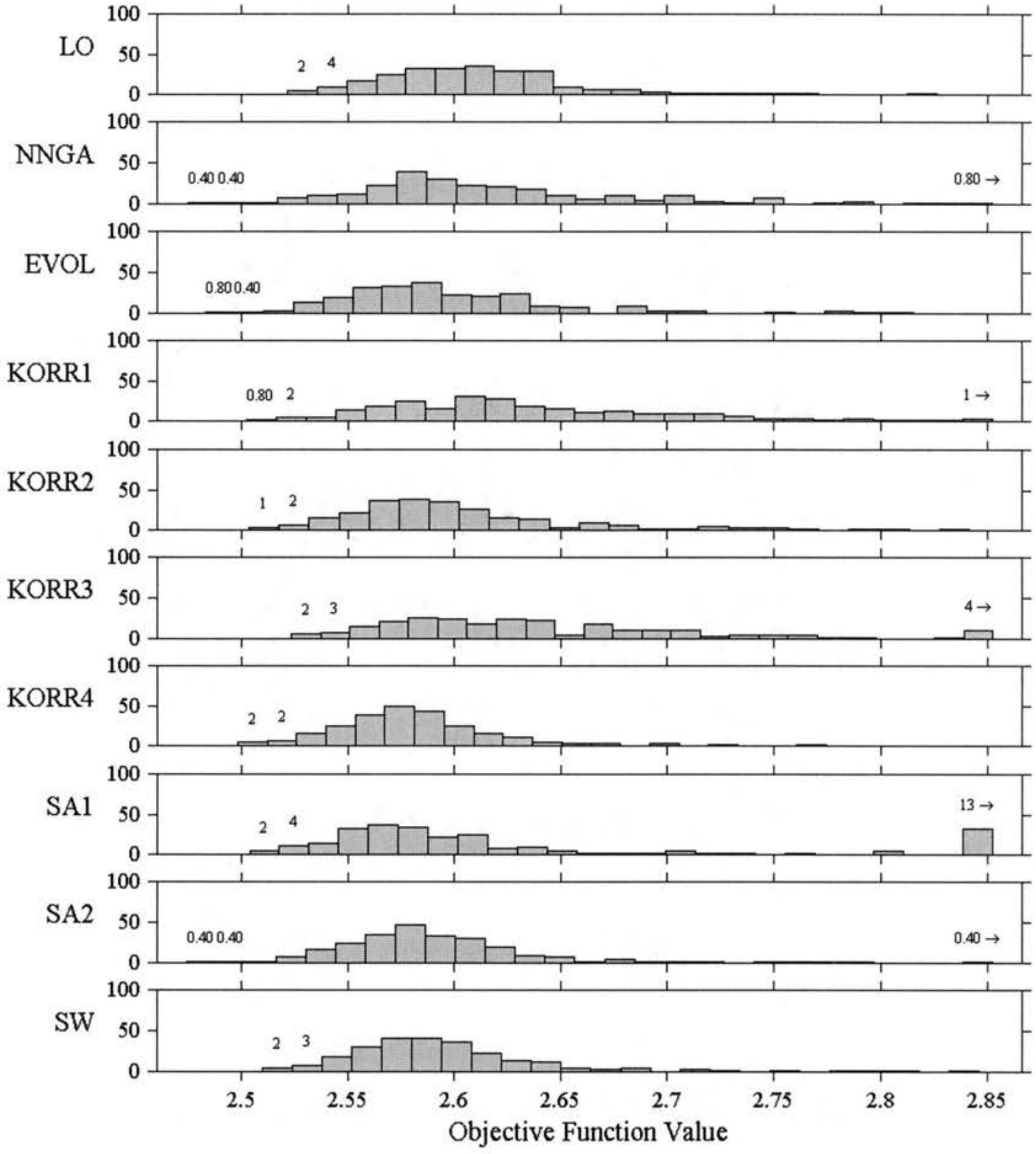

Figure 5.5 Histograms of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the Flare Training Data. See figure 5.1 for a more detailed explanation of the information in this figure.

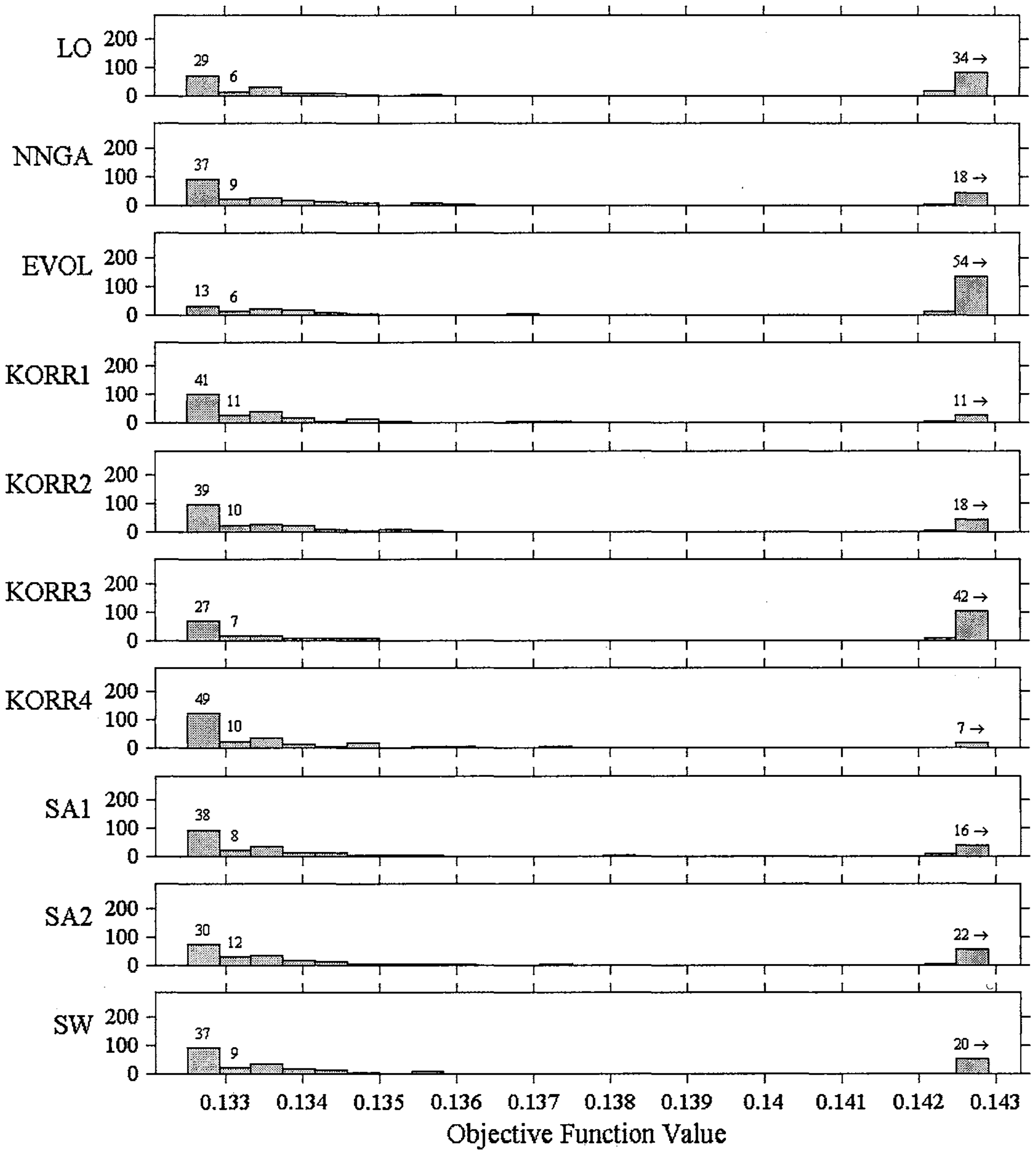

Figure 5.6 Histograms of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the Mackey-Glass Training Data. See figure 5.1 for a more detailed explanation of the information in this figure.

bins. If a number with an arrow appears over the far right bin, it indicates that the histogram has been truncated to better display the results. In that case, the number is the percentage of values contained in and to the right of the bin.

As can be seen from the histograms in figures 5.1-5.6, there is no single algorithm that dominates all others across the training data sets. The box plots in figures 5.7-5.12 provide an alternative way to characterize the information contained in the histograms. The box plots also show that no single algorithm dominates all others across the trading data sets. The box plots indicate the median, upper, and lower quartile, upper and lower adjacent values, and outside values. The median is displayed as a solid dot and left and right end of the boxes indicate the upper and lower quartiles. The upper and lower adjacent values, and outside values are based on the fences. An upper fence would be calculated as the upper quartile plus 1.5 times the interquartile range, where the interquartile range is the upper quartile minus the lower quartile. The lower fence would be analogously calculated. An upper adjacent value is the maximum point within the upper fence. The vertical lines outside the box indicate the adjacent values. Any value that falls outside the adjacent values or vertical lines is considered an outside value and is plotted with an open circle in figures 5.7-5.12. Therefore, the minimum and maximum objective function values obtained by the respective algorithms are displayed as vertical lines, or if necessary the most extreme open circle.

The histograms and boxplots do indicate that a large number of local optimums exist for all the neural network training data sets. The histograms and boxplots display the unique characteristics of each of the data sets. The histograms do not greatly differ

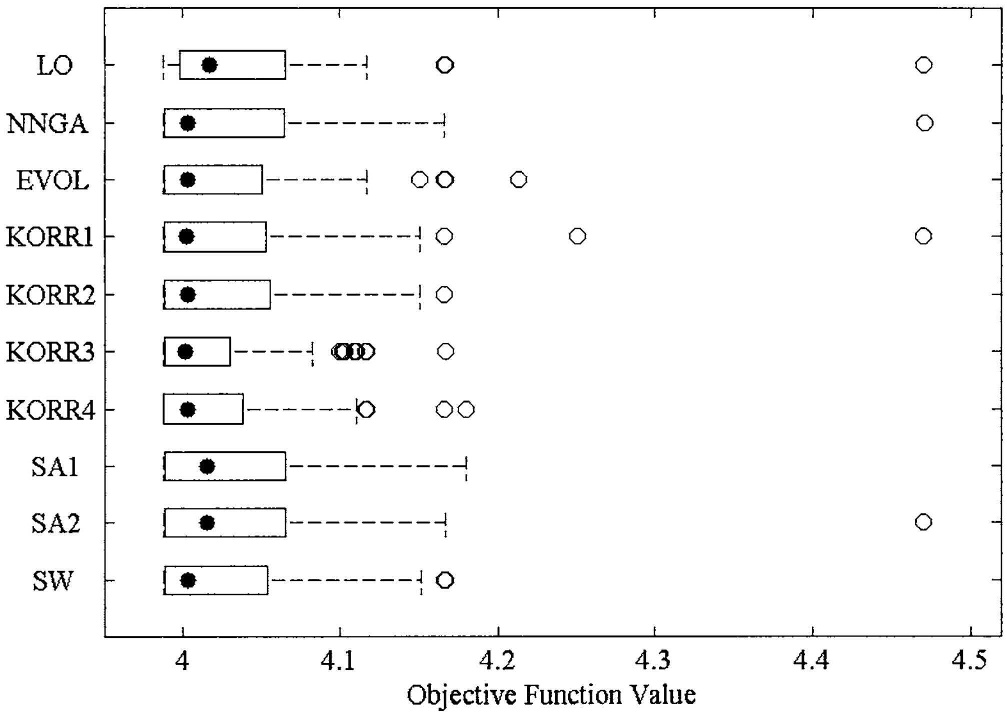

Figure 5.7 Boxplot of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the Bilinear Training Data. The boxplots indicate the median, upper and lower quartiles, upper and lower adjacent values, and outside values. In the box plot, the solid dot indicates the median and the right and left ends of the box are the upper and lower quartiles. The vertical lines or whiskers outside the box mark the highest (lowest) data points within a range defined by the upper (lower) quartile + (-) 1.5 times the interquartile range. Any values outside of the whiskers are considered outside values and are plotted by open circles.

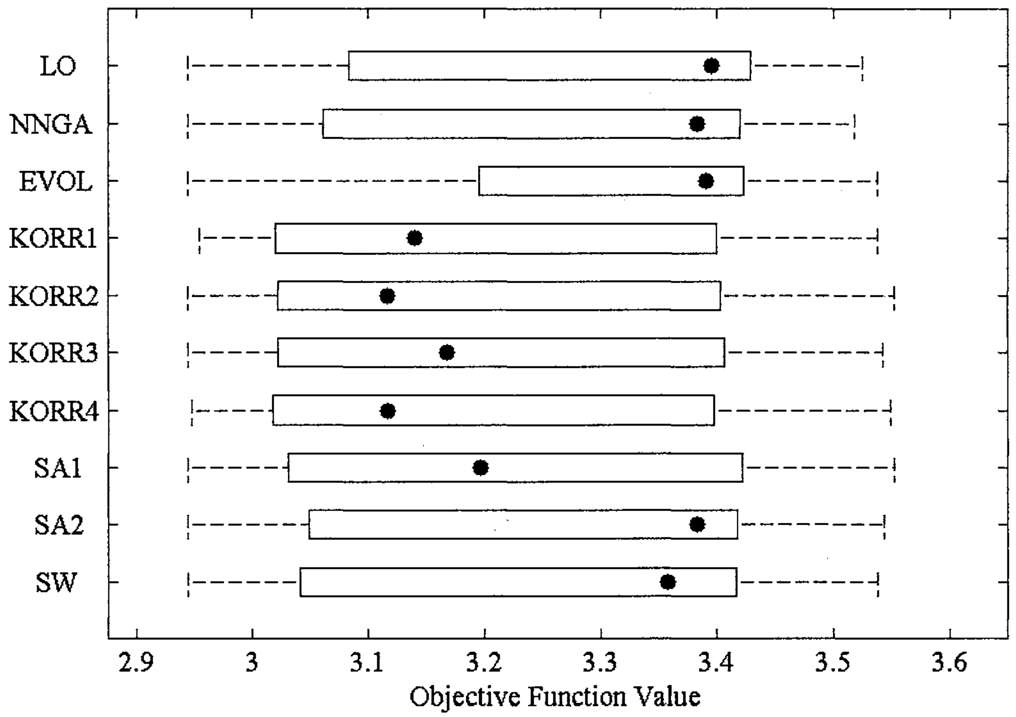

Figure 5.8 Boxplot of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the Dax Training Data. See figure 5.7 for a more detailed explanation of the information in this figure.

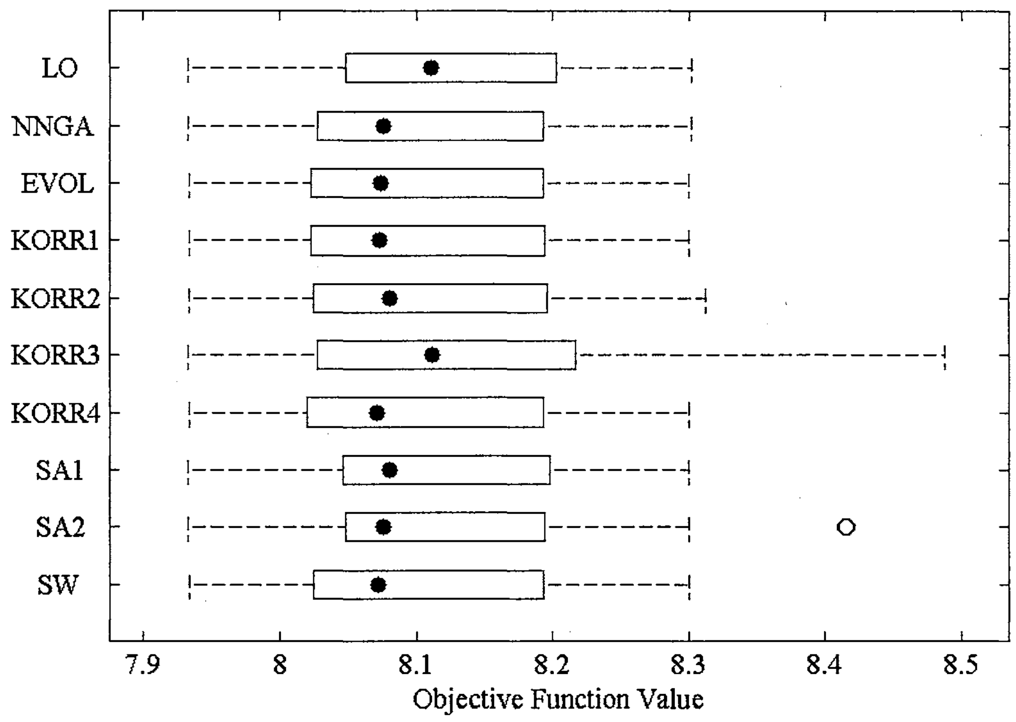

Figure 5.9 Boxplot of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the JYUS Training Data. See figure 5.7 for a more detailed explanation of the information in this figure.

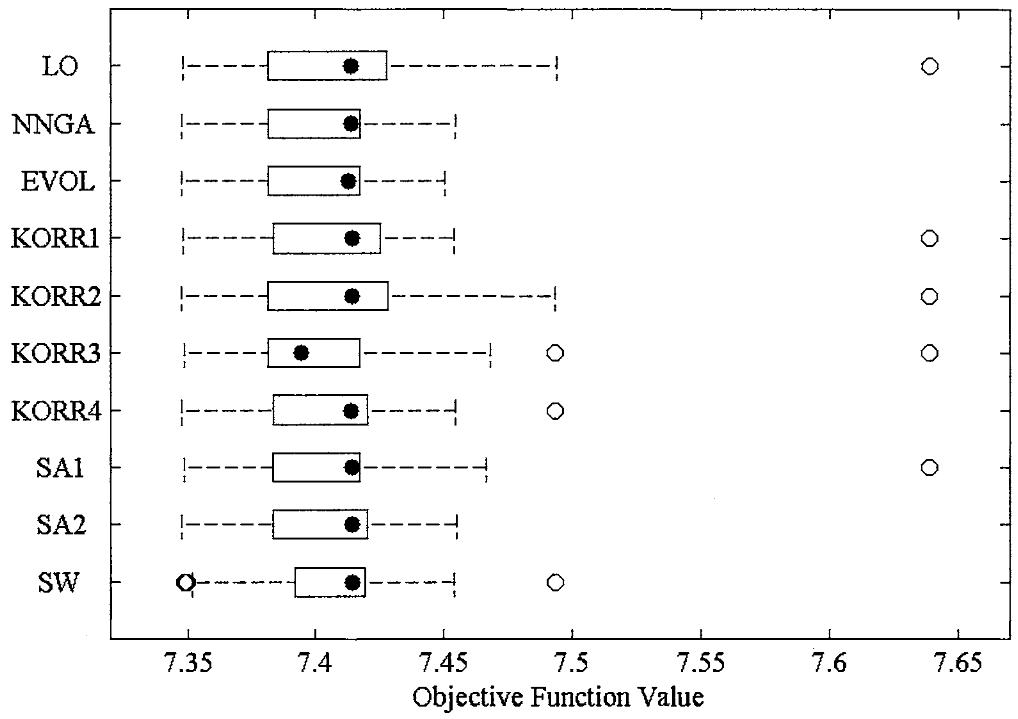

Figure 5.10 Boxplot of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the JYUSTTR Training Data. See figure 5.7 for a more detailed explanation of the information in this figure.

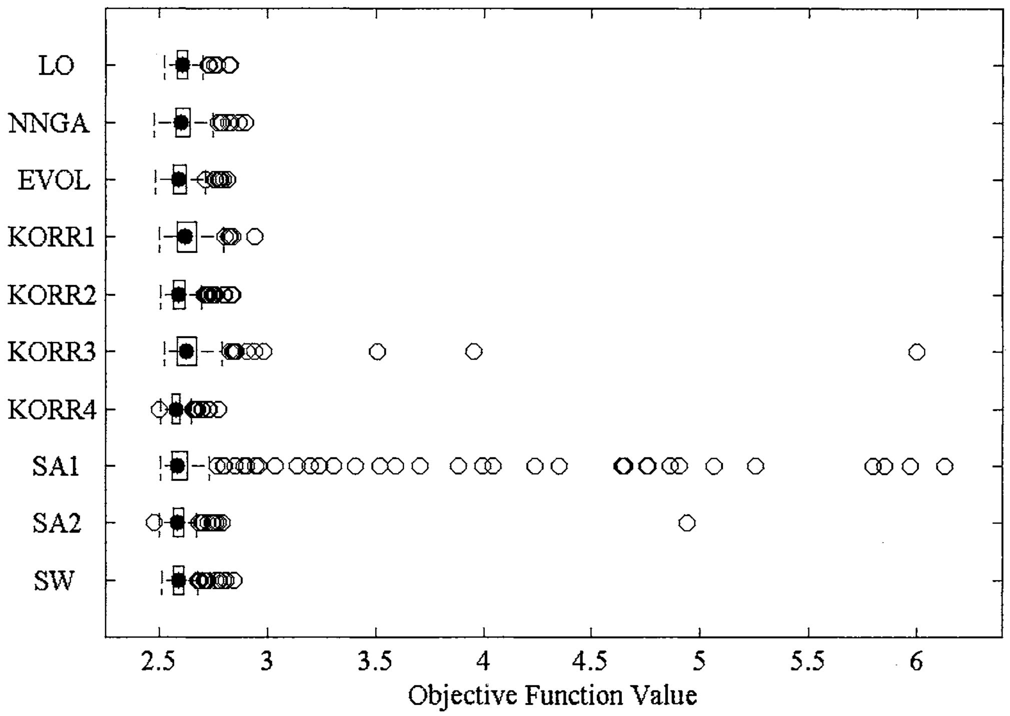

Figure 5.11 Boxplot of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the Flare Training Data. See figure 5 for a more detailed explanation of the information in this figure.

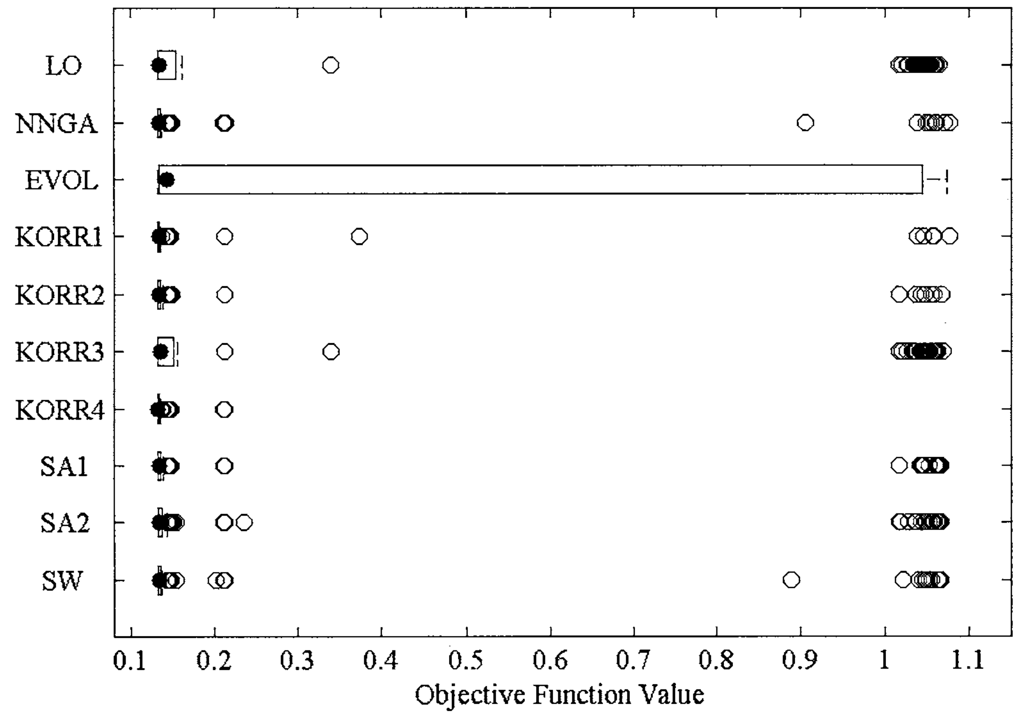

Figure 5.12 Boxplot of Objective Function Values from Random Restarts of Different Optimization Algorithms for Neural Network Training on the Mackey-Glass Training Data. See figure 5 for a more detailed explanation of the information in this figure.

from each other across algorithms within the same data set but do differ greatly across different data sets. The histogram on the Flare data set is the only one that approaches anything resembling a Normal distribution but it does have a left skew and outliers on the right. Some of the distributions are very obviously bimodal. As can be seen in figure 5.6, the most bimodal of the results are on the Mackey-Glass data set. The maximum and minimum values obtained for all but the KORR4 algorithm are slightly over 1.0 and a little more then .13. This range of values can, however, be misleading. To get a better idea of the difference in fit between the models associated with the maximum and minimum objective function values, table 5.1 lists the r-squared and adjust r-squared values for all the algorithms and data sets. For the Mackey-Glass data set the r-squared goes down from a near perfect fit at .999 to a little less then .97. Larger drops can be seen in some of the other data sets, especially when considering the adjusted r-squared. For example, the DAX data shows a large drop. The neural network models on the JYUS and JYUSTTR data sets show a relatively poor fit.

From looking at the histograms and boxplots, it can be seen that no single algorithm consistently outperforms all others. More importantly, with respect to the research objectives of this study as defined in section 1.4, the local optimization algorithm is not consistently dominated by any of the global algorithms. However, the global algorithms do provide on average marginally more probability of obtaining a lower converged objective function value as opposed to the local optimization routines. However, with one exception, the local routine obtained a sufficient number of convergences at the minimum value, or very near the minimum. This statement is examined more closely in the following paragraph.

TABLE 5.1 **R-Squared Across all Optimization Algorithms and Data Sets for Neural Network Models with the Minimum and Maximum Object Function Values Across Restarts.**

	Bilinear				DAX				JYUS			
	Minimum Obj Func Value		Maximum Obj Func Value		Minimum Obj Func Value		Maximum Obj Func Value		Minimum Obj Func Value		Maximum Obj Func Value	
Algorithm	R^2	Adj R^2	R^2	Adj R^2	R^2	Adj R^2	R^2	Adj R^2	R^2	Adj R^2	R^2	Adj R^2
LO	.310	.269	.218	.172	.334	.278	.155	.084	.102	.066	.055	.018
NNGA	.310	.269	.218	.171	.334	.278	.150	.078	.102	.066	.057	.019
EVOL	.310	.269	.272	.228	.334	.278	.144	.072	.102	.066	.057	.019
KORR1	.310	.269	.218	.172	.334	.278	.144	.072	.102	.066	.057	.019
KORR2	.310	.269	.274	.231	.333	.277	.144	.072	.102	.066	.057	.019
KORR3	.310	.269	.274	.231	.335	.279	.149	.077	.102	.066	.040	.001
KORR4	.310	.269	.274	.231	.334	.278	.144	.072	.102	.066	.057	.019
SA1	.310	.269	.274	.231	.335	.280	.147	.075	.102	.066	.057	.019
SA2	.310	.269	.218	.172	.335	.279	.143	.071	.102	.066	.044	.006
SW	.310	.269	.274	.231	.334	.278	.144	.072	.102	.066	.057	.019

Note: The Adj R^2 heading is the Adjusted R^2.

(continued)

TABLE 5.1 (Continued) **R-Squared Across all Optimization Algorithms and Data Sets for Neural Network Models with the Minimum and Maximum Object Function Values Across Restarts.**

	JYUSTTR				Mackey-Glass				Flare			
	Minimum Obj Func Value		Maximum Obj Func Value		Minimum Obj Func Value		Maximum Obj Func Value		Minimum Obj Func Value		Maximum Obj Func Value	
Algorithm	R^2	Adj R^2	R^2	Adj R^2	R^2	Adj R^2	R^2	Adj R^2	R^2	Adj R^2	R^2	Adj R^2
LO	.055	.012	.002	-.043	.999	.999	.968	.966	.648	.418	.605	.347
NNGA	.058	.015	.029	-.015	.999	.999	.968	.965	.654	.428	.595	.331
EVOL	.054	.012	.030	-.014	.999	.999	.968	.965	.653	.427	.607	.350
KORR1	.058	.015	.002	-.043	.999	.999	.969	.966	.650	.422	.589	.321
KORR2	.055	.012	.002	-.043	.999	.999	.968	.966	.650	.422	.603	.344
KORR3	.055	.012	.002	-.043	.999	.999	.968	.965	.647	.417	.161	-.386
KORR4	.050	.007	.024	-.020	.999	.999	.997	.997	.651	.423	.612	.359
SA1	.055	.012	.002	-.043	.999	.999	.969	.966	.650	.422	.143	-.416
SA2	.055	.012	.030	-.014	.999	.999	.968	.966	.654	.429	.310	-.141
SW	.057	.015	.024	-.020	.999	.999	.969	.966	.650	.420	.602	.343

Note: The Adj R^2 heading is the Adjusted R^2.

Table 5.2 presents the probability of obtaining a solution within 5% and 20% cutoff points or percentiles. The cutoff points for the given percentiles are calculated by pooling all unique objective function values across all the algorithms for a given data set and then calculating the given percentiles. In other words, for each of the Bilinear, DAX, JYUS, and JYUSTTR data sets, a total of 5000 (500 restarts for each of the 10 algorithms) objective function values for each data set are created by pooling the final converged objective function values from each algorithm. For the Flare and Mackey-Glass problems, a total of 2500 objective function values are pooled since on these two data sets only 250 restarts were used for each of the algorithms.

Ideally, all the minimums could be enumerated. However, this obviously cannot be done analytically, therefore, the pooled minimums found across all the algorithms serve as a proxy for the set of all minimums. More precisely, if the pooled objective function values serve as an unbiased proxy for the range and distribution of minimums for the particular data set, then the calculated percentiles will be unbiased with respect to the true percentiles. In table 5.2 we can see that for the Bilinear training data set, 4.4% of the objective function values found by the local optimization routine were within the 5th percentile of the minimums found across all algorithms and similarly 14.2% of objective function values were within the 20th percentile. In comparison, for the Bilinear data set, 7 out of 9 of the global algorithms had higher probabilities of obtaining a minimum within the 5th percentile of all minimums found.

Further examining table 5.2, it can be seen that, on average, as compared to the local optimization routines, the global algorithms do show an increase in probability of obtaining a low objective function value. However, the degree to which this is true

Table 5.2 Probability of Obtaining an Objective Function Value Within the Given Percentile of All Objective Function Values Obtained Across All Algorithms for the Specific Data Set.

	Bilinear		DAX		JYUS	
Algorithm	5th percentile	20th percentile	5th percentile	20th percentile	5th percentile	20th percentile
LO	.044	.142	.050	.164	.020	.138
NNGA	.092	.194	.054	.176	.080	.218
EVOL	.074	.260	.022	.128	.086	.228
KORR1	.104	.190	.074	.252	.096	.240
KORR2	.090	.246	.056	.226	.090	.216
KORR3	.138	.302	.052	.232	.022	.178
KORR4	.112	.306	.078	.276	.094	.262
SA1	.038	.184	.056	.190	.038	.180
SA2	.040	.152	.066	.174	.032	.134
SW	.052	.234	.046	.190	.056	.212

	JYUSTTR		Flare		Mackey-Glass	
Algorithm	5th percentile	20th percentile	5th percentile	20th percentile	5th percentile	20th percentile
LO	.020	.138	.024	.128	.028	.152
NNGA	.080	.218	.052	.160	.048	.204
EVOL	.086	.228	.064	.244	.004	.076
KORR1	.096	.240	.028	.112	.040	.312
KORR2	.090	.216	.044	.220	.056	.232
KORR3	.022	.178	.024	.116	.024	.188
KORR4	.094	.262	.076	.316	.040	.396
SA1	.038	.180	.088	.272	.028	.232
SA2	.032	.134	.056	.228	.036	.160
SW	.056	.212	.044	.204	.044	.240

varied depending upon the data set and the percentile level. For all data sets, the local optimization routine either beat or was very close to at least one of the global optimization routines. The results in table 5.2, would tend to support the statement reported earlier: "the local routine obtained a sufficient number of convergences at the minimum value, or very near the minimum". This is especially true in light of the computational requirements of the global algorithms.

We may not only be interested in the distribution of objective function value. The absolute or best minimum objective function value is also important. Tables 5.3-5.8 display the following statistics computed across the restarts: mean, median, standard deviation, and maximum and minimum values obtained. For the Bilinear and Mackey-Glass data sets, all the algorithms found the same minimum. On the Dax data set, the local optimization routine found a lower minimum than 5 of the global optimization routines and matched the minimum found by the other global routines. For the JYUS and JYUSTTR data sets, the local optimization routine found a minimum that was only slightly larger then the lowest minimum found. On both data sets, the local optimization routine found lower minimums than several of the global routines.

Although the local optimization routines were very competitive with the global routines in finding the minimum objective function values, the local routines were outperformed by the global routines on most data sets with respect to the mean and median. The one exception was for the JYUSTTR data set in which the local routine outperformed all but one of the global routines with respect to the median. From looking at the histograms and boxplots in figures 5.1-5.12, the global routines did on average

Table 5.3 Statistics for Objective Function Values from Random Restarts of Optimization Algorithms for Training a Neural Network on the Bilinear Data Set.

Algorithm[a]	Mean	Median	Standard Deviation	Maximum	Minimum
LO	4.0280	4.0170	.0404	4.4703	3.9880
NNGA	4.0251	4.0032	.0472	4.4706	3.9880
EVOL	4.0213	4.0031	.0422	4.2130	3.9880
KORR1	4.0232	4.0022	.0498	4.4700	3.9880
KORR2	4.0247	4.0031	.0436	4.1663	3.9880
KORR3	4.0145	4.0019	.0341	4.1664	3.9880
KORR4	4.0222	4.0031	.0406	4.1795	3.9880
SA1	4.0295	4.0153	.0438	4.1795	3.9880
SA2	4.0316	4.0156	.0505	4.4703	3.9880
SW	4.0225	4.0031	.0429	4.1666	3.9880

Note: The numbers in the table above represent the indicated statistics computed across 500 restarts of the algorithms from new random starting values. The statistics are computed for the objective function values after the algorithm has indicated convergence for each of the restarts.

[a]The abbreviation LO represents the particular local optimization routine employed in this training data set, for example, a quasi-Newton algorithm in this case. The rest of the algorithms are global optimization algorithms with the following meaning: NNGA – a neural network specific genetic algorithm, SW – the Solis-Wets algorithm, EVOL – a evolutionary strategy (ES) from Schwefel (1995), KORR1 through KORR4 – 4 different variations of ES algorithms from Schwefel (1995), and SA1 and SA2 are two variations of simulated annealing. See 4.4 for details of the algorithms.

Table 5.4 Statistics for Objective Function Values from Random Restarts of Optimization Algorithms for Training a Neural Network on the DAX Data Set.

Algorithm	Mean	Median	Standard Deviation	Maximum	Minimum
LO	3.3004	3.3955	.1798	3.5238	2.9442
NNGA	3.2780	3.3830	.1815	3.5176	2.9442
EVOL	3.3063	3.3906	.1693	3.5379	2.9442
KORR1	3.1907	3.1394	.1835	3.5377	2.9536
KORR2	3.1938	3.1161	.1869	3.5516	2.9444
KORR3	3.2096	3.1674	.1849	3.5418	2.9442
KORR4	3.1803	3.1161	.1789	3.5490	2.9467
SA1	3.2263	3.1953	.1900	3.5516	2.9442
SA2	3.2667	3.3817	.1835	3.5429	2.9444
SW	3.2510	3.3568	.1865	3.5377	2.9444

Note: See table 5.3 for an explanation of the various entries.

Table 5.5 Statistics for Objective Function Values from Random Restarts of Optimization Algorithms for Training a Neural Network on the JYUS Data Set.

Algorithm	Mean	Median	Standard Deviation	Maximum	Minimum
LO	8.1297	8.1111	.0898	8.3018	7.9325
NNGA	8.1133	8.0756	.0970	8.3016	7.9324
EVOL	8.1096	8.0735	.0962	8.3005	7.9340
KORR1	8.1122	8.0731	.0995	8.3005	7.9335
KORR2	8.1170	8.0808	.0986	8.3121	7.9335
KORR3	8.1239	8.1113	.0914	8.4878	7.9324
KORR4	8.1028	8.0708	.0967	8.3005	7.9335
SA1	8.1210	8.0808	.0952	8.3005	7.9324
SA2	8.1159	8.0756	.0911	8.4153	7.9324
SW	8.1066	8.0721	.0886	8.3005	7.9340

Note: See table 5.3 for an explanation of the various entries.

Table 5.6 Statistics for Objective Function Values from Random Restarts of Optimization Algorithms for Training a Neural Network on the JYUSTTR Data Set.

Algorithm	Mean	Median	Standard Deviation	Maximum	Minimum
LO	7.4113	7.4138	0.0316	7.6387	7.3479
NNGA	7.4032	7.4138	0.0220	7.4547	7.3478
EVOL	7.4034	7.4130	0.0218	7.4504	7.3478
KORR1	7.4139	7.4144	0.0435	7.6387	7.3479
KORR2	7.4121	7.4142	0.0444	7.6387	7.3478
KORR3	7.4020	7.3945	0.0247	7.6387	7.3485
KORR4	7.4071	7.4138	0.0209	7.4932	7.3478
SA1	7.4070	7.4142	0.0227	7.6387	7.3485
SA2	7.4088	7.4142	0.0203	7.4549	7.3478
SW	7.4090	7.4142	0.0200	7.4932	7.3488

Note: See table 5.3 for an explanation of the various entries.

Table 5.7 Statistics for Objective Function Values from Random Restarts of Optimization Algorithms for Training a Neural Network on the Flare Data Set.

Algorithm	Mean	Median	Standard Deviation	Maximum	Minimum
LO	2.6096	2.6063	.0452	2.8260	2.5219
NNGA	2.6169	2.6003	.0660	2.8969	2.4752
EVOL	2.5975	2.5871	.0538	2.8150	2.4832
KORR1	2.6342	2.6209	.0678	2.9390	2.5023
KORR2	2.6016	2.5887	.0563	2.8413	2.5037
KORR3	2.6590	2.6227	.2459	6.0022	2.5236
KORR4	2.5815	2.5784	.0384	2.7745	2.4984
SA1	2.7890	2.5846	.6316	6.1310	2.5040
SA2	2.6009	2.5836	.1554	4.9396	2.4744
SW	2.5957	2.5878	.0492	2.8461	2.5101

Note: See table 5.3 for an explanation of the various entries.

Table 5.8 Statistics for Objective Function Values from Random Restarts of Optimization Algorithms for Training a Neural Network on the Mackey-Glass Data Set.

Algorithm	Mean	Median	Standard Deviation	Maximum	Minimum
LO	.3434	.1339	.3807	1.0637	.1325
NNGA	.1776	.1335	.1870	1.0770	.1325
EVOL	.4906	.1429	.4439	1.0733	.1325
KORR1	.1568	.1333	.1304	1.0756	.1325
KORR2	.1681	.1335	.1698	1.0673	.1325
KORR3	.3382	.1348	.3760	1.0694	.1325
KORR4	.1350	.1329	.0089	.2122	.1325
SA1	.1989	.1335	.2296	1.0657	.1325
SA2	.2252	.1336	.2687	1.0657	.1325
SW	.1997	.1335	.2285	1.0659	.1325

Note: See table 5.3 for an explanation of the various entries.

provide marginally more probability of obtaining a lower minimum. Therefore, we would expect slightly lower mean and median statistics for the global routines. However, the local optimization routine outperformed the global routines 63% of the time with respect to the standard deviation of objective function values.

Although the global routines did on average provide slightly more probability of obtaining a lower minimum, the local routines were nonetheless competitive. The local routines had a lower standard deviation of results and obtained minimum objective function values at the minimum. The one exception was for the Flare data set. As can be seen in figures 5.5 and 5.11, the Evolutionary Strategy routine KORR3 had a higher minimum then the local routine but the other eight global routines found lower minimums. However, the lowest minimum found by the local routine is only around 2% higher then the lowest found by the simulated annealing routine SA2. It can also be seen that out of all the algorithms, the local routine has the lowest standard deviation of objective function values. The local optimization routine does outperform 4 of the global routines with respect to the mean and 2 of the global routines with respect to the median value of objective function values. Looking at table 5.2, the local routine does obtain a reasonable number of solutions, as compared to the global routines, within the 5th and 20th percentiles.

The results discussed above are best interpreted in the light of the computing time required for the various training algorithms. Table 5.9 shows the computing time required for the various global optimization algorithms relative to the computing time required for the particular local optimization used on the given training data set. The numbers in the table show the ratio of the training time for the global optimization

Table 5.9 Ratio of Training Times for Global Optimization Algorithms in Comparison to Local Optimization Algorithms

Data Set	NNGA	EVOL	KORR1	KORR2	KORR3	KORR4	SA1	SA2	SW
Bilinear	145	174	175	175	175	175	174	185	159
DAX	48	59	61	60	60	64	60	63	54
JYUS	93	111	106	106	106	106	113	119	93
JYUSTTR	58	68	66	66	67	66	68	70	56
Flare	4	4	4	4	5	4	4	4	4
Mackey-Glass	12	14	14	13	14	13	15	14	13
Average:	60	71	71	71	71	71	72	76	63

Note: The numbers indicate the ratio of the average training time for the global optimization routine divided by the average training time for the local optimization routine. For example, for the DAX neural network model, the NNGA took on average 48 times longer to train then the local optimization routine. The training times are averaged across all restarts. For the global optimization algorithms, the training times are taken from the final configurations as given in appendix A, tables A.1-A.5.

routine divided by the training time for the local optimization routine. For example, for the DAX neural network model, the NNGA training time averaged 48 times longer than the local optimization routine. For a particular data set and algorithm, the training times were computed as the total computing time consumed for the full number of restarts, e.g. 500 for the Bilinear training data. For the stochastic global algorithms, the training times are taken from the run of the algorithm with the combination of algorithm parameters presented in one of the tables A.1 through A.5 in appendix A, i.e. the winning combination of parameters. Each of the global algorithms was a hybrid algorithm whereby the global algorithm provided starting values for the respective local algorithm for that training set. Therefore, the training times for the global algorithms are the sum of its local algorithm time plus the training time for the respective global algorithm.

The obvious observation from the training times presented in table 5.9 is that the global algorithms took much more time to train then the local algorithms. However, the relative computational requirements between the local and global algorithms were less pronounced for the larger neural network models trained on the Flare and Mackey-Glass training data sets. The global algorithms had the worst performance relative to local algorithm on the Bilinear data set. The obvious cause of this is that the Bilinear data set is easy to learn and the quasi-Newton local optimization algorithm converges very quickly on this problem. For the larger and harder to learn problems Flare and Mackey-Glass, the global algorithms were at less of a disadvantage. The disadvantage is the smallest on the Flare data set. The large size of the neural network model used on the Flare data set, with 211 parameters to estimate, necessitated the use of the conjugate gradient algorithm as opposed to the more efficient quasi-Newton algorithm used on the other problems.

The training times for the EVOL, and KORR1-KORR4 algorithms are equal. In reality, there was some difference in training times, however, training times were rounded for simplicity and thus the reported relative training times for the evolutionary strategy algorithms are identical. The code base for the KORR1-KORR4 algorithms is identical with changes to the calling parameters of the underlying subroutine invoking different functioning of the crossover operator, which differentiates the various KORR algorithms. These differences in operation of the KORR algorithms are apparently dominated by the computational demands of the other processes in each iteration of the algorithm, not the least of which is the calculation of the outputs of the neural network given the current values for the weights. The EVOL algorithm is conceptually simpler then the KORR algorithm, however, it is probably coincidence that the training time is so similar to the KORR algorithms. The EVOL algorithm could be inefficiently coded relative to the KORR algorithms and as stated above, other processes may dominate the computational demands.

Since the global algorithms take a great deal more time relative to the local routines, a greater number of restarts could be performed by a local routine relative to a global. An increased number of restarts would increase the relative performance of the local routine. Table 5.10 presents the probabilities from table 5.2 normalized with respect to computing time required. Each of the probabilities for the global algorithms given in table 5.2 is divided by the associated relative computing time given in table 5.9. Therefore, the probabilities are adjusted to assume that the global algorithms use the

Table 5.10 Adjusted Probability, Assuming Equal Training Times, of Obtaining an Objective Function Value Within the Given Percentile of all Objective Function Values Obtained Across all Algorithms for the Specific Data Set.

	Bilinear		DAX		JYUS	
Algorithm	5th percentile	20th percentile	5th percentile	20th percentile	5th percentile	20th percentile
LO	0.0440	0.1420	0.0500	0.1640	0.0200	0.1380
NNGA	0.0006	0.0013	0.0011	0.0037	0.0009	0.0023
EVOL	0.0004	0.0015	0.0004	0.0022	0.0008	0.0021
KORR1	0.0006	0.0011	0.0012	0.0041	0.0009	0.0023
KORR2	0.0005	0.0014	0.0009	0.0038	0.0008	0.0020
KORR3	0.0008	0.0017	0.0009	0.0039	0.0002	0.0017
KORR4	0.0006	0.0017	0.0012	0.0043	0.0009	0.0025
SA1	0.0002	0.0011	0.0009	0.0032	0.0003	0.0016
SA2	0.0002	0.0008	0.0010	0.0028	0.0003	0.0011
SW	0.0003	0.0015	0.0009	0.0035	0.0006	0.0023

	JYUSTTR		Flare		Mackey-Glass	
Algorithm	5th percentile	20th percentile	5th percentile	20th percentile	5th percentile	20th percentile
LO	0.0200	0.1380	0.0240	0.1280	0.0280	0.1520
NNGA	0.0014	0.0038	0.0130	0.0400	0.0040	0.0170
EVOL	0.0013	0.0034	0.0160	0.0610	0.0003	0.0054
KORR1	0.0015	0.0036	0.0070	0.0280	0.0029	0.0223
KORR2	0.0014	0.0033	0.0110	0.0550	0.0043	0.0178
KORR3	0.0003	0.0027	0.0048	0.0232	0.0017	0.0134
KORR4	0.0014	0.0040	0.0190	0.0790	0.0031	0.0305
SA1	0.0006	0.0026	0.0220	0.0680	0.0019	0.0155
SA2	0.0005	0.0019	0.0140	0.0570	0.0026	0.0114
SW	0.0010	0.0038	0.0110	0.0510	0.0034	0.0185

same amount of computing time that the local algorithm does for that particular data set. In effect, the global algorithms are allowed a smaller number of restarts relative to the local algorithm. The probabilities for the local routine could have alternatively been adjusted upward to assume the local algorithm was allowed to run as long as one of the global algorithms, in effect, increasing the number of restarts for the local algorithm. However, what is more important in table 5.10 are the relative probabilities as opposed to the absolute levels.

We can see from table 5.10 that if our goal is to obtain a solution within say the 5th percentile of solutions, the local optimization routine outperforms all the global algorithms on every problem. The local routine also outperforms in the context of obtaining solutions within the 20th percentile. The global algorithms performed the best on the Flare data set, relatively speaking. The global algorithms were at the least disadvantage on the Flare data set. However, the global algorithms were still dominated by the local algorithm when adjusting for computing time.

One weakness of this type of analysis is that it depends on the particular implementation of the global algorithms in this study. In particular, an improved method for switching from the global routine to the local routine could potentially cut a great deal of time off the computing time of the global algorithms. However, even ignoring computing time, the local optimization routine is competitive for most of the problems in this research. Even a simple doubling of the computing time required for the global algorithms, relative to a local routine, would make the local routine superior to the global routines in most of the cases presented in this research.

As has been pointed out previously, the computational time required to pick a good set of algorithm parameters in the pre-testing stage for the global algorithms was ignored in the analysis above. If a user intends to perform a large number of restarts for the purposes of obtaining a good set of neural networks, then for the SW algorithm, with only 6 possible configurations, a pre-testing stage would only add 60 restarts. However, for the NNGA algorithm, assume we perform 10 restarts for each of the 72 possible configurations. The pre-testing stage would more then double the computational requirements with 720 extra restarts. A different procedure for the global algorithms would have been to drop the second stage and simply perform a number of restarts for each of the configurations, pooling the results across all configurations. However, a conservative test of the efficiency of the global algorithms is to compare the local algorithm against a well configured global algorithm. That is, if the global algorithm does not significantly outperform the local algorithm with the benefit of hindsight, then perhaps these global algorithms are not performing according to users preconceived expectations. The question remains if there is significant difference in performance between the various configurations for the global optimization algorithm parameters? The next section presents the results of a test to investigate if there is a significant performance difference between the various configurations.

5.4 Pre-testing Bias

The procedure, as described in section 4.4.1, to pick the algorithm parameters for the global algorithms could introduce pre-testing bias into the results presented in the previous sections. For example, for the EVOL algorithm on the bilinear training data, we have 14 different configurations, each representing a different combination of algorithm parameters. In the pre-testing stage, 10 restarts are run for each of these configurations. The configuration with the lowest mean objective function value computed across the restarts is chosen to perform the full-scale simulations, i.e. 500 restarts. The simulation results presented in this chapter are from the full-scale simulations. There is some pre-testing bias introduced into the results if the optimal set of algorithm parameters significantly outperforms the other combinations of algorithm parameters.

In practice, the small advantage seen by the global algorithms over the local algorithms might not be obtainable outside of a procedure like that used in this research to pick an "optimal" set of algorithm parameters. It should be noted that the pre-testing procedure could have been incorporated into the estimation procedure for the global algorithms and the reported results could have contained this information. However, this would certainly further handicap the global algorithms in viewing the results in the context of computational time. The procedure followed in this study to report the results is conservative in the sense that if the results for the global algorithms do not outperform relative to the results for the local optimization procedure, even in the context of potential bias, then is no need for further analysis with respect to pre-testing for the global algorithms.

For a given data set and algorithm, to test if there is a statistically significant difference for the objective function values between each configuration, an F-test is calculated in the context of a regression model. Consider the following regression model:

$$\mathbf{Q} = \beta_0 + \sum_{i=1}^{n} \beta_i I_i + \epsilon \tag{5.1}$$

where Q is the objective function value, e.g. from (4.6) or (4.7), for a restart and I_i is an indicator variable as follows:

$$(5.2) \qquad I_i = \begin{cases} 1, & \text{if Q is from configuration } i \\ 0, & \text{otherwise,} \end{cases}$$

where a configuration, as in preceding discussions in this study, refers to a specific set of algorithm parameters. For example, consider the EVOL algorithm on the Bilinear training data. Referring to table 4.3, we have 14 different configurations, each with 10 restarts. Therefore, in (5.1) the number of classes or configurations is $N = 14$ and the regression has a total of 140 observations; 14 configurations times 10 restarts for each configuration. Note that an econometrician would refer to each "configuration" as a "class", however, in keeping with the language in this study, we will continue to use the term configuration. A test for a statistically significant difference between the mean values for Q between classes or configurations is then an F-test on the regression with a null hypothesis of:

$$(5.3) \qquad H_0 : \beta_1 = \beta_2 = \ldots = \beta_n = 0.$$

Note that this test doesn't test which configurations are statistically different, or in other words which β_i's are different from zero, only that at least one of the configurations is significantly different.

Table 5.11 presents the results for the above described F-tests for comparing variability of objective function values across configurations for the global algorithms. For most algorithms and data sets, we cannot reject the null hypothesis given in (5.3). However, at a five or ten percent significance level, we can reject the null hypothesis of no significant difference in objective function values across configurations for 21 out of

TABLE 5.11 Probabilities for F-tests From Regressions Comparing the Variability of Objective Function Values Across Configurations.

Algorithm	Bilinear	DAX	JYUS
NNGA	.0876*	.8133	.7325
EVOL	.0007**	.5477	.5920
KORR1	.7163	.4813	.1459
KORR2	.6827	.0067**	.9657
KORR3	.0136**	.1015	.3550
KORR4	.4418	.0003**	.3301
SA1	.9862	.5594	.2011
SA2	.8663	.0064**	.8498
SW	.0022**	.0565*	.8355

Algorithm	JYUSTTR	Mackey-Glass	Flare
NNGA	.6429	.0001**	.0012**
EVOL	.6659	.0326**	.0395**
KORR1	.7999	.2858	.0001**
KORR2	.5410	.5808	.0001**
KORR3	.0462**	.2807	.0001**
KORR4	.3349	.0891*	.0001**
SA1	.1728	.8020	.0001**
SA2	.0022**	.9449	.0002**
SW	.1437	.1142	.7447

*Significant at the 10% significance level.
**Significant at the 5% significance level.

the 54 algorithms and data sets. Out of these 21, 18 are significant at the five percent level. For each of the Bilinear, DAX, JYUSTTR, and Mackey-Glass data sets, there are a few algorithms for which there does appear to be some pretesting bias introduced by the procedure to pick the algorithm parameters for the global algorithms. This could account for some of the small out-performance of those algorithms relative to the local algorithm. Of particular significance is the Flare data set where at a five percent significance level, there is evidence of pretesting-bias on eight of the nine algorithms. This is noteworthy because for the Flare data set, with respect to the objective function values obtained and ignoring computational time, the global algorithms did appear to outperform the local algorithm. The evidence of pretesting bias for the global algorithms on the flare data set could explain their better performance relative to the local algorithms on this data set. Furthermore, the R-squared from the aforementioned regressions were relatively high for the flare data set. For the Flare data set, a single low value of .10 for the R-square was observed for one of the algorithms. The other R-squares were much higher with 4 data sets producing a regression with an R-squared above .90. The simulations on the Mackey-Glass data set produced regressions with an average R-squared around .25. The simulations on the other data set produced regressions with lower R-squares ranging from .05 to .30.

5.5 Conclusions

Even ignoring the relative computational requirements of the various algorithms, the results presented in figures 5.1-5.12 and tables 5.1-5.10 fail to provide any convincing justification for using stochastic global optimization algorithms to train neural networks. No single algorithm consistently outperforms all others and more importantly, the local optimization routine is not dominated by any of the global optimization algorithms. The global algorithms do on average provide marginally more probability of obtaining a lower converged objective function value as opposed to the local optimization routines.

However, in general, the local optimization routines obtained solutions at the minimum value, or very near the minimum.

Looking at the results in the context of the relative training times presented in table 5.9 adds weight to the results. For all the neural network models and training data sets, the global algorithms took considerably more time to train then the local optimization routines. Stochastic global optimization algorithms as a class of algorithms are computationally expensive. However, it is generally expected that as a tradeoff for increased computational time, the global algorithm will obtain a much lower objective function value then a local algorithm. For estimating neural network parameters, the simulations presented in this study only show marginally more probability of obtaining a lower minimum. With respect to the training times, certainly a more sophisticated stopping method could be investigated for switching from the global algorithms to the local optimization algorithm. This could considerably reduce the computational time consumed by the global algorithms. In addition, optimization of the code for the global algorithms could improve the computational efficiency. Nonetheless, the marginal gains, if any, in the quality of the solution obtained by the stochastic global algorithms investigated in this study do not justify the universal application of these algorithms for training neural networks.

A computationally demanding pre-testing procedure was used to obtain a reasonable set of user-definable algorithm parameters for the stochastic global algorithms. The extra computational time required for this pre-testing procedure was excluded from the analysis presented in this chapter. The pre-testing procedure should have theoretically given an unfair advantage to the global algorithms. Nonetheless, the global algorithms failed to substantially outperform, or even match in some cases, the local algorithms with respect to the magnitude of the solutions found. Stochastic global optimization algorithms could be useful in situations where the neural network objective function is discontinuous. The local algorithms used in this research require a continuous and differentiable objective function.

Chapter 6

SUMMARY AND CONCLUSIONS

6.1 Introduction

The first section of this chapter presents a summary of the results presented in the previous chapter. Conclusions to be drawn from these results are also reported. The last two sections discuss the limitations of the study and give suggestions for further research.

6.2 Summary of Results, and Conclusions

In this research, the relative speed and accuracy of 9 alternative global optimization methods in estimating the weights of neural networks is compared to local optimization methods. The stochastic global algorithms investigated were 2 simulated annealing algorithms, 1 simple random stochastic algorithm, 1 genetic algorithm and 5 evolutionary strategy algorithms. The algorithms are compared by performing multiple estimations from random starting values on 6 function approximation problems and analyzing the running time and distribution of the final objective function values over the multiple estimations. On two of the training data sets, 250 random restarts were run and on the other four, 500 random restarts were run. The results were displayed graphically in the form of histograms and boxplots. In addition, various statistics were reported such as the mean, median, minimum, and maximum of the objective function values computed across the restarts.

The results indicated that a large number of local minimums exist for all the neural network training data sets considered in this study. There was no single algorithm that dominated all others across the training data sets. More importantly, with respect to the research objectives of this study, the local optimization algorithm is not consistently dominated by any of the global algorithms. However, the global algorithms do provide on average marginally more probability of obtaining a lower converged objective function value as opposed to the local optimization routines. The higher probability is demonstrated in the slightly lower mean and median values for the objective function values from the global algorithms as compared to the local algorithm. However, on average, the local optimization routine did have a lower standard deviation of objective function values across the data sets. The local routine obtained a sufficient number of convergences at the minimum value, or very near the minimum. In 5 of the 6 training problems, the local optimization routine found a solution that was at the lowest minimum found across all algorithms, or within .0001 of it.

The stochastic global algorithms required much more computing time then the local routines. On average, the global routines required 60 to 70 times as much computing time. However, for the two largest training data sets with 43 and 211 neural network weights to estimate, the difference in training times was much less. The global algorithms for the largest problem with 211 weights took about 4 times longer then the local routine and 14 times longer for the training problem with 43 weights. Since the global algorithms take a great deal more time relative to the local routines, a greater number of restarts could be performed by a local routine relative to a global. This would increase the relative performance of the local routine. Adjusting the results to account for the computing time showed that if the goal is to obtain a solution within the 5th percentile of solutions, the local optimization routine outperforms all the global algorithms on every problem. The weakness of this type of analysis is that it is dependent on the particular implementation of the global algorithms in this study. In particular, an improved method for switching from the global routine to the local routine could potentially cut a great deal of time off the computing time of the global algorithms.

In conclusion, the results indicate that with respect to the specific algorithms studied, there is little evidence to show that a global algorithm should be used over a more traditional local optimization routine for training neural networks. Further, neural networks should not be estimated from a single set of starting values whether a global or local optimization method is used. The results strictly apply only to the estimation methods and problems considered. There may be problems where global optimization methods are superior. However, even ignoring computational time, there is still little evidence to support the use of stochastic global algorithms for training neural networks. The results presented in this study add significantly to the body of literature concerning the usefulness of stochastic global optimization algorithms for training neural networks. With respect to the range of data sets and algorithms studied, no previous study has presented simulation results as extensive as those presented in this research.

6.3 Limitations of Study and Directions for Study

The greatest limitation of this study is the limited number of training data sets and global algorithms examined. Extending the analysis to a larger number of data sets would either add weight to the results presented in this study or could discover types of problems on which the global algorithms are effective. For the smaller estimation problems investigated in this study, the local optimization routine was clearly superior to the global routines. However, for the largest estimation problem in this study the global algorithms, with one exception, did show an advantage in their ability to find a solution with a slightly smaller objective function value. In this problem, with respect to the solution found, the advantage of the best global optimization algorithm over the local routine was less then 2%. However, perhaps the larger and more complex neural network estimation problems would benefit from using a stochastic global optimization algorithm.

The global optimization field is an area in which a great deal of research is taking place. New algorithms and improvements to existing algorithms are being researched. Other types of stochastic global algorithms could be investigated, for example Ant algorithms. Besides stochastic algorithms, other categories of global algorithms, such as function smoothing techniques, could be investigated. The existing algorithms in this research could be improved by implementing an intelligent method of switching from the global routines to the associated local routine.

Bibliography

Aarts, E., and J. Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing.* Chichester, England: John Wiley & Sons, 1989.

Alander, J.T. "Indexed Bibliography of Genetic Algorithms and Neural Networks." University of Vaasa, Department of Information Technology and Production Economics, Report 94-1-NN, available via anonymous ftp: site ftp.uwasa.fi directory cs/report94-1 file gaNNbib.ps.Z.

Angeline, P.J., G.M. Saunders, and J.B. Pollack. "An Evolutionary Algorithm That Constructs Recurrent Neural Networks." *IEEE Transactions on Neural Networks* 5(January 1994):54-65.

Antonisse, J. "A New Interpretation of Schema Notation That Overturns the Binary Encoding Constraint," *Proceedings of the Third International Conference on Genetic Algorithms.* J.D. Schaffer, ed., pp. 86-91, San Mateo, CA: Morgan Kaufmann Publishers, 1989.

Avriel, Mordecai. *Nonlinear Programming Analysis and Methods.* Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1976.

Baba, N. "A New Approach for Finding the Global Minimum of Error Function of Neural Networks." *Neural Networks* 2(1989):367-373.

Baba, N., Y. Mogami, M. Kohzaki, Y. Shiraihi, and Y. Yoshida. "A Hybrid Algorithm for Finding the Global Minimum of Error Function of Neural Networks and Its Applications." *Neural Networks* 7(1994): 1253-1265.

Bäck, T. *Evolutionary Algorithms Theory and Practice.* New York: Oxford University Press, 1996.

___. "Optimal Mutation Rates Genetic Search." *Proceedings of the Fifth International Conference on Genetic Algorithms.* S. Forrest, ed., pp. 2-8, San Mateo, CA: Morgan Kaufmann, 1993.

Bäck T. and S. Khuri. "An Evolutionary Heuristic for the Maximum Independent Set Problem." *Proceedings of the First IEEE Conference on Evolutionary Computation*, pp. 531-535, IEEE Press, 1994.

Bäck, T., and H.P. Schwefel. "An Overview of Evolutionary Algorithms for Parameter Optimization." *Evolutionary Computation* 1(1993):1-23.

___. "Evolutionary Computation: An Overview." P*roceedings of the IEEE Conference on Evolutionary Computation.* pp. 20-29, 1996.

Bäck, T., D. Fogel, D. Whitley, and P. Angeline. "Mutation." *Handbook of Evolutionary Computation*. T. Bäck, D. Fogel, and Z. Michalewicz, ed., New York: Oxford University Press, 1997.

Bäck, T., F. Hoffmeister, and H.P. Schwefel. "A Survey of Evolution Strategies." *Proceedings of the Fourth International Conference on Genetic Algorithms*. R. Belew, and L. Booker, ed., pp. 2-9, Los Altos, CA: Morgan Kaufmann Publishers, 1991.

Bäck, T., G. Rudolph, and H.P. Schwefel. "Evolutionary Programming and Evolution Strategies: Similarities and Differences." *Proceedings of the Second Annual Conference on Evolutionary Programming.* D.B. Fogel and W. Atmar, ed., pp. 11-22, San Diego, CA: Evolutionary Programming Society, 1993.

Barnard, E. "Optimization for Training Neural Nets." *IEEE Transactions on Neural Networks* 3, 2(March 1992):232-240.

Barnes, N.M., A.W. O'Neill, and D. Wood. "Rapid, Supervised Training of a Two-layer, Opto-electronic Neural Network Using Simulated Annealing." *Optics Communications* 87(1992):203-206.

Barron, A.R. "Universal Approximation Bounds for Superpositions of a Sigmoidal Function." Manuscript, Department of Statistics, University of Illinois, Champange Urbana, IL, 1991.

Bartlett, P., and T. Downs. "Training a Neural Network with a Genetic Algorithm." Department of Electrical Engineering, University of Queensland, Australia, Technical Report, January 1990.

Bartlett, P.L. "For Valid Generalization, the Size of the Weights is More Important Than the Size of the Network," *Advances Neural Information Processing Systems 9*. M.C. Mozer, M.I. Jordan, and T. Petsche, ed., pp. 134-140, Cambridge, MA: The MIT Press, 1997.

Battiti, R. "First and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method." *Neural Computation* 4(1992):141-166.

Belew, R.K., J. McInerney, and N.N. Schraudolph. "Evolving Networks: Using the Genetic Algorithm with Connectionist Learning." Cognitive Computer Science Research Group, Computer Science & Engr. Dept., Univ. of California at San Diego, CSE Technical Report #CS90-174, June 1990.

Binder, K. *Monte Carlo Methods Statistical Physics*. New York: Springer-Verlag, 1978.

Bishop, C.M. *Neural Networks for Pattern Recognition*. Oxford: Oxford University Press, 1996.

Boender, C.G.E, and H.E. Romeijn. "Stochastic Methods." *Handbook of Global Optimization*. R. Horst and P.M. Pardalos, ed., pp. 829-869, Dordrecht, The Netherlands: Kluwer Academic Publishers, 1995.

Booker L., D. Fogel, D. Whitley, and P. Angeline. "Recombination." *Handbook of Evolutionary Computation*. T. Bäck, D. Fogel, and Z. Michalewicz, ed., pp. C3.3:1-C.3.3:27, Oxford Press, 1997.

Bretthorst, Larry G. *Bayesian Spectrum Analysis and Parameter Estimation*. vol. 48 of *Lecture Notes Statistics*, Springer, 1988.

Brill, F.Z., D.E. Brown, and W.N. Martin. "Fast Generic Selection of Features for Neural Network Classifiers." *IEEE Transactions on Neural Networks* 3, 2(1992):324-328.

Brunelli, Roberto. "Training Neural Nets through Stochastic Minimization." *Neural Networks* 7, 9(1994):1405-1412.

Carroll, S.M., and B.W. Dickinson. "Construction of Neural Nets Using the Radon Transform." *Proceedings of the IEEE Conference on Neural Networks (Washington DC)*. New York: IEEE Press, pp. 607-611, 1989.

Cerny, V. "Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm." *Journal of Optimization Theory and Applications* 45(1985):41-51.

Chang, E.J., and R.P. Lippmann. "Using Genetic Algorithms to Improve Pattern Classificiation Performance." *Advances Neural Information Processing 3*. R.P. Lippmann, J.E. Moody, and D.S. Touretsky, ed., pp. 797-803, San Mateo, CA: Morgan Kaufmann, 1991.

Chen, Y.M., and R.M. O'Connell. "Active Power Line Conditioner with a Neural Network Control." *IEEE Transactions on Industry Applications*. 33(July/August 1997): 1131-1136.

Chow, Tommy, and Chi-Tat Leung. "Performance Enhancement Using Nonlinear Processing." *IEEE Transactions on Neural Networks* 7, 4(July 1996):1039-1042.

Cohen, Barak, David Saad, and Emanuel Marom. "Efficient Training of Recurrent Neural Network with Time Delays." *Neural Networks* 10, 1(1997):51-59.

Corana, A., M. Marchesi, C. Martini, and S. Ridella. "Minimizing Multimodal Functions of Continuous Variables with the "Simulated Annealing" Algorithm." *ACM Transactions on Mathematical Software* 13, 3(September 1987):262-280.

Cybenko, G. "Approximation by Superpositions of a Sigmoid Function." *Mathematics of Control Signals and Systems* 2(1989):303-314.

Davis, L., Ed. *Handbook of Genetic Algorithms*, New York: Van Nostrand Reinhold, 1991.

Day, S.P., and D.S. Camporese. "A Stochastic Training Technique for Feed-Forward Neural Networks." *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, vol. 1, pp. 607-612, 1990.

Dean Hougen, D., J. Fischer, and D. Johnam. "A Neural Network Pole Balancer that Learns and Operates on a Real Robot Real Time." *Proceedings of the Machine-Learning Conference-Conference on Learning Theory, Workshop on Robot Learning*. pp. 73-80, July 10, 1994.

DeJong, K. "Analysis of the Behavior of a Class of Genetic Adaptive Systems." PhD Dissertation. Dept. of Computer and Communication Sciences, Univ. of Michigan, Ann Arbor, 1975

Derwent, D. "A Better Way to Control Pollution." *Nature* 331(1988):575-578.

Ergezinger, S., and E. Thomsen. "An Accelerated Learning Algorithm for Multilayer Perceptrons: Optimization Layer by Layer." *IEEE Transactions on Neural Networks* 6, 1(January 1995):31-42.

Erkmen, I., and A. Ozdogan. "Short Term Load Forecasting Using Genetically Optimized Neural Network Cascaded with a Modified Kohonen Clustering Process." *Proceedings of the 1997 IEEE International Symposium on Intelligent Control*, pp. 107-112, 1997.

Eshelman, L.J. and J.D. Schaffer. "Crossover's Niche." *Proceedings of the Fifth International Conference on Genetic Algorithms*. S. Forrest, ed., pp. 9-14, San Mateo, CA: Morgan Kaufmann Publishers, 1993.

Fang, Luyuan, and Tao Li. "A Globally Optimal Annealing Learning Algorithm for Multilayer Perceptrons with Applications." *Proceedings of the 4th Australian Joint Conference on Artificial Intelligence (AI'90),* pp. 21-23, Perth, Australia, World Scientific Publishing, Singapore, November, 1990.

Farmer, J.D. "Chaotic Attractors of an Infinite-Dimensional Dynamical System." *Physica D* 4(1982):366-393.

Floudas, C.A. *Deterministic Global Optimization: Theory, Algorithms and Applications.* Dordrecht: The Netherlands: Kluwer Academic Publishers, 1999.

Fogel, D.B. "An Introduction to Simulated Evolutionary Optimization." *IEEE Transactions on Neural Networks* 5, 1(January 1994):3-14.

Franses, Philip Hans, and Dick van Dijk. *Non-linear Time Series Models Empirical Finance.* Cambridge: Cambridge University Press, 2000.

Frenzel J.F. "Genetic Algorithms: A New Breed of Optimization." *IEEE Potentials* 12(October 1993): 21-24.

Fukushima, K., and S. Miyake. "Neocognition: A New Algorithm for Pattern Recognition Tolerant of Deformations and Shifts Position." *Pattern Recognition.* 15(1984):455-469.

Funabashi, K. "On the Approximate Realization of Continuous Mappings by Neural Networks." *Neural Networks* 2(1989):183-192.

Gallant, R., and H. White. "On Learning the Derivatives of an Unknown Mapping with Multilayer Feedforward Networks." *Neural Networks* 5(1992):129-138.

Gallant, A.R., D.A. Hsieh, and G.E Tauchen. "On Fitting a Recalcitrant Series: The Pound/Dollar Exchange Rate, 1974-83." *Nonparametric and Semiparametric Methods Econometrics and Statistics.* W.A. Barnett, J. Powell, and G.E. Tauchen, ed., pp. 199-240. Cambridge: Cambridge University Press, 1991.

Geman, S., and D. Geman. "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6(1984):721-741.

Geman, S., E. Bienenstock, and R. Doursat. "Neural Networks and the Bias/Variance Dilemma." *Neural* Computation 4(1995):1-58.

Gill, P., W. Murray, and M. Wright, *Practical Optimization.* London: Academic Press, 1993.

Goffe, W.L., and G.D. Ferrier. "Simulated Annealing: An Initial Application in Econometrics." *Computer Science in Economics and Management* 5(1992):133-146.

Goffe, W.L., G.D. Ferrier, and J. Rogers. "Global Optimization of Statistical Functions with Simulated Annealing." *Journal of Econometrics* 60(1994):65-99.

Goldberg, D. "Genetic Algorithms and Walsh Functions: Part 1, A Gentle Introduction." *Complex Systems* 3(1989a):129-152.

___. *Genetic Algorithms Search, Optimization, and Machine Learning*. MA: Addison-Wesley Reading, 1989b.

___. "Real-Coded Genetic Algorithms, Virtual Alphabets and Blocking." *Complex Systems* 5(1991):119-167.

Granger, C.W., and A.P. Andersen. *An Introduction to Bilinear Time Series Models*. Gottingen: Vandenhoek and Ruprecht, 1978.

Gray, Perry, William Hart, Laura Painton, Cindy Phillips, Mike Trahan, and John Wagner. "A Survey of Global Optimization Methods." Sandia National Laboratories, Albuquerue, NM, http://www.cs.sandia.gov/opt/survey/, accessed on March 10, 1992.

Guo, Z., and R.E. Uhrig. "Using Genetic Algorithms to Select Inputs for Neural Networks." *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks, COGANN-92*, pp. 223-234, *1992*

Hagan, M.T., and M. Menhaj. "Training Feedforward Networks with the Marquardt Algorithm." *IEEE Transactions on Neural Networks* 5, 6(1994):989-993.

Hagan, M.T., H.B. Demuth, and M. Beale. *Neural Network Design.* Boston: PWS Publishing Company, 1996.

Hancock, P.J.B. "Genetic Algorithms and Permutation Problems: A Comparison of Recombination Operators for Neural Structure Specification." *Combinations of Genetic Algorithms and Neural Networks (COGANN workshop)*. D. Whitley ed., IEEE Computer Society Press, 1992.

Harp, S., T. Samad, and A. Guha. "Towards the Genetic Synthesis of Neural Networks." *Proceedings of the Third International Conference on Genetic Algorithms*. J. D. Schaffer, ed., pp. 360-369, San Mateo, CA: Morgan Kaufmann, 1989.

Heistermann, J. "Different Learning Algorithms for Neural Networks - A Comparative Study." *Parallel Problem Solving from Nature, Workshop Proceedings*. Y. Davidor, H.P. Schwefel, and R. Manner, ed., pp. 368-396, Springer Verlag, 1994.

Herdy, M. "Application of the Evolution Strategy to Discrete Optimization Problems." *Proceedings of the First International Conference on Parallel Problem Solving from Nature (PPSN).* H.P. Schwefel and R. Männer, ed., pp.188-192, Dortmund, Germany, 1990.

Hjalmarsson, and A. Juditsky. "Nonlinear Black-Box Modeling System Identification: a Unified Overview." *Automatica* 31, 12(1995):1691-1724.

Holland, J.H. *Adaption Natural and Artificial Systems.* Ann Arbor: Univ. of Michigan Press, 1975.

Hornik, K., M.B. Stinchcombe, and H. White. "Multilayer Feedforward Networks Are Universal Approximators." *Neural Networks* 2(1989): 359-366.

Hornik, K., M. Stinchcombe, and H. White. "Universal Approximation of an Unknown Mapping and Its Derivatives Using Multilayer Feedforward Networks." *Neural Networks* 3(1990):551-560.

Horst, R., and H. Tuy. *Global Optimization: Deterministic Approaches, third edition.* Berlin: Springer-Verlag, 1996.

Horst, R., and P.M. Pardolos. *Handbook of Global Optimization.* Dordrecht, The Netherlands: Kluwer Academic Publishers, 1995.

Horst, R., P. Pardalos, and N. Thoai. *Introduction to Global Optimization.* Boston: Kluwer Academic Publishers, 1995.

Ingber, L. "Very Fast Simulated Re-Annealing." *Journal of Mathematical Computer Modelling* 12(1989):967-973.

Ingber, L. "Simulated Annealing: Practice Versus Theory." *Mathematical Computer Modelling* 18, 11(1993):29-57.

Janikow, C.Z., and Michalewicz, Z. "An Experimental Comparison of Binary and Floating Point Representations Genetic Algorithms." *Proceedings of the Fourth International Conference on Genetic Algorithms.* pp. 31-36, Morgan Kaufman Publishers, 1991.

Johansson, E.M., F.U. Dowla, and D.M. Goodman. "Backpropagation Learning for Multilayer Feed-Forward Neural Networks Using the Conjugate Gradient Method." *International Journal of Neural Systems* 2, 4(1992): 291-301.

Johnson, N.L., and S. Kotz. *Distributions Statistics: Continuous Univariate Distributions-1.* New York: Hougton Mifflin, 1970.

Johnson, N.L., and S. Kotz. *Distributions Statistics: Continuous Multivariate Distributions.* New York: Wiley, 1972.

Keane, A.J. "The Options Design Exploration System." Reference Manual and Users Guide – Version B3.1. Computational Engineering and Design Centre, University of Southhampton, U.K., 2002.

Khachaturyan, A., A. Semenovskaya, and B. Vainshtein, "The Thermodynamic Approach to the Structure Analysis of Crystals." *Acta Crystallographica* 31(1981):742-754.

Khuri S., and T. Bäck. "An Evolutionary Heuristic for the Minimum Vertex Cover Problem." *KI-94 Workshops (Extended Abstracts).* J. Kunze and H. Stoyan, ed., pp. 83-84, Gesellschaft für Informatik e. V., Bonn, 1994.

Khuri S., T. Bäck, and J. Heitkötter. "The Zero/One Multiple Knapsack Problem and Genetic Algorithms." *Proceedings of the 1994 ACM Symposium on Applied Computing.* E. Deaton, D. Oppenheim, J. Urban, and H. Berghel, ed., pp. 188-193, New York: ACM Press, 1994a.

___. "An Evolutionary Heuristic Approach to Combinatorial Optimization Problems." *Proceedings of the 22nd Annual ACM Computer Science Conference.* D. Cizmar, ed., pp. 66-73, New York: ACM Press, 1994b.

Kinsella, J.A. "Comparison and Evaluation of Variants of the Conjugate Gradient Method for Efficient Learning Feed-Forward Neural Networks with Backward Error Propagation." *Network* 3(1992):27-35.

Kirkpatrick, S., C.D. Gelatt, and M.P. Vecchi. "Optimization by Simulated Annealing." *Science* 220(May 1983):671-680.

Kitano, H. "Neurogenetic Learning: An Integrated Method of Designing and Training Neural Networks using Genetic Algorithms." *Physica D* 75(1994):225-228.

Knowles, Joshua, David Corne, and Mark Bishop. "Evolutionary Training of Artificial Neural Networks for Radiotherapy Treatment of Cancers." *Proceedings of 1998 International Conference on Evolutionary Computation (ICEC'98)*, ed., pp. 398-403, 1998.

Knowles, Joshua, David Corne, and Mark Bishop. "Evolutionary Training of Artificial Neural Networks for Radiotherapy Treatment of Cancers." *Proceedings of*

1998 International Conference on Evolutionary Computation (ICEC'98). pp. 398-403, 1998.

Kuan, C.M., and H. White. "Artificial Neural Networks: An Econometric Perspective." *Econometric Reviews* 13(1994):1-91.

Laarhoven, P.J.M. van, and E. Aarts. *Simulated Annealing: Theory and Applications.* Dordrecht, Holland: Kluwer Academic Publishers, 1987.

Lapedes, A., and R. Farber. "Nonlinear Signal Processing Using Neural Networks: Prediction and System Modeling." Technical Report LA-UR-87-2662, Los Alamos National Laboratory, Los Alamos, NM, 1987.

Lee, S.W. "Off-Line Recognition of Totally Unconstrained Handwritten Numerals Using Multilayer Cluster Neural Networks." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 18, 6(June 1996): 648-652.

Likartsis, A, I. Vlachavas, and L.H. Tsoukalas. "A New Hybrid Neural-Genetic Methodology for Improving Learning." *Proceedings of the 9th IEEE International Conference on Tools with Artificial Intelligence*, pp. 32-36, 1997.

Mackey, M.C., and L. Glass. "Oscillation and Chaos Physiological Control Systems." *Science* 197(1977):287-289.

Maniezzo, V. "Genetic Evolution of the Topology and Weight Distribution of Neural Networks." *IEEE Transactions on Neural Networks* 5, 1(1994):39-53.

Masters, T. *Advanced Algorithms for Neural Networks: A C++ Sourcebook.* New York: John Wiley and Sons Inc., 1995.

___. *Practical Neural Network Recipes C++.* New York: Academic Press, 1993.

McLoone, S., and G.W. Irwin. "Fast Parallel Off-Line Training of Multilayer Perceptrons." *IEEE Transactions on Neural Networks* 8, 3(May 1997):646-653.

Metropolis, N., A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines." *Journal of Chemical Physics* 21(1953):1087-1092.

Michalewicz, Zbigniew . *Genetic Algorithms + Data Structures = Evolution Programs. 3rd edition* Berlin: Springer-Verlag, 1996.

Miller, G., P. Todd, and S. Hedge. "Designing Neural Networks Using Genetic Algorithms." *Proceedings of the Thrid International Conference on Genetic Algorithms*. J. D. Schaffer, ed., pp. 379-384, San Mateo, CA: Morgan Kaufmann, 1989.

Minsky, M., and S. Papert. *Perceptrons.* Cambridge: MIT Press, 1969.

Mohler, R.R. *Bilinear Control Processes.* New York: Academic Press, 1973.

Møller, M.F. "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning." *Neural Networks* 6(1993):525-533.

Montana, D., and L. Davis. "Training Feedforward Neural Networks Using Genetic Algorithms." *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* pp. 762-767, San Mateo, CA: Morgan Kaufmann, 1989.

Neruda, R. "Genetic Algorithms and Neural Networks: Making Use of Parameter Space Symmetries." *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks* (IJCNN'2000), vol. 1, pp. 293-298, 2000.

___. "Yet Another Genetic Algorithm for Feed-forward Neural Networks." *Proceedings of the 9th International Conference on Tools with Artificial Intelligence* (ICTAI '97). Z. Rakowski, ed., pp. 375-380. Newport Beach, California, IEEE Computer Society, November 4-7, 1997.

Neumaier, Arnold, available at http://solon.cma.univie.ac.at/~neum/glopt.html, accessed on May 10, 2001.

Omatu, S, and M. Yoshioka. "Self-Ttuning Neuro-PID Control and Applications." *Proceedings of the 1997 IEEE Conference on Systems, Man, and Cybernetics*, vol. 3, pp. 1985-1989, 1989.

Omatu, S., and S. Deris. "Stabalization of Inverted Pendulum by the Genetic Algorithm." *Proceedings of the 1996 IEEE Conference on Emerging Technologies and Factory Automation*, ETFA'96. vol. 1, pp. 282-287, 1996.

Pincus, M. "A Monte Carlo Method for the Approximate Solution of Certain Types of Constrained Optimization Problems." *Operations Research* 18(1970):1225-1228.

Pinter, J.D. *Global Optimization Action.* Dordrecht: Kluwer, 1996.

Porto, V.W., D.B. Fogel, and L.J. Fogel. "Alternative Neural Network Training ethods." *EE Expert* 10(June 1995):16-22.

Powell, M.J.D. "Restart Procedures for the Conjugate Gradient Method." *Mathematical Programming.* 12(1977):241-254.

Prechelt, Lutz. "Proben1 – A Set of Neural Network Benchmark Problems and Benchmarking Rules." Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, available via ftp.ira.uka.de/pub/papers/techreports/1994/1994-21.ps.Z, 1994.

Pujol, J.C.F, and P. Riccardo. "Evolution of the Topology and the Weights of Neural Networks using Genetic Programming with a Dual Representation." *Applied Intelligence*. 8(1998):73-84.

Radcliffe, N.J. "Genetic Neural Networks on MIMD Computers." Doctoral dissertation, University of Edinburgh, Edinburgh, Scotland, 1990.

Rechenberg, I. "Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biolgischen Evolution." Stuttgar: Frommann-Holzboog Verlag, 1973.

Ripley, B.D. "Statistical Aspects of Neural Networks." *Networks and Chaos: Statistical and Probabilistic Aspects*. O.E. Barndorff-Nielsen, J.L. Jensen, and W.S. Kendall, ed., London: Chapman & Hall, 1993.

Rooij, A.J.F. van, L.C. Jain, and R.P. Johnson. *Neural Network Training Using Genetic Algorithms*. Singapore: World Scientific Publishing Co., 1996.

Rumelhart D.E., G.E. Hinton, and R.J. Williams. "Learning Internal Representations by Error Propagation." *Parallel Distributed Processing: Explorations the Microstructure of Cognition, Vol. 1.* D.E. Rumelhart and J.L. McClelland ed., pp. 318-362, Cambridge, MA: MIT Press, 1986.

Rumelhart, D.E., J.L. McClelland, and the PDP Group. "Parallel Distributed Processing." *Explorations the Microstructure of Cognition, Vol. 1: Foundation.* Cambridge: MIT Press, 1986.

Ryoo, H., and N. Sahinidis. "A Branch-and-Reduce Approach to Global Optimization." *Journal of Global Optimization* 8(1996):107-139.

Sarle, Warren S. "Neural Network Implementation SAS Software." *Proceedings of the Nineteenth Annual SAS Users Group International Conference.* April, 1994a.

Sarle, Warren S. "Neural Networks and Statistical Models." *Proceedings of the Nineteenth Annual SAS Users Group International Conference.* April 1994b.

Schaffer, J.D., D. Whitley, and L. Eshelman. "Combination of Genetic Algorithms and Neural Networks: A Survey of the State of the Art." *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks, COGANN-92*, pp. 1-37, 1992.

Schoenauer, M., and Z. Michalewicz. "Boundary Operators for Constrained Parameter Optimization Problems." *Proceedings of the 7th International Conference on Genetic Algorithms*. pp.320-329, East Lansing, Michigan, July 19-23, 1997.

Schwefel, H.P. *Evolution and Optimum Seeking*. New York: Wiley, 1995.

___. *Numerical Optimization of Computer Models*. Chichester: Wiley, 1981.

___. *Numerische Optimierung von Computer-Modellen mittel der Evolutionsstrategie*, vol. 26 of *Interdisciplinary Systems Research*. Basel: Birkhäuser, 1977.

Sexton, R.S., R.E. Dorsey, and J.D. Johnson. "Beyond Backpropagation: Using Simulated Annealing for Training Neural Networks." *Journal of End User Computing* 11, 3(1999a):3-10.

___. "Optimization of Neural Networks: A Comparative Analysis of the Genetic Algorithm and Simulated Annealing." *European Journal of Operational Research* 114(1999b):589-601.

___. "Toward Global Optimization of Neural Networks: A Comparison of the Genetic Algorithm and Backpropagation." *Decision Support Systems* 22, 2(1998):171-185.

Shang, Y., and B. Wah. "Global Optimization for Neural Network Training." *Computer* 29, 3(1996):45-54.

Sima, J. "Back-propagation is Not Efficient." *Neural Networks* 9, 6(1996):1017-1023.

Sjöberg, J., Q. Zhang, L. Ljung, A. Benveniste, B. Delyon, P.Y. Glorennec, H. Hjalmärsson, and A. Juditsky. "Nonlinear Black-Box Modelling in System Identification: Mathematical Foundation." *Automatica* 31(December 1995):1725-1750.

Skinner, A.J., and J.Q. Broughton. "Neural Networks Computational Materials Science: Training Algorithms." *Modelling Simulation Materials Science Engineering* 3, 3(1995):371-390.

Smagt, P.P. van der. "Minimization Methods for Training Feedforward Neural Networks." *Neural Networks* 7, 1(1994):1-11.

Solis, F.J., and J.B. Wets. "Minimization by Random Search Techniques." *Mathematics of Operations Research* 6(1981):19-30.

Styblinski, M.A., and T.S. Tang. "Experiments Nonconvex Optimization: Stochastic Approximation with Function Smoothing and Simulated Annealing." *Neural* Networks 3(1990):467-483.

Syswerda, G. "Schedule Optimization Using Genetic Algorithms." *Handbook of Genetic Algorithms*. L. Davis, ed., pp. 332-349, New York: Van Nostrand Reinhold, 1991.

Szu, H., and R. Hartley. "Fast Simulated Annealing." *Physics Letters A* 122(June 1987):157-162.

Tang, Z., and G. Koehler. "Deterministic Global Optimal FNN Training Algorithms." *Neural Networks* 7, 2(1994):301-311.

Trippi, Robert R., and Turban Efraim. *Neural Networks Finance and Investing.* Chicago: Probus, 1992.

Vanderbilt, D., and S.G. Louie. "A Monte Carlo Simulated Annealing Approach to Optimization over Continuous Variables." *Journal of Computational Physics* 56(1984):259-271.

Vecchi, M.P. and S. Kirkpatrick. "Global Wiring by Simulated Annealing." *IEEE Transactions on Computer-Aided Design,* vol. CAD-2 , 4(October 1983):215-222.

Vignaux, G.A., and Z. Michalewicz. "A Genetic Algorithm for the Linear Transportation Problem." *IEEE Transactions on Systems, Man and Cybernetics* 21, 2(1991):445-452.

Visual Numerics. *IMSL Math/Library version 3.0.* Houston, TX, 1997.

Werbos, P. "Beyond Regression: New Tools for Prediction and Analysis the Behavioral Sciences." unpublished Ph.D. Dissertation, Harvard University, Department of Applied Mathematics, 1974.

White, H. "ANN: A Little Knowledge Can Be a Dangerous Thing." Second Moment, available at www.secondmoment.org, accessed on Oct. 5, 2002.

___. "Learning Neural Networks: A Statistical Perspective." *Neural Computation* 1(1989):425-464.

White, S.R. "Concepts of Scale Simulated Annealing." *Proceedings of the IEEE International Conference on Computer Design*, ICCD'84 pp. 646-641, New York, October, 1984.

Whitely D., T. Starkweather, and C. Bogart. "Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity." *Parallel Computing* 14, 3(1990):347-361.

Whitley, D. "A Genetic Algorithm Tutorial." *Statistics and Computing* 4(1994):65-85.

___. "Genetic Algorithms and Neural Networks." *Genetic Algorithms Engineering and Computer Science.* G. Winter, J. Periaux, M. Galan and P. Cuesta, ed. pp. 203-216, New York: John Wiley, 1995.

Wright, A.H. "Genetic Algorithms for Real Parameter Optimization." *Foundations of Genetic Algorithms,* G.J.E. Rawlins, ed., pp. 205-218, San Mateo, CA: Morgan Kaufmann Publishers, 1991.

Xinxing, Yang , and Jiao Licheng. "Fast Global Optimization Neural Network and its Application Data Fusion." *Proceedings of the Fourth International Conference on Signal Processing,* (ICSP'98), vol. 2, pp. 1351-1354, 1998.

Xu, Lei, Adam Krzyzak, and Alan Yuille. "On Radial Basis Function Nets and Kernal Regression: Statistical Consistency, Convergence Rates, and Receptive Field Size." *Neural Networks* 7(1994):609-628.

Yan, W., and Z. Zhu. "A New Evolutionary Computation Method." *Proceedings of the IEEE 1997 National Aerospace and Electronics Conference,* (NAECON 1997), vol. 2, pgs. 803-807, 1997.

Yan, W., Z. Zhu, and R. Hu. "A Hybrid Genetic/BP Algorithm and Its Application for Radar Target Classification." *Proceedings of the 1997 IEEE National Aerospace and Electronics Conference,* NAECON, vol. 2, pp. 981-984, 1997.

Yao, Xin, "Evolving Artificial Neural Networks." *Proceedings of the IEEE* 87, 9(September 1999):1423-1447.

APPENDIX A

STOCHASTIC GLOBAL OPTIMIZATION ALGORITHM PARAMETERS

Tables A.1 through A.5 present the chosen values for the user-definable parameters of the stochastic global optimization algorithms. An example of a user-definable parameter would be the standard deviation of mutation, which is common to all the stochastic global algorithms. The performance of the algorithms may depend upon wisely choosing the values of the various algorithm parameters. Section 4.4.1 presents the details of the procedure for picking the algorithm parameter values presented in tables A.1 through A.5. For most of the algorithm parameters, it is difficult to see any discernable pattern in the values chosen. However, one can make some general observations about some of the parameters. For example, for the standard deviation of mutation s, a parameter common to all the algorithms[1], the value chosen is generally much less then the maximum value of 1 tried for most of the algorithms.

In the NNGA and SW algorithms, the standard deviation of mutation stays constant throughout the operation of the algorithm. All the other algorithms have some sort of mechanism to adjust the standard deviation of mutation throughout the operation of the algorithm. Therefore, the reported standard deviation of mutation for the EVOL, KORR1-KORR4, and SA1 and SA2 algorithms is the beginning standard deviation of

[1] For the simulated annealing algorithm, the standard deviation of mutation is synonymous with the parameter called temperature.

Table A.1 Values for the User-Definable Parameters of the NNGA Algorithm.

Data Set	b – bias	r_c – probability of crossover	r_m – probability of mutation	s- std. dev. of mutation
Bilinear	5	.8	.2	0.12
Dax	5	.8	.2	.5
JYUS	20	.2	.8	.12
JYUSTTR	20	.2	.8	.12
Flare	10	.8	.6	1.0
Mackey-Glass	10	.8	.6	1.0

Note: See section 4.3.2 for a detailed explanation of the NNGA algorithm and its parameters. The values in the table are for the various user definable parameters that were chosen by the procedure described in section 4.4.1.

Table A.2 Values for the User-Definable Parameters of the EVOL Algorithm.

Data Set	s- std. dev. of mutation[a]	a_s - adjustment factor for s
Bilinear	.250	.850
Dax	.125	.999
JYUS	.125	.999
JYUSTTR	.125	.999
Flare	.060	.999
Mackey-Glass	.250	.999

Note: See section 4.3.3 for a detailed explanation of the EVOL algorithm and its parameters. The values in the table are for the various user definable parameters that were chosen by the procedure described in section 4.4.1.
[a]This column reports the beginning standard deviation of mutation. The algorithm adjusts s as it progresses.

Table A.3 Values for the User-Definable Parameters of the SW Algorithm.

Data Set	s- std. dev. of mutation
Bilinear	.25
Dax	.50
JYUS	.25
JYUSTTR	.50
Flare	.5
Mackey-Glass	.125

Note: See section 4.3.5 for a detailed explanation of the SW algorithm and its parameters. The values in the table are for the various user definable parameters that were chosen by the procedure described in section 4.4.1.

Table A.4 Values for the User-Definable Parameters of the KORR1, KORR2, KORR3, and KORR4 Algorithms.

Algorithm	Data Set	s – std. dev. of mutation[a]	τ' - adjustment factor for s	τ - adjustment factor for s
KORR1	Bilinear	.125	.183	.359
	Dax	.5	.033	.076
	JYUS	.03	.183	.359
	JYUSTTR	.03	.046	.090
	Flare	.03	.006	.023
	Mackey-Glass	1	.054	.138
KORR2	Bilinear	.25	.046	.090
	Dax	1	.033	.076
	JYUS	.25	.091	.180
	JYUSTTR	.03	.091	.180
	Flare	.06	.012	.046
	Mackey-Glass	.03	.054	.138
KORR3	Bilinear	.03	.183	.359
	Dax	1	.033	.076
	JYUS	.5	.183	.359
	JYUSTTR	.06	.183	.359
	Flare	.03	.006	.023
	Mackey-Glass	.03	.027	.069
KORR4	Bilinear	.5	.183	.359
	Dax	.5	.033	.076
	JYUS	.06	.046	.090
	JYUSTTR	.25	.091	.180
	Flare	.03	.006	.023
	Mackey-Glass	.5	.108	.276

Note: See section 4.3.3 for a detailed explanation of the KORR1, KORR2, KORR3, and KORR4 algorithms. The values in the table are for the various user definable parameters that were chosen by the procedure described in section 4.4.1.
[a]This column reports the beginning standard deviation of mutation. The algorithm adjusts s as it progresses.

mutation at the start of the algorithms operation. Note that for the SA1 and SA2 algorithms, the beginning standard deviation of mutation is the beginning parameter temperature given by $T_p^{(0)}$. For these algorithms, the beginning standard deviation of mutation is probably not as critical a value as the standard deviation of mutation for the

Table A.5 Values for the User-Definable Parameters of the SA1 and SA2 Algorithms.

Algorithm	Data Set	$T_p^{(0)}$ - beginning parameter temperature[a]	$T_p^{(M)}$ - ending parameter temperature	α - ending temperature ratio	c' - scale factor
SA1	Bilinear	.01	.001	1	.1
	Dax	.125	.001	.0001	4
	JYUS	.02	.001	1	.5
	JYUSTTR	.01	.001	.01	2
	Flare	.02	.001	1	2
	Mackey-Glass	.12	.001	.01	2
SA2	Bilinear	.03	.00001	.002	4
	Dax	.01	.001	1	2
	JYUS	.03	.001	.01	40
	JYUSTTR	.01	.00001	.0004	18
	Flare	.03	.00001	.01	4
	Mackey-Glass	.03	.001	.01	4

Note: See section 4.3.4 for a detailed explanation of the SA1 and SA2 simulated annealing algorithms and their parameters. The values in the table are for the various user definable parameters that were chosen by the procedure described in section 4.4.1.
[a]The beginning acceptance criterion temperature $T_a^{(0)}$ is set equal to the beginning parameter temperature $T_p^{(0)}$ but the acceptance criterion temperature is scaled by the factors c and c'.

NNGA and SW algorithms where the standard deviation stays fixed throughout the operation of the algorithm. However, a large value for the beginning standard deviation of mutation can lead to saturated hidden neurons, which could freeze the learning process of any algorithm. There are several notable exceptions to the general rule of a small standard deviation of mutation, namely a value of 1 for both the Flare and Mackey-Glass problems for the NNGA algorithm. For the Flare data set, with 211 model parameters or neural network weights to estimate, we would expect a smaller standard deviation of mutation would be best. The Mackey-Glass problem is also one of the larger problems with 43 model parameters. There are several examples of large standard deviations of mutation for the KORR algorithms. However, as mentioned before, the values reported in table A.4 are for the beginning standard deviation of mutation and the KORR algorithms have a mechanism to adjust the mutation rate as the algorithm progresses.

For the NNGA algorithm, the final chosen values for the probabilities of crossover r_c and mutation r_m varied. A high value of .8 for r_c was chosen for 4 of the 6 problems with a relatively low value of .2 for the remaining two problems. The converse was true for r_m, the probability of mutation. A relatively low probability of mutation of .2 was chosen for 2 of the problems with higher values of .6 or .8 for the remaining four problems. An interesting observations is that the lower values of r_m are associated with the higher values of r_c and visa-versa. In other words, there appears to be an inverse relationship between the probability of crossover and of mutation. Note that a probability of mutation of .2 is still high compared to most genetic algorithm implementations. The rational for the relatively high probability of mutation is given in section 4.1.1. There does not appear to be any pattern in the values for the bias parameter b. For the EVOL algorithm, consistent with the results presented by Keane, a value of .999 was chosen for a_s, the adjustment factor for the standard deviation, for 5 out of 6 of the problems.

VITA 2

Lonnie Hamm

Candidate for the Degree of

Doctor of Philosophy

Thesis: A COMPARISON OF STOCHASTIC GLOBAL OPTIMIZATION METHODS: ESTIMATING NEURAL NETWORK WEIGHTS

Major Field: Agricultural Economics

Biographical:

Education: Graduated from Hooker High School, Hooker, Oklahoma in May 1986; Received Bachelor of Science degree in Accounting from Oklahoma State University, Stillwater, Oklahoma in May 1991; Received Master of Science degree with a major in Agricultural Economics at Oklahoma State University in July 1995. Completed the requirements for the Doctor of Philosophy degree with a major in Agricultural Economics at Oklahoma State University in August 2003.

Experience: Employed by Oklahoma State University, Department of Agricultural Economics, as a graduate research assistant, June 1992 to July 1997 and June 2000 to December 2001. Employed as a research associate by Telesis Management Incorporated, a Commodity Trading Advisor, August 1997 to December 2000, Santa Barbara, California.