Variations of the Simulated Annealing Algorithm

By

Kandyce Elizabeth Smith

B.S. April 1997, Embry-Riddle Aeronautical University

A Research Paper submitted to

The Faculty of

The School of Engineering and Applied Science
of the George Washington University in partial satisfaction
of the requirements for the degree of Master of Science

April 28, 2000

Research performed at NASA Langley Research Center

Abstract

Neural network training methods used by Mathworks Matlab's Neural Network Toolbox are deterministic optimizers that tend to find local optimums. Using a probabilistic optimizer, such as a simulated annealing-based optimizer, to minimize the error should achieve a global optimum, or at least a better local optimum. The neural network simulator and candidate simulated annealing optimizers were written in Fortran 90. The simulator calculates the error between actual and desired output for a given input. The optimizer is used to minimize the error with the expectation of finding a global minimum over the parameters that define a neural network. In general, the Matlab methods trained the neural networks better than the candidate simulated annealing training methods. However, in a couple of trials, a simulated annealing search led to better answers than were obtained using deterministic methods alone.

Table of Contents

List of Figures iii

Chapter 1 Introduction 1

Chapter 2 Simulated Annealing 4

2.1 Global Optimization 4

2.2 Background 5

2.3 Algorithm 6

2.4 Modifications to the Code 9

Chapter 3 Neural Networks 15

3.1 Background 15

3.2 Training Methods 19

3.3 Error Function 20

Chapter 4 Fortran 90 21

4.1 Dynamic Memory Allocation and Pointers 21

4.2 Matrix Manipulation 21

4.3 Modules 22

Chapter 5 Experiments and Results 23

Chapter 6 Conclusions 35

References 36

List of Figures

Figure 3.1 Diagram of Neural Network Layout 16
Figure 3.2 Diagram of Neural Network Calculation for Layer i 17
Figure 3.3 Application of Activation Function for Layer i 17
Figure 3.4 Activation Functions – Graphs and Equations 18

Chapter 1
Introduction

Global optimization has been a very common field of research. One global optimization method, simulated annealing, was first used for discrete problems but has been extended to include continuous functions. Simulated annealing "explores the function's entire surface and tries to optimize the function while moving both uphill and downhill" [Goffe]. When talking about minimization, downhill moves are referred to as function evaluations that are less then the previous function evaluation whereas uphill moves are greater then previous function evaluations. "Thus it is largely independent of the starting values, often a critical input in conventional algorithms. Further, it can escape from local optima and go on to find the global optimum by the uphill and downhill moves. Simulated annealing also makes less stringent assumptions regarding the functions than do conventional algorithms (it need not even be continuous). Because of the relaxed assumptions, it can more easily deal with functions that have ridges and plateaus. Finally, it can optimize functions that are not defined for some parameter values" [Goffe]. Many researchers have contributed to developing techniques for optimizing continuous functions utilizing simulated annealing including Corana, et al., Dekkers and Aarts, and Goffe, Ferrier, and Rogers.

The purpose of the research being reported in this paper was to provide a testing environment in which to evaluate simulated annealing in the training of neural networks and to perform such evaluations. The simulated annealing algorithm (in several variations) and the architectures and training sets for the neural networks to be used in the tests were provided by the Guidance and Control Branch (GCB) of NASA Langley Research Center, through the guidance of Dr. Daniel Giesy. Neural network training was chosen as a test example because of experience among members of the GCB that neural network training was frequently a multi-modal (more than one local optimum) optimization problem. The testing progressed through several contrived test examples of increasing complexity to examples of actual engineering interest in an attempt to find a test example of sufficient multi-modality to provide the desired testing environment.

The design of artificial neural networks is inspired by the architecture of a neuron and multiple input-single output connections of neurons in biological neural networks. Some input data is processed by both networks and output is produced based on that data. Artificial neural networks have more regular structure, typically with fewer neurons and connections than functioning biological neural networks. For the purpose of the paper, "neural network", or any form thereof, refers to an artificial neural network.

A neural network is determined by the architecture and parameter settings. The architecture of the neural network consists of neurons, connections, and activation functions. The parameter settings are the weights and biases of the network. Training a neural network involves choosing values of the parameters. The training is formulated as an optimization problem. Some metric is devised to measure how well the neural network is performing the intended function. The values of the parameters are adjusted in an attempt to find the best value of this metric.

In MathWorks Matlab, neural networks are trained using deterministic optimization methods. In general, deterministic optimization methods are fast, however, they may converge to a sub-optimal set of parameters. In an attempt to obtain an optimal set of parameters, simulated annealing was used. Using simulated annealing to train neural networks should achieve a global optimum, or, at least, converge to a better local optimum from a given starting point than a deterministic method does. A comparison of the two methods was completed as part of the scope of this research.

Neural networks are common in today's research environment. Some applications of neural networks include aerospace, automotive, banking, defense, electronics, medical devices, robotics and transportation [Demuth]. This research is motivated by the use of neural networks in the field of guidance and control of the aerospace industry.

Chapter 2 is an explanation of global optimization and the simulated annealing global optimizer. An outline of the algorithm and modifications made to the algorithm are included in the chapter. A definition of a neural network along with Matlab training methods are

explained in Chapter 3. The Fortran 90 features used in the research are explored in Chapter 4. Chapter 5 is an description of the test cases that were used to exercise the simulated annealing algorithm. The results of the test cases and the conclusions of the research are discussed in Chapter 6.

Chapter 2
Simulated Annealing

The ideas of global optimization and simulated annealing are presented in this chapter. The simulated annealing algorithm is laid out along with modifications made to the basic algorithm.

2.1 Global Optimization

Deterministic optimization methods such as the Levenberg-Marquardt training method in Matlab are relatively fast but may converge to a sub-optimal set of parameters. To avoid this possibility, global optimization methods, specifically probabilistic methods, were considered for training neural networks.

According to Horst, global optimization is defined as follows [Horst]:

> Given an objective (or cost) function $f : X \rightarrow R$ and a constraint set, C, one gathers under the wording *global minimization* the questions related to the following problems:
>
> i) Find the [greatest] lower bound of f on C (on the whole of C); ($\mathcal{P}$)
>
> ii) Find a global minimum of f on C, i.e. $\bar{x} \in C$ such that $f(\bar{x}) \leq f(x)$ *for all* $x \in C$.
>
> The issue of *local minimization* has the same set of goals as a starting point, but is content *in fine* with characterizing, finding, approximating, etc., elements $\bar{x}$ of C satisfying $f(\bar{x}) \leq f(x)$ for all $x \in C \cap N$, where N is some neighborhood of $\bar{x}$. So, in global minimization, (one decides) from the beginning that the neighborhood N is to be the whole space.

There are two types of global optimization methods, deterministic and stochastic. "The disadvantage of deterministic methods is that they find the global minimum only after an exhaustive search over (X) and additional assumptions on *f*. The faster among these methods have the additional disadvantage that even more assumptions must be made about *f*, or that there is no guarantee of success.

Stochastic methods, in contrast, can almost all be proven to find a global minimum with an asymptotic convergence guarantee in probability, i.e. these methods are asymptotically

successful with probability 1. Furthermore, the computational results of the stochastic methods are, in general, better than those of the deterministic methods." [Dekkers] It should be noted that the more assumptions a method makes about the objective function, the fewer optimization problems fall within the scope of that method.

Stochastic methods include two-phase methods, random search methods, random function approach, and simulated annealing. The stochastic method chosen for this study by GCB was simulated annealing.

2.2 Background

Most optimization algorithms are based on a "downhill" search method, which means that a sequence of design values $x_1, x_2, \ldots$ is generated with $f(x_1) > f(x_2) > \cdots$. In a deterministic method, $x_{k+1} = x_k + s_k$ where the increment s_k is determined by the value of the function (and perhaps the derivatives) at x_k (and perhaps $x_{k-1}, x_{k-2}, \ldots$) in a deterministic fashion. In a simple random search, candidate increments of s might be generated at random until the search finds an s_k for which, with $x_{k+1} = x_k + s_k$, one achieves $f(x_k) > f(x_{k+1})$.

Simulated annealing is derived from the thermodynamic principle of annealing or slow cooling metals to achieve a highly ordered, crystalline state of lowest energy. The material is cooled starting at a high temperature. The temperature is lowered to allow the material to form highly ordered internal structures until an equilibrium state is reached. Because of the slow decrease of annealing temperature, the cooling metal is allowed to escape from high energy crystalline equilibria to lower energy states.

Simulated annealing is not a purely "downhill" method. A global optimization algorithm, simulated annealing (SA) is an iterative random search procedure that allows for uphill moves based on meeting a certain probabilistic criterion, which means that a design point x_{k+1} may be allowed for which $f(x_k) < f(x_{k+1})$. Occasional uphill moves might allow for escape from the neighborhood of a non-global local minimum to find a global or at least a better local minima. Candidate steps are generated from simulated annealing like the simple

random search and any downhill steps will be accepted, however, an uphill step might be accepted. The decision on whether to accept an uphill step depends on just how far uphill the function evaluation goes (a small step uphill is more likely to be accepted than a large one) and what the current "annealing temperature" is (this is an optimization parameter which starts large and decreases over the course of the optimization – the smaller the annealing temperature, the less likely that a given uphill step will be accepted). For a given cost function, the cost function is evaluated many times before each temperature reduction, retaining design points with the "better" cost function values and discarding approximately half of the inferior cost function values, with the expectation of finding a global minimum. As with the cooling metal, the cost function is allowed to escape from a basin of a sub-optimal local minimum due to the slow decrease of annealing temperature.

Simulated annealing was first used for discrete optimization problems such as the traveling salesman problem. Since then, the simulated annealing algorithm has been adapted to solve continuous problems. Research on continuous problems can be found in [Corana], [Kirkpatrick], [Dekkers], [Goffe], and [Vanderbilt] to name a few.

2.3 Algorithm

The SA algorithm is adapted from the Corana, et al., [Corana] algorithm with some necessary changes. The Corana, et al., algorithm will be presented below (with some minor changes) and the modifications will be discussed in Section 2.4.

Here is how Corana, et al., introduce their algorithm [Corana]:

> Let $\vec{x}$ be a vector in R^n and $(x_1, x_2, \ldots, x_n)$ its components. Let $f(\vec{x})$ be the function to minimize and let $a_1 < x_1 < b_1, \ldots, a_n < x_n < b_n$ be the n variables, each ranging in a finite interval. f does not need to be continuous but it must be bounded.

Actually, the box constraint should use weak inequalities as in $\vec{a} \leq \vec{x} \leq \vec{b}$ and f need only be bounded from below the box.

Corana, et al., state the algorithm as [Corana]:

Step 0 (Initialization)

Choose

A starting point $\overrightarrow{x_0}$.

A starting step vector $\overrightarrow{v_0}$.

A starting temperature T_0.

A termination criterion ε and a number of successive temperature reductions to test for termination N_ε.

A test for step variation N_s and a varying criterion $\vec{c}$.

A test for temperature reduction N_T and a reduction coefficient r_T.

Set i, j, m, k to 0. i is the index denoting successive points, j denotes successive cycles along every direction, m describes successive step adjustments, and k increases with successive temperature reductions.

Set h to 1. h is the index denoting the coordinate along which the trial point is generated, starting from the last accepted point.

Compute $f_0 = f(\overrightarrow{x_0})$.

Set $\overrightarrow{x_{opt}} = \overrightarrow{x_0}$, $f_{\text{opt}} = f_0$.

Set $n_u = 0$, $\quad u = 1,\ldots,n$.

Set $f_u^* = f_0$, $\quad u = 0, -1,\ldots,-N_\varepsilon+1$.

Step 1

Starting from the point $\overrightarrow{x_i}$, generate a random point $\overrightarrow{x'}$ along the direction h:

$$\overrightarrow{x'} = \overrightarrow{x_i} + r v_{m_h} \overrightarrow{e_h}$$

where r is a random number generated in the range [-1,1] by a pseudorandom number generator; $\overrightarrow{e_h}$ is the vector of the hth coordinate direction; and v_{m_h} is the component of the step vector $\overrightarrow{v_m}$ along the same direction.

Step 2

If the hth coordinate of $\overrightarrow{x'}$ lies outside the definition domain of f, that is, if $x'_h < a_h$ or $x'_h > b_h$, then return to step 1.

Step 3

Compute $f' = f(\overrightarrow{x'})$.

If $f' \leq f_i$, then accept the new point:

set $\overrightarrow{x_{i+1}} = \overrightarrow{x'}$,

set $f_{i+1} = f'$,

add 1 to i,

add 1 to n_h,

if $f' < f_{\text{opt}}$, then set

$\overrightarrow{x_{opt}} = \overrightarrow{x'}$,

$f_{\text{opt}} = f'$.

end if;

else ($f' > f_i$) accept or reject the point with acceptance probability p (Metropolis move):

$$p = \exp\left(\frac{f_i - f'}{T_k}\right).$$

In practice, a pseudorandom number p' is generated in the range [0,1] and is compared with p. If $p' < p$, the point is accepted, otherwise it is rejected.

In the case of acceptance:

set $\overrightarrow{x_{i+1}} = \overrightarrow{x'}$,
set $f_{i+1} = f'$,
add 1 to i,
add 1 to n_h.

Step 4

Add 1 to h.

If $h \leq n$, the goto step 1;

Else set h to 1 and add 1 to j.

Step 5

If $j < N_s$, then goto step 1;

Else update the step vector $\overrightarrow{v_m}$:

for each component index u the new step vector component v'_u is

$$v'_u = v_{m_u}\left(1 + c_u \frac{n_u / N_s - 0.6}{0.4}\right) \qquad \text{if } n_u > 0.6N_s,$$

$$v'_u = \frac{v_{m_u}}{1 + c_u \dfrac{0.4 - n_u / N_s}{0.4}} \qquad \text{if } n_u < 0.4N_s,$$

$$v'_u = v_{m_u} \qquad \text{otherwise.}$$

Set $\overrightarrow{v_{m+1}} = \overrightarrow{v'}$,
set j to 0,
set n_u to 0, $\quad u = 1, \ldots, n$,
add 1 to m.

The aim of these variations in step length is to maintain the average percentage of accepted moves at about one-half of the total number of moves. The rather complicated formula used is discussed at this end of this chapter. The c_u parameter controls the step variation along each uth direction.

Step 6

If $m < N_T$, then go to step 1;

Else, reduce the temperature T_k:

set $T_{k+1} = r_T \cdot T_k$,
set $f_k^* = f_i$,

add 1 to k,

set m to 0.

It is worth noting that a temperature reduction occurs every $N_s \cdot N_T$ cycles of moves along every direction and after N_T adjustments.

Step 7 (terminating criterion)

If:

$$\left| f_k^* - f_{k-u}^* \right| \leq \varepsilon, \qquad u = 1, \ldots, N_\varepsilon$$

$$f_k^* - f_{opt} \leq \varepsilon$$

then stop the search;

else:

add 1 to i,

set $\overrightarrow{x_i} = \overrightarrow{x_{opt}}$,

set $f_i = f_{\text{opt}}$.

Go to step 1.

The simulated annealing algorithm, including the Corana algorithm, is capable of climbing out of a sub-optimal basin. However, the search of the Corana algorithm is limited to the hyper-rectangle of points $\vec{x}$ whose coordinates fall between those of given vectors $\vec{a}$ and $\vec{b}$ and the Corana algorithm uses only coordinate direction searches. These limitations lead to the modifications discussed in the next section.

2.4 Modifications to the Method

A number of modifications were considered for the Corana algorithm. The changes to the code and the respective steps being changed will be stated. Reasons for the modifications will be discussed below. Most of the changes are a result of testing and were added with the expectation of improving the performance of the algorithm. This aspect will be discussed along with the experiment results in Chapter 5, Experiments and Results.

The Guidance and Control Branch (GCB) design philosophy for simulated annealing included random search directions and eliminating the bounding box. A random direction search was preferred because in deterministic searches, altering only one component of the design vector at a time is known [Nocedal] to produce an inefficient search. The bounding box was eliminated because a guess had to be made about the size of the box. If the size of

the box was chosen too small, it could eliminate the best answer and too large a box with standard "uniform distribution" search step length would be wasteful.

All searches have the form

$$\vec{x}' = \vec{x}_i + z\vec{R}.*\vec{y}$$

where

z is a random variable in $(0, \infty)$,

$\vec{y}$ is a random vector uniformly distributed over the surface of the unit ball in n-space,

$\vec{R}$ is a scaling vector to be adjusted from time to time as sampling gives information about the geometry of the cost function, and

.* is a component by component multiplication.

This allowed for generation of trial search steps. Step 2, Step 4, and the step vector, $\vec{v}$, of the Corana algorithm were eliminated. As the trials progressed, algorithm modifications took the form of changing the distribution of z and changing the scheme for updating $\vec{R}$.

The Guidance and Control Branch never came up with a satisfactory scheme for adjusting $\vec{R}$. Some adjustment schemes had $\vec{R}$ diverging to infinity. This coupled with the tendency of the tangent sigmoid activation function (see Section 3.1) to saturate at outputs of +1 or –1 for large magnitude inputs. When this happens, the simulated annealing search was deprived of information needed to make progress toward a minimum. These large $\vec{R}$ values allowed search steps to get so large that the computer capacity was overflowed, and answers of NaN (Not a Number) were returned. Other schemes caused $\vec{R}$ to shrink to zero so fast that the search could not make any progress.

First Attempt

For this method, z was determined by

$$z = \frac{r}{1-r^2}$$

where r is uniformly distributed on (0,1) so z was randomly distributed over $(0,\infty)$. All components of $\vec{R}$ were set equal. $\vec{R}$ was adjusted by

$$\overrightarrow{R_{new}} = \overrightarrow{R_{old}}\left(1 + c_u \frac{n_s / N_s - 0.6}{0.4}\right) \quad \text{if } n_s > 0.6\text{n}^*N_s,$$

$$\overrightarrow{R_{new}} = \frac{\overrightarrow{R_{old}}}{1 + c_u \frac{0.4 - n_s / N_s}{0.4}} \quad \text{if } n_s < 0.4\text{n}^*N_s,$$

$$\overrightarrow{R_{new}} = \overrightarrow{R_{old}} \quad \text{otherwise,}$$

the same manner as the step vector $\overrightarrow{v_m}$ (Step 5), to maintain the number of accepted moves to approximately one half of the total number of moves, where n_s is the number of accepted trials in this phase of the search.

However, the $\vec{R}$ update was not successful in practice, and alternate $\vec{R}$-updates were sought. The search steps grew too large. This appeared to be because $\vec{R}$ grew large. It was also observed at this point that this first choice for z gave a random variable with infinite expected value which would also tend to make search steps too big.

Geometric Mean

Another method used to update the scaling vector, $\vec{R}$, was to calculate $\overrightarrow{ams}$, the average magnitude of all the accepted vectors. $\overrightarrow{ams}$ was initialized to 0 and updated by

$$\overrightarrow{ams_i} = \overrightarrow{ams_{i-1}} + \left|\overrightarrow{x_{i-1}} - \overrightarrow{x_i}\right|$$

which was updated every time the function value for the $\overrightarrow{x_i}$ vector was less than the function value for the $\overrightarrow{x_{i+1}}$ vector. The notation $\left|\overrightarrow{x_{i-1}} - \overrightarrow{x_i}\right|$ represents the vector of component by component absolute values of the vector $\overrightarrow{x_{i+1}} - \overrightarrow{x_i}$. The calculation for z became

$$z = \frac{r}{\sqrt{1-r} \cdot \left(1 + \sqrt{1-r}\right)}$$

where r is again uniformly distributed over (0, 1). This random variable z serves the same "spread (0,1) over (0,∞)" function as $\frac{r}{1-r^2}$, but has expected value 1. In this case and the subsequent cases, $\vec{R}$ is a vector and its components did not have to have the same value. $\vec{R}$ was first updated by

$$\overrightarrow{R_{new}} = \sqrt{\frac{\overrightarrow{R_{old}} \cdot \overrightarrow{ams_{n_u}}}{n_u}}$$

where n_u was the number of accepted function evaluations, where all operations on vectors are done component by component. After the above calculation, each component of $\vec{R}$ was updated according to the "First Attempt" method described above. This method tended to not escape local minima.

Constrained Geometric Mean I

A third method tried was to keep a running tally of the maximum step for each component which leads to a downhill step. This was done by

$$\overrightarrow{ams_{new}} = \max\left(\overrightarrow{ams_{old}}, \left|\overrightarrow{x_{old}} - \overrightarrow{x_{new}}\right|\right)$$

The "max" here is a component by component array operation. Then the function adj_fac was created. An adjustment factor based on the relative size of the best successful step in component i compared to the old $\vec{R}$ (i) is calculated by the function. The function adj_fac is a monotone increasing on [0,∞), reciprocally symmetric (adj_fac(1/x) = 1/adj_fac(x)), and bounded by a user-set bound (this study used the bound 2). $\vec{R}$ is calculated first as

$$\vec{R}(i) = \vec{R}(i) \cdot adj_fac\left(\frac{\overrightarrow{ams}(i)}{\vec{R}(i)}\right)$$

then each component of $\vec{R}$ was updated according to the "First Cut" method described above.

Constrained Geometric Mean II

Another version of the constrained geometric mean was implemented without the component update of $\vec{R}$ at the end of the second loop.

Optimum $\vec{R}$ Update

Starting from the constrained geometric mean, another vector was added to the algorithm. This time, if a function evaluation was accepted and it was better than the current f_{opt}, $\overrightarrow{R_{opt}}$ would be set to the current $\vec{R}$. If the termination criterion was not met, $\vec{R}$ would be set to $\overrightarrow{R_{opt}}$ and the algorithm would continue.

Optimum $\vec{R}$ Update II

This update used the same idea as Optimum $\vec{R}$ Update; however, the components of $\vec{R}$ were not updated according to the "First Attempt" method.

Corana, et. al., penalized

After many changes to the Corana algorithm and many unsuccessful tries, a decision was made to go back to the original Corana algorithm for the simulated annealing. This decision was a result of the tries at step-size control had problems (either growing too big or shrinking too small), which led to more methods being tried. Most solutions wandered into the saturation region of the sigmoid functions used as an activation function in the neural networks. This caused insensitivity of the objective function to variation of design variables when the function evaluations were in the saturation region. Some variables got so out of control that non-numeric (NaN) values were returned from neural network calculation.

The new methodology was to modify the error function in the neural network module (Section 3.3). A penalty against very large design variable values was added to the error function with the expectation of eliminating large values for components of $\vec{x}$. The error function was changed from

$$e(\vec{x}) = \sum_{j=1}^{n}\sum_{i=1}^{m}\left(vd_{j,i} - vo_{j,i}\right)^2$$

to

$$e(\vec{x}) = \sum_{j=1}^{n}\sum_{i=1}^{m}\left(vd_{j,i} - vo_{j,i}\right)^2 + \vec{\delta} \bullet \left|\vec{x}\right|$$

where $|\vec{x}|$ is the vector of absolute values of components of $\vec{x}$, $\bullet$ represents vector dot product, and the components of $\vec{\delta}$ were all set, for this test, to 10^{-6}. A bounding box for the Corana, et al., method was picked based on numerical experience with the earlier tests.

Chapter 3
Neural Networks

This chapter is dedicated to giving the reader a basic understanding of what a neural network is and how a neural network functions. Different types of training methods for neural networks and the error function calculated as a result of the neural network are also discussed.

3.1 Background

According to the newsgroup comp.ai.neural-nets there is no universally accepted definition of a neural network. However, the newsgroup suggest that most people would accept "that a neural network is a network of many simple processors ("units"), each possibly having a small amount of local memory. The units are connected by communication channels ("connections") which usually carry numeric data, encoded by any of various means" [newsgroup].

Since Mathworks Matlab Neural Network Toolbox is used as part of the research, the Mathworks definition should also be stated: a neural network is a composition of "many simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the network function is determined largely by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of connections (weights) between elements" [Demuth]. The "simple elements" are also referred to as neurons. A "neuron layer", or "layer" for short, is made up of one or several neurons. A neural network may have one or more layers. The last layer of a neural network is referred to the output layer and all other layers are referred to as hidden layers.

The use of neural networks in research dates back to 1954. During this time, the main concentration for neural networks was pattern recognition. Most work involved single layer networks. Multi-layer networks were not theoretically proven and the importance was misunderstood by researchers of that time involved with neural networks. Shortly thereafter,

the single layer networks were deemed lacking in ability to sufficiently recognize a multiple number of patterns and essentially all funding for research on the subject was cut. However, some researchers continued their work on neural networks and eventually proved the worth of multi-layer networks. Nowadays, the usefulness of neural networks appears in a multitude of concentrations. [Wasserman]

Neural networks can be feedforward or feedback networks. Feedforward networks are only commanded by the input and do not correct for disturbances (such as noise) added to the system. Feedback networks correct for disturbances added to the system and therefore are more accurate then feedforward networks. [Nise] For this research, however, feedforward networks were used because GCB was interested in training feedforward networks for use in control systems.

As mentioned before, neural networks is constructed of a single layer or multiple layers. Typical neural networks have three or fewer layers. For each layer the output from the previous layer is the input for the next layer, as diagramed in Figure 3.1.

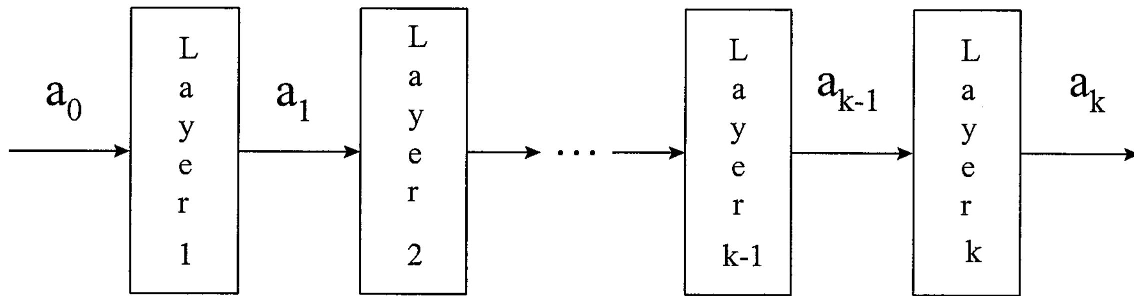

Figure 3.1 Diagram of Neural Network Layout

Each layer consists of a weighting matrix and an activation function. The network can also contain a bias vector, but it is optional. The elements of both the weighting matrix and bias vector are adjustable parameters of the neural network. The weighting matrix connects each input to all the outputs of the layer. The bias vector allows the user to shift the solution space in various dimensions and allows for flexibility in most problems. For a given layer, the input, a_i, is multiplied by the weighting matrix, M_i, and the result is added to the bias vector, B_i, if present, yielding an intermediate value z_i, as shown in Figure 3.2.

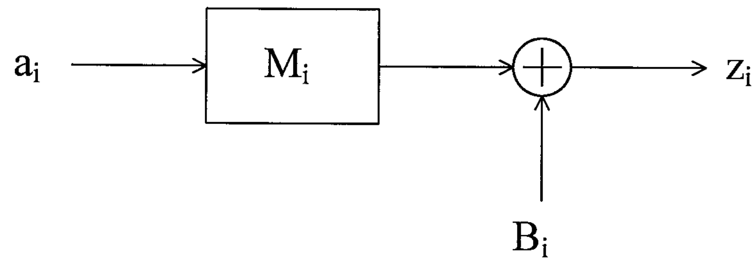

Figure 3.2 Diagram of Neural Network Calculation for Layer i

Once the intermediate value, z_i, is determined, the activation function is applied to each component in the z_i vector, yielding a new a_{i+1} vector. The a_{i+1} vector then becomes the input for the next layer, as shown in Figure 3.3 below.

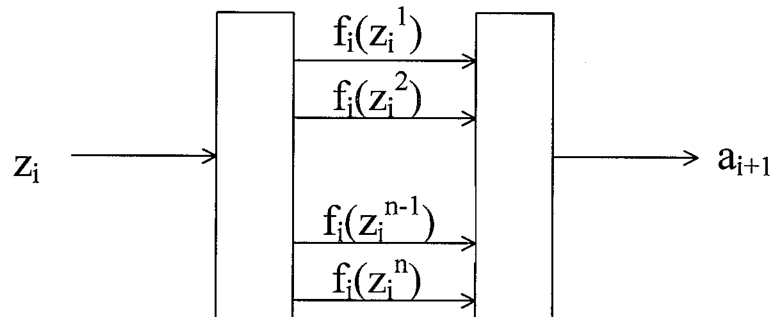

Figure 3.3 Application of Activation Function for Layer i

An activation function is a map from the inputs to the outputs of the network. "Activation functions for the hidden layers are needed to introduce nonlinearity into the network. Without nonlinearity, hidden layers would not make nets more powerful than just plain preceptrons (which do not have any hidden layers, just input and output). The reason is that a composition of linear functions is again a linear function. However, it is the nonlinearity (i.e. the capability to represent nonlinear functions) that makes multilayer networks so powerful" [newsgroup]. In practice, neural networks are constructed with the activation function of the output layer as linear; however, this is not required by the definition of the net.

There are five activation functions that were programmed into the test software. These, as defined in Matlab, are step (or hardlim), pure linear, saturated linear, log sigmoid, and

tangent sigmoid functions. Figure 3.4 shows the equations of the activation functions and the corresponding graphs.

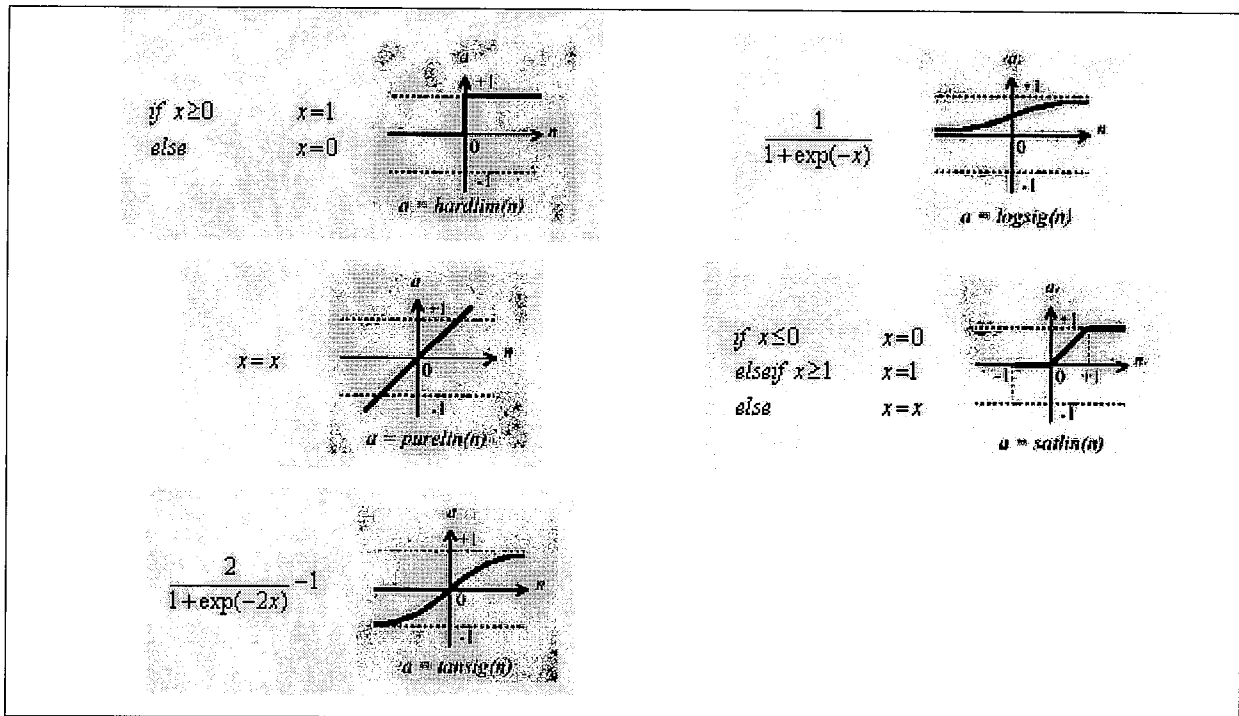

Figure 3.4 Activation Functions – Graphs and Equations

It should be noted that the logarithmic sigmoid and tangent sigmoid functions are susceptible to a saturation region. The tansig function approaches ±1 at the outer bounds, whereas the logsig function ranges between (0,1). Since there are no limits on x, i.e. x approaches ±∞, the values of x can become very large. Theoretically, if x < y, then tansig(x) < tansig(y); in particular, if x and y are different, so are tansig(x) and tansig(y). However, a computer only represents numbers to about 15 or 16 decimal places. When x gets large enough, even though tansig(x) < 1 theoretically, tansig(x) is so close to 1 that the computer representation of tansig(x) is 1. If x and y are large enough, x and y might have distinct computer representations, but when tansig(x) and tansig(y) are computed, the computer representation for both is 1. This means that if x was changed to y to try to improve a cost function f, the computer representation of the cost values f(x) and f(y) have the same computer representation. This provides no information to the optimization as to which design point is better.

3.2 Training Methods

Neural networks are not programmed, they are trained by example. "As children need to know nothing about comparative physiology to recognize their mothers, programmers need not provide neural networks with quantitative descriptions of objects being recognized, nor sets of logical criteria to distinguish such objects from similar objects. Instead, we show a neural network examples of objects (faces, parts or scenes) along with their identifications (mother, carburetor, or mountains). It memorizes these by altering values in the weight matrix, and will produce the proper response when an object is seen again" [Wasserman].

Neural networks are trained with supervised or unsupervised training. Only an input is required with unsupervised training and the neural network is allowed to "organize" itself as to give a consistent output every time that same input is used. A training set consisting of inputs and associated target outputs is required for supervised training. The weighting matrices and bias vectors of the neural network training method is adjusted in the network in an attempt to minimize the error.

There are numerous training methods for neural networks. One of the types of training methods in Matlab is backpropagation, a supervised training method. Backpropagation refers to "the manner in which the gradient is computed for nonlinear multilayer networks" [Demuth]. Specifically, "input vectors and the corresponding output vectors are used to train a network until it can approximate a function, associate input vectors with specific output vectors, or classify input vectors in an appropriate way as defined by the user" [Demuth].

The specific backpropagation method chosen to train the networks in Matlab was the Levenberg-Marquardt algorithm. This algorithm "uses this approximation to the Hessian matrix in the following Newton-like update

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}]^{-1}\mathbf{J}^T\mathbf{e}\text{''}$$

[Demuth]. It "was designed to approach second-order training speed without having to compute the Hessian matrix" [Demuth]. This particular method was chosen for baseline training in this study because the "algorithm appears to be the fastest method for training moderate-sized feedforward neural networks (up to several hundred weights). It also has a

very efficient MATLAB implementation, since the solution of the matrix equation is a built-in function, so it's attributes become even more pronounced in a MATLAB setting" [Demuth].

3.3 Error Function

A training pair for neural network is an ordered pair {vi, vd} of vectors such that the training input vector vi is of the correct size to be input to the neural network and the target output vector vd is the same size as the output vectors of the neural network. The actual output of the neural network resulting from the input vi is represented by vo. In training a network, a set of training pairs may be employed, with the training inputs represented by TI = [vi_1, vi_2, …, vi_n] and the associated target outputs by TO = [vd_1, vd_2, …, vd_n]. The neural network, acting on each of the training input vectors, produces a set of calculated output vectors, CO = [vo_1, vo_2, …, vo_n]. These output vectors also depend on the neural network parameters. The point of training the network is to make these output vectors match the corresponding target vectors as nearly as possible. To measure the extent to which the calculated outputs fail to match the target outputs, an error function is used. This function results in a scalar value that is defined as follows:

$$e(\vec{x}) = \sum_{j=1}^{n} \sum_{i=1}^{m} \left(vd_{j,i} - vo_{j,i}\right)^2$$

The inner sum is the square of the Euclidian distance between the target output and the calculated output for the j-th training pair. The error function is than the sum of these squared distances over all training pairs.

Chapter 4
Fortran 90

Because of the added features of Fortran 90, it was chosen over Fortran 77 to use for programming of this research. An explanation of these features and examples of each, where applicable, are described in this chapter.

4.1 Dynamic Memory Allocation and Pointers

"In Fortran 77, all the storage that was required during execution of a program could be determined and allocated by the compiler. This is known as *static storage allocation*, as opposed to *dynamic storage allocation*, which means that storage may be allocated or deallocated during the execution of a program" [Brainerd, 139]. This allows for memory to be assigned or freed up during the execution of a program. If running long codes with lots of arrays, the memory may not be sufficient to execute the program. With dynamic storage allocation, arrays can be allocated and deallocated, thus allowing for more memory to be available for the execution of the program.

Pointers are another method of memory management. A pointer is "a free-floating name that may be associated dynamically with or 'aliased to' some data object" [Brainerd, 265]. Dynamic memory management, pointers, and modules were used in the research.

4.2 Matrix Manipulation

With the simplified methods of matrix manipulation in Fortran 90, programming in Fortran 90 has become simplified. The following example shows the Fortran 77 method compared to the Fortran 90 method.

Fortran 77:
```
do i = 1, tt
    do i = 1, 3
        sn(i,j) = snless(i,j)
    end do
```

```
                end do
Fortran 90:     sn(1:tt,1:3) = snless(1:tt,1:3)
```

As seen by the Fortran 90 method, the programming has become simplified and allows for a more robust programming style.

4.3 Modules

"A module is a program unit that is not executed directly, but contains data specifications and procedures that may be utilized by other program units via the USE statement" [Brainerd]. The use of modules allows for isolating data and calculations for one part of a program from other parts of the program. The simulated annealing algorithm and the neural network are both set up in modules. This allows the simulated annealing algorithm to not have to have knowledge of the cost function being evaluated but to be able to call on the cost function for the value simulated annealing assigns to a given set of design variables.

As part of this research, one module was written which performed the simulated annealing optimization search. With each modification to the simulated annealing algorithm, this module was modified to incorporate it. Another module was written which could perform the calculation of a feedforward neural network with an arbitrary number of layers, and of nodes per layer.

Chapter 5
Experiments and Results

Neural networks were trained both in Matlab and using simulated annealing. The results of the training for several data sets are presented in this chapter. For each data set, the size of the training set, activation functions, number of layers, and number of neurons per layer are specified. This information, listed under "Training Data" for each case, defines the cost function to be minimized using simulated annealing or the Matlab Neural Network Toolbox. For each simulated annealing run, the respective modification to the code is stated. The initial temperature is only stated when it was changed from the default of 100,000.

Case 1 – Training Data

Size of training set	16 x 2
Activation functions	Log sigmoid, pure linear
Number of layers	2
Number of neurons per layer	1st – 3 2nd – 1

Case 1a

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	9.987e-7

Case 1b

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	9.978e-7

Case 1c

Training algorithm	Simulated Annealing
Code modification	First Attempt
Starting point	All zeros
Initial fopt	3677.12
Final fopt	1934.01

Case 1d

Training algorithm	Simulated Annealing
Code modification	First Attempt
Starting point	Close to solution of Case 1a
Initial fopt	3180.67
Final fopt	103.75

Case 1e

Training algorithm	Simulated Annealing
Code modification	First Attempt
Starting point	All zeros
Initial fopt	2308.12
Final fopt	1323.00

Case 1f

Training algorithm	Simulated Annealing
Code modification	First Attempt
Starting point	All zeros
Initial fopt	3677.12
Final fopt	409.07

Case 1g

Training algorithm	Simulated Annealing
Code modification	First Attempt
Starting point	All zeros
Initial fopt	3677.12
Final fopt	1430.91

As seen by the results of Case 1, simulated annealing did not improve on Matlab's training routines. Because of the nature of the function in the training set, the output was easily

matched by Matlab with the target output. Because the two Matlab tests produced the same answer, it was suspected that the problem was too simple to be multi-modal, so a larger, more difficult training set was introduced. The simulated annealing tests gave poor results and the simulated annealing code was modified.

Case 2 – Training Data

Size of training set	51 x 2
Activation functions	Log sigmoid, pure linear
Number of layers	2
Number of neurons per layer	1st – 3 2nd – 1

Case 2a

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	9.783

Case 2b

Training algorithm	Matlab
Starting point	All zeros
Initial fopt	n/a
Final fopt	11.629

Case 2c

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	11.938

Case 2d

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	7.915

Case 2e

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	11.978

Case 2f

Training algorithm	Simulated Annealing
Code modification	Geometric Mean
Starting point	Solution of Case 2e
Initial fopt	11.978
Final fopt	11.978

The Matlab tests showed that this case was indeed multi-modal. The simulated annealing run was started at the worst case solution of the Matlab training runs to determine if the simulated annealing code could improve on the solution. Unfortunately, the simulated annealing code did not improve on the starting value. Another data set was tried to try to gain some insight into the workings of this version of the simulated annealing algorithm.

Case 3 – Training Data

Size of training set	51 x 2
Activation functions	Log sigmoid, pure linear
Number of layers	2
Number of neurons per layer	1st – 3 2nd – 1

Case 3a

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	0.142

Case 3b

Training algorithm	Matlab
Starting point	All zeros
Initial fopt	n/a
Final fopt	0.135

Case 3c

Training algorithm	Simulated Annealing
Code modification	Geometric Mean
Starting point	Solution of Case 3a
Initial fopt	0.142
Final fopt	0.142

Again, starting simulated annealing from the worst case Matlab solution, simulated annealing produced the same results as the Matlab run. Yet another training set was created to test.

Case 4 – Training Data

Size of training set	51 x 2
Activation functions	Log sigmoid, pure linear
Number of layers	2
Number of neurons per layer	$1^{st} - 3$ $2^{nd} - 1$

Case 4a

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	0.0056

Case 4b

Training algorithm	Matlab
Starting point	All zeros
Initial fopt	n/a
Final fopt	0.0050

Case 4c

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	0.0059

Case 4d

Training algorithm	Matlab
Starting point	Layer 1 at all zeros, Layer 2 at random starting point
Initial fopt	n/a
Final fopt	0.0050

Case 4e

Training algorithm	Simulated Annealing
Code modification	Geometric Mean
Starting point	Solution of Case 4b
Initial fopt	0.0050
Final fopt	0.0050

Simulated annealing again produced the same results as the Matlab training run of Case 4b. Further modifications to the simulated annealing code were made. In general, from this point out, the second best Matlab training point was used as the simulated annealing starting point.

Case 5 – Training Data

Size of training set	200 x 2
Activation functions	Tangent sigmoid, pure linear
Number of layers	2
Number of neurons per layer	1st – 3 2nd – 1

Case 5a

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	0.0339

Case 5b

Training algorithm	Matlab
Starting point	All zeros
Initial fopt	n/a
Final fopt	55.847

Case 5c

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	0.0273

Case 5d

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	0.0331

Case 5e

Training algorithm	Simulated Annealing
Code modification	Geometric Mean
Starting point	All zeros
Initial fopt	66.78
Final fopt	11.59

Case 5f

Training algorithm	Simulated Annealing
Code modification	Constrained Geometric Mean
Starting point	Solution of Case 5a
Initial fopt	0.0339
Final fopt	0.0339

Note: All NaN's in output file

Case 5g

Training algorithm	Simulated Annealing
Code modification	Constrained Geometric Mean
Starting point	Solution of Case 5a
Initial Temperature	1000
Initial fopt	0.0339
Final fopt	0.0339

Note: All NaN's in output file

Case 5h

Training algorithm	Simulated Annealing
Code modification	Constrained Geometric Mean II
Starting point	Solution of Case 5a
Initial Temperature	1000
Initial fopt	0.0339
Final fopt	0.0339

Note: All NaN's in output file

Case 5i

Training algorithm	Simulated Annealing
Code modification	Optimum R Update
Starting point	Solution of Case 5a
Initial fopt	0.0339
Final fopt	0.0339

For results of Cases 5f, 5g, and 5h, simulated annealing did not improve on the starting value and, instead of converging to an answer, ran design variables off of the range of the computer. These tests led to further modifications of the code. For each of the cases that can be compared, Matlab produced better results. It was decided to obtain a data set that had been used in other research for this next case. This data comes from some unpublished research of Dr. Peiman Maghami of GCB. Dr. Maghami was attempting to train a neural network to serve as an attitude controller for the Lewis spacecraft (a different attitude control study for this spacecraft is described in [Maghami]).

Case 6 – Training Data

Size of training set	347 x 2
Activation functions	Tangent sigmoid, pure linear
Number of layers	2
Number of neurons per layer	1st – 15 2nd – 1

Case 6a

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	354.09

Case 6b

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	1.188e7

Case 6c

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	1.567e6

Case 6d

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	400.11

Case 6e

Training algorithm	Simulated Annealing
Code modification	Optimum R Update
Starting point	Solution of Case 6d
Initial fopt	400.11
Final fopt	400.11

Note: All NaN's in output file

The simulated annealing run for this case also did not converge to an answer, instead ran design variables off of the computer's range. The same training set was used for this next case with further modifications to the simulated annealing code.

Case 7 – Training Data

Size of training set	347 x 2
Activation functions	Tangent sigmoid, pure linear
Number of layers	2
Number of neurons per layer	1st – 10 2nd – 1

Case 7a

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	1.857e7

Case 7b

Training algorithm	Matlab
Starting point	Random starting point
Initial fopt	n/a
Final fopt	1.730e6

Case 7c

Training algorithm	Simulated Annealing
Code modification	Optimum R Update
Starting point	Solution of Case 6a
Initial fopt	1.857e7
Final fopt	5467.79

Note: All NaN's in output file

Case 7d

Training algorithm	Simulated Annealing
Code modification	Optimum R Update II
Starting point	Solution of Case 6a
Initial fopt	1.857e7
Final fopt	4880.40

Note: All NaN's in output file

Case 7e

Training algorithm	Simulated Annealing
Code modification	Corana, et. al., penalized
Starting point	Solution of Case 7a
Initial Temperature	5
Initial fopt	1.857e7
Final fopt	1.225e7
Penalty value	0.855
SSE value	1.225e7

In both cases 7c and 7d, the terminal behavior of the simulated annealing iterations was to diverge to values outside the limits of floating point arithmetic on the computer. However, in both cases intermediate values of the iterates provided better cost function values than either Matlab run by more than two orders of magnitude. This is an acceptable comparison because the best values for x_{opt} and f_{opt} are stored as the solutions. A better result was determined by case 7e than the starting point from the Matlab solution in case 7a, however, the solution was not an improvement over case 7b.

The x_{opt} from Cases 7c and 7d were used as starting points for Matlab training runs to complete the convergence to a minimum.

Case 7f

Training algorithm	Matlab
Starting point	Solution of Case 7c
Initial fopt	5467.79
Final fopt	5273.71

Case 7g

Training algorithm	Matlab
Starting point	Solution of Case 7d
Initial fopt	4880.40
Final fopt	4479.46

Althought Matlab produced better results than the simulated annealing runs in cases 7f and 7g, Matlab would have not reached these results without the aid of simulated annealing. Cases 7a and 7b trained by Matlab from random starting points yielded cost function values in the 10^6 and 10^7 range, presumably local minima. Cost function values in the range of 10^3 to 10^4 were determined from cases 7c and 7d before the solution wandered out of control, possibly a failure of the dynamic step size adjustment. This demonstrates that simulated annealing is capable of escaping from sub-optimal local minima to find better cost function values. Cases 7f and 7g were not cases of "Matlab beats simulated annealing again". Instead, Matlab was using a downhill algorithm to finish converging to a local minimum from a favorable starting point located by simulated annealing.

Chapter 6
Conclusions

In the course of this study, about 1400 lines of Fortran 90 code were written and maintained through several algorithm modifications to test simulated annealing as a training method for neural netowrks. Several variations of the simulated annealing algorithm were proposed by the Guidance and Control Branch (GCB) and tested using this software. Results of these tests were reported back to GCB. Presented in this paper is the foundation for possible modifications to a simulated annealing algorithm for continuous functions. Several variations were suggested which proved not to work well on the neural network training problem.

For the 7 training sets, 25 Matlab training runs were performed for comparison. At least 2 Matlab runs were performed for each training set. The second best result from Matlab runs was used for starting points for the simulated annealing training tests. Of the 17 simulated annealing runs, 6 produced results that went out of control (as indicated by the "All NaNs in output file"), and 4 did not improve on the non-global local minimum starting point. The remaining 7 runs settled in local minima. It is important to note that cases 7f and 7g could never have been run without the information gotten from simulated annealing cases 7c and 7d, respectively. This is a geniune simulated annealing contribution to the network training.

This is a proof-of-concept study. The Fortran 90 language allowed for ease of programming yet the software has not reached the point of production. The simulated annealing algorithm could use further development, particularly in the adjustment of the scale factors. A study of the initial parameters should be completed to determine what starting temperature, initial step size, reduction coefficient, etc., should be used.

References

Brainerd, Walter, Charles Goldberg, and Jeanne Adams. *Programmers Guide to Fortran 90 Second Edition* UNICOMP, 1994.

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*, Vol. 13, No. 3, 262-280, September 1987.

Dekkers, Anton and Emile Aarts. "Global Optimization and Simulated Annealing" *Mathematical Programming 50*, 367-393, 1991.

Demuth, Howard and Mark Beale. *Neural Network Toolbox User's Guide Version 3.0* The Mathworks, Inc. 1998.
http://www.mathworks.com/access/helpdesk/help/pdf_doc/nnet/nnet.pdf

Giesy, Daniel. "Sensitivity of Discrete Linear System Output Error to Variation in the Weighting Matrices and Bias Vectors of a Feedforward Neural Network Controller" NASA/TP-1998-Draft, October 1998.

Goffe, William, Gary Ferrier, and John Rogers. "Global optimization of statistical functions with simulated annealing" *Journal of Econometrics 60*, 65-99, 1994.

Horst, Reiner and Panos Pardalos. *Handbook of Global Optimization* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1995.

Kirkpatrick, S., C. Gelatt, Jr., and M. Vecchi. "Optimization by Simulated Annealing" *Science*, Volume 220, 671-680, 13 May 1983.

Maghami, Peiman. “Enhanced Attitude Control Experiment for SSTI Lewis Spacecraft” *AIAA Guidance, Navigation, and Control Conference*, New Orleans, LA, AIAA 97-3527, August 11-13, 1997.

“newsgroup” Archive name: ai-faq/neural-nets/part1, URL: ftp://ftp.sas.com/pub/neural/FAQ.html, Maintainer: Warren S. Sarle (saswss@unx.sas.com)

Nise, Norman. *Control Systems Engineering Second Edition* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1995.

Nocedal, Jorge and Stephan Wright. *Numerical Optimization* Springer, New York City, NY, 1999.

Vanderbilt, David and Steven Louie. “A Monte Carlo Simulated Annealing Approach to Optimization over Continuous Variables” *Journal of Computational Physics*, Volume 36, 259-271, 1984.

Wasserman, Philip and Tom Schwartz. “Neural Networks, Part 2: What are they and why is everybody so interested in them now?” *IEEE Expert*, 10-15, Spring 1988.